Algorithms for the Analysis and Synthesis of Tree Structured Communication Networks

Dorota M. Huizinga and Ewa Kubicka

Abstract
This paper describes a comprehensive approach to the analysis and synthesis of tree structured communication networks. First, a class of models for tree structured communication networks is proposed. Then, performance parameters such as communication delays and network reliability are defined, and efficient algorithms for calculating these parameters are provided. Subsequently, an application of a powerful, tree generating algorithm to the synthesis of optimal communication networks is described. The universal approach of this algorithm allows for its use in conjunction with the proposed model and the algorithms for calculating values of performance parameters. The paper shows sample optimal tree structured networks resulting from applying the synthesis algorithm for various optimization parameters.

1. INTRODUCTION

The cost-effective analysis and synthesis of communication networks has been the subject of broad research in the past decade [Ahuja, et al., 1993, Balakrishnan et al., 1991, Berry et al., 1994, Berry et al., 1995 and Berry et al., 1997, Sharma, et al., 1991,]. Since most nontrivial network synthesis problems are NP-complete [Berry, et al., 1995], the proposed solutions usually rely on heuristic approaches and deliver sub-optimal results [Banerjee et al, 1992, Berry et al., 1994, Berry at al., 1997, Minoux, 1989]. The cost functions in network synthesis problems are defined to model network behavior. These models reflect all-purpose network performance parameters as well as application-specific parameters. Examples of all-purpose parameters include the well studied latency, throughput, or fault tolerance [Stallings, 1999]. Examples of application-specific parameters consist of the average inter-node distance used in the design of the interconnection networks [Dandamudi et al., 1990] or a minimum end-to-end delay used in multimedia conferencing [Ramanathan at el, 1992].

Thus, an effective network synthesis methodology must be based on a sound model with realistic and efficiently computable cost functions that reflect the values of the performance parameters of the model.

The work presented in this paper addresses the above issues by proposing a novel, comprehensive approach to the analysis and synthesis of tree structured communication networks. Tree structured communication networks are of great importance as they describe topologies of hierarchical systems, can be used to define efficient packet routing in non-tree structured systems, or can be applied as access networks to other network structures [Berry et al., 1995]. Such networks are the focus of our work.

Specifically, in this paper, we propose a class of models for tree structured communication networks. The class is general enough to encompass a variety of performance parameters. For this class of models, we describe efficient, dynamic programming algorithms that calculate the following performance parameters: the worst case many-to-one communication delay, the worst case many-to-many communication delay, the average many-to-many communication delay, and the average size of a subnetwork. The values of these parameters define cost functions for the application of the network synthesis algorithm outlined in the subsequent part of this paper. The algorithm is based on the efficient generation and examination of all trees of a given order. It has been shown in our previous work [Kubicka, 1996], [Huizinga et al., 1997] that the number of steps per tree performed by this exhaustive algorithm is bound by a constant that does not depend on the number of nodes in the tree. Thus, to our knowledge, this is the fastest algorithm in its category. Expanding on our previous work, this paper illustrates the application of this algorithm to the synthesis of tree structured networks in conjunction with the algorithms for calculating network performance parameters. Sample optimal networks generated by the synthesis algorithm for various optimization parameters are shown in the last part of the paper. In conclusion, we discuss strengths and limitations of this work, and we outline future directions of our research.

In the next section the class of models for tree structured communication networks is defined. The algorithms that calculate network performance parameters are described in section three. Section four illustrates the application of the network synthesis algorithm and shows sample results of this application. The conclusion and future work are outlined in section five.

2. THE CLASS OF MODELS

Consider a tree structured communication architecture consisting of n processors (nodes), where n is an integer greater than zero. This configuration can be viewed as a rooted tree of order n, i.e. a tree consisting of n nodes (see Fig.1). Let's denote this tree by $\mathtt{T}$.

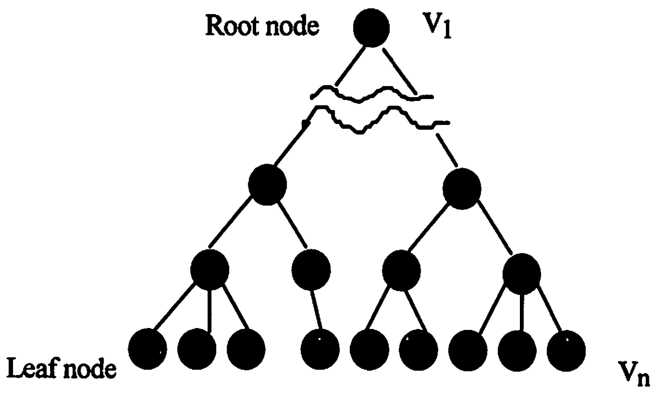


Fig.1. A tree structured data communication network with n nodes labeled v_1 through v_n.

An internal node of the hierarchy $\mathtt{T}$, can send/receive packets directly to/from its children and its parent. A leaf node can directly communicate only with its parent node, and the root node can directly communicate only with its children. If a node sends a packet to another node in a tree network, then there is exactly one path that this packet could travel. The time needed to travel this path depends on the path length and the congestion at each of the path nodes. The average communication delay between any two nodes for tree network $\mathtt{T}$ is one of its essential parameters. An $O(n^2)$, dynamic programming algorithm evaluating this parameter is described in the next section. However, synthesizing (constructing) a tree network that minimizes the average of communication delays between any two nodes is much more challenging. This is because the number of trees grows exponentially with the tree order n [Beyer et al., 1980, Wright et al., 1986].

In general, there are a number of parameters essential for evaluating network performance. Examples of such parameters include the worst case many-to-one communication delay, the average many-to-many communication delay, or the average size of the sub-network. We will denote any of these parameters by `OptimizationParameter`.

The class of the system models described below encompasses a wide range of such network parameters which can be calculated using a bottom-up approach.

The following paragraph outlines the model and its assumptions.

1. For tree network `T` of order n, let's denote an optimization parameter to be calculated by `OptimizationParameter`.

2. For each node of `T`, `OptimizationParameter` is evaluated in the "bottom up" manner. This means that for node v_i $1 \le i \le n$, with m children, denoted $v_{i_1}, v_{i_2}, \ldots, v_{i_m}$, `OptimizationParameter(`$v_i$`)` is a function `f` of the optimization criteria calculated for all children, topology of the subtree "below" v_i, denoted by `Top(`v_i`)`, and system dependent parameters $p_1, \ldots, p_w$. Therefore:

   $$\mathtt{OptimizationParameter}(v_i)=\mathtt{f}(\mathtt{OptimizationParameter}(v_{i_1}), \ldots, \mathtt{OptimizationParameter}(v_{i_m}), \mathtt{Top}(v_i), p_1, \ldots, p_w)$$

   For a leaf node, the value of the optimization parameter is set to a constant.

3. The value of the optimization parameter calculated for root v_1 of tree `T`, constitutes the final performance result for this tree, i.e. `OptimizationParameter(`v_1`)`.

Note, that since function "f" used in calculating `OptimizationParameter(`v_i`)` can be an arbitrary cost function, the above model encompasses a wide range of the networking scenarios.

In the next section we will describe dynamic programming algorithms for determining values of various network parameters. The algorithms are consistent with the class of models described above.

3. ALGORITHMS FOR CALCULATING NETWORK PARAMETERS

The subsequent sections describe algorithms for determining communication delays and reliability in tree networks. We will use the following notation for calculating communication delays.

For a tree network T, of order n, let T_{init} denote the maximum initialization time, i.e. the maximum time needed for the system startup. Let T_{net} denote the maximum communication delay between a child node and a parent node, i.e. the maximum time needed for a child node to send a packet directly to its parent, or the other way around. Let T_{cong} denote the congestion parameter, i.e. the parameter that determines the impact of traffic congestion on packet delivery delay in the system.
Let the node's v outdegree, i.e. the number of children for node v, be denoted by od(v). Let Penalty(v) be defined for node v as a function of od(v), i.e. the function that reflects the impact of the node outdegree on the node's transmission delay for each packet. Thus, T_{cong}*Penalty(od(v)) depicts node's delay due to its traffic congestion.
For any two nodes in a tree network, there is exactly one path between them. The communication delay between the extreme nodes of this path depends on the number of path links, transmission time along each path link (T_{net}) and traffic congestion delay at each path node v (T_{cong}*Penalty(od(v))).
The algorithms 1, 2, and 3 use the above assumptions in calculating properties of communication networks.

