Analyzing an Improvement of MPLS-Net Structures for the Decrease of Dialogue Transmission Delay

Feng Huining, Chen Qimei

Department of Computer Science and Technology, Nanjing University
Telecommunication Technology Institute, Nanjing University
No.22 Hankou Road, Nanjing, Jiangsu, 210093, China
E-mail: mawn@nju.edu.cn

Abstract:

As the network traffic increases, dialogue transmission delay becomes more and more significant recently. To optimize current MPLS networks and enhance their capability, a new analytical method based on trees is proposed here. From this view, network details are better traced in a discrete mathematics context. On identifying a network as a directed graph and dividing it into multiple trees, emphasis is placed on those component trees instead of the original graph to obtain a more trenchant understanding of it. Various types of those trees are defined, analyzed on their performance, and converted to and fro. The conversion between each other is bestowed a high practical value, since the entire network can be optimized in such an easier divide-and-conquer way. The viability of combining individual converted component trees to attain a whole converted network is proved at the end of this article.

Keywords:

Net application, MPLS-Net, Transmission delay, Bidirected share tree, Unidirected share tree

Dialogue application has been greatly increased in recent years. To enable audio and video communication via networks, an extremely high transmission capability is required. MPLS partially solved this problem. However, it turns out that poor design compromise the advantage of MPLS. Here, we propose a new method to analyze MPLS-Net structures. We view those structures as multiple trees. From this viewpoint, their composition can be better understood and special optimization can be done on them.

1. Conceptual Description of Source Trees and Share Trees

The implementation of a simplified network is presented in Fig. 1. The tree structure in this figure can be viewed as either two source trees or one share tree. Note that the ellipse area with a dotted edge represents a probable network. There are four sorts of routers in this illustration: ***merging points*** (those denoted by M_i, which enable two-way connections and provide functions to merge data streams from various sources so that they are forwarded concurrently), ***source nodes*** (those denoted by S_i, at which there is no other routers in the tree can arrive in one or more steps), ***destination nodes*** (those denoted by D_i, from which any other routers in the tree cannot be reached in one or more steps), and ***ordinary nodes*** (those denoted by N_i that do not fall in the above three categories). The last three sorts of routers are also known as ***non-merging points***, which can only be connected in one predestined direction and have no merging function.

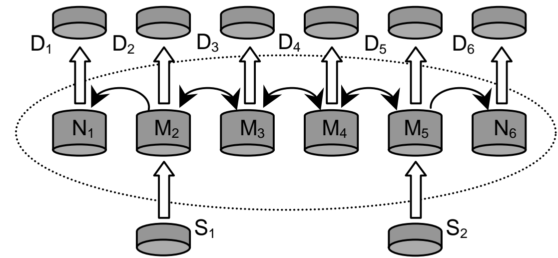

Fig. 1 Implementation of a simplified network

In an abstraction, we view these structures as two-level trees, neglect those merging points and ordinary nodes, and only keep in mind the location and relationship of source nodes and destination nodes. We assume that all such tree structures are ***bipartite***, that is to say, we can separate the nodes (essentially, routers) belonging to one such tree into two groups so that between each of the two nodes in the same group there are no edges. These negligence and assumption would not impede our thorough study, as the theorem in the latter part of this article implies. ***Source trees*** are those directed bipartite trees with a single source node and an unlimited number of destination nodes; ***share trees*** are those directed bipartite trees with multiple source nodes instead. At the first thought one may realize that several source trees with the same set of destination nodes can be combined to make a single share tree. True, in most of the practical cases, share trees are a more economical way to represent those tree structures than source trees. For instance, we convert the network in Fig. 1 into two source trees and into one share tree as shown in Fig. 2.

