Faster Algorithms for Construction of Recovery Trees Enhancing QoP and QoS

Weiyi Zhang, *Member, IEEE*, Guoliang Xue, *Senior Member, IEEE*, Jian Tang, *Member, IEEE*, and Krishnaiyan Thulasiraman, *Fellow, IEEE*

***Abstract*—Médard *et al.* proposed an elegant recovery scheme (known as the MFBG scheme) using red/blue recovery trees for multicast path protection against single link or node failures. Xue *et al.* extended the MFBG scheme and introduced the concept of quality of protection (QoP) as a metric for multifailure recovery capabilities of single failure recovery schemes. They also presented polynomial time algorithms to construct recovery trees with good QoP and quality of service (QoS). In this paper, we present faster algorithms for constructing recovery trees with good QoP and QoS performance. For QoP enhancement, our $O(n + m)$ time algorithm has comparable performance with the previously best $O(n^2(n + m))$ time algorithm, where n and m denote the number of nodes and the number of links in the network, respectively. For cost reduction, our $O(n + m)$ time algorithms have comparable performance with the previously best $O(n^2(n+m))$ time algorithms. For bottleneck bandwidth maximization, our $O(m \log n)$ time algorithms improve the previously best $O(nm)$ time algorithms. Simulation results show that our algorithms significantly outperform previously known algorithms in terms of running time, with comparable QoP or QoS performance.**

***Index Terms*—Bottleneck bandwidth, protection and restoration, quality of protection (QoP), quality of service (QoS), redundant trees.**

I. Introduction

PROTECTION and restoration in high-speed networks are important issues that have been studied extensively [1], [9], [10], [16], [19]–[21]. They have important applications in both SONET and WDM networks [14], [15], [26]. Path protection and link protection schemes were proposed in [1] and [19]–[21]. Protection cycles and self-healing rings were discussed in [6] and [26]. The p-cycle protection scheme was proposed in [7]. Itai and Rodeh in [8] proposed to use multi-trees to achieve single link or single node failure restoration. For multicast protection and restoration, Médard *et al.* in [10]–[12] extended the multi-tree scheme by constructing a pair of redundant trees on arbitrary link-redundant or node-redundant networks. These schemes are applicable to IP, WDM, SONET, and ATM networks to provide multicast protection and restoration in case of single link or node failure [14], [16], [20]–[22].

In [11], Médard *et al.* presented an elegant scheme (known as the MFBG scheme) to construct a pair of directed spanning trees from a common root node s in a way that the failure of any edge (or a node other than the root node) in the graph leaves each vertex still connected with the root node using at least one of the trees, provided that the network is edge-redundant (or vertex-redundant). They named one of the trees the *red tree* (denoted by T^R) and the other the *blue tree* (denoted by T^B). They showed that, for any edge-redundant (or vertex-redundant) network, there exists a pair of red/blue trees which provides fast recovery from single link (node) failure. They also presented $O(n^3)$ time algorithms for constructing such a pair of red/blue trees for any edge-redundant (or vertex-redundant) network, where n is the number of nodes in the network.

In [27], [28], Xue *et al.* introduced the concept of Quality of Protection (QoP) and Quality of Service (QoS) of red/blue trees. The QoP of a pair of red/blue trees T^R and T^B is defined as the maximum integer k such that there exists an instance of k simultaneous link failures that T^R and T^B can survive. Depending on different design goals, the pair of red/blue trees could be constructed to maximize the QoP, to minimize the total cost, or to maximize the bottleneck bandwidth. Xue *et al.* in [28] presented an $O(n^2(m + n))$ time heuristic for constructing a pair of red/blue trees with enhanced QoP, two $O(n^2(m + n))$ time heuristics for constructing a pair of red/blue trees with low total cost, and an $O(nm)$ time algorithm for constructing a pair of red/blue trees with maximum bottleneck bandwidth, where n and m are the number of nodes and the number of links in the network, respectively.

In this paper, we present improved algorithms over those presented in [28]. For QoP enhancement, we present an $O(m + n)$ time algorithm that exhibits performance comparable to that of the algorithm of [28]. For cost reduction, we consider the case where edges have equal costs, and present two $O(m + n)$ time algorithms that exhibit performances comparable to those of the algorithms of [28]. For bottleneck bandwidth maximization, we present an $O(m \log n)$ time algorithm, improving the $O(mn)$ time algorithm of [28]. The algorithms presented in this paper excel both in terms of running time and in terms of QoP or QoS performance. Compared with the algorithms of [11] (which do not consider QoP or QoS), our algorithms are faster and have better QoP or QoS performance. Compared with the algorithms of [28], our algorithms have similar QoP or QoS performance, while having much faster running times.

The remainder of this paper is organized as follows. In Section II, we present some preliminaries which will be used in later sections. In Sections III–V, we present our algorithms

Manuscript received June 6, 2005; revised October 19, 2006, approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor C. Qiao. The work of G. Xue was supported in part by the Army Research Office under Grant W911NF-04-1-0385 and in part by the National Science Foundation under ITR Grant ANI-0312635 and Grant CCF-0431167. The work of K. Thulasiraman was supported by the National Science Foundation under Grant ANI-0312435. An earlier version of this paper appeared in IEEE INFOCOM'2005.

W. Zhang is with the Department of Computer Science, North Dakota State University, Fargo, ND 58105 USA (e-mail: weiyi.zhang@ndsu.edu).

G. Xue is with the Department of Computer Science and Engineering at Arizona State University, Tempe, AZ 85287-8809 USA (e-mail: xue@asu.edu).

J. Tang is with the Department of Computer Science, Montana State University, Bozeman, MT 59717-3880 USA (e-mail: tang@cs.montana.edu).

K. Thulasiraman is with the School of Computer Science, University of Oklahoma, Norman, OK 73019 USA (e-mail: thulasi@ou.edu).

Digital Object Identifier 10.1109/TNET.2007.900705

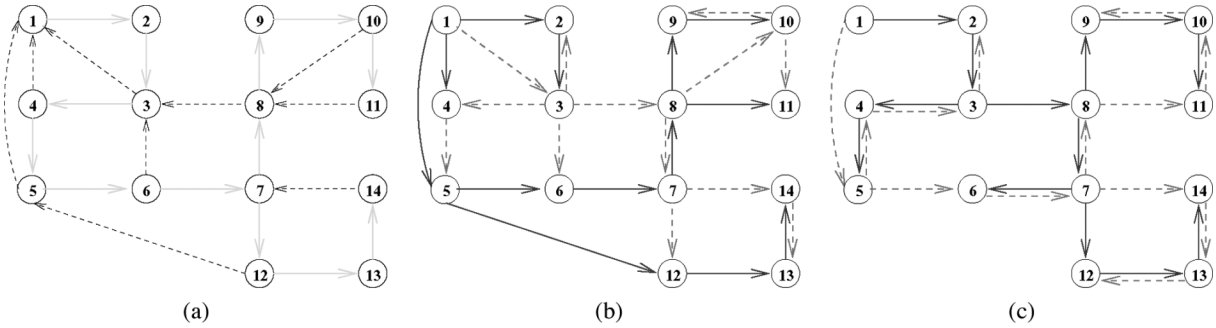


Fig. 1. (a) Sample network (ignoring edge directions, not distinguishing solid from dashed), a DFS tree (in solid edges), and back edges (in dashed edges). (b) Pair of single-link failure recovery trees (edges in the red tree are dashed, and edges in the blue tree are solid) with enhanced QoP. (c) Pair of single-link failure recovery trees (edges in the red tree are dashed, and edges in the blue tree are solid) with low cost.

for constructing a pair of red/blue trees with enhanced QoP, reduced cost, or maximum bottleneck bandwidth, respectively. In Section VI, we present computational results. We conclude this paper in Section VII.