3.1 Worst Case Many-To-One Communication Delay

A routing problem is said to be many-to-one if more than one transmission packet can have the same destination [Leighton, 1992]. The worst case many-to-one communication delay is a parameter that determines the maximum amount of time necessary for an arbitrary node to communicate with a designated node in a network. Accurate estimation of this parameter is essential especially in real-time time applications, where the guaranteed worst case timing behavior is often critical.
In order to present our many-to-one communication model, let's consider a tree network system shown in Fig.1. Without loss of generality, let's assume that the designated node in our many-to-one communication model is the root of a tree network. We will call DelayToOne the optimization parameter in this system. Thus, DelayToOne is an instance of OptimizationParameter in our class of communication models, and it represents the worst case communication delay from any node to the root. For each node in tree T, there is exactly one path to the root.
Let arrayTree, be an array of size n, representing tree network T such that arrayTree[i] corresponds to the i'th node of T in the preorder traversal.
Thus, for the many-to-one communication delay model, each entry i, where $1 \leq i \leq n$, of the arrayTree, will hold the structure called nodeRecord:

```
nodeRecord  =  {
    int NumberOfChildren; //number of children for this node
    int FirstChild;
    int RightSibling;
    int Parent;
    float DelayToOne; //worst case all-to-one delay for subtree
                      //rooted at this node
 }
```

Presented below algorithm 1 calculates the worst-case communication delay from any node to the root. The algorithm progresses in the bottom-up manner, treating each node v as a root of a subtree rooted at v, and calculating the value of the slowest path from any of the leaf nodes to v. For each node v, this value is stored in variable `DelayToOne`. Thus, the value of `DelayToOne` for a parent node is calculated by finding the maximum of `DelayToOne` among its children, and adding to it $T_{net}+T_{cong}$`*Penalty(od(v))`.

Algorithm 1 – Computation of the worst case many–to-one communication delay.

//Given : `arrayTree[1..n]` of `nodeRecord` structures corresponding to tree

// network `T`

// Objective: find the worst case many-to-one communication delay in `T`

```
1 for (v=n; v ≥ 1; v=v-1) //Examine all records of the arrayTree starting from
                          //the n^th, ending with 1^st
2   currentNode=arrayTree[v];
3   if (currentNode.NumberOfChildren==0)        // It's a leaf node
4        currentNode. DelayToOne= $T_{init}$;
5   else {        //It's a non-leaf node
6        slowestChildDelayToOne=findMaxChildDelayToOne(currentNode)
         // search all the children of the current node to find the child that
         //maximizes DelayToOne and
         // store its DelayToOne value in variable slowestChildDelayToOne
7        currentNode.DelayToOne =
              slowestChildDelayToOne
              $+T_{net}+T_{cong} \times$ Penalty(arrayTree[i].NumberOfChildren);
         }
```

Theorem 1.
Algorithm 1 calculates the worst case many-to-one communication delay for a tree network `T`, defined above. The value of this worst case delay is stored in the variable `DelayToOne` of `nodeRocord` corresponding to the root of `T`; (i.e. in `arrayTree[1].DelayToOne`).

Proof.
We proceed by the Strong Form of Induction on n, where n is the order of tree `T`.
Step 1 of induction.
Let n=1. It is clear that for a tree consisting of just one node, algorithm 1 trivially finds the worst case delay by the direct assignment of T_{init} to the `arrayTree[1].DelayToOne` (line 4 of algorithm 1).
Inductive step.
Assume now that the above conjecture is true for all tree networks of order n <=k. We will show that it is also true for an arbitrary `T'` of order k+1. A rooted tree `T'` of order k+1, consists of subtrees connected to root r of `T'`, as depicted in Fig.2.
Note that in order to calculate the worst case delay for the root r of `T'`, we need to add to the delay of its "slowest" child, the transmission time from this child to `r`, and the penalty for congestion at the root (line 7 of algorithm 1). Each of the proper subtrees of tree `T'` has no more than k nodes. Thus, according to our inductive assumption the variable `DelayToOne` for each child of `r` holds the worst case delay for its subtree, and the `slowestChildDelay` can be found by taking the maximum of `DelayToOne` over all children of r (line 6 of algorithm 1).

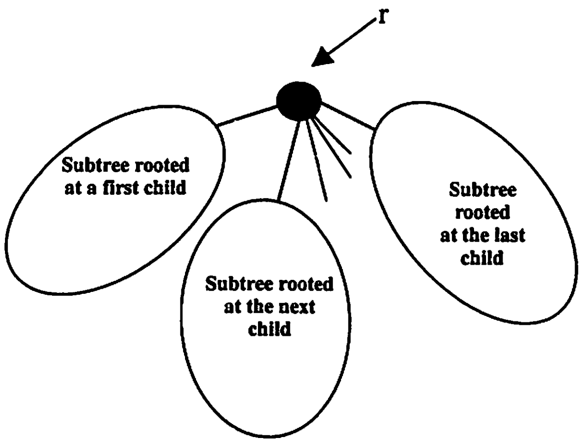

r
Subtree rooted at a first child
Subtree rooted at the next child
Subtree rooted at the last child

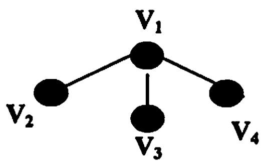
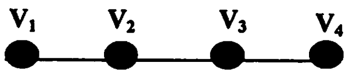
Fig. 2. A rooted tree `T'` of order k+1.

Since the **arrayTree** represents the tree **T′** in the preorder traversal and the main loop of the algorithm 1 iterates from the largest to the smallest array index (from n down to 1), it is guaranteed that the value of **DelayToOne** for each node of the tree **T′** is calculated in a bottom-up manner, i.e. the calculations for all children precede the calculation for the parent, validating lines 6 and 7 of the algorithm □.

Lemma 1.
For tree network **T** of order n, the computation complexity of the algorithm 1 is O(n).

Proof.
It is sufficient to notice that each node of the tree **T** is visited at most twice; once due to the main loop in line 1 and the second time in line 6 while searching for the slowest child of the current node □.

To illustrate calculations of parameter **DelayToOne** let's consider the following example. Let **T** be a tree of order 4, let $T_{init} = 1$, $T_{net} = 2$, $T_{cong} = 1$, and $Penalty(od(v)) = (od(v))^2$.
The value of **DelayToOne** depends on the topology of the tree network which is illustrated by an example in table 1.

Topology : star	Topology: path
	
For V_4,V_3, and V_2 we have: DelayToOne = $T_{init} = 1$ For V_1 we have: **DelayToOne = $T_{init} + T_{net} + T_{cong} \times (od(V_1))^2 = 1 + 2 + 1 \times 3^{2=} 1+2+1 \times 3^2 = 12$**	For V_4 we have: DelayToOne= $T_{init} = 1$ For V_3 we have: DelayToOne = $T_{init} + T_{net} + T_{cong} \times od(V_3))^2$

	For V_2 we have: DelayToOne= T_{init} +T_{net}+ $T_{cong} \times od(V_3))^2$ + T_{net} + $T_{cong}(od(V_2))^2$ For V_1, and therefore for the whole tree T, we have **DelayToOne= T_{init} +T_{net}+ $T_{cong}(od(V_3))^2)$+ T_{net} + $T_{cong}(od(V_2))^2$+ T_{net}+ $T_{cong}(od(V_1))^2)$= = $1+2+1 \times 1^2 +2 + 1 \times 1^2+2 + 1 \times 1^2$ =10**

Table 1. Values of `DelayToOne` (worst case many-to-one communication delay) calculated for two different network topologies.

