Distributed Linear Time Construction of Colored Trees for Disjoint Multipath Routing[⋆]

Srinivasan Ramasubramanian, Mithun Harkara, and Marwan Krunz

Department of Electrical and Computer Engineering
University of Arizona, Tucson, AZ 85721

Abstract. Disjoint multipath routing (DMPR) is an effective strategy to achieve robustness in networks, where data is forwarded along multiple link- or node-disjoint paths. DMPR poses significant challenges in terms of obtaining loop-free multiple (disjoint) paths and effectively forwarding the data over the multiple paths, the latter being particularly significant in datagram networks. One approach to reduce the number of routing table entries for multipath forwarding is to construct two trees, namely red and blue, rooted at a destination node such that the paths from a source to the destination on the two trees are link/node-disjoint. This paper develops the first distributed algorithm for constructing the colored trees whose running time is linear in the number of links in the network. The paper also demonstrates the effectiveness of employing generalized low-point concept rather than traditional low-point concept in the DFS-tree to reduce the average path lengths on the colored trees.

1 Introduction

Multipath routing (MPR) is an effective strategy to achieve robustness [1], load balancing [2], congestion reduction [3], low power consumption [4], and increased throughput. It operates by transmitting data over multiple paths. In general, the multiple paths from a source to a destination may have common links (or nodes) as long as the shared links (or nodes) have sufficient resources. To improve the transmission reliability and avoid shared-link (or node) failures, the multiple paths can be selected to be link- or node-disjoint. In this case, the MPR approach is referred to as *disjoint multipath routing* (DMPR). DMPR provides better robustness compared to the generic MPR. However, it may be inefficient with respect to other metrics such as the overall energy consumption [4] in a wireless ad hoc or sensor network.

DMPR has been extensively studied in the context of wired networks [5, 6], where the multiple paths are often employed for failure resiliency purposes. Only one of the paths, referred to as the primary path, is used at any instant. Upon a failure, the connection is rerouted over a backup path. If the backup path is the same for any link (or node) failure that affects the primary path, then the primary and backup paths must be link- (or node-) disjoint. In applications such as transmission of multiple description encoded video streaming, the two link-disjoint paths are used simultaneously. Two independently encoded video streams are transmitted along two link-disjoint paths [7]. If

[⋆] The research developed in this paper is supported by National Science Foundation under grants 0325979, 0435490, and EEC-0333046.

multiple paths are employed for increased throughput, then the data may be split over multiple paths.

Motivation. Implementation of generic MPR and DMPR poses two main challenges. The first is related to the computation of loop-free multiple paths. Several centralized algorithms (or equivalently those that assume a global network knowledge) have been proposed for the DMPR problem in the context of failure resiliency in wired connection-oriented networks. For large-scale wired or wireless networks, a distributed solution that relies only on local information is preferred. Distributed multipath routing algorithms in the literature are developed purely in the context of wireless networks. MPR approaches based on Dynamic Source Routing (DSR) [8–10] require the destination to select maximally disjoint paths among the received route requests. MPR approaches based on AODV routing [11–15] do not guarantee finding disjoint paths. The only well-known generic multipath routing employed in the wired datagram network is the OSPF algorithm, where the choice of paths is limited to those of equal cost.

The second challenge of implementing MPR (or DMPR) techniques is related to forwarding of data over the multiple paths. In typical connection-oriented networks, the end-to-end path is clearly identified using, for example, connection identifiers or labels. The nodes maintain a routing table that specifies the output port for each label. Each path requires a unique identifier. Hence, the size of the routing table at each node is directly proportional to the number of multiple paths. In contrast, datagram networks rely on the destination address in the packet header for forwarding packets over one path. To implement MPR or DMPR techniques in such networks, every node must maintain a set of preferred neighbors to reach a destination, such that the paths are loop-free (and disjoint, if needed). Forwarding of packets to meet such constraints must be based on destination address and some "additional" information (e.g. source address, labels, etc.). The intermediate nodes must be aware of this additional information or otherwise, it must be carried in every packet header. The choice of the additional information used in forwarding along multiple paths determines the overhead involved.

To reduce the routing table overhead, hence reduce lookup time, a novel multipath routing strategy called *colored trees* (CT) was developed [16]. Every node in the network has two preferred neighbors to the destination: *red* and *blue*. A packet transmitted from a source is marked with one of the two colors. An intermediate node that receives the packet forwards it to its preferred neighbor based on the color of the packet. Thus, the routing table at a node has only two entries (for every destination node). The network may be viewed as two trees (red and blue) that are rooted at the drain. The two paths from a given source to the drain on the two trees are link/node-disjoint.