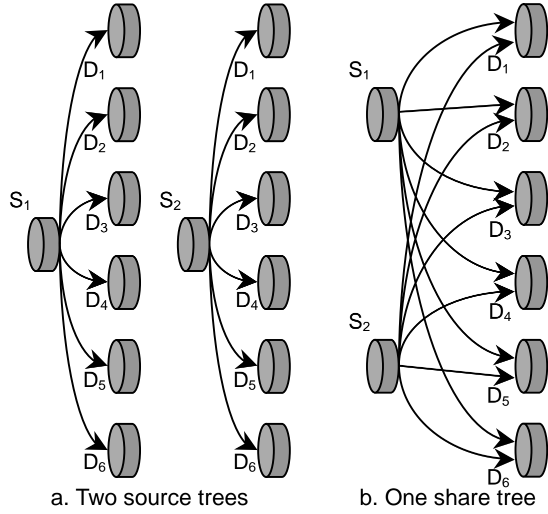

Fig. 2 Tree representations of Fig. 1

Employing share trees in FEC allocation makes it possible to assign multiple source nodes a single FEC so as to decreases the consumption of the limited label space. Unfortunately, source trees are widely used in reality because they are easier and more natural to establish at times when assigning FECs. Reorganizing source trees to obtain equivalent share trees of a smaller amount proves to be an arduous task. True, one can invariably achieve this goal by taking these steps: firstly, separate every source tree to obtain distinct source-destination pairs; and secondly, gather all such source-destination pairs, randomly combine them to form share trees, compare the amount of those share trees, and finally decide the best result after trying all the possibilities. This maneuver is demonstrated in the following procedures:

```
procedure step1;
    for every sourceTree st do
        for every st.child do
            add(st.parent, st.child);

procedure step2;
    combined : boolean = false;
    if number of (parent, child) pairs > 1 then
        for every (parent, child) pair pc1 do
            for every (parent, child) pair pc2 do
                if (pc1 <> pc2) and
                    canCombine(pc1, pc2) then
                begin
                    pc3 := combine(pc1, pc2);
                    delete(pc1);
                    delete(pc2);
                    add(pc3);
                    combined := true;
                    step2;
                    delete(pc3);
                    add(pc2);
                    add(pc1);
                end;
    if (not combined) and
        (number of (parent, child) pairs < min)
    then begin
        min := number of (parent, child) pairs;
        minSourceTree := (parent, child) pairs;
    end;
```

However, this algorithm is in fact impracticable. Consider that procedure step2 is a recursive one, doing $o(n^2)$ operations on each call. In the worst case where each node is to be combined in a resulting share tree, the levels of the recursion sum up to n; so, the operations done in one such successful search amount to $o[(n^2)^n] = o(n^{2n})$. Suppose this algorithm is implemented on a computer that operates 10^7 times in a second, for an original source tree with only 10 nodes, it would take over 300,000 years!

The above ponderous algorithm has been improved in several ways, but the result is still far from satisfactory. Currently, none of the attempts has successfully limited the number of operations to o(n) or $o(n^2)$. Those attempts include reducing the recursion to non-recursive iterations and omitting those tries that are obviously infeasible.

To be more conclusive, we alternatively state this problem in a decision way, "suppose we have an unlimited number of bins and n objects, some of which can be placed together in a bin according to some rules (specified explicitly in the subroutine *canCombine*), and given an input k, do these n objects fit in k bins?"[2] Once we solve this problem, our subsequent task is to find the smallest k. Yet, we realize that the famous NP-complete problem of bin packing [3] is reducible to this problem, so we are assured that this one is so intractable that it is also NP-complete and probably cannot be solved in polynomial time. [4] As a result, the use of share tree, though attractive, is still limited.

However, hope still exists. Similar to the idea presented in *Reference [5]*, an approximation algorithm may be a reasonable roundabout solution. An innovatory strategy to reorganize an enormous amount of source trees will definitely incur a revolution in this field.