3.2 Worst Case Many-To-Many Communication Delay

A many-to-one routing problem can be generalized to a many-to-many routing problem if multiple destinations for packets are allowed [Leighton,1992]. The worst-case many-to-many communication delay is a parameter that determines the maximum amount of time necessary for a packet to travel between any two arbitrary nodes of a tree network.

Again, let's consider a tree network system shown in Fig.1. The optimization criterion called `DelayToMany` for this system is the worst-case communication delay between any two nodes of `T`. We are going to assume that the communication time is symmetrical, i.e. the amount of time needed to send a packet from node V_i to V_j is equal to the amount of time needed to send a packet from V_j to V_i. Let T_{init}, T_{net}, T_{cong}, the `Penalty(od(v))` function, and `arrayTree`, be defined as at the beginning of section 3. Let the new optimization parameter in `nodeRecord` be denoted by variable `DelayToMany`.

Thus, the `nodeRecord` in this model holds the following data.

```
 nodeRecord   =   {
     int NumberOfChildren; //number of children for this node
     int FirstChild;
     int RigthSibling;
     int Parent;
```

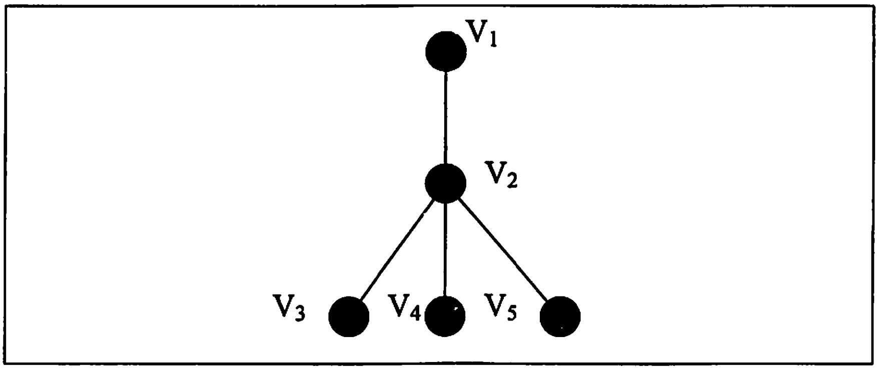
```
    float DelayToOne;  //worst case many-to-one delay for subtree
                       //rooted at this node
    float DelayToMany; //worst case many-to-many delay for subtree
                       //rooted at this node
}
```

Presented below algorithm 2 calculates the worst-case communication delay between any two nodes in a tree network. The algorithm progresses in the bottom-up manner, treating each node v as a root of a subtree rooted at v. For each such node v, the algorithm calculates the value of the slowest path in the subtree rooted at v and stores its value in variable `DelayToMany`. The value of `DelayToMany` is calculated by finding the maximum among the paths that go through v, and the paths that do not go through v. The proof of theorem 2 details the most essential steps of this algorithm.

Algorithm 2 – Computation of the worst case many–to-many communication delay.
// Given : `arrayTree[1..n]` of `nodeRecord` structures corresponding to // tree network `T`
// Objective: find the worst case many-to-many communication delay in `T`
// Comment: `DelayToOne` for each `nodeRecord` is calculated as in Algorithm 1.

```
1 for (v=n; v ≥ 1; v=v-1) //Examine all records of the arrayTree starting from
//the n^th, ending with 1^st
2   currentNode=arrayTree[v]; slowestRPathDelayToMany=0;
3   if (currentNode.NumberOfChildren==0)        // It's a leaf node
4       currentNode.DelaytoMany= $T_{int}$;
    else {    //It's a non-leaf node
5       slowestChildDelayToOne=findMaxChildDelayToOne(currentNode);
        //search all the children of the current node to find the child that
        //maximizes DelayToOne.
6       if (currentNode.NumberOfChildren>1){
7
    secondSlowestChildDelayToOne=findSecondMaxChildDelayToOne(curr
    entNode);
        //find the second child that maximizes DelayToOne among all other
        //children of the current node (excluding the slowest one)and store its
        //DelayToOne in SecondSlowestChildDelay
8           slowestRPathDelayToMany =
          = slowestChildDelayToOne+secondSlowestChildDelayToOne +
          2 × $T_{net}$+ $T_{cong}$ × Penalty(currentNode.NumberOfChildren);}//end
          if
      //slowestRPathDelayToMany holds the value of the slowest path
      //between any/two nodes of the subtree rooted at v that goes through
      //v
      else
9         slowestRPathDelayToMany =currentNode.DelayToOne;
10        //there is only one child
11    slowestNRPathDelayToMany=findMaxChildDelayToMany(currentN
ode);
      //find the child that maximizes DelayToMany among all children of
      // the current node and store
      //its DelayToMany value in variable slowestNRPathDelayToMany;
      // this represents the slowest path that does not go through v in the
      //subtree rooted at v.
12    if (slowestRPathDelayToMany >slowestNRPathDelayToMany)
13        currentNode.DelayToMany=slowestRPathDelayToMany;
      else
14        currentNode.DelayToMany=slowestNRPathDelayToMany;
      // the worst case many-to-many delay is a maximum of the slowest of
      // all rootpaths and non-rootpaths
```

To illustrate how this parameter is calculated, let's consider the following example. Let T be a tree network of order 4, let T_{init}=1, T_{net}=2, T_{cong}=1, and Penalty(od(v))= $(od(v))^2$. The values of DelayToMany for star and path network topologies of T are calculated in table 2.



Consider a rooted tree T of order 5 depicted above.
Assume that:
$T_{init}=1$; $T_{net}=2$; $T_{cong}=1$;
Penalty(v)=$[od(v)]^2$;
Then, for the leaf nodes V_5, V_4, and V_3, we have the following values:
DelayToMany= $T_{init}=1$;
For the node V_2 Algorithm 2 computes:
slowestRPathDelayToMany=$1+1+2\times 2+1\times 3^2=15$
slowestNRPathDelayToMany=1
DelayToMany=Maximum(15,1)=15
For the node V_1 and therefore for the whole tree T, we have:
slowestRPathDelayToMany=$1+2+3^2+2+1=15$
slowestNRPathDelayToMany=15
DelayToMany=15

Table 2. Example for algorithm 2.

Theorem 2.
Algorithm 2 calculates the worst case many-to-many communication delay for a tree network `T`, defined above. The value of this worst case delay is stored in the **`arrayTree`** record corresponding to the root of tree `T`, i.e. in **`arrayTree[1].DelayToMany`**.

Proof.
Similar to theorem 1, the proof proceeds through the strong form of induction, and the first step of induction is trivially true due to the assignment of **`Tinit`** to the **`currentNode.DelayToMany`** in line 4. Let's assume that our conjecture is true for an arbitrary tree `T` of order k or less. We will show that it is also true for a tree `T'` of order k+1. Again, since the algorithm works in a bottom-up manner, the value of **`DelayToMany`** is calculated for the offspring nodes before it is calculated for the parent nodes (line 1 of algorithm 2). This is essential because in order to calculate the worst case many-to-many communication delay for a tree `T'` rooted at root `r`, we need to examine all children of r first. Since all subtrees rooted at children of `r` have an order less or equal to k, we can apply to them our inductive hypothesis. Note that the longest communication path between any two nodes in tree `T'` can either go through the root r of this tree, or not. The algorithm calculates the time of the longest path that goes through the root r by identifying the largest and the second to largest values of **`DelayToOne`** among all children of r (**`slowestChildDelayToOne`** in line 5 and

secondSlowestChildDelayToOne in line 7). The time needed to communicate between these two slowest children, which includes the communication to and from the parent ($2 \times T_{net}$), and T_{cong} ×Penalty(currentNode.NumberOfChildren), which is the congestion delay at the root, is then added to their delays (line 8). If r has only one child then the time of the longest path that goes through r is simply equal to its DelayToOne. The algorithm calculates the time of the longest path that does not go through the root r by finding the maximum of DelayToMany among all children of r (line 10). Thus the value of DelayToMany at root r of T′ is determined by taking the maximum of the longest paths that go through the root (slowestRPathDelayToMany) and the paths that do not go through the root (slowestNRPathDelayToMany) in lines 11-14 □.