The goal of this paper is to develop a linear-time distributed algorithm for constructing two colored trees. The rest of the paper is organized as follows. Section 2 describes the network model and problem definition. Section 3 discusses the related work in obtaining colored trees using generalized path augmentation technique and maintaining the partial order in a distributed manner using local information. Section 4 develops the linear-time distributed algorithm for constructing the two colored trees. Section 5 presents the performance comparison of the distributed algorithm with traditional and generalized low-point concepts. Our Conclusions are presented in Section 6.

2 Problem Statement

Consider a network $\mathcal{G}(\mathcal{N}, \mathcal{L})$ composed of a set of nodes $\mathcal{N}$ and a set of links $\mathcal{L}$. The links are assumed to be bi-directional. The terminology of *arc* is used to refer to a directed link between two nodes. An arc from node i to j is represented as $i \rightarrow j$. Given a drain node $d \in \mathcal{N}$, the goal is to construct two trees $\mathcal{R}$ and $\mathcal{B}$ (referred to as the red and blue trees, respectively) rooted at d that minimize the average path length from a source to the drain such that the CT-LD and CT-ND versions of the problem satisfy the link-disjoint and node-disjoint path constraints, respectively. These constraints are stated as follows. Let $\mathcal{P}^{\mathcal{R}}_{sd}$ and $\mathcal{P}^{\mathcal{B}}_{sd}$ denote the paths from a node s to the drain d on trees $\mathcal{R}$ and $\mathcal{B}$, respectively.

Link-disjoint path constraint: $\forall s \in \mathcal{N} \setminus \{d\}$ and $\forall i, j \in \mathcal{N}$

$$i \rightarrow j \in \mathcal{P}^{\mathcal{R}}_{sd} \;\; \Rightarrow \;\; (i \rightarrow j \notin \mathcal{P}^{\mathcal{B}}_{sd}) \wedge (j \rightarrow i \notin \mathcal{P}^{\mathcal{B}}_{sd}).$$

Node-disjoint path constraint: $\forall s \in \mathcal{N} \setminus \{d\}$ and $\forall i \in \mathcal{N} \setminus \{s, d\}$

$$i \in \mathcal{P}^{\mathcal{R}}_{sd} \;\; \Rightarrow \;\; (i \notin \mathcal{P}^{\mathcal{B}}_{sd}).$$

A network must be two-node-connected (two-edge-connected) to obtain a solution to the CT-ND (CT-LD) problem [17].

3 Generalized Path Augmentation

The generalized path augmentation algorithm [18] is a heuristic developed in the context of robust multicasting. It may be applied to the problem at hand by simply reversing the direction of the links in the trees constructed. It starts by choosing an arbitrary directed cycle $(d, v_1, ..., v_k, d)$ in $\mathcal{G}$ with at least three nodes ($k \geq 2$). If this cycle does not include all the nodes of $\mathcal{G}$, then a path that starts and ends on that cycle and that passes through at least one node not on the cycle is chosen for augmentation. The algorithm continues with path augmentation until all the nodes in the network are considered.

Medard et al. [17] developed a centralized algorithm that selects a cycle and successive paths at random. Xue et al. [18] developed a generalized version of the centralized path augmentation approach (referred to as the XCT algorithm in the rest of the paper) by specifying certain criteria for selecting paths for augmentation, which depend on the problem objective (e.g. minimizing average delay or cost, maximizing bandwidth, etc.).

The XCT algorithm is based on partial ordering of nodes in the network. The partial order $\prec$ of the nodes on the blue tree $\mathcal{B}$ is defined as follows. If $u \rightarrow v \in \mathcal{B}$, then v precedes u in the partial order, represented as $v \prec u$ (the algorithm in [18] employs partial order on both the red and blue trees for link-disjoint paths. However, the explanation here has been simplified based on the partial order on the blue trees and node-disjoint paths). The partial ordering satisfies the transitive relationship, i.e., if $u \prec v \prec w$, then $u \prec w$. The generalized approach is now described for the construction of two colored trees for the CT-ND problem.

The XCT algorithm for constructing two trees that satisfy the node-disjoint path constraint follows four steps:

1. Initialize $\mathcal{R}$ and $\mathcal{B}$ to contain the root node d only. Initialize the partial order of the nodes to be the empty set.
2. Find a cycle $(d, v_1, ..., v_k, d)$. Let $v_k \rightarrow v_{k-1} \rightarrow ... \rightarrow v_1 \rightarrow d$ be the *red chain* and $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_k \rightarrow d$ be the *blue chain*. Add the blue chain to $\mathcal{B}$ and the red chain to $\mathcal{R}$. Update the precedence relation with $v_1 \prec v_2 \prec ... \prec v_k \prec d$.
3. Stop if $\mathcal{B}$ spans all the nodes in $\mathcal{G}$.
4. Find a path $(x, v_1, ..., v_k, y)$ that connects any two distinct nodes x and y on $\mathcal{B}$ and any k nodes not on $\mathcal{B}$, $k \geq 1$, such that $x \prec y$. Let $v_k \rightarrow v_{k-1} \rightarrow ... \rightarrow v_1 \rightarrow x$ be the red chain and $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_k \rightarrow y$ be the blue chain. Add the blue chain to $\mathcal{B}$ and the red chain to $\mathcal{R}$. Update the precedence relation with $x \prec v_1 \prec v_2 \prec ... \prec v_k \prec y$. Go to Step 3.