2. Two Kinds of Share Trees -- Bidirected and Unidirected Ones

In our study, share trees are divided into two kinds: bidirected ones and unidirected ones, both of which manifest distinct features. As an example of bidirected share tree, consider the network implementation in Fig. 1. In the structure, four merging points (M_2~M_5) are depicted. Suppose that two data streams depart from S_1 and S_2, respectively, and are both directed to the same destination D_3. The route of the stream from S_1 is $S_1 \rightarrow M_2 \rightarrow M_3 \rightarrow D_3$; and the route of the other stream, $S_2 \rightarrow M_5 \rightarrow M_4 \rightarrow M_3 \rightarrow D_3$. If time is appropriate, the two streams are merged on M_3 and forwarded to D_3 together. In a similar vein, those data streams to D_1 and D_2 are merged on M_2; those to D_4 are merged on M_4; and those to D_5 and D_6, on M_5. It is thus easy to

understand that in this example, implementing a ***bidirected share tree*** (whose concept is loosely defined as those share trees with several two-way connections between various merging points by a straightforward means) necessitates four merging points to be employed.

As an alternative, a ***unidirected share tree*** contains far fewer merging points while at the same time enables the same connections. The share tree in Fig. 3 is the unidirected counterpart of that in Fig. 1. After converting the bidirected share tree in the latter figure into a unidirected one as we have done, only one merging point is necessary. As a result, the network is simplified, though an extra node is added. [1]

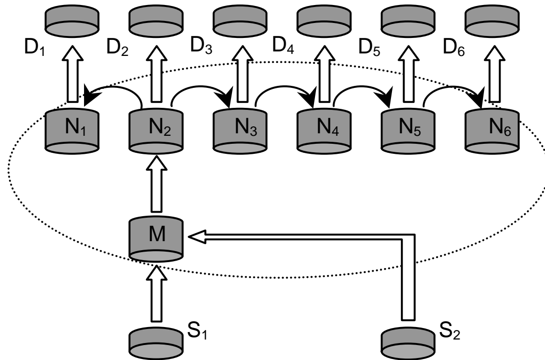

Fig. 3 Implementation of a simplified network

We then proceed to calculate the advantages of this change. Let the average delay time of a data package be t_1 on a merging point and t_2 on a non-merging point. Since merging points employ a more sophisticated algorithm and provide more flexible functions, t_1 is considerably greater than t_2. Also, we let the time of merging two such packages be t_m. In the source tree in Fig. 1, it takes time $2t_1$ for a package sent from S_1 to arrive at M_3, and $3t_1$ for a package from S_2 to arrive at M_3. After merging, they are sent together to D_3. This latter part of the process takes time t_m+t_2. So, the total time spent on the two packages, from their being sent till their arrival, is $3t_1+t_m+t_2$. As to the unidirected implementation depicted in Fig. 3, it takes only $t_1+t_m+3t_2$ to accomplish the same goal. Obviously, the latter approach performs much better, though it is at the cost of building up extra connections and increasing the burden of a single merging point. Generally speaking, this drawback, if taken into account at the very beginning of establishing a network, appears to be insignificant. Furthermore, with an elegant network arrangement, a merging point empowered by the current hardware technology is able to take up some (but not many) of its peers' work without appreciable overload. The savings of decrease in the number of merging points and the performance enhancement in the long run are both attractive.

3. Algorithm on Converting Bidirected Share Trees to Unidirected Share Trees

We have successfully converted a bidirected share tree to its unidirected counterpart. Yet, one cannot be satisfied by this instinctive conversion. Can we find an algorithm to finish this process in a universally practicable way? The answer is positive. In the coming discussion, basic knowledge of graph theory is required.

1) Establish a matrix representing the original bidirected share tree.

Here a binary matrix is employed with each of its elements assuming the value of either 0 or 1. The matrix is n *x* n large, depending on the total number of nodes in the tree. One of its elements, for example the one at position [*i*, *j*], denotes the fact that an edge leading directly from node *i* to node *j* exists if it has a value of 1, or otherwise there is no such edge. To illustrate this, the matrix representation of the share tree shown in Fig. 1 is given in Tab. 1.