Lemma 2.
For tree network T of order n, the computation complexity of algorithm 2 is O(n).
Proof.
It is sufficient to note that each node of tree T is visited at most three times: once as a parent node and twice as a child □.

3.3 The Average Many-To-Many Communication Delay

The average amount of time needed for a packet to travel between any two nodes of the network is another parameter that determinates this network's performance.
Again, let T_{init}, T_{net}, T_{cong}, and arrayTree, be defined as at the beginning of section 3. Let DelaySumToMany denote the sum of delays for all paths in the subtree rooted at the currentNode. In addition to DelaySumToMany, nodeRecord for this model needs to hold the order of the subtree rooted at the currentNode, the sum of delays for the paths that end at the currentNode (DelaySumREPaths), and the sum of delays for the paths that do not end at the current node (DelaySumNREPaths).

Thus, the nodeRecord for this model holds the following data.

```
nodeRecord  =   {
    int NumberOfChildren; //number of children for this node
    int order; //number of nodes in the tree rooted at the current node
    int FirstChild;
    int RightSibling ;
    int Parent;
    float DelaySumREPaths ;//sum of delays for the paths that end at
                           // the current node
    float DelaySumNREPaths ;//sum of delays for the paths that do
                           // NOT end at currentNode
    float DelaySumToMany  //sum of delays for all paths in the subtree
                           // rooted at currentNode
}
```

Algorithm 3 calculates the average many-to-many communication delay between any two nodes in a tree network. Similarly to algorithms 1. and 2, algorithm 3 progresses in the bottom-up manner, treating each node v as a root of a subtree rooted at v. For each such node v, the algorithm calculates the values of the sum of delays for all paths that end at v (***DelaySumREPaths***), and the sum of delays for all paths that do not end at v (***DelaySumNREPaths)***. Thus, the sum of values of these delays, calculated for the root of tree T, and divided by the number of all possible paths T, results in the average many-to-many communication delay for the tree stored in ***AverageDelayToMany***. The proof of theorem 3 details the most essential steps of this algorithm.

Algorithm 3 – Computation of the average of many–to-many communication delay.
// Given : `arrayTree[1..n]` of `nodeRecord` structures corresponding
// tree network `T`
// Objective: find the average many-to-many communication delay in `T`

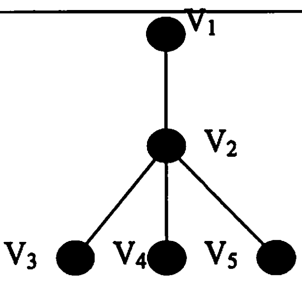
```
1 for (v=n; v≥ 1; v=v-1) //Examine all records of the arrayTree starting from
     //the nth, ending with 1st
2    { currentNode=arrayTree[v];
3    if (currentNode.NumberOfChildren==0) // It's a leaf node
4       { currentNode.DelaySumREPaths= $T_{init}$;
5        currentNode.DelaySumNREPaths=0;}
     else            //It's a non-leaf node
6      { currentNode.DelaySumREPaths=
      DelaySumREPaths(arrayTree,currentNode)
         //call function DelaySumREPaths
7    currentNode.DelaySumNREPaths=
         DelaySumNREPaths(arrayTree,currentNode)} // call function
         //DelaySumNREPaths
8    currentNode.DelaySumToMany  = currentNode.DelaySumREPath
                                    +currentNode.NREPaths;}
```

```
9  } //end of loop
10 if ( n>1)
11      averageDelayToMany =
        currentNode.DelaySumToMany/choose(n,2);
        //The sum of delays for all paths is divided by the number of paths (n
        //choose 2)
12 else
13      averageDelayToMany=$T_{init}$;

DelaySumREPaths(arrayTree,currentNode)
// Objective:for the subtree rooted at the currentNode, determine the sum of
// communication delays for all the paths that end at the currentNode
1    runningSum=0;
2    tempChild=currentNode.FirstChild;
3    for( i=1; i$\leq$currentNode.NumberOfChildren ;i++) {
4         runningSum=runningSum+tempChild.DelaySumREPaths
                  + tempChild.order$\times$( $T_{net}$+
                  $T_{cong}$$\times$Penalty(currentNode.NumberOfChildren));
5         tempChild=tempChild.RightSibling;}
6    return runningSum;

DelaySumNREPaths(arrayTree,currentNode)
// Objective:for the subtree rooted at the currentNode, determine the sum of
//communication delays for all the paths that do NOT end at the currentNode.
1         runningSum=0;
2         tempChild=currentNode.FirstChild;
3         for( i=1; i$\leq$currentNode.NumberOfChildren ;i++) {
4              runningSum=runningSum+tempChild.DelaySumNREPaths;
5              nextChild=tempChild;
6              for (j=i+1; i$\leq$currentNode.NumberOfChildren; i++) {
7                   nextChild=nextChild.rightSibling;
8                   runningSum=runningSum
                    +tempChild.DelaySumREPaths$\times$ nextChild.order
                    +nextChild.DelaySumREPaths.tempChild.order
                    +tempChild.order$\times$ nextChild.order
                    $\times$(2$\times$Tnet +Penalty(currentNode.NumberofChildren));
9         tempChild=tempChild.RightSibling;}
10   return runningSum;
```

To illustrate how algorithm 3 works, let's consider the following example depicted in table 3.



Consider a rooted tree T of order 5 depicted above.
Assume that:
$T_{init}=1$; $T_{net}=2$; $T_{cong}=1$;
$Penalty(v)=[od(v)]^2$;

Then, for the leaf nodes V_5, V_4, and V_3, we have the following values:
DelaySumRPaths=1;
DelaySumNRPaths=0;

For the node V_2 Algorithm 3 computes:
$$\text{DelaySumRPaths} = [1+1\times(2+1\times3^2)] + [1+1\times(2+1\times3^2)] + [1+1\times(2+1\times3^2)] = 36.$$
$$\text{DelaySumNRPaths} = 0 + [1\times1 + 1\times1 + 1\times1 + (2\times2 + 3^2)] + + [1\times1 + 1\times1 + 1\times1 + (2\times2 + 3^2)] + + 0 + [1\times1 + 1\times1 + 1\times1 + (2\times2 + 3^2)] + 0 = 45.$$

For the node V_1 we have:
$\text{DelaySumRPaths} = 36 + 4\times(2 + 1^2) = 48.$
$\text{DelaySumNRPaths} = 45.$
$\text{DelaySumToMany} = 48 + 45 = 93.$

Finally, for the whole tree T, the value of the optimization parameter is:
AverageDelayToMany = 93/(5 choose 2) = 9.3.

Table 3. Example for Algorithm 3.

Theorem 3.
Algorithm 3 determines the average many-to-many communication delay for a tree network `T`, defined above. The value of average many-to-many delay is stored in variable **`averageDelayToMany`**.

Proof.
Again, the proof proceeds through the strong form of induction, and our inductive conjecture is true for n=1 due to the direct assignment in line 13. Let's assume that the inductive hypothesis is true for any tree `T` of order less or equal to k. Consider now a tree `T'` of order k+1, as illustrated in figure 2.

The algorithm first calculates the sum of delays for all paths that end at the `currentNode` (which is corresponds to root `r`) by calling function `DelaySumREPaths`. Note that for each of the children of `currentNode` the value of this sum had already been calculated. Each path that ends at a child node of the `currentNode` needs to be now extended to the new root, i.e. `currentNode`.
Thus, for each of these paths the communication delay needs to be increased by T_{net} + T_{cong}×`Penalty(currentNode.NumberOfChildren)`. The order of the subtree rooted at this child (`tempChild.order`) determines the number of such paths, including the path from this child to the `currentNode`, which explains line 4 of function `DelaySumREPaths`. These values are added for all children of the `currentNode` producing the total that is returned in variable `runningSum` in line 6 of function `DelaySumREPaths`.