II. Preliminaries

As in [11] and [28], we use an undirected graph to model the network. We use *vertices* and *nodes* interchangeably, as well as *edges* and *links*. We call a 2-edge connected graph *edge-redundant* and call a 2-vertex connected graph *vertex-redundant*. We use $\langle u, v\rangle$ to denote an *undirected edge* connecting nodes u and v and use (u, v) to denote a *directed edge* from node u to node v. Refer to [25] for other graph theoretic notations not defined here.

Definition 2.1: Let $G(V, E, s)$ be an undirected graph with vertex set V and edge set E, where $s \in V$ is a distinguished *root node*. Let T^R and T^B be a pair of directed trees such that there is a directed path p_v^R in T^R from s to every vertex $v \in V$ and a directed path p_v^B in T^B from s to every vertex $v \in V$.

1) T^R and T^B form a pair of *single-link failure recovery trees* if for any chosen edge $\langle u, v\rangle \in E$ and any $w \in V$, p_w^R and p_w^B do not both use edge $\langle u, v\rangle$.
2) T^R and T^B form a pair of *single-node failure recovery trees* if for any node $v \in V \setminus \{s\}$, and any node $w \in V \setminus \{s, v\}$, p_w^R and p_w^B do not both contain node v.

We will use *recovery trees* (also known as *red/blue trees*) to denote either a pair of single-node failure recovery trees or a pair of single-link failure recovery trees when the exact meaning can be derived from the context. □

We briefly illustrate the concept of recovery trees using Fig. 1. The sample network can be obtained by ignoring the directions of the edges in Fig. 1(a). Fig. 1(b) shows a pair of single-link failure recovery trees rooted at node 1, where the tree with solid links is T^B and the tree with dashed links is T^R. Before any link failure, every node is reachable from the root node via either the red tree or the blue tree. Suppose link $\langle 7, 8\rangle$ fails. Then, nodes 8–11 are disconnected from the root node on T^B, and nodes 7, 12, 13, and 14 are disconnected from the root node on T^R. However, nodes 8–11 are still connected with the root node on T^R and nodes 7, 12, 13, and 14 are still connected with the root node on T^B. This pair of trees is not a pair of single-node failure recovery trees. When node 8 fails, nodes 9–11 are disconnected from the root node in both trees. The pair of trees in Fig. 1(c) is a pair of single-node failure recovery trees.

Xue *et al.* in [28] noticed that, although a pair of single-link failure recovery trees only guarantees surviving against a single-link failure, it can also survive multiple link failures, provided that the failed links are *isolated*. For example, the pair of recovery trees in Fig. 1(b) can survive up to nine simultaneous link failures such as $\langle 1, 2\rangle$, $\langle 1, 4\rangle$, $\langle 4, 5\rangle$, $\langle 5, 6\rangle$, $\langle 3, 8\rangle$, $\langle 9, 10\rangle$, $\langle 10, 11\rangle$, $\langle 7, 12\rangle$, and $\langle 13, 14\rangle$. However, the pair of recovery trees in Fig. 1(b) cannot survive ten or more simultaneous link failures. This led them to introduce the concept of quality of protection (QoP), formally defined in the following.

Definition 2.2: Let T^R and T^B be a pair of single-link failure recovery trees. The *quality of protection* (QoP) of T^R and T^B is defined as the maximum integer k such that there exists an instance of k simultaneous link failures that T^R and T^B can survive. The MaxQoP problem asks for a pair of single-link recovery trees with maximum QoP. □

By this definition, the QoP of the pair of recovery trees in Fig. 1(b) is 9. Recovery trees with maximum QoP are sought when the design goal is to maximize the capability for multi-failure recovery. Besides optimizing the QoP metric, we are also interested in optimizing QoS metrics such as cost and bottleneck bandwidth as discussed in the following.

If network resource usage is a major concern, we should seek a pair of recovery trees with minimum total cost, measured as the sum of the costs of the links used by the pair of trees [28]. We consider the case where all edges have equal cost. A pair of single-link (single-node) failure recovery trees is called a pair of *min-cost single-link (single-node) failure recovery trees* if it has the minimum total cost among all single-link (single-node) failure recovery trees with the same root. If a bandwidth requirement is specified, we should seek a pair of recovery trees whose bottleneck bandwidth meets the requirement, where the bottleneck bandwidth of a pair of recovery trees is defined as the minimum edge bandwidth among the edges used by one of the trees. A pair of single-link (single-node) failure recovery trees is called a pair of *max-bandwidth single-link (single-node) failure recovery trees* if it has the maximum bottleneck bandwidth among all single-link (single-node) failure recovery trees with the same root.

In [28], Xue *et al.* presented polynomial time algorithms for constructing a pair of recovery trees with enhanced QoP, re-

Algorithm 1 DFS(G, v, u)

Input: The global arrays $D[]$ and $L[]$ are initialized to zero and the global variable N is initialized to 1. The set of (directed) *tree edges* $\mathcal{T}$ and the set of (directed) *back edges* $\mathcal{B}$ are initialized to $\emptyset$.

Output: DFS tree $\mathcal{T}$, back edges $\mathcal{B}$, DFS number $D[]$ and lowpoint number $L[]$.

```
1: D[v] := N; L[v] := D[v]; N := N + 1;
2: for each vertex w adjacent from v do
3:    if (D[w] = 0) {w is a descent of v} then
4:       T := T ∪ {(v, w)}; DFS(G, w, v);
         L[v] := min(L[v], L[w]); parent[w] := v;
5:    else if (w ≠ u) {w is an ancestor of v} then
6:       B := B ∪ {(v, w)}; L[v] := min(L[v], D[w]);
7:    end if
8: end for
```

duced cost, or maximum bandwidth. In this paper, we present *faster* algorithms for these problems. Our algorithms are based on the *Depth First Search* (DFS) technique [23], which is briefly described as Algorithm 1. We assume that the graph G has n vertices and m edges.

If (u, v) is a directed edge in the DFS tree $\mathcal{T}$, then u is called a *parent* of v, and v is called a *child* of u. Node x is called an *ancestor* of node y (y is called a *descendant* of x) if there is a directed path from x to y in $\mathcal{T}$. If x is an ancestor of y in $\mathcal{T}$, we use $\pi(x, y)$ to denote the directed path from x to y in $\mathcal{T}$. In the description of the Algorithm 1, v is the current vertex to be visited, u is the parent of v in the DFS tree if v is not the root node s. The DFS *number* of v, denoted by $D[v]$, indicates the order that node v is visited during the depth first search. A DFS tree assigns a unique direction to each edge in G and classifies these directed edges into *tree edges* and *back edges*. Let $\langle u, v\rangle$ be an undirected edge of G such that $D[u] < D[v]$. If $(u, v) \in \mathcal{T}$, we correspond the undirected edge $\langle u, v\rangle = \langle v, u\rangle$ to the directed tree edge (u, v). Otherwise, we correspond the undirected edge $\langle u, v\rangle = \langle v, u\rangle$ to the directed back edge (v, u). The *lowpoint number* of node v, denoted by $L[v]$, is the smaller of $D[v]$ and the smallest/lowest DFS number of a vertex u that can be reached from v by a sequence of zero or more tree edges followed by a back edge. The concepts of acceptable back edge and maximal back edge defined in the following play an important role in our construction of recovery trees.