The above algorithm may be applied to the link-disjoint case by relaxing the condition in Step 4 that x and y have to be distinct and maintaining partial ordering of edges instead of nodes. The algorithm is guaranteed to obtain two trees that satisfy the link-disjoint (node-disjoint) constraint if the network is two-edge-connected (two-node-connected). The approach may be combined with depth-first-search numbering to obtain an $O(L)$ algorithm to construct the colored trees [19].

The algorithms developed in [18] and [19] assume a complete knowledge of network topology; i.e., they are centralized algorithms. For large networks, a distributed implementation is essential. In such a distributed implementation, nodes are assumed to have only neighborhood information.

3.1 Maintaining the Partial Order in a Distributed Fashion

The crux in developing such a distributed algorithm is to identify a mechanism to manage the partial order in a distributed fashion, where each node relies only on local information. Consider the example network in Figure 1.

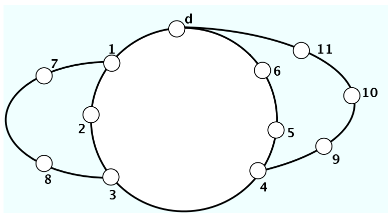

Fig. 1. Example network to illustrate partial ordering and path augmentation used to develop the distributed colored-tree construction algorithm.

Let the first cycle selected by the centralized algorithm be $(d, 1, 2, 3, 4, 5, 6, d)$. Considering one particular direction in the cycle (corresponding to say the blue tree), the partial ordering of the nodes would be $1 \prec 2 \prec 3 \prec 4 \prec 5 \prec 6 \prec d$. There are two options for selecting a path for augmentation: 1–7–8–3 or 4–9–10–11–d. The algorithm by Medard et al. selects a path at random while that by Xue et al. selects a path based on a certain metric. Without loss of generality, assume that the path 4–9–10–11–d is chosen for augmentation. The partial ordering of these paths must be such that: (1) node 4 precedes node 9 ($4 \prec 9$); node 11 precedes node d ($11 \prec d$); and nodes 9, 10, and 11 must appear in the same order as in the path ($9 \prec 10 \prec 11$). However, it is to be noted that there is no explicit ordering between the nodes 9, 10, and 11 in the new path and the nodes 5 and 6 in the old path. Some of the valid global ordering of the nodes that satisfy the above partial order are:

1. $1 \prec 2 \prec 3 \prec \underline{4} \prec 5 \prec 6 \prec \mathbf{9} \prec \mathbf{10} \prec \mathbf{11} \prec \underline{d}$
2. $1 \prec 2 \prec 3 \prec \underline{4} \prec \mathbf{9} \prec \mathbf{10} \prec \mathbf{11} \prec 5 \prec 6 \prec \underline{d}$
3. $1 \prec 2 \prec 3 \prec \underline{4} \prec \mathbf{9} \prec 5 \prec \mathbf{10} \prec 6 \prec \mathbf{11} \prec \underline{d}$

It is the choice of which of these global orderings is selected that distinguishes various approaches. In order to develop a distributed algorithm employing only local information, we select the first ordering, namely $1 \prec 2 \prec 3 \prec \underline{4} \prec 5 \prec 6 \prec \mathbf{9} \prec \mathbf{10} \prec \mathbf{11} \prec \underline{d}$. Given that the first cycle is formed, the global ordering of the nodes is fixed. The path selection starts from the node that is the highest in the order. If a new path can be selected for augmentation from the highest node, then such a path is chosen. The nodes in the new path are added to the global order just before the node from which the path was computed. Once the highest node exhausts all possibilities (adding paths through each of its neighbor), then the path search begins with the next node in the list.

4 Linear-Time Distributed Construction of Colored Trees

The linear-time distributed algorithm for constructing the colored trees works in two phases: (1) Distributed DFS numbering and generalized low-point computation, and (2) Distributed path augmentation. The distributed algorithm is sequential in nature and requires only neighborhood information.

4.1 Distributed DFS Numbering and Generalized Low-Point Computation

We assign DFS numbers to the nodes in the network starting from the drain. The drain is assigned the DFS number 1. In order to help compute paths for augmentation without backtracking, we compute the *generalized low-point value* of a node.