Tab. 1 Matrix representation of the share tree in Fig. 1

	S_1	S_2	M_2	M_3	M_4	M_5	N_1	N_6	D_1	D_2	D_3	D_4	D_5	D_6
S_1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
S_2	0	1	0	0	0	1	0	0	0	0	0	0	0	0
M_2	0	0	1	1	0	0	1	0	0	1	0	0	0	0
M_3	0	0	1	1	1	0	0	0	0	0	1	0	0	0
M_4	0	0	0	1	1	1	0	0	0	0	0	1	0	0
M_5	0	0	0	0	1	1	0	1	0	0	0	0	1	0
N_1	0	0	0	0	0	0	1	0	1	0	0	0	0	0
N_6	0	0	0	0	0	0	0	1	0	0	0	0	0	1
D_1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
D_2	0	0	0	0	0	0	0	0	0	1	0	0	0	0
D_3	0	0	0	0	0	0	0	0	0	0	1	0	0	0
D_4	0	0	0	0	0	0	0	0	0	0	0	1	0	0
D_5	0	0	0	0	0	0	0	0	0	0	0	0	1	0
D_6	0	0	0	0	0	0	0	0	0	0	0	0	0	1

From this matrix, we can discern useful facts. We assume that from each node there is a circular edge from it to itself, so the elements on the diagonal of the matrix always assume the value 1. If a certain column of the matrix contains only one 1-element, the node named on the top of it must be a source node; if a certain row of the matrix contains only one 1-element, the node named to the left of it must be a destination node; and if a certain column of the matrix contains more than three 1-elements, the node named on the top of it must be a merging point. We will not consider those nodes that are both sources and destinations, since they are actually solitary.

2) Find the transitive closure of the matrix.

The transitive closure of a matrix reveals the relationship between pairs of nodes, according to which it can be decided whether it is possible for a package from the first node to reach the second in one or more steps. An element at [*i*, *j*] valued 1 in the closure means that a directed path exists from node *i* to node *j*, and vice versa.

To determine the transitive closure of a graph, we will

introduce the Warshall algorithm here: [2]

```
void transitiveClosure(boolean[][] A, int n, boolean[][] R)
    int i, j, k;
    Copy A into R.
    Set all main diagonal entries, rij, to true.
    for (k = 1; k ≤ n; k ++)
        for (i = 1; i ≤ n; i ++)
            for (j = 1; j ≤ n; j ++)
                rij = rij ∨ (rik ∧ rkj)
```

The time complexity of this algorithm is $o(n^3)$, and with an improvement of maintaining each row of the original matrix bitwise in one or more computer words, the complexity can be reduced to $o(n^2)$. [2]

As to the previous example, the transitive closure of the matrix we have given is in Tab. 2.

Tab. 2 Transitive closure of the matrix in Tab. 1

	S_1	S_2	M_2	M_3	M_4	M_5	N_1	N_6	D_1	D_2	D_3	D_4	D_5	D_6
S_1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
S_2	0	1	1	1	1	1	1	1	1	1	1	1	1	1
M_2	0	0	1	1	1	1	1	1	1	1	1	1	1	1
M_3	0	0	1	1	1	1	1	1	1	1	1	1	1	1
M_4	0	0	1	1	1	1	1	1	1	1	1	1	1	1
M_5	0	0	1	1	1	1	1	1	1	1	1	1	1	1
N_1	0	0	0	0	0	0	1	0	1	0	0	0	0	0
N_6	0	0	0	0	0	0	0	1	0	0	0	0	0	1
D_1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
D_2	0	0	0	0	0	0	0	0	0	1	0	0	0	0
D_3	0	0	0	0	0	0	0	0	0	0	1	0	0	0
D_4	0	0	0	0	0	0	0	0	0	0	0	1	0	0
D_5	0	0	0	0	0	0	0	0	0	0	0	0	1	0
D_6	0	0	0	0	0	0	0	0	0	0	0	0	0	1

3) Based on the above two matrices, arrange the resulting unidirected share tree.