After that the algorithm proceeds to calculating the sum of delays for the paths in the subtree rooted at `currentNode` that do not end at the `currentNode` by calling function `DelaySumNREPaths`. For each of the children of the `currentNode` this function adds its `DelaySumToMany` to the `runningSum` (line 4 of the function). This accounts for all paths included completely in the subtree rooted at this child. The paths that go through the `currentNode` are accounted for in lines 6, 7 and 8 of the function. Note that all paths that end at one child of the `currentNode` can be extended via the `currentNode` to any other child of the `currentNode`. Thus for each pair of `currentNode` children: `tempChild` and `nextChild`, line 8 calculates the sum of delays for all paths that start in a subtree rooted at `tempChild` going through the `currentNode` and ending in the subtree rooted at `nextChild`. The loop in line 6 of the function assures that all not previously considered siblings of the child `tempChild` are accounted for. Since the main loop iterates through all the children of the current node, the final result returned in line 10 contains the sum of delays of all paths that do not end at the `currentNode`.

In order to conclude the proof we need to notice that line 11 of the algorithm 3 calculates the average by dividing the `DelaySumToMany` by the number of paths in tree `T` (n choose 2).

Lemma 3.
For tree network `T` of order n, the computation complexity of algorithm 3 is $O(n^2)$.

Proof.
Note that each node of tree network `T` is visited once as a parent node. Additionally, due to the nested loop in function *DelaySumNREPath*, each child node is visited at most as many times as many siblings it has □.

3.4 The Average Size of a Subnetwork

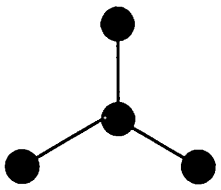
Consider a node or link failure in a tree network. Such a failure leads to a partition of the network into disconnected subnetworks each of which is a subtree of the original tree. The parameter that determines the average size of such a subnetwork allows to estimate the average number of nodes that stay connected during network partition.

An average order of a subtree for tree `T`, `Ave(T)` can be expressed by the following formula:

$$\text{Ave}(T) = \frac{\sum_{\text{all subtrees } T_s \text{ of } T} \text{order of } T_s}{\text{number of all subtrees } T_s \text{ of } T}$$

To illustrate the above formula let's consider the following tree `T` of order 4 and its subtrees.

T:



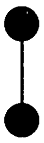
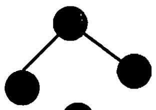
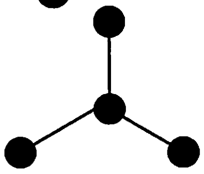
Type of subtree	amount	order of the subtree	number of vertices
	4	1	4
	3	2	6
	3	3	9
	1	4	4

Fig.3. Example of calculations of the average order of subtree for tree **T** of order 4.

Thus, we get the following result:

$$\mathbf{Ave\,(T)} = \frac{4+6+9+4}{4+3+3+1} = \frac{23}{11} = 2.09$$

Let **arrayTree** of defined below **nodeRecords** represents tree T in preorder traversal. Note that for the **nodeRecord**, in addition to the fields already defined in the previous sections, we introduce four new fields needed to calculate the average size of the subtree. For each tree rooted at the **currentNode** (see algorithm 4) these fields hold the number of its subtrees containing the root, the number of subtrees not containing the root, and the corresponding to the sets of these subtrees, the number of vertices in each of them.

Thus, the following fields are defined for the **nodeRecord**.

```
nodeRecord  = {
   int NumberOfChildren;  //number of children for this node
   int FirstChild;
   int RightSibling ;
   int Parent;
   int NumberOfRSubtrees;  // number of all  subtrees containing this
                           // node (Rs)
   int NumberOfRVertices;  // total number of  vertices in all subtrees
                           //containing this node (Rv)
   int NumberOfNRSubtrees; // number of subtrees not containing this
                           //node (Ns)
   int NumberOfNRVertices;// total number of vertices in all subtrees
                           // not containing this node (Nv)
 }
```

Our goal is to compute the average order of a subtree for a given tree T. We will concentrate on the calculations of the number of subtrees and vertices defined in the `nodeRecord`. We will use the following abbreviated notation. Let **Rs** denote the number of subtrees of tree `T` containing root `r`, let **Rv** denote the total number of vertices in all subtrees counted in **Rs**. Similarly, let **Ns** denote the number of subtrees of `T` not containing `r`, and let **Nv** denote the total number of vertices in all subtrees counted in **Ns**. Then, of course the following holds for tree `T`:

$$\mathbf{Ave\,(T)} = \frac{\mathbf{Rv + Nv}}{\mathbf{Rs + Ns}}$$

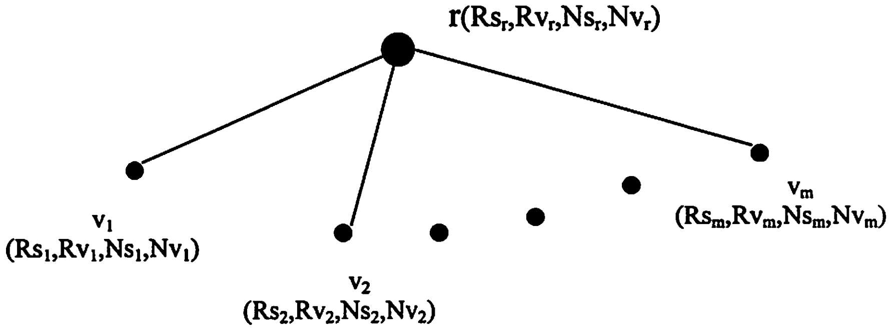


Fig. 4. Tree `T` with root r and children $v_1,v_2, \ldots v_m$.

Let $\mathbf{r}$'s children are denoted by $v_1, v_2, \dots v_m$. Let's assume that values Rs_i, Rv_i, Ns_i, Nv_i are already calculated for every subtree with the root at v_i $1 \le i \le m$. This situation is depicted in figure 4.

The correlation between values of Rs_r, Rv_r, Ns_r, Nv_r calculated for root r, and values of Rs_i, Rv_i, Ns_i, Nv_i, where $1 \le i \le m$, calculated for r's children, is established in the next lemma.

Lemma 4.
For the notation described above the following equalities hold:

(a) $Rs_r = \prod_{i=1}^{m} (1 + Rs_i)$

(b) $Rv_r = Rs_r + \sum_{j=1}^{m} (Rv_j \prod_{i \ne j} (1 + Rs_i)) = Rs_r(1 + \sum_{j=1}^{m} \frac{Rv_j}{(1 + Rs_j)})$

(c) $Ns_r = \sum_{i=1}^{m} (Rs_i + Ns_i)$

(d) $Nv_r = \sum_{i=1}^{m} (Rv_i + Nv_i)$

Proof:
For tree T with root r, let T_i denote the branch with the root at v_i. Consider now a subtree of T containing r. Its intersection with T_i is either empty or forms a subtree containing v_i. Thus, to select a subtree of T containing r we have $1+Rs_i$ possibilities for each branch and consequently

$$Rs_r = \prod_{i=1}^{m} (1 + Rs_i).$$

The total number of vertices in all subtrees of T_i containing v_i is given by Rv_i. Those vertices are repeated as many times as many subtrees of $T-T_i$ containing r we have, and this quantity is given by $\prod_{i \ne j} (1 + Rs_i)$. Thus the total number of vertices is all subtrees of containing r, not counting the root itself is equal to $\sum_{j=1}^{m} (Rv_j \prod_{i \ne j} (1 + Rs_i))$. The root has to be present in each one of Rs_r subtrees and therefore this value is added to the previous to obtain Rv_r.