The *low-point value* of a node n is traditionally defined as the lowest DFS-index of a node that can be reached from n by using DFS-tree[1] edges and at most one back edge. The *low-point path* of node n is the path traversed to reach the low-point node. The low-point path of a node n is of the form $n \rightarrow i_1 \rightarrow i_2 \rightarrow \ldots \rightarrow i_k \rightarrow n'$ ($k \geq 0$) such that: (1) node n is the DFS-parent of node i_1, (2) node i_{j-1} is the DFS-parent of node i_j $(2 \leq j \leq k)$, (3) the DFS-index of n' is lower than that of n; and (4) the DFS-index of n' is the lowest among all such possible paths. The algorithm developed in [19] employs the traditional low-point value and path.

We define the *generalized low-point value* (GLPV) of a node n as the lowest DFS-index of a node that can be reached from node n by traversing a sequence of nodes with

[1] A DFS-tree is a tree rooted at the drain and the arcs in the tree are directed away from the drain. A back edge is an edge that connects a higher DFS-index node to a lower DFS-index node. The *low-point node* of a node n is the node whose DFS-number is the LPV of node n.

increasing DFS-index with the exception of the last hop. The *generalized low-point path* of a node n is of the form $n \rightarrow i_1 \rightarrow i_2 \rightarrow \ldots \rightarrow i_k \rightarrow n'$ $(k \geq 0)$, such that: (1) the DFS-index of n is lower than that of i_1, (2) the DFS-index of i_{j-1} is lower than that of i_j $(2 \leq j \leq k)$, (3) the DFS-index of node n' is lower than that of node n, and (4) the DFS-index of n' is the lowest among all such possible paths. The *generalized low-point neighbor* (GLPN) of a node n is defined as that neighbor of node n which is on its generalized low-point path.

The GLPV and GLPN of a node are computed during the distributed DFS numbering phase. The algorithm to assign the DFS-indices and compute the GLPV and GLPN is shown in Figure 2. The DFS-indices of all the nodes are initialized to -1. We incorporate hop count as a metric to compute the shortest generalized low-point path among those available. Note that the linear-time algorithm developed in this paper will work with the traditional low-point of a node, however, the path length optimization cannot be made as the arcs are forced to be on the DFS-tree, except the last hop.

Notation	**Comment**
$\texttt{dfs}[\texttt{n}]$	DFS-index of node $\texttt{n}$.
$\texttt{dfsparent}[\texttt{n}]$	DFS-parent of node $\texttt{n}$.
$\texttt{glpv}[\texttt{n}]$	Generalized low-point value of node $\texttt{n}$.
$\texttt{glpn}[\texttt{n}]$	Generalized low-point neighbor of node $\texttt{n}$.
$\texttt{glpd}[\texttt{n}]$	Generalized low-point distance (hop count) of node $\texttt{n}$.

```
DFS(parent, n, currdfs)
1.      if $\texttt{dfs}[\texttt{n}] > 0$ return currdfs;
2.      $\texttt{dfs}[\texttt{n}] = \texttt{currdfs}$; $\texttt{dfsparent}[\texttt{n}] = \texttt{parent}$; $\texttt{currdfs} = \texttt{currdfs} + 1$;
3.      for every neighbor $\texttt{i} \neq \texttt{parent}$ of n do:
3.A.        currdfs = DFS(n, i, currdfs);
3.B.        if $(\texttt{dfs}[\texttt{i}] < \texttt{dfs}[\texttt{n}])$ and $(\texttt{dfs}[\texttt{i}] \leq \texttt{glpv}[\texttt{n}])$
3.B.i.          $\texttt{glpv}[\texttt{n}] = \texttt{dfs}[\texttt{i}]$; $\texttt{glpn}[\texttt{n}] = \texttt{i}$; $\texttt{glpd}[\texttt{n}] = 1$;
3.C.        else if $(\texttt{dfs}[\texttt{i}] > \texttt{dfs}[\texttt{n}])$ and $(\texttt{glpv}[\texttt{i}] < \texttt{glpv}[\texttt{n}])$
3.C.i.          $\texttt{glpv}[\texttt{n}] = \texttt{glpv}[\texttt{i}]$; $\texttt{glpn}[\texttt{n}] = \texttt{i}$; $\texttt{glpd}[\texttt{n}] = \texttt{glpd}[\texttt{i}] + 1$;
3.D.        else if $(\texttt{dfs}[\texttt{i}] > \texttt{dfs}[\texttt{n}])$ and $(\texttt{glpv}[\texttt{i}] = \texttt{glpv}[\texttt{n}])$ and $(\texttt{glpd}[\texttt{i}] < \texttt{glpd}[\texttt{n}] - 1)$
3.D.i.          $\texttt{glpn}[\texttt{n}] = \texttt{i}$; $\texttt{glpd}[\texttt{n}] = \texttt{glpd}[\texttt{i}] + 1$;
4.      return currdfs;
```

Fig. 2. Algorithm to assign DFS-indices to the nodes and compute generalized low-point value and neighbor of a node.