a. Among the columns of the transitive closure on the top of which appear the names of merging points, pick out those consisting exactly of the same group of elements. For instance, given the above-mentioned transitive closure, we would pickup the columns of M_2~M_5, because M_2~M_5 are all merging points and their corresponding columns have exactly the same elements $(1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0)^T$. If such a group of at least two columns is found, merging functions of those points can be substituted by a newly-introduced merging point, and we will be proceeding to step *b*.; otherwise, we should leap to step *e*. and put the algorithm to an end. Attention should be paid to those selected merging points. They in fact constitute a strongly connected directed graph themselves: if not, suppose M_a cannot reach a certain M_b, since M_b can always reach itself according to our premise, then the column of M_a cannot be the same as that of M_b, and they must belong to two different selections.

b. Create a new merging point M, and make it function as a combination of all those selected merging points. To do this, one needs simply to add a connection between the new merging point and one of the old ones (according to any of our wishes) and make the new one the end-point of every such edge: its start-point is not among the selected merging points while its end-point (previous to this change) is. Returning to our incomplete example, we add a merging point M, link it with M_2 (or any one of M_3~M_5), and redirect the two edges from S_1 and S_2 to it instead of M_2 and M_5, respectively. Except for this change, we maintain all the existing connections.

c. Rearrange the direction of all the connections between the merging points (new and old), and change each of the old merging points into a non-merging point. The direction of the connection between the new merging point and an old one is from the former to the latter. The direction of the connection between each pair of the old merging points is so arranged that from the new merging point all of the old ones are in reach. We assure that for each share tree there is always one and only one such solution, because all the old merging points compose of a strongly connected directed graph. After this, since the previous two-way connections have become one-way, and none of the nodes in the tree connects directly to the selected merging points except themselves, it is now safe to convert those merging points to non-merging points. In the previous example, we direct the connections $M \rightarrow M_2$, $M_2 \rightarrow M_3$, $M_3 \rightarrow M_4$, and $M_4 \rightarrow M_5$ to make possible the transmission of packages from M to any of M_2~M_5; and finally we deprive M_2~M_5 of the superiority as merging points.

d. Repeat steps a~c.

e. End the algorithm.

4. The Algorithm's Topological Feature

Firstly, we apply our algorithm to a share tree and convert it from a bidirected form to a unidirected one. Secondly, we restore the original form of the share tree, place it in a larger share tree environment in which it is but a subgraph, and apply our algorithm to this larger share tree. Will the two results on the inner share trees differ? Our answer is negative.

Theorem Suppose a network environment in which places a bidirected share tree. The environment itself is viewed as a larger bidirected share tree containing the former one. If there is no two-way connection between two nodes one of which lies in the inner tree and the other is out of it, then whether to apply the conversion algorithm to the inner tree or to the outer network environment as a whole, the modification done on the inner tree is the same.

proof: We need only to prove that when we process step *a*. of the algorithm, any merging point selection in the two cases, if it is relevant to the inner tree, remains the same. This is because after selecting the same group of merging points whose merging functions we are to substitute with a new merging point, the rest of

the algorithm definitely does the same thing.
∵ there is no two-way connection between a node inside the inner tree and a node outside the inner tree,
∴ a merging point inside the inner tree is not connected with any merging point outside it.
∵ we have proved that in each selection in step *a*., the merging points belonging to the group forms a strongly connected directed graph,
∴ when the algorithm is applied to the outer environment, we will not pick out a group of merging points some of which are in the inner tree while the others are not;
∴ modifications relevant to the inner tree are in fact done on a group of merging points all of which belong to the inner tree.
∵ in the second case (lager tree), the nodes outside the inner tree that can reach one of the inside merging point can also reach its peers in a selected group,
∴ the differences of columns in the transitive closure corresponding to distinct groups still lie in the inner tree: the outside nodes have nothing to do with it;
∴ whether to take the outside nodes into account or not, the selections of inside merging points to be substituted remain the same;
∴ whether to apply the conversion algorithm to the inner tree or to the outer network environment, the modification done on the inner tree is the same. □

This theorem reveals an important feature of the algorithm: it maintains the topological structure of the previous network. This agreeable feature places two advantages in front of the network practitioners:

1) It is possible to understand a large area of network by a gradual improvement in the understanding of its subnetworks, though small they may be. Either combining several modified subnetworks that satisfy the precondition of the theorem or enlarging a modified subnetwork of this kind step by step, using the algorithm at times when necessary, results in a large modified network. The modification done in this way on the large one is just the same as applying the algorithm to it directly.