A subtree of T not containing r is entirely contained inside T_i for some i. Thus, it is a subtree of T_i either containing v_i or not. Therefore we have $Rs_i + Ns_i$ of those subtrees for $1 \le i \le m$ and as a result we obtain

$$Ns_r = \sum_{i=1}^{m} (Rs_i + Ns_i).$$

Similarly, we receive $Nv_r = \sum_{i=1}^{m} (Rv_i + Nv_i)$ □.

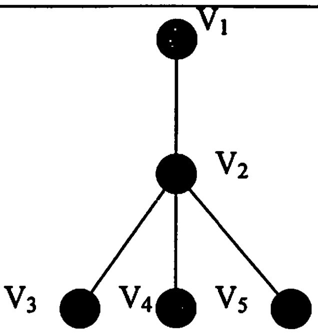
Lemma 4 leads us to the following algorithm that calculates the average size of a subnetwork.

Algorithm 4 – Computation of the average size of a subnetwork in a tree network.
//Given : `arrayTree[1..n] of nodeRecord` structures corresponding
// to tree network T
// Objective: find the average size of a subnetwork in T

```
1 for (v=n; v≥ 1; v=i-1)  //Examine all records of the arrayTree starting from
                           // the nth ending with 1st
2   currentNode=arrayTree[v];
3   if (currentNode.NumberOfChildren==0){        // It's a leaf node
4       currentNode.NumberOfRSubtrees=1;
5       currentNode.NumberOfRVertices=1;
6       currentNode.NumberOfNRSubtrees=0;
7       currentNode.NumberofNRVertices=0;}
8   else {   //It's a non-leaf node
9       SumNumberOfRSubtrees=1; SumNumberofRVertices=0;
10      SumNumberofNRSubtrees=0; SumNumberOfNRVertices=0;
                                       //initialize running sums
11      tempChild=currentNode.FirstChild
12      for(i=1;i≤currentNode.NumberOfChildren;i++){ //loop through all
                                                    //children
13          SumNumberOfRSubtrees=SumNumberOfRSubtrees
                                  × (1+tempChild.NumberOfRSubtrees);
14          SumNumberOfNRSubtrees=SumNumberOfNRSubtrees
                                  + temptChild.NumberOfRSubtrees
                                  + tempChild.NumberOfNRSubtrees;
15          SumNumberOfNRVertices= SumNumberOfNRVertices
                                  + tempChild.NumberOfRVertices
```

```
                                   + tempChild.NumberofNRVertices;
16          tempChild=tempChild.RightSibling;}
17      tempChild=currentNode.FirstChild;
18      for(i=1;i≤currentNode.NumberOfChildren;i++){ //loop again
                                                  // through all children
19          SumNumberOfRVertices= SumNumberOfRVertices
                                   + tempChild.NumberOfRVertices
                                  /(1+tempChild.NumberOfRSubtrees);
            temptChild=tempChild.RightSibling;}
21  } //end else
22  currentNode.NumberOfRSubtrees=SumNumberOfRSubtrees;
23  currentNode.NumberOfRVertices=
        SumNumberOfRSubtrees × (1 +SumNumberOfRVertices);
24  currentNode.NumberOfNRSubtrees=SumNumberofNRSubtrees.
25  currentNode.NumberOfNRVertices=SumNumberOfNRVertices.
26  averageSizeOfSubtree=(currentNode.NumberOfRVertices+currentNode.
    NumberOfNRVertices)/(currentNode.NumberOfRSubtrees+currentNode.
    NumberOfNRSubtree)
```

Table 4 illustrates an application of algorithm 4.



Consider a rooted tree T depicted in the above figure.

For the leaf nodes V_5, V_4, and V_3, we have the following values:
NumberOfRSubtrees = 1.
NumberOfRVertices = 1.
NumberOfNRSubtrees = 0.
NumberOfNRVertices = 0.

For the node V_2 Algorithm 4 computes:
SumNumberOfRSubtrees = 1 × (1 + 1) × (1 + 1) × (1 + 1) = 8.
SumNumberOfNRSubtrees = 0 + (1 + 0) + (1 + 0) + (1 + 0) = 3.
SumNumberOfNRVertices = 0 + (1 + 0) + (1 + 0) + (1 + 0) = 3.
SumNumberOfRVertices = 0 + 1/(1 + 1) + 1/(1 + 1) + 1/(1 + 1) = 3/2.
NumberOfRSubtrees = 8.
NumberOfRVertices = 8 × (1 + 3/2) = 20.
NumberOfNRSubtrees = 3.
NumberOfNRVertices = 3.

Finally, for the node V_1 and therefore for the whole tree T, we have:
SumNumberOfRSubtrees = 1 x (1 + 8) = 9.
SumNumberOfNRSubtrees = 0 + (8 + 3) = 11.
SumNumberOfNRVertices = 0 + (20 + 3) = 23.
SumNumberOfRVertices = 0 + 20/(1 + 8) = 20/9.

NumberOfRSubtrees = 9.
NumberOfRVertices = 9 x (1 + 20/9) = 29.
NumberOfNRSubtrees = 11.
NumberOfNRVertices = 23.

Now, the optimization parameter for the tree T can be computed:
AverageSizeOfSubtree = (29 + 23)/(9 + 11) = 2.6.

Table 4. Example for algorithm 4

Theorem 4.
Algorithm 4 determines the average size of a subnetwork for a tree network `T`, defined above. The value of this average is stored in variable **`averageSizeOfSubtree`**.

Proof.
Again, the proof is by induction by the order of the tree. In order to show that algorithm 4 calculates the average size of subtree in tree `T` it is sufficient to notice that lines 12 through 16 are the algorithmic implementations of equations (a), (c) and (d) of lemma 4. Lines 17 through 21, and line 23 are the algorithmic implementation of equation (b). The assignment of variable **`averageSizeOfSubtree`** in line 26 according to the formula for **`Ave(T)`** concludes the proof □.

Lemma 5.
For tree T of order n, the computation complexity of algorithm 4 is O(n).
Proof.
It is sufficient to note that each node of tree T is visited at most three times: once as a parent node and twice as a child □.

4. SYNTHESIS OF OPTIMAL TREE NETWORKS

4.1 The Network Synthesis Algorithm

The algorithms presented in the previous section determined specific parameters for a given tree network T of order n. In many situations, however, it is desirable to identify a network topology that optimizes a desired parameter. For example, we may want to identify a tree network that minimizes the average many-to-many communication delay, or maximizes the average size of the subnetwork, or both.

While devising a low complexity algorithm for any optimization problem is usually the best approach, in many cases exhaustive search is inevitable. The algorithm described below was designed to deal with such problems. The strength of it lies in its flexibility; i.e. the same approach can be used to analyze a number of various communication models, including all those described in section 3. The objective of the algorithm is to identify the topology of a tree network that optimizes a predefined parameter. The algorithm is based on the methodologies for generating all trees of a given order proposed in [Beyer et al., 1980] and [Wright et al., 1986], which utilize the concept of canonical level sequences for representation of trees.

Subsequently, the necessary definitions will be introduced, the data structures used in the algorithm will be described, and the algorithm principles will be detailed.

Let T be a rooted tree of order n (number of nodes is equal to n), where the numbers 1, 2,...,n are assigned to the nodes of T by the preorder traversal.

Definition 1. The level of a vertex i, denoted by `l(i)` is one more than the distance from the vertex i to the root (e.g. the level of the root is 1).

Definition 2. The level sequence of T, denoted by L(T), is the sequence of the levels of all its vertices obtained by the preorder traversal of T.

Note, that a tree can have many different level sequences since interchanging relative positions of any two sibling rooted subtrees may change its level sequence.

Definition 3. The canonical level sequence of T is lexicographically the largest level sequence of T. (The canonical level sequence gives a unique representation of T.)

The following paragraph defines and exemplifies the successor function that transforms any canonical level sequence other than [1,2,2,...,2] into the next canonical level sequence in decreasing lexicographical order [Beyer et al., 1980].