Given a two-edge-connected network, the GLPV of a node n is lower than or equal to the DFS-index of its DFS-parent. Given a two-node-connected network, the GLPV of a node n is strictly lower than the DFS-index of its DFS-parent. The GLPV provides a mechanism to identify if the network is two-edge or two-node connected for reaching the drain in linear time. Every node sends a `DFS` message to all its neighbors (except its parent) and receives a `DFSRETURN` message in response. The number of `DFS` and `DFSRETURN` messages sent are $2|\mathcal{L}| - (|\mathcal{N}| - 1)$ each. At the end of the distributed DFS-numbering phase, every node in the network is aware of the DFS numbers of its neighbors. The neighbors of a node are arranged in an increasing order of their DFS-indices.

4.2 Distributed Path Augmentation

An overview of the steps involved in the distributed path augmentation is shown in Figure 3. The drain is initialized to the TOKEN state, indicating that it is already added to the trees and has the authority to initiate path search. The other nodes are initialized to the UNVISITED state.

Distributed Path Augmentation Algorithm

1. Arrange the neighbors in the neighbor list in an increasing order of their DFS-indices.
2. On receiving a TOKEN message, initiate path search along every node in the neighbor list, one at a time.
   (a) Every node that receives the SEARCH message forwards it sequentially to every node in the neighbor list according to some forwarding rules.
   (b) When a SUCCESS message returns from a neighbor, the value of msg.flagNewNodeAdded is stored in the neighbor list.
3. Forward the TOKEN message to every node if the flagNewNodeAdded flag for this neighbor is TRUE. The neighbor list is traversed in the reverse direction. Every node finishes its operation and sends a RETURN message back.
4. After receiving a RETURN message from all the neighbors to whom the token message was sent, send RETURN message to the parent that sent the TOKEN message.

Fig. 3. Overview of the steps involved in the distributed algorithm for computing colored trees.

Path search. The drain initiates a path search sequentially along its neighbors. The first search is for a cycle while the others are for a path. On receiving a SEARCH message, a node in the UNVISITED state changes itself to the VISITED state, which indicates that the node is part of the path being chosen for augmentation. The SEARCH message is then forwarded to one of the neighbors (based on the forwarding rules discussed later). The drain always responds to a SEARCH message with a SUCCESS. When a SEARCH message reaches any other node, that node responds with a SUCCESS message if it is in the CYCLE state[2]. If a node in the TOKEN state, it is configured to send a SUCCESS message, then the paths for augmentation may start and end at the same node, resulting in a solution for the CT-LD case. If the node in the TOKEN state is configured to respond with a FAILURE, then the result would be a solution for the CT-ND case.

A node in the CYCLE state responds to a SEARCH message with a SUCCESS message in which msg.flagNewNodeAdded is set to FALSE, indicating no new nodes are added further down the path. This enables the receiving node to not forward the search token to a node that is already on the cycle. As a rule, *a path search token may be forwarded from node i to node j only if node j was added to the path through a path search message from node i to j*. Note that as paths are being searched from the highest node in the global-order list (maintained in a distributed manner), any node that is on the cycle that receives the search message must be lower in the global-order list than the node that initiated the message. Upon receiving a SUCCESS message, a node in the VISITED state

[2] A node in the CYCLE state indicates that it has already been added to the colored trees. It has not received the TOKEN message to initiate path search.

changes to the CYCLE state. It adds the node from which it received the SEARCH message as its parent on the blue tree and the node from which it received the SUCCESS as its parent on the red tree. The flagNewNodeAddedvariable for that neighbor is set to the value indicated in the message. The node then sends a SUCCESS message to the node from which it received the SEARCH message with the msg.flagNewNodeAdded set to true, indicating that it was newly added to the path.

Forwarding search token. A node that has the path search token attempts to augment a path through each of its neighbor. The node then forwards the token to those eligible neighbors, traversing the ordered list in the reverse direction (opposite to the order in which the SEARCH messages were initiated), one at a time. An eligible neighbor is one for which the variable flagNewNodeAdded is set to true. Such an order reversal for passing the token helps maintain a consistent global ordering in a distributed manner across all the nodes in the network. A node that receives a TOKEN changes its state from CYCLE to TOKEN, starts the path search along each of its neighbors, and forwards the token to its eligible neighbors.

Once the tokens are returned by all neighbors, the node sets its state to FINISH and returns the token to the node from which it first received the token. The token finally reaches the drain, indicating that all nodes in the network are in the FINISH state, at which point the algorithm terminates.

4.3 Forwarding Rules for Path Augmentation Without Backtracking

The cycle and paths required for the distributed path augmentation approach are computed using four types of messages. A node sends out a SEARCH message to obtain a path (or cycle) for augmentation. In order to obtain a path in a distributed fashion without backtracking, we develop certain forwarding rules with some additional information in every message.