Definition 2.3: Given a DFS tree $\mathcal{T}$ of graph G, together with a pair of current red/blue trees T^R and T^B spanning *a subset of the network nodes* (*including the root node* s), a back edge (u, w) of $\mathcal{T}$ is called an *acceptable back edge with respect to* T^R *and* T^B, if node u is not on T^R and T^B while node w is on both T^R and T^B. We will use the term *acceptable back edge* if T^R and T^B can be implied from the context. An acceptable back edge (u, w) is called a *maximal back edge* if it is impossible to reach a node on T^R and T^B from u by a sequence of one or more tree edges followed by a back edge. □

Intuitively, if (u, w) is an acceptable back edge, then both T^R and T^B contain node w, but not node u. Let v be the nearest ancestor of u (on the DFS tree $\mathcal{T}$) that is on both T^R and T^B, then the tree path from v to u, concatenated with the back edge (u, w) forms a path (or cycle, in case $v = w$) connecting nodes v and w via nodes not yet on T^R and T^B. This enables us to construct a pair of recovery trees efficiently in Section III. The concept of maximal back edge will be used in Section IV to efficiently construct a pair of recovery trees with low cost.

Without loss of generality, we assume that $\mathcal{T}$, T^R and T^B are all rooted at node s and that nodes in G are relabeled so that $D[v] = v$ for every node v. Fig. 1(a) shows a DFS tree $\mathcal{T}$ of a graph G whose edge set can be obtained by ignoring the directions of both the solid edges and the dashed edges in Fig. 1(a). The solid links represent *tree edges* and the dashed links represent *back edges*. Suppose we are at the stage when only the root node 1 is on the current red/blue trees. Then back edges (3, 1), (4, 1) and (5, 1) are the only acceptable back edges. Among these, (5, 1) is the only maximal back edge.

It is worth noting that the concepts of acceptable back edge and maximal back edge rely on the current T^B and T^R (under construction). It is possible for a back edge which is non-acceptable at a point in time to become acceptable later. For example, if node 3 had been added onto the current T^B and T^R, then back edges (6, 3) and (8, 3) would become acceptable back edges, and (8, 3) would become a maximal back edge. If (u, v) is a back edge, we say it is an *outgoing* back edge of node u, and an *incoming* back edge of node v. This concept extends to acceptable back edges and maximal back edges. All of our algorithms depend on the DFS tree $\mathcal{T}$ rooted at node s. When we talk about *ancestors* or *descendants* of nodes, it is understood that they are with respect to the DFS tree $\mathcal{T}$. We will use the *voltage technique* of [11]. For single-node failure recovery trees, each node u is assigned a positive *blue voltage* $v^B(u) > 0$. The root node s also has a *red voltage* $v^R(s) = 0$. For single-link failure recovery trees, each node u is assigned a *blue voltage* $v^B(u)$ and a *red voltage* $v^R(u)$ such that $v^B(u) > v^R(u)$, with the root node s having a zero red voltage, $v^R(s) = 0$. To speed up the computation, at each node u, we also maintain (using the variable $v_{max}(u)$) *the maximum voltage that is smaller than* $v^B(u)$.

III. Enhancing Quality of Protection

In this section, we present an $O(n + m)$ time algorithm for constructing a pair of single-link failure recovery trees with QoP performance comparable to the $O(n^2(n + m))$ time algorithm of [28]. In [28], Xue *et al.* noted that the construction of red/blue trees is closely related to the *ear decomposition* of a graph [25]. An *open ear* of a graph $\mathcal{G}$ is a maximal path whose internal vertices have degree 2 in $\mathcal{G}$. A *closed ear* of a graph $\mathcal{G}$ is a cycle C such that all vertices of C except one have degree 2 in $\mathcal{G}$. An (open) *ear decomposition* of $\mathcal{G}$ is a decomposition $P_0, \ldots, P_k$ such that P_0 is a cycle and P_i for $i \geq 1$ is an open ear of $P_0 \cup \cdots \cup P_i$. A *closed-ear decomposition* of $\mathcal{G}$ is a decomposition $P_0, \ldots, P_k$ such that P_0 is a cycle and P_i for $i \geq 1$ is either an open ear or a closed ear of $P_0 \cup \cdots \cup P_i$. For example, for the sample network in Fig. 1(a) (ignoring edge directions), we have a closed-ear decomposition with 8 ears (where P_6 is a closed ear): $P_0 = (1, 2, 3, 1)$, $P_1 = (1, 4, 3)$, $P_2 = (1, 5, 4)$, $P_3 = (5, 6, 3)$, $P_4 = (6, 7, 8, 3)$, $P_5 = (5, 12, 7)$, $P_6 = (8, 9, 10, 8)$, $P_7 = (8, 11, 10)$, $P_8 = (12, 13, 14, 7)$.

Xue *et al.* [28] proved that the QoP of a pair of red/blue trees is the same as the number of ears in the corresponding closed-ear decomposition of the subgraph of G induced by the pair of red/blue trees. Consequently, it is desirable to use more ears in the construction of the red/blue trees if QoP is our major concern.

We have two design goals here. First, we want to construct a pair of recovery trees efficiently. Second, we want to construct the pair of recovery trees using as many ears as possible. Following the discussion immediately after Definition 2.3, we construct the pair of recovery trees T^R and T^B by adding ears that correspond to acceptable back edges. There are two advantages to this approach. First, by finding ears starting from an acceptable back edge we can construct the pair of recovery trees in linear time, which is asymptotically optimal. Second, by adding an ear whenever we see an acceptable back edge, we can construct a pair of recovery trees using relatively more ears. Therefore, the pair of recovery trees so constructed should have high QoP. Our algorithm is listed as Algorithm 2.

Algorithm 2 EnhancedQoP(G, s)

1: Compute DFS tree $\mathcal{T}$ rooted at s. Relabel the nodes so that $D[v] = v$ for every node v. Initialize T^R and T^B to the empty tree. Initialize the queue $markedQ$ and the stack $markedS$ to empty.
2: Mark node s, insert it into $markedQ$ and add it onto T^B and T^R. Set the voltages $v^B(s)$ and $v^R(s)$ such that $v^B(s) > v^R(s) = 0$. Set $v_{max}(s) := 0$.
3: **while** ($markedQ$ is NOT EMPTY) **do**
4: Dequeue a node u from $markedQ$.
5: **for** (each adjacent node w of u) **do**
6: **if** ((w, u) is a back edge) **then**
7: // mark nodes from w to its marked ancestor.
8: **while** (w is not marked) **do**
9: Mark w. Push w onto $markedS$. $w := parent[w]$;
10: **end while**
11: Pop all the elements on $markedS$ and insert them into $markedQ$ while popping.
12: **else if** ((u, w) is an *acceptable back edge*) **then**
13: // add an ear to T^R and T^B.
14: Let v be u's nearest ancestor (in $\mathcal{T}$) that is on the current T^B and T^R.
15: **if** $((v^B(w) > v^B(v)))$ **then**
16: $p_s := w$; $p_t := v$;
17: **else**
18: $p_s := v$; $p_t := w$;
19: **end if**
20: $\pi(v, u)$ concatenated with (u, w) forms a path or cycle (in case $v = w$) connecting w and v. Let this path or cycle be $(p_s, x_1, \ldots, x_k, p_t)$. Add $p_s \rightarrow x_1 \rightarrow \cdots \rightarrow x_k$ to T^B. Add $p_t \rightarrow x_k \rightarrow \cdots \rightarrow x_1$ to T^R. Assign voltages s.t. $v^B(p_s) > v^B(x_1) > v^R(x_1) > \cdots > v^B(x_k) > v^R(x_k) > v_{max}(p_s)$. Set $v_{max}(x_i) := v^R(x_i)$, $i = k, k-1, \ldots, 1$. $v_{max}(p_s) := v^B(x_1)$.
21: **end if**
22: **end for**
23: **end while**