Definition 4. Let L(T) = [l(1),l(2),..,l(q),..,l(p),..,l(n)] denote the canonical level sequence of tree T with p being the largest vertex number with level greater than 2, and q being the vertex number of its parent. The successor S(T) of tree T with the canonical level sequence L is the tree corresponding to the canonical level sequence obtained by applying the following function to L:

$$
\begin{array}{lll}
\mathrm{SUCC}(L) = [s_1, s_2, \ldots, s_n], & & \text{where} \\
 & s_i = l(i) & \text{if } 1 \le i < p \\
\text{and} & s_i = l(i-p+q) & \text{if } p \le i \le n.
\end{array}
$$

For example, the successor S(T) of the tree T with the level sequence L(T)=[1,2,3,3,3,2,2], has the following level sequence L(S(T))=[1,2,3,3,2,3,3] (see Fig. 5). For this example the position of p is equal to 5 and the position q is equal to 2.

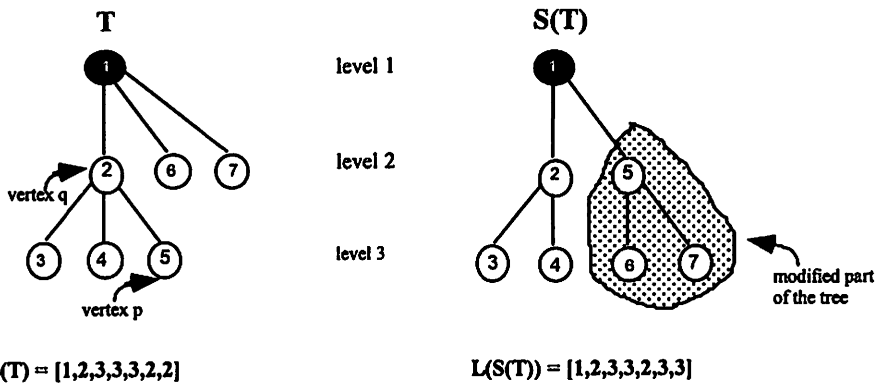


Fig. 5. A sample tree and its successor.

The successor function, denoted by **SUCC**, changes the levels of the vertices from p to n, i.e., the first p-1 vertices in a level sequence remain unchanged. It has been shown in [Beyer et al., 1980] that the average number of vertices changed by **SUCC** is not greater than two, regardless of the order n of the trees. An array called `arrayTree` of n records is used to represent a tree `T` of order n. The i-th record corresponds to the i-th vertex in the preorder traversal of the tree `T`. Each `nodeRecord` of `arrayTree` has the following structure:

```
nodeRecord  =   {
    int NumberOfChildren;  //number of children for this node
    int FirstChild;
    int RigthSibling;
    int Parent;
    int Level;   // level of this node (distance from the root)
    type OptimizationParameter  //value of the optimization
                           // parameter for the subtree  rooted at this node
    }
```

In this representation, every vertex i of tree `T` can be viewed as the root of the tree formed by all the vertices "below" i, or more precisely, `T(i)` is induced by all the vertices v of `T` such that i belongs to the unique path from v to the root of `T`. Note that the above data structure contains the value of the optimization criterion for vertex i, which depends on the topology of `T(i)`. It also contains the level of each vertex i and, therefore, the level sequence of the whole tree can be identified. The successor function `SUCC` can be easily modified to produce not only the level sequence of the current tree successor, but also its whole structural representation given in `arrayTree`, namely `SUCC(T)`. Since the vertices p, p+1,...,n are the only vertices whose levels change, the `nodeRecords` should be modified only for these vertices and their ancestors. Therefore, when `SUCC` is applied to generate a new tree, the only vertices v whose `nodeRecord` has to be changed are vertices from p to n, vertex q, and all the vertices along the new path from vertex n to the root of `SUCC(T)`. Thus, these are the only vertices for which the `OptimizationParameter` field has to be recalculated.

The analysis begins with tree network `T` having lexicographically the largest canonical level sequence of all trees of order n. This tree, represented by the level sequence `L(T)`=[1,2,3,...,n], consists of a single path from the root to the leaf. For each node of the initial tree, the value of the `OptmizationParameter` is calculated using the "bottom-up" approach. The value of `OpmizationParameter` for the root of `T` constitutes the value of the optimization criterion for the whole tree. Subsequently, the successor function `SUCC` is applied to the current tree, the successor tree `SUCC(T)` is generated, and the values of `OptimizationParameter` is recalculated for the required nodes. The procedure for applying function `SUCC` and recalculating the values of `OptmizationParameter` is repeated until the tree corresponding to lexicographically the smallest level sequence is reached, i.e., sequence [1,2,2,...,2] representing the star topology.
Subsequently the steps of the algorithm are described. Without loss of generality, it is assumed that function `OptmizationParameter` is to be minimized for all trees of order n.

Algorithm 5 – Synthesis of a tree network that optimizes the value of optimization criterion `OptmizationParameters` among all trees of order n.
//Given: the order of the tree network n
//Objective: synthesize (construct) a tree network `T` that optimizes parameter //`OptmizationParameter` among all tree networks of order n. Use //`arrayTree[1..n]` of `nodeRecords` to represent `T`.

```
1   LevelSequence(arrayTree)=[1,2,...,n];
    //initialize arrayTree to respresnt a path topology of a tree network, i.e L
    // is initialized to the lexicographically largest level sequence of order n,
    // i.e. (/L=[1,2,...,n])
```

```
2  CalculateOptimizationParameter(arrayTree,n,1);
   //using a bottom-up method, calculate value of OptimizationParameter for
   each //nodeRecord of //arrayTree (from n downto 1).
3  Minimium=arrayTree[1].OptimizationParameter; minTree=arrayTree;
   // initialize the current minimum to the value of the OptimizationParameter
   at the // root of T initialize minTree
4  do
5      arrayTree = SUCC(arrayTree);// (L=SUCC(L)) apply the successor
                                  //function to arrayTree;
6      RecalculateOptimizationParameters(arrayTree,p,q ) //Using a bottom
       // up method, modify and recalculate the value of
       // OptimizationParameter ONLY //for those elements of arrayTree
       // which were affected by applying SUCC //function( p and q are as in
       // definition 4 of SUCC)
7      if arrayTree.OptimizationParameter<Minimum {
               Minimum= arrayTree.OptimizationParameter; //update the
                                                          //minimum.
               minTree=arrayTree; }
8  while( LevelSequence(arrayTree) ≠ [1,2,2,...,2]); //end the loop when the
               // level sequence of the tree /represents a star topology, or
               // L=[1,2,,2]
```

Upon completion of the algorithm, the variable `Minimum` holds the optimal value of the optimization criterion for all rooted trees of order n, and an optimal tree can be reconstructed by tracing array `minTree`.
Since algorithm 5 examines all rooted trees of a given order, its running time is proportional to the number of those trees. Let T_n denote the number of rooted trees of order n. The following formula, taken from [Harary at el, 1973], describes T_n, and thus complexity of algorithm 5.

$$(1)\qquad T_n = \frac{0.4399237 \times 2.9557649^n}{n^{\frac{3}{2}}} + O(\frac{2.9557649^n}{n^{\frac{5}{2}}})$$

The following theorem shows effectiveness of Algorithm 5.

Theorem 5 [Kubicka , 1996]:
Algorithm 5 computes the minimum (maximum) value of optimization criterion over all rooted trees of a given order n. The average number of steps per tree the algorithm performs is bounded by a constant which does not depend upon n.

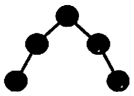
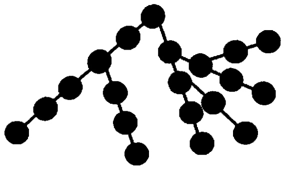
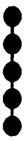
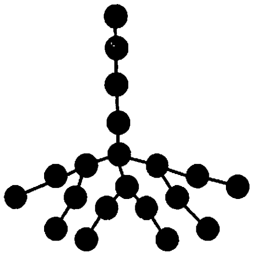
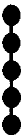
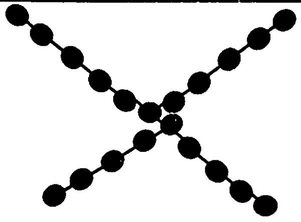
The complete and elaborate proof of theorem 5 can be found in [Kubicka,1996].