Let a node x, which has been already added to the trees, attempt a path search by sending a message through its neighbor y. Every search message msg contains the following fields: (1) msg.source is the source of the message, (2) msg.sourcedfs is the DFS-index of the source; (3) msg.specialFlag is a flag based on which the message may be routed to a different neighbor other than the default lowest DFS-index neighbor; and (4) msg.glpv is the GLPV of the source that initiated the message which could be modified at an intermediate node. A SEARCH message initiated by node x has the msg.specialFlag set to DEFAULT. If node y is not added to the colored trees, it forwards the message to one of its neighbors according to the rules shown in Figure 4.
Case 1: $\mathtt{dfs}[\mathtt{x}] > \mathtt{dfs}[\mathtt{y}]$.
In this case, there exists a path from y to the drain by successively traversing the lowest DFS-index neighbor from y to reach the drain. As the drain is already a part of the cycle, the message either reaches the drain or any other node that is already added to the trees (in CYCLE state) without backtracking. The specialFlag in the message is set to DEFAULT (refer to Step 4.A of Figure 4).

Case 2: $\mathtt{dfs}[\mathtt{x}] < \mathtt{dfs}[\mathtt{y}]$ *and* x *is the DFS-parent of* y.
In this case, if there exits a node z in the neighborhood of y whose DFS-index is lower than that of x, then the message could be forwarded to node z. If such a node does not

Notation	Comment
`msg`	Message received by node `y`.
`msg.source`	Source node of message `msg`.
`msg.sourcedfs`	DFS-index of the source node of message `msg`.
`msg.specialFlag`	Special flag field in the message.
`msg.glpv`	Generalized low-point value indicated by a node in the message.
`newmsg`	Message sent by node `y`.

Rules to forward a message.

```
1.       newmsg.source = y;  newmsg.sourcedfs = dfs[y];
2.           if (msg.specialFlag = PARENTFLAG)
2.A.             z = lowpoint[y];
2.B.             if (dfs[z] = glpn[y]) newmsg.specialFlag = DEFAULT;
2.C.             else newmsg.specialFlag = PARENTFLAG;
3.           else if (msg.specialFlag = LOWPOINTFLAG)
3.A.             if (glpv[y] < msg.glpv)
3.A.i.               z = glpn[y]; newmsg.specialFlag = PARENTFLAG;
3.B.             else
3.B.i.               z = dfsparent[y];
3.B.ii.              newmsg.specialFlag = LOWPOINTFLAG; newmsg.glpv = msg.glpv;
4.           else
4.A.             if the lowest DFS-index neighbor is not the same as msg.source
4.A.i                z = lowest DFS-index neighbor;  newmsg.specialFlag = DEFAULT;
4.B.             else if (msg.source = dfsparent[y])
4.B.i.               z = glpn[y];  newmsg.specialFlag = PARENTFLAG;
4.C.             else if (msg.source = glpn[y])
4.C.i.               z = dfsparent[y];  newmsg.specialFlag = LOWPOINTFLAG;
4.C.ii.              newmsg.glpv = msg.sourcedfs;
4.D.             else if (msg.sourcedfs < dfs[y])
4.D.i.               z = glpn[y];  newmsg.specialFlag = PARENTFLAG;
4.E.             else //Comment: msg.sourcedfs < dfs[y]
4.E.i.               z = lowest DFS-index neighbor that is not the same as msg.source;
4.E.ii.              newmsg.specialFlag = DEFAULT;
5.           Send newmsg to node z.
```

Fig. 4. Rules to forward a `SEARCH` message when received by a node `y` that is not added to the trees yet.

exist, then node `y` forwards the message to its GLPN. The `specialFlag` in the message is set to `PARENTFLAG` indicating that the message was received from a DFS-parent, hence must be forwarded to the GLPN successively (refer to Step 4.B of Figure 4). The message is forwarded along the generalized low-point path (refer to Step 2 of Figure 4). The node that forwards this message to the low-point node resets the flag to `DEFAULT`. From the low-point node onwards, the message is forwarded to the lowest DFS-index neighbor until it reaches the drain. Since the generalized low-point path does not involve any loops, a path is chosen for augmentation without backtracking.

The above two cases are sufficient if the colored trees are constructed to satisfy the CT-LD constraint. Hence, steps 1, 2, 4.A, 4.B, 4.E and 5 are sufficient in the set of rules to construct the colored trees satisfying CT-LD constraint. Note that if conditions 4.A and 4.B fail, then it implies that node x is the lowest DFS-index neighbor of y and is not its parent. Then, there must exist one node z in the neighborhood of y such that $\texttt{dfs}[x] < \texttt{dfs}[z] < \texttt{dfs}[y]$. One such obvious node is the DFS-parent of y. The message is forwarded along this neighbor with the DEFAULT flag. The message follows the lowest DFS-index neighbor successively to reach a node already added to the trees. The node at which the path augmentation terminates could be the same node that initiated the path search message, as the construction needs to satisfy the CT-LD constraint only.