The algorithm starts with T^R and T^B initialized to contain the root node s only. We make a node *marked* when it is visited to ensure linear time complexity. The algorithm uses a queue $markedQ$ to hold marked nodes that are not processed. It uses a stack $markedS$ to insert the marked nodes into $markedQ$ in a desired order. Initially, only the root node s is marked and inserted into the queue $markedQ$. With the exception of node s, all other nodes which are in $markedQ$ are candidate nodes to be added to the current T^R and T^B, but are not on the current T^R and T^B yet. There are two major steps in Algorithm 2. The first major step (Lines 7 through 11) is the *marking step*. Once we find that (w, u) is a back edge and w is not marked, we mark w and all of its unmarked ancestors (Lines 8 through 10) and insert these nodes into $markedQ$ in reverse order (Line 11). The second major step (Lines 13 through 20) is the *adding step*. Starting from an acceptable back edge (u, w), where u is the marked node that has just been deleted from $markedQ$, we add the ear formed by the tree path from v to u, followed by the back edge (u, w), where v is the nearest ancestor of u that is already on the current T^R and T^B. The process continues until all nodes are added to T^R and T^B.

We illustrate Algorithm 2 with the sample network (together with a DFS tree $\mathcal{T}$) shown in Fig. 1(a). Node 1 is the root node. In Line 1, we construct the DFS tree $\mathcal{T}$, which is shown in Fig. 1(a) where solid edges are *tree edges* and dashed edges are *back edges*. T^R, T^B, the queue $markedQ$ and the stack $markedS$ are both initialized to empty. In Line 2, we mark node 1 and insert it onto T^R and T^B, as well as into $markedQ$. We assign voltages at node 1 such that $v^B(1) > v^R(1) = 0$. Also, we set $v_{max}(1) := 0$. In Line 4, we dequeue node $u = 1$ from $markedQ$. Next, in Line 5, we check each adjacent node w of u (in the order 2, 3, 4, 5). With $w = 2$, the condition in Line 6 is not true, nor is the condition in Line 12. So, control returns to Line 5 with w updated to 3. Since (3, 1) is a back edge, the condition at Line 6 is true. The WHILE-loop in Lines 8 through 10 marks node 3, push 3 onto $markedS$, sets $w = 2$; then marks node 2, and push it onto $markedS$, sets $w = 1$; and exits the WHILE-loop (since 1 is already marked). Then in Line 11, nodes 2, 3 are popped from $markedS$ and inserted into $markedQ$ in that order. Now control returns to Line 5 with w updated to 4. Since (4, 1) is a back edge, the condition at Line 6 is true. Node 4 is marked, pushed onto $markedS$, popped from $markedS$, and inserted into $markedQ$. Now control returns to Line 5 with w updated to 5. Similarly, node 5 is marked and inserted into $markedQ$. At this moment, the elements in $markedQ$ are 2, 3, 4, 5.

Next we dequeue node $u = 2$ in Line 4. We check each adjacent node w of u (in the order 1, 3). For each of the two values of w, the conditions in Lines 6 and 12 are both false. So node $u = 3$ is dequeued from $markedQ$.

We check each neighbor w (1, 2, 4, 6, 8) of $u = 3$. Since (3, 1) is an acceptable back edge, the condition in Line 12 is satisfied (with $w = 1$). So we will add an ear to T^R and T^B. To find u's nearest ancestor on T^R and T^B, we follow the parent pointers

$u = 3 \rightarrow 2 \rightarrow 1$ to find that u's nearest ancestor on T^R and T^B is node $v = 1$. Since the condition $(v^B(w) > v^B(v))$ is not true, we set $p_s = v = 1$ and $p_t = w = 1$. In Line 20, the links (1, 2) and (2, 3) are added to T^B, the links (1, 3) and (3, 2) are added to T^R. Now nodes 2 and 3 are also on T^B and T^R. We assign the voltages for nodes 2 and 3 according to the rules of [11] such that $v^B(1) > v^B(2) > v^R(2) > v^B(3) > v^R(3) > v_{max}(1)$. Now $v_{max}(3)$ is set to $v^R(3)$, $v_{max}(2)$ is set to $v^R(2)$, $v_{max}(1)$ is updated to $v^B(2)$, in that order. Now control returns to Line 5 with w updated to 2. None of the conditions at Lines 6 and 12 is true. So control returns to Line 5 with w updated to 4. Similarly, none of the conditions at Lines 6 and 12 is true. Now control returns to Line 5 with w updated to 6. Since (6, 3) is a back edge, the condition at Line 6 is true. Lines 8 through 11 mark node 6 and insert it into $markedQ$. Similarly, for $w = 8$, nodes 8 and 7 are marked and inserted into $markedQ$ in the reverse order.At this point, the nodes in $markedQ$ are 4, 5, 6, 7, 8. Following this way, additional ears are added to T^B and T^R in the following order: (1, 4, 3), (1, 5, 4), (5, 6, 3), (6, 7, 8, 3), (5, 12, 7), (12, 13, 14, 7), (8, 9, 10, 8), (8, 11, 10). This leads to the pair of red/blue trees as shown in Fig. 1(b).

Lemma 3.1: After each execution of the statement of the WHILE-loop in Line 3 of Algorithm 2 (Line 4 through Line 22), the following hold true.

1) If a node u is marked, then all ancestors of u in $\mathcal{T}$ are also marked.
2) If a node u is on T^R and T^B, then all ancestors of u in $\mathcal{T}$ are also on T^R and T^B.

If G is 2-edge connected, then every node of G will be marked and added to T^R and T^B at the end of the algorithm. □

Proof: Initially, only the root node s is marked and is on T^B and T^R. Therefore, the two claims of the lemma are true. Every time a node is marked during the WHILE-loop, we also mark all of its unmarked ancestors (see Lines 8 through 10). Therefore, the first claim of the lemma is true after the execution of Line 4–Line 22.

Every time some nodes are added to T^R and T^B during the process, we find either a path connecting two nodes (w and v) already on T^R and T^B via some nodes not yet on T^R and T^B, or a cycle consisting of exactly one node ($w = v$) already on T^R and T^B and some nodes not yet on T^R and T^B. However, the path or cycle chosen by the algorithm consists of a tree path (which is $\pi(v, u)$) concatenated with an acceptable back edge (which is (u, w)). Before this path or cycle is added to T^R and T^B, node v is already on T^R and T^B. Therefore, all ancestors of node v are already on T^R and T^B. When this path or cycle is added to T^R and T^B, we add all the nodes on the tree path $\pi(v, u)$ (node that v is already on T^R and T^B). Let $y \neq v$ be a node that is on the tree path $\pi(v, u)$ and x be an ancestor of y in $\mathcal{T}$. If y is on $\pi(v, u)$, then y is added to T^R and T^B together with x. If y is not on $\pi(v, u)$, then y is also an ancestor of v, and therefore is already on T^R and T^B before this step. This proves that the second claim of the lemma holds true after each addition of a path or a cycle.

Next, we will prove that every node will be marked at the end of the algorithm. To the contrary, assume that node y is not marked after running Algorithm 2. Without loss of generality, we further assume that all of y's ancestors are marked. Let x denote the parent node of y. Since G is a 2-edge connected graph, the edge $\langle x, y\rangle$ is not a *bridge* [25]. Therefore, there exists a descendant z of y (in tree $\mathcal{T}$) and an ancestor α of y (α could coincide with x) such that (z, α) is a back edge [23]. By our assumption, α is marked. At the time node α is dequeued from $markedQ$ and the incoming back edge (z, α) is processed in Line 6 of the algorithm, every ancestor of z (including node y) should be marked when we finish processing the back edge (z, α). The contradiction proves our claim.

Finally, we prove that every node will be added to T^R and T^B. We proved that every node will be marked. A node v is marked by our algorithm because it has an ancestor x which is already marked and a descendant y that has a back edge (y, u) where u is already on T^R and T^B and is either x or an ancestor of x. When node y is dequeued from $markedQ$, y and all its ancestors that have not been added to T^R and T^B will be added to T^R and T^B. This proves that every marked node will be added to T^R and T^B. □

Theorem 3.1: Assume that G is a 2-edge connected graph with n vertices and m edges. Algorithm 2 computes a pair of single-link failure recovery trees correctly. Furthermore, the worst-case running time of the algorithm is $O(m + n)$. □

Proof: It follows from Lemma 3.1 that when Algorithm 2 terminates, all nodes are added to T^R and T^B. Each time a path/cycle is found, we grow T^R and T^B according to the voltage rules [11]. It follows from [11] that T^R and T^B form a pair of single-link failure recovery trees.

Line 1 of the algorithm constructs the DFS tree $\mathcal{T}$ and computes the corresponding information in $O(m + n)$ time [23]. Line 2 requires $O(1)$ time. In the rest of the proof, we will show that $O(n + m)$ time is sufficient for *all executions of Lines 3 through 15*.

Each node will be marked exactly once, pushed onto the stack $markedS$, popped off the stack and inserted into the queue $markedQ$, and eventually dequeued from the queue. When a node u is dequeued from $markedQ$, we examine all the edges adjacent with node u. All these examinations (when u loops over all nodes in G) require a total execution time of $O(n+m)$. When some nodes are marked in the inner WHILE-loop on Line 7, these nodes will be pushed onto the stack $markedS$ and then popped off the stack and inserted into the queue $markedQ$. The time required for these operations is proportional to the number of nodes marked during this step. Therefore, all these stack and queue operations combined require $O(n)$ time. When some marked nodes (in a path/cycle) are added to T^R and T^B, the time required to find the nearest ancestor v and the time required to assign the voltages to the nodes newly added to T^R and T^B is proportional to the number of nodes newly added T^R and T^B. Therefore, the total time required for adding nodes to T^R and T^B and assigning voltages is $O(n)$. This proves that the time complexity of Algorithm 2 is $O(n + m)$. □

Notice that the $O(n+m)$ time complexity of Algorithm 2 improves that of the corresponding algorithm in [28] (which represents the current best algorithm for enhancing QoP)by a factor

Algorithm 3 ReducedCostE(G, s)

1: Compute DFS tree $\mathcal{T}$ rooted at s. Relabel the nodes so that $D[v] = v$ for every node v. Initialize T^R and T^B to the empty tree. Initialize the queue $markedQ$ to empty. Every node is *unmarked* and *unvisited*.
2: Mark node s, insert it into $markedQ$ and add it onto T^B and T^R. Set the voltages $v^B(s)$ and $v^R(s)$ such that $v^B(s) > v^R(s) = 0$. Set $v_{max}(s) := 0$.
3: **while** (*markedQ* is NOT EMPTY) **do**
4: Dequeue a node u from *markedQ*.
5: **for** (each adjacent node w of u) **do**
6: **if** ((w is neither marked nor visited) **and** ((w, u) is a maximal back edge)) **then**
7: // add an ear to T^R and T^B.
8: Let v be the nearest ancestor (in $\mathcal{T}$) of node w on the current T^B and T^R.
9: **if** ($(v^B(u) > v^B(v))$) **then**
10: $p_s := u$; $p_t := v$;
11: **else**
12: $p_s := v$; $p_t := u$;
13: **end if**
14: $\pi(v, w)$ concatenated with the (w, u) forms a path (or a cycle in case $v = u$) connecting u and v. Let this path or cycle be $(p_s, x_1, \ldots, x_k, p_t)$. From v to w, mark each unmarked node on this path and insert it into the queue $markedQ$; add $p_s \to x_1 \to \cdots \to x_k$ to T^B; add $p_t \to x_k \to \cdots \to x_1$ to T^R.
15: Assign voltages s.t. $v^B(p_s) > v^B(x_1) > v^R(x_1) > \cdots > v^B(x_k) > v^R(x_k) > v_{max}(p_s)$.
16: Set $v_{max}(x_i) := v^R(x_i)$, $i = k, k-1, \ldots, 1$; $v_{max}(p_s) := v^B(x_1)$.
17: **else if** (node w is neither marked nor visited) **then**
18: Record w as visited.
19: **end if**
20: **end for**
21: **end while**

Algorithm 4 ReducedCostV(G, s)

1: Compute DFS tree $\mathcal{T}$ rooted at s. Relabel the nodes so that $D[v] = v$ for every node v. Sort the children of each node in nondecreasing order of lowpoint numbers. Initialize T^R and T^B to the empty tree. Initialize the queue $markedQ$ to empty. Every node is *unmarked*.
2: Mark node s, push it onto *markedQ*, and add it onto T^B and T^R. Set voltage $v^B(s) > 0$, set $v_{max}(s) := 0$.
3: **while** (*markedQ* is NOT EMPTY) **do**
4: Dequeue a marked node u from *markedQ*.
5: **for** (each unmarked child node w of u) **do**
6: // add an ear to T^R and T^B.
7: **while** (w is not marked) **do**
8: Let $t(w)$ be the tagged child of w;
9: **if** ($L[t(w)] < D[u]$ **or** $L[t(w)] = 1$) **then**
10: Mark node w, insert w into $markedQ$; $w := t(w)$;
11: **else**
12: Mark node w. insert w into $markedQ$;
13: **if** ($v^B(L[w]) > v^B(u)$) **then**
14: $p_s := L[w]$; $p_t := u$;
15: **else**
16: $p_s := u$; $p_t := L[w]$;
17: **end if**
18: $\pi(u, w)$ concatenated with $(w, L[w])$ forms a path or cycle (in case u is the root node) connecting u and $L[w]$; assume this path or cycle be $(p_s, x_1, \ldots x_k, p_t)$; add $p_s \to x_1 \to \ldots \to x_k$ to T^B; add $p_t \to x_k \to \ldots \to x_1$ to T^R;
19: Assign voltages s.t. $v^B(p_s) > v^B(x_1) > \ldots > v^B(x_k) > v_{max}(p_s)$; set $v_{max}(x_k) := v_{max}(p_s)$; $v_{max}(x_i) := v^B(x_{i+1})$, $i = k-1, \ldots, 1$; $v_{max}(p_s) := v^B(x_1)$;
20: **end if**
21: **end while**
22: **end for**
23: **end while**

of $O(n^2)$, and improves that of the algorithm in [11] (which does not consider QoP) by a factor of $O(n^3/(n+m))$. In Section VI we will see that, on all test cases conducted, Algorithm 2 runs significantly faster than other two aforementioned algorithms while exhibiting performance comparable to that of the corresponding algorithm in [28], confirming our theoretical analysis.