2) Since the algorithm maintains original topological structures except that some new merging points are added, the original efforts to optimize networks is perfectly preserved after the application of the algorithm. One can ameliorate a large network area by respectively optimizing its component subnetworks.

5. Practical Use of the Algorithm

To conclude on our algorithm and its important feature, a network designer can employ this effective strategy to convert a bidirected network to lessen the two-way connections in it freely. The work of minimizing this kind of connections is sometimes strictly necessary to lower the expense of establishing a large network.

Practicing the algorithm in a broad network context is somewhat challenging, though it is much easier to consider one of its subnetworks at each time rather than taking into account the whole network initially. Fig. 4 below shows an imaginary network in practice.

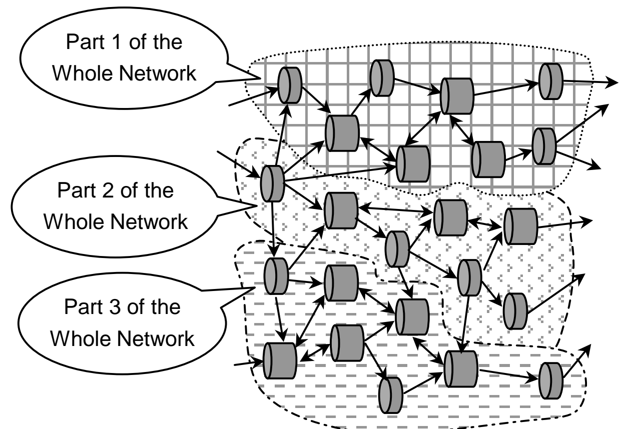

Fig. 4 An imaginary network in practice

To simplify our work, we separate the whole network area into three parts. A careful division allows that the border of each part satisfies our stated precondition. After this, we apply the algorithm on the three components and get the same result as if the algorithm is applied to the entire area. As a result, three extra merging points are added to distinct parts respectively when the algorithm fulfills its task.

6. Remarks

The effort to analyze and optimize MPLS networks is mainly aimed at improving the performance of net-based dialogue application. To accommodate increasing audio and video traffic and to enable real-time dialogue communication via networks, a clear view of the current implementation and its optimization are indispensable.

Trees are a focused concept in networking. The two kinds of tree structures, source trees and share trees, reveal an essence of FEC allocation in MPLS. As an advantageous network representation, share trees are generally highlighted. Thus it is important to recognize the two kinds of share trees: bidirected ones and unidirected ones. An algorithm used in the conversion between them is presented, whose feature opens a new gate to network designers.

References

[1]. D Ooms, B Sales, Alcatel, W Livens, Colt Telecom, A Acharya, IBM, F Griffoul, Ulticom, F Ansari, Bell Labs. *Framework for IP Multicast in MPLS* [EB/OL]. http://community.roxen.com/developers/idocs/drafts/draft-ietf-mpls-multicast-05.html.2001-1-23/2001-9-30.

[2]. Sara Baase, Allen Van Gelder. *Computer Algorithms – Introduction to Design and Analysis* [M]. Higher Education Press, Pearson Education Ltd. 2001-7

[3]. Y. Narahari. *Data Structures and Algorithms* [EB/OL], http://lcm.csa.iisc.ernet.in/dsa/index.html. 2001-07-10

[4]. Himanshu Gupta. *Result Verification Algorithms for Optimization Problems* [J]. Technical Report, UIUC. 1995-5-12

[5]. Giorgio Gambosi, Alberto Postiglione, Maurizio Talamo. *On-Line Maintenance of an Approximate Bin-Packing Solution* [J]. *Nordic Journal of Computing*. Summer 1997