However, if the colored tree construction were to satisfy the CT-ND constraint, then the nodes that start and terminate the paths must be distinct, for which the following rules are developed.

Case 3: $\texttt{dfs}[x] < \texttt{dfs}[y]$ *and* x *is the GLPN of* y.
In this case, $\texttt{dfs}[x]$ is the GLPV of node y. The message in this case is forwarded to the DFS-parent of y and msg.specialFlag is set to LOWPOINTFLAG, indicating that the message was obtained from a low-point node, hence must be forwarded to the DFS-parent. In addition, the glpv field in the outgoing message is set to $\texttt{dfs}[x]$ (obtained from the sourcedfs field of the received message, refer Step 4.C in Figure 4). Such a forwarding is continued until the message reaches a node whose GLPV is lower than msg.glpv, from where the message follows the generalized low-point path with the specialFlag in the message changed to PARENTFLAG (refer to Step 3 Fig. 4). From this point on, the forwarding of the message takes place similar to that of Case 2.

Note that when a message is forwarded to the DFS-parent upon msg.specialFlag = LOWPOINTFLAG, the message cannot reach the node that started the path search process, namely x. This would imply that the path search for augmentation started and ended at the same node. This in turn implies that no node in the DFS-tree beneath node x had a GLPV lower than $\texttt{dfs}[x]$. This contradicts the fact that when a network is two-vertex connected, then the GLPV of a child of x is strictly lower than $\texttt{dfs}[x]$. Hence, there exists an intermediate node whose GLPV is lower than that of $\texttt{dfs}[x]$.

It can also be easily shown that the generalized low-point path taken from the intermediate node does not loop back to any of the nodes in the path through the DFS-parents as the intermediate nodes in the former path would have a GLPV value strictly lower than that at the intermediate nodes in the latter path. Hence, a path is chosen for augmentation without backtracking.

Case 4: $\texttt{dfs}[x] < \texttt{dfs}[y]$ *and* x *is neither the DFS-parent nor the GLPN of node* y.
In this case, node x is the lowest DFS-index node in the neighborhood of y and is not the GLPN of y. This implies that the GLPV of y is strictly lower than $\texttt{dfs}[x]$. The generalized low-point path of node y, by definition, does not contain x. Therefore, the message is forwarded to the GLPN of y with the specialFlag set to PARENTFLAG. The path taken by the message from then on is similar to that discussed in Case 2.

If the construction were to satisfy the CT-ND constraint, the forwarding algorithm will not reach Step 4.E as one of the earlier four cases would definitely hold true.

Every node attempts to find a path through each of its neighbor (except through the node from which it received the token), the number of SEARCH messages sent in the network is $2|\mathcal{L}| - (|\mathcal{N}| - 1)$. Every SEARCH message has a corresponding SUCCESS message. In addition, every node except the drain receives the `TOKEN` message to initiate a path search and sends a `RETURN` message. The total number of messages sent in the network is $8|\mathcal{L}| - 2|\mathcal{N}| + 2$. As the distributed algorithm is sequential in nature, the number of messages sent directly provides the running time of the distributed algorithm, which is linear in the number of links in the network.

Proof of correctness. The distributed algorithm is based on the path augmentation technique [17]. Hence, the proof of correctness of the algorithm follows from [17] and is not repeated in this paper due to space constraints.

5 Performance Evaluation

The linear-time distributed algorithm developed in this paper is evaluated on random topologies with 100, 200, 300, and 400 nodes. The topologies were constructed using Waxman's model [20]. The effectiveness of employing the generalized low-point (GLP) is studied by comparing the performance of the algorithm with that employing the traditional low-point (TLP) concepts. For each network size, twenty different topologies were simulated and the average results are shown in Table 1 for the CT-ND case. The "average minimum (maximum) path length" refers to the lowest (highest) path length among the two paths, averaged over all the nodes in the network. It is observed that a significant reduction in the average path lengths is obtained by employing the generalized low-point concept, which allows optimization of hop-count on the low-point path. The number of messages used in both the approaches were the same. Similar results were obtained for the CT-LD case and are not shown here due to space constraints.

Number of Nodes	Average Number of Links	Average Red Path Length		Average Blue Path Length		Average Minimum Path Length		Average Maximum Path Length		Average Total Path Length		Reduction in Average Total Path Length
		TLP	GLP	TLP	GLP	TLP	GLP	TLP	GLP	TLP	GLP	
100	**774.2**	6.82	5.53	10.85	5.69	4.81	3.80	12.86	7.42	17.67	11.22	36.5%
200	**1382.65**	11.68	11.49	14.36	8.10	7.65	6.08	18.39	13.51	26.04	19.59	24.8%
300	**2585.56**	15.21	14.24	20.40	9.05	9.78	7.10	25.83	16.19	35.61	23.29	34.6%
400	**4540.45**	16.49	15.99	20.56	8.67	10.82	6.93	26.23	17.73	37.05	24.66	33.4%

Table 1. Comparison of the results of the distributed algorithm to compute colored trees satisfying CT-ND constraint employing traditional low-point and generalized low-point concepts.