4.2 Examples of Optimal Tree Structured Networks

We have applied the synthesis algorithm to design of optimal networks for parameters described in section 3. Results of these experiments for network sizes of 5 and 20 nodes are shown in the table below.

The optimization parameters were calculated by algorithms described in section three. For `DelayToOne`, `DelayToMany` and `averageDelayToMany` we have used the following constant values: $T_{init}=1$, $T_{net}=2$, $T_{cong}=1$, and a non-linear penalty function for the node congestion: $\mathtt{Penalty(od(v))} = \mathtt{(od(v))}^2$.

The synthesis algorithm was applied to generate the optimal networks for each of the discussed parameters. Sample results are shown in table 5. There are 12,826,228 different tree network topologies with 20 nodes. The minimal value of the worst case many-to-one communication delay (`DelayToOne`) is equal to 22, and this value is achieved for 34 network topologies. Table 5 depicts one of these optimal topologies. The minimal value the worst case many-to-many communication delay (`DelayToMany`) is equal to 33, and there are seven networks that achieve this value. One of these networks is shown in table 5. Contrary to the above two parameters, the average many-to-many communication delay (`averageDelayToMany`) is minimized by only one tree network. The minimal value of `averageDelayToMany` parameter is equal to 18.47 and the optimal network topology is shown in table 4. There are six topologies that maximize the average size of a subnetwork for tree networks with 20 nodes. The optimal average size is equal to 12.72 and one of the optimal topologies is depicted in table 5.

Optimization Parameter	Optimal Topology for Network with n Nodes and the value of the Optimization Parameter	
	n=5 (Total number of tree networks with 5 nodes = 9)	n=20 (Total number of tree networks with 20 nodes = 12,826,228)
The Worst Case Many-to-One Communication Delay `DelayToOne`	 `DelayToOne =10`	 `DelayToOne=22`
The Worst Case Many-to-Many Communication Delay `DelayToMany`	 `DelayToMany=13`	 `DelayToMany=33`
The Average Many-to-Many Communication Delay `averageDelayToMany`	 `averageDelayToMany=6.5`	 `averageDelayToMany=18.47`

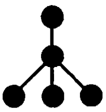
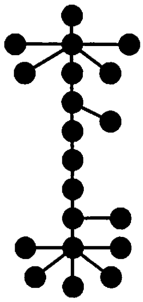
The Average size of a Subnetwork `averageSizeOfSubtree`	 `averageSizeOf Subtree=2.6`	 `averageSizeOf Subtree=12.72`

Table 5. Optimal topologies for tree structured networks of size 5 and 20.

5. CONCLUSION

We have proposed a novel, comprehensive approach to the analysis and synthesis of tree structured communication networks. Unlike the existing work, the class of tree network models described in this paper is universal and can be used in a variety of applications. We have substantiated our model by describing efficient, dynamic programming algorithms for calculating performance parameters such as communication delays and the average size of a subnetwork. Building on our previous work, we have illustrated a powerful application of an algorithm for generating and examining all trees of a given order to the network synthesis problem. The synthesis algorithm outlined in section four uses our proposed tree structured communication network model. The optimization parameters in this algorithm are calculated using the dynamic programming algorithms described in section three. Unlike most of the existing work, our network synthesis algorithm always produces an optimal solution. Sample optimal networks shown in section four, exhibit complex properties with non-trivial topologies, validating both the difficulty of the network synthesis problem and the power of the synthesis algorithm.

An obvious limitation of this algorithm is its inability to analyze networks with large number of nodes, since the number of trees grows prohibitively fast with order size. Therefore, the algorithm can be used to design optimal hierarchies for small order networks or as a preliminary analysis of large systems. We believe that in a many cases the above method can be applied to smaller networks and the obtained tree structures can be extrapolated to approximate optimal solutions for large networks, or for developing low complexity heuristics.

Our current research is evolving in two directions. One is further study of network models, performance parameters and cost functions, such as measures of reliability and fault tolerance. The second direction is an attempt to increase the domain size of the network synthesis algorithm by applying pruning techniques and parallelism to it.

REFERENCES

[Ahuja et al., 1993] Ahuja, R., Magnati, T., and Orlin J. *Network Flows: Theory, Algorithms and Applications,* Prentice Hall, 1993.

[Balakrishnan et al, 1991] Balakrishnan, A., Magnati, T., Shulman, A. and Wong, R. Models for Planning Capacity Expansion in Local Access Telecommunication Networks", *Annals of Operation Research,* vol 33, pp. 239-289.

[Banerjee, et al., 1992] Banerjee, S., Mukherjee, B., Sarkar, D., Heuristic algorithms for constructing near-optimal multihop lightwave networks, *IEEE INFOCOM, 1992.*

[Berry et al, 1997] Berry, L., McMahon, G., Murtagh, B., Sugden, S., and Welling, L., An Integrated LP-Genetic Algorithms Approach to Communication Network Design, *Proc. IFIP Workshop on Traffic Management and Synthesis of ATM Networks,* Canada, September, 1997.

[Berry et al., 1995] Berry, L., Murtagh, B., Sugden, S. and McMahon, G. Application of a Genetic-based Algorithm for Optimal Design of Tree Structured Networks" *Proc. Regional Teletraffic Engineering Conference of the International Teletraffic Congress, South Africa, 1995, pp.361-370.*

[Berry et al., 1994] Berry, L., Murtagh, B., and Sugden, S., A Genetic-based Approach to Tree Network Synthesis with Cost Constraints", *Proc. of EUFIT'94, Germany, September 1994, pp. 626-629.*

[Beyer et al., 1980] Beyer, T., and Hedetniemi, S.M., Constant time generation of rooted trees, *SIAM J. Comput.*, 9 (1980) 706–712.

[Dandamudi et al., 1990] Dandamudi, S.P., and Eager, D.L., Hierarchical Interconnection Networks for Multicomputer Systems, *IEEE Trans. Computers,* 39 (1990), 786-797.

[Harary, F. et al., 1973] Harary, F., Palmer E., *Graphical Enumeration*, Academic Press, New York and London 1973.

[Huizinga et al., 1997] Huizinga D.M., and Kubicka, E.., A Tree Generating Algorithm for Designing Optimal Hierarchical Distributed Systems, *Proc. of ACM Symposium on Applied Computing*, San Jose, CA, Feb. 28- March 2, 1997, pp. 345-353.

[Kubicka, 1996] Kubicka, E., An efficient method of examining all trees, *Combinatorics, Probability, and Computing,* Vol 5, 1996, pp-403-413.

[Leighton, 1992] Leighton, F.T, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Pub., 1992.

[Minoux, 1989] Minoux, M. Network Synthesis and Optimum Network Design: Models, solution methods and applications, *Networks*, vol. 19 (1989) pp. 313-360.

[Polya, 1937] Polya, G., Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und Chemische Verbindungen. *Acta Math.* Vol 68 (1937) 145-154.

[Ramanathan at el., 1992] Ramanathan, S., Rangan, P.V., Vin, H.M., and Kaeppner, T., Optimal Communication Architectures for Multimedia Conferencing in Distributed Systems, *Proc. of 12-th International Conference on Distributed Computing Systems*, 1992, 46-53.

[Stallings, 1999] Stallings, W. *Data and Computer Communications*, 6th Edition, Prentice Hall PTR, 1999.

[Sharma, 1991] Sharma, U., Misra, K.B., Bhattacharji, A.K., Applications of an Efficient Search Technique for Optimal Design of Computer Communication Networks. *Micro Electronics and Reliability,* vol 31, pp. 337-341.

[Wright et al., 1986] Wright, R.A., Richmond, B., Odlyzko A.,and McKay, B.D., Constant time generation of free trees, *SIAM J. Comput.*, 15 (1986) 540 – 548.