IV. Reducing Cost

In [28], Xue *et al.* presented $O(n^2(m+n))$ time algorithms for constructing a pair of single-link/single-node failure recovery trees with low total cost. In this section, we focus on the case of the problem where the cost of the trees are measured by the number of edges used, and present two $O(n+m)$ time algorithms for constructing a pair of recovery trees with low total cost. The algorithm for constructing a pair of single-link failure recovery trees is listed as Algorithm 3. The algorithm for constructing a pair of single-node failure recovery trees is listed as Algorithm 4.

Algorithm 3 is similar to Algorithm 2 in the following sense. For each dequeued node u, if it has an incoming *maximal back edge* (w, u), we add a path/cycle, which connects w's nearest ancestor on the current recovery trees with node u, onto T^B and T^R. Recall that in Algorithm 2 an outgoing *acceptable back edge* triggers such an operation.

Algorithm 3 and Algorithm 2 differ in the following important aspects. The goal of Algorithm 2 is to have more ears in the ear decomposition while the goal of Algorithm 3 is to have fewer ears in the ear decomposition. Therefore, Algorithm 2 adds an ear whenever an *acceptable back edge* is encountered, while Algorithm 3 adds an ear only when a *maximal back edge* is encountered. To enhance QoP performance, Algorithm 2 uses a stack $markedS$ to reverse the order of the nodes to be added into the queue $markedQ$. This heuristic is not used in Algorithm 3. Another difference is that in Algorithm 3, a node is marked if and only if it is on both T^R and T^B. This is not the case for Algorithm 2.

In Line 6 of Algorithm 3, we need to check whether (w, u) is a maximal back edge. This is not an $O(1)$ time operation. However, the total time required by all executions of Line 6 in the algorithm is $O(n+m)$. This can be accomplished in the following way. We can check whether (w, u) is a back edge in

$O(1)$ time. To check whether back edge (w, u) is a maximal back edge, we need to check the lowpoint number of each child node of node w: (w, u) is a maximal back edge if and only if for any child node x of w, node $L[x]$ is not on T^R and T^B ($L[x]$ is not marked), where $L[x]$ is the lowpoint number of node x. This requires a time proportional to the degree of node w. Since the sum of node degrees in a graph is twice the number of edges in the graph, the total time required by all executions of Line 6 in the algorithm is $O(n + m)$.

Using the above analysis and an argument similar to the one used in the proof of Theorem 3.1, we can show that the running time of Algorithm 3 is $O(n + m)$. Since we are using maximal back edges to add ears, and a maximal back edge is also an acceptable back edge, we can use an argument similar to that used in the proof of Theorem 3.1 to prove the following theorem.

Theorem 4.1: Assume that G is a 2-edge connected graph with n vertices and m edges. Algorithm 3 computes a pair of single-link failure recovery trees correctly. Furthermore, the worst-case running time of the algorithm is $O(m + n)$. □

Applying Algorithm 3 to the sample network shown in Fig. 1(a) (ignoring edge directions), we can construct the pair of red/blue trees T^R and T^B as shown in Fig. 1(c), where the ears added are (1, 2, 3, 4, 5, 1), (3, 8, 7, 6, 5), (7, 12, 13, 14, 7), and (8, 9, 10, 11, 8).

Computing a pair of single node failure recovery trees with low cost can also be accomplished in linear time. For this purpose, we need the concept *tagged child* of a node defined in the following.

Definition 4.1: A child node v of a node u is called the *tagged child* of u, denoted by $t(u)$, if for any other child w of u, we have either $L[v] < L[w]$ or $L[v] = L[w]$ but $D[v] < D[w]$. We may simply say that v is a tagged child, if the parent node u can be implied from the context. □

Notice that, starting from a node u, we can go to a child node v of u and then follow some tree path concatenated with a back edge to reach an ancestor of u if and only if $L[v] < u$. While we are searching for an ear starting from node u along a tree path to a node v, we can check the lowpoint number of the tagged child $t(v)$ to see whether we can go further down the tree at v: we can go further down if and only if $L[t(v)] < u$. The concept of tagged child enables us to design an efficient algorithm for constructing a pair of single-node failure recovery trees with low cost. Algorithm 4 is an $O(n + m)$ time algorithm for constructing a pair of single-node failure recovery trees with low cost.

Algorithm 4 is similar to Algorithm 3, but differs from it in the following aspects.

1) Algorithm 3 deals with link failure recovery, while Algorithm 4 deals with node failure recovery. In Algorithm 4, we do not allow adding a cycle onto the current T^B and T^R except the first ear.
2) Algorithm 3 checks incoming back edges (line 6) to trigger a process of adding an ear onto the current T^B and T^R (lines 7–16). Algorithm 4 checks tree edges (line 5) to trigger such a process (lines 6–21). The reason for the change is that we cannot guarantee the ear triggered by checking a back edge is a path. Following the tree edges, we can guarantee that the end nodes of each ear are different, except the first ear.
3) Algorithm 4 sorts the adjacency lists so that the child nodes with smaller lowpoint numbers appear to the front of the list (for child nodes with equal lowpoint numbers, we break ties using their DFS number). This ensures that the first child of a node is always its tagged child. In addition, this sorting (for all nodes in G) can be done in $O(n)$ time using bucket sort [5].

Lemma 4.1: During the execution of Algorithm 4, if a node u is on T^B and T^R, then all ancestors of u in the DFS tree $\mathcal{T}$ will also be on T^B and T^R. □

Proof: Initially, only the root node s is on the recovery trees. Therefore, this property is true. Every time new nodes are added to the recovery trees, we add a path or a cycle, where the path or cycle is formed by the tree path $\pi(u, w)$ contacted with a back edge from w to $L[w]$. Therefore, this property is kept true after each addition of a path or cycle. This proves the lemma. □

Theorem 4.2: Assume that G is a 2-vertex connected graph with n nodes and m edges. Then Algorithm 4 correctly constructs a pair of single-node failure recovery trees T^B and T^R in $O(m + n)$ time. □

Proof: We will prove the correctness first. Since the graph is 2-vertex connected, the root node s has exactly one child in the DFS tree $\mathcal{T}$ [5]. The algorithm first finds a relatively long cycle to add the first ear to T^B and T^R. The cycle is relatively long because we extend the tree path by following the tagged child as long as the lowpoint number of the tagged child is not bigger than the DFS number of the root node. This is the only cycle that will be added to T^R and T^B in the algorithm.

When node u is dequeued from the queue $markedQ$ in Line 4 of the algorithm, u is already on T^R and T^B. Let w be any child of u. It follows from the property of DFS trees of 2-connected graphs [5] that $L[w] < D[u]$. Therefore, we can find a tree path starting from u to w and continuously extending the path via the tagged child as long as the lowpoint number of the tagged child is smaller than $D[u]$ (ensuring that we can reach an ancestor of node u). Let this tree path be $u \rightarrow w \rightarrow \cdots \rightarrow v$. This tree path, concatenated with the maximal back edge $(v, L[v])$, forms a path from node u to $L[v]$ via nodes on the tree path from w to v. From the way we extend the tree path, we know that $L[v] < D[u]$. Therefore, $L[v]$ (the node whose DFS number is equal to $L[v]$) is an ancestor of u, which (recall Lemma 4.1) implies that $L[v]$ is also on T^B and T^R. Using the voltage rules of [11], we can add the nodes on this path to T^B and T^R. Since every child node of u can be added to T^B and T^R in this way, the algorithm will add every node onto T^B and T^R. This proves the correctness of the algorithm.

Next we will prove the linear time complexity of the algorithm. Line 1 takes $O(n + m)$ time to perform DFS [23] and to relabel the nodes so that $D[v] = v$. It also takes an additional $O(n)$ time to sort the children of each node (in nondecreasing lowpoint numbers) using bucket sort [5]. Line 2 takes $O(1)$ time. In the following, we will prove that the total time required by Lines 3 through 23 is $O(n)$.

We first notice that each node will be marked, inserted into the queue $markedQ$, and then dequeued from the queue *exactly once*. Each time a marked node u is dequeued from $markedQ$, we will add an ear (to T^R and T^B) corresponding to each unmarked child w of u. The ear corresponding to the unmarked child w of u is constructed by following the tree edge corre-

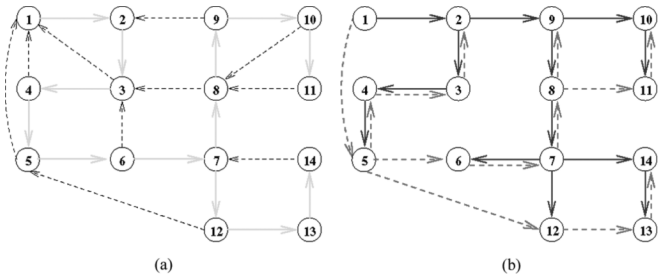


Fig. 2. Sample 2-vertex connected network and corresponding red/blue trees. (a) DFS tree of a 2-vertex connected graph. (b) Red/blue trees by Reduced-CostV.

sponding to a tagged child as long as the lowpoint number of the tagged child is equal to 1 or smaller than $D[u]$. Due to the pre-sorting in Line 1, the first child of each node is guaranteed to be a tagged child of that node. Therefore, the time required to find an ear is proportional to the number of nodes in the ear that are not already on T^R and T^B. While we are extending this ear, all the nodes on this ear will be marked and inserted into the queue $markedQ$ (see Lines 10 and 12). The operations in Lines 13 through 19 also require time proportional to the number of nodes in the ear that are not already on T^R and T^B. This proves that the time required to add an ear onto T^B and T^R is proportional to the number of nodes in the ear that are not already on T^R and T^B. Since the final T^R and T^B each has exactly n nodes, the total time required by Lines 3 through 23 is $O(n)$. This proves that the time complexity of Algorithm 4 is $O(n+m)$. □

We illustrate Algorithm 4 with the sample network whose DFS tree is shown in Fig. 2(a). In Line 1, T^B, T^R, and the queue $markedQ$ are all initialized to empty. We also sort the children of each node in nondecreasing order of their lowpoint numbers. In Line 2, the root node 1 is marked and inserted onto T^B and T^R, as well as the queue $markedQ$. We also assign voltages at node 1 such that $v^B(s) > 0$, and $v_{max}(s) = 0$. In Line 4, we dequeue node $u = 1$. Node u has only one child $w = 2$, which is unmarked. Since w's tagged child $t(w) = 3$ has lowpoint number $L[t(w)] = 1$, the condition in Line 9 is true. Therefore, we mark node $w = 2$, insert it into $markedQ$, and update w so that $w = t(w) = 3$. Since w's tagged child $t(w) = 4$ has lowpoint number $L[t(w)] = 1$, the condition in Line 9 is true. Therefore, we mark node $w = 3$, insert it into $markedQ$, and update w so that $w = t(w) = 4$. Similarly, node 4 is marked and inserted into $markedQ$. Next control returns to Line 7 with $w = 5$. Since w's tagged child $t(w) = 6$ has lowpoint number $L[6] = 2$, the condition in Line 9 is not true. Therefore, we mark node 5, insert it into $markedQ$, and find the ear $1 \to 2 \to 3 \to 4 \to 5 \to 1$. We add the links (1, 2), (2, 3), (3, 4) and (4, 5) onto T^B, and the links (1, 5), (5, 4), (4, 3) and (3, 2) onto T^R. We assign the voltages for all nodes according to the rules of [11] such that $v^B(1) > v^B(2) > v^B(3) > v^B(4) > v^B(5) > v_{\max}(1)$. Now $v_{\max}(5)$ is set to $v_{\max}(1)$, $v_{\max}(4)$ is set to $v^B(5)$, $v_{\max}(3)$ is updated to $v^B(4)$, $v_{\max}(2)$ is set to $v^B(3)$, $v_{\max}(1)$ is updated to $v^B(2)$, in that order. Here we finish the check of node 1. At this moment, the content of $markedQ$ is 2, 3, 4, 5. Following the algorithm, we will add ears onto T^B and T^R in the following order: $2 \to 9 \to 8 \to 7 \to 6 \to 5, 7 \to 12 \to 5, 9 \to 10 \to 11 \to 8, 7 \to 14 \to 13 \to 12$.

Algorithm 5 MaxBandE(G, s)

1: Sort all edge bandwidth values in non-decreasing order.
2: Set b_L to the smallest bandwidth value of G; Set b_U to 1 plus the maximum bandwidth value of G; DONE := 0;
3: **while** (**not** DONE) **do**
4: Let b_M be the median of all bandwidths smaller than b_U, but not smaller than b_L.
5: Let G' be the subgraph of G obtained by removing all edges whose bandwidths are smaller than b_M.
6: **if** (G' is 2-edge connected) **then**
7: $b_L := b_M$;
8: **else**
9: $b_U := b_M$;
10: **end if**
11: **if** (there is no edge whose bandwidth is smaller than b_U but not smaller than b_L) **then**
12: DONE := 1; $b_{BN} := b_L$;
13: **end if**
14: **end while**
15: Let G' be the subgraph of G containing only edges with bandwidths at least b_{BN}; Apply ReducedCostE or EnhancedQoP on G' to construct a pair of single-link failure recovery trees.

This leads to the pair of red/blue trees T^R and T^B shown in Fig. 2(b).

Notice that the $O(n+m)$ time complexities of Algorithms 3 and 4 improve those of the corresponding algorithms in [28] (which represents the current best algorithm for constructing a pair of red/blue trees with low total cost) by a factor of $O(n^2)$. In Section VI we will see that, on all test cases conducted, Algorithms 3 and 4 run significantly faster than their corresponding counterparts in [28] and [11].

V. Maximizing Bottleneck Bandwidth

In [28], Xue *et al.* presented $O(nm)$ time algorithms to construct a pair of single failure recovery trees with maximum bottleneck bandwidth. In this section, we present $O(m \log n)$ time algorithms to construct a pair of single failure recovery trees with maximum bottleneck bandwidth. The key idea here is to find the maximum bottleneck bandwidth and then apply one of our linear time algorithms to construct the recovery trees of the subgraph obtained by removing the links whose bandwidths are smaller than the maximum bottleneck bandwidth. Therefore, it is worth noting that our $O(m \log n)$ time algorithms can construct a pair recovery trees with maximum bottleneck bandwidth as well as enhanced QoP or QoS performance, which have not been addressed by previous works [11], [28].

An $O(m \log n)$ time algorithm for constructing the maximum-bandwidth single-link recovery trees is presented as Algorithm 5. To simplify the description and analysis of the algorithm, we assume that no two edges have the same bandwidth. This does not result in loss of generality, as the IDs of edges can be used to distinguish the edges with equal bandwidth.

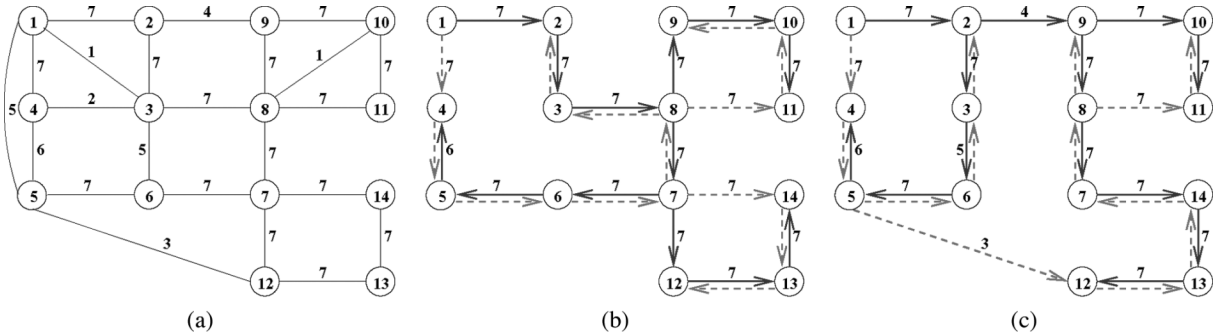


Fig. 3. Red/blue trees for link/node recovery with maximum bottleneck bandwidth. (a) Sample network. (b) Red/blue trees for edge failure recovery. (c) Red/blue trees by node failure recovery.

Algorithm 6 MaxBandV(G, s)

1: Sort all edge bandwidth values in non-decreasing order.
2: Set b_L to the smallest bandwidth value of G; Set b_U to 1 plus the maximum bandwidth value of G; DONE := 0;
3: **while** (**not** DONE) **do**
4: Let b_M be the median of all bandwidths smaller than b_U, but not smaller than b_L.
5: Let G' be the subgraph of G obtained by removing all edges whose bandwidths are smaller than b_M.
6: **if** (G' is 2-vertex connected) **then**
7: $b_L := b_M$;
8: **else**
9: $b_U := b_M$;
10: **end if**
11: **if** (there is no edge whose bandwidth is smaller than b_U but not smaller than b_L) **then**
12: DONE := 1; $b_{BN} := b_L$;
13: **end if**
14: **end while**
15: Let G' be the subgraph of G containing only edges with bandwidths at least b_L; Apply Algorithm ReducedCostV on G' to construct a pair of single-node failure recovery trees.

Theorem 5.1: Assume that G is a 2-edge connected graph with n nodes and m edges. Then Algorithm 5 correctly constructs a pair of single-link failure recovery trees with maximal bottleneck bandwidth as well as additional enhanced QoP or QoS performance in $O(m \log n)$ time. □

Proof: Lines 1–14 of the algorithm compute the maximum bottleneck bandwidth b_{min} such that the subgraph G' of G consisting of all edges with bandwidth values at least b_{BN} is 2-edge connected. In Line 15, we apply ReducedCostE or EnhancedQoP to compute a pair of single-link failure recovery trees in G'. This proves the correctness of the algorithm.

Line 1 takes $O(m \log m) = O(m \log n)$ time. Line 2 takes $O(1)$ time. The WHILE-loop in lines 3–14 performs $O(\log m) = O(\log n)$ iterations, where each iteration requires $O(m)$ time [23]. Therefore, the time complexity of lines 1–14 is $O(m \log n)$ time. Since Line 14 takes $O(n + m)$ time, the total time complexity of Algorithm 5 is $O(m \log n)$. □

Let us illustrate Algorithm 5 with the sample network shown in Fig. 3(a), where the label of an edge indicates the bandwidth of that edge. First, the algorithm computes the maximum bottleneck bandwidth (for 2-edge connectivity) to be 6. In Line 15, we construct a 2-edge connected network with maximum bottleneck bandwidth 6. Next, we use algorithm ReducedCostE to find a pair of single-link failure recovery trees with low cost, based on the subgraph with links of bandwidths of 6 or more. The resulting T^B and T^R are shown in Fig. 3(b), where solid edges are for T^B, and dashed edges are for T^R.

Similarly, we can construct a pair of maximum bottleneck bandwidth single-node recovery trees in $O(m \log n)$ time. The difference is that here we need to find the maximum bandwidth value B such that the subgraph of G induced by the edges with bandwidth at least B is 2-vertex connected. Algorithm 6 shows how to construct such a pair of recovery trees.

Theorem 5.2: Assume that G is a 2-vertex graph with n nodes and m edges. Then Algorithm 6 correctly constructs a pair of single-node failure recovery trees with maximal bottleneck bandwidth as well as additional enhanced QoP or QoS performance in $O(m \log n)$ time. □

For the sample network shown in Fig. 3(a), we find out that the maximum bottleneck bandwidth for 2-vertex connectivity is 3. Applying algorithm ReducedCostV, we find a pair of single-node failure recovery trees as shown in Fig. 3(c).

VI. Numerical Results

In this section, we present numerical results to confirm our theoretical analysis of the algorithms. Our implementation was based on LEDA [13], a C++ class library. All tests were performed on a 1.0 GHz Linux PC with 1 G bytes of memory. We have implemented Algorithm 2 of this paper (denoted by QoP in this section), Algorithm 3 of this paper (denoted by CostE), and Algorithm 4 of this paper (denoted by CostV). We did not implement Algorithms 5 and 6 of this paper, as *they have been proved to be optimal algorithms* for the bottleneck bandwidth maximization problems. For purposes of comparison, we have also implemented the algorithm of [11] for single-link failure recovery (denoted by MFBG-E), the algorithm of [11] for single-node failure recovery (denoted by MFBG-V), the algorithm of [28] for enhancing QoP (denoted by XCT-QoP), the algorithm of [28] for low cost single-link failure recovery (denoted by XCT-CostE), and the algorithm of [28] for low cost single-node failure recovery (denoted by XCT-CostV). These algorithms are evaluated by the following performance metrics: (1) quality