6 Conclusions

This paper develops a linear-time distributed algorithm for the construction of colored trees for link/node-disjoint multipath routing to a particular drain in the network. The total number of messages sent in the network is shown to be $8|\mathcal{L}| - 2|\mathcal{N}| + 2$. The paper also demonstrates that significant reduction in the average path lengths may be obtained by employing generalized low-point concept in a DFS-tree rather than the traditional low-point concept.

References

1. Ye, Z., Krishnamurthy, S., Tripathi, S.: A framework for reliable routing in mobile adhoc networks. In: Proceedings of IEEE INFOCOM'03. (2003) 270–280
2. Pham, P.P., Perreau, S.: Performance analysis of reactive shortest path and multipath routing mechanism with load balance. In: Proceedings of IEEE INFOCOM. Volume 1. (2003) 251–259
3. Murthy, S., Garcia-Luna-Aceves, J.J.: Congestion-oriented shortest multipath routing. In: Proceedings of IEEE INFOCOM. Volume 3. (1996) 1028–1036
4. Ganesan, D., Govindan, R., Shenker, S., Estrin, D.: Highly resilient energy-efficient multipath routing in wireless sensor networks. ACK SIGMOBILE Mobile Computing and Communications Review **4**(5) (2001) 11–25
5. Bhandari, R.: Survivable Networks: Algorithms for Diverse Routing. Kluwer Academic Publishers (1999)
6. Grover, W.D.: Mesh-based Survivable Networks: Options and Strategies for Optical, MPLS, SONET and ATM Networking. Prentice Hall Publishers, New Jersey, USA (2003)
7. Begen, A.C., Altunbasak, Y., Ergun, O.: Multi-path selection for multiple description encoded video streaming. In: Proceedings of IEEE International Conference on Communications. Volume 3. (2003) 1583–1589
8. Lee, S., Gerla, M.: Split multipath routing with maximally disjoint paths in ad hoc networks. In: Proceedings of IEEE ICC. (2001) 3201–3205
9. Nasipuri, A., Das, S.R.: On-demand multipath routing for mobile ad hoc networks. In: Proceedings of IEEE International Conference on Computer Communications and Networks. (1999) 64–70
10. Wu, J.: An extended dynamic source routing scheme in ad hoc wireless networks. In: Proceedings of 35th Annual Hawaii International Conference on System Sciences. (2002) 3832–3838
11. Marina, M.K., Das, S.R.: On-demand multipath distance vector routing in ad hoc networks. In: Proceedings of IEEE ICNP. (2001) 14–23
12. Park, V.D., Corson, M.S.: A highly adaptive distributed routing algorithm for mobile wireless networks. In: Proceedings of IEEE INFOCOM. (1997) 1405–1413
13. Raju, J., Garcia-Luna-Aceves, J.J.: A new approach to on-demand loop-free multipath routing. In: Proceedings of IEEE International Conference on Computer Communications and Networks (ICCCN). (1999) 522–527
14. Valera, A., Seah, W.K.G., Rao, S.V.: Cooperative packet caching and shortest multipath in mobile adhoc networks. In: Proceedings of IEEE INFOCOM. (2003) 260–269
15. Lee, S., Gerla, M.: Aodv-br: Backup routing in ad hoc network. In: Proceedings of IEEE WCNC. (2000) 1311–1316
16. Ramasubramanian, S., Krishnamoorthy, H., Krunz, M.: Disjoint multipath routing using colored trees. Technical Report, University of Arizona (2005)
17. Medard, M., Barry, R., Finn, S., Gallager, R.: Redundant trees for preplanned recovery in arbitrary vertex- redundant or edge redundant graphs. IEEE/ACM Transactions on Networking **7**(5) (1999) 641–652
18. Xue, G., Chen, L., Thulasiraman, K.: Quality-of-service and quality-of-protection issues in preplanned recovery schemes using redundant trees. IEEE Journal on Selected Areas in Communication **21**(8) (2003) 1332–1345
19. Zhang, W., Xue, G., Tang, J., Thulasiraman, K.: Linear time construction of redundant trees for recovery schemes enhancing QoP and QoS. In: Proceedings of IEEE INFOCOM, Miami, FL, USA (2005) 2702–2710
20. Waxman, B.M.: Routing of multipoint connections. IEEE Journal of Selected Areas in Communications **6**(9) (1988) 1617–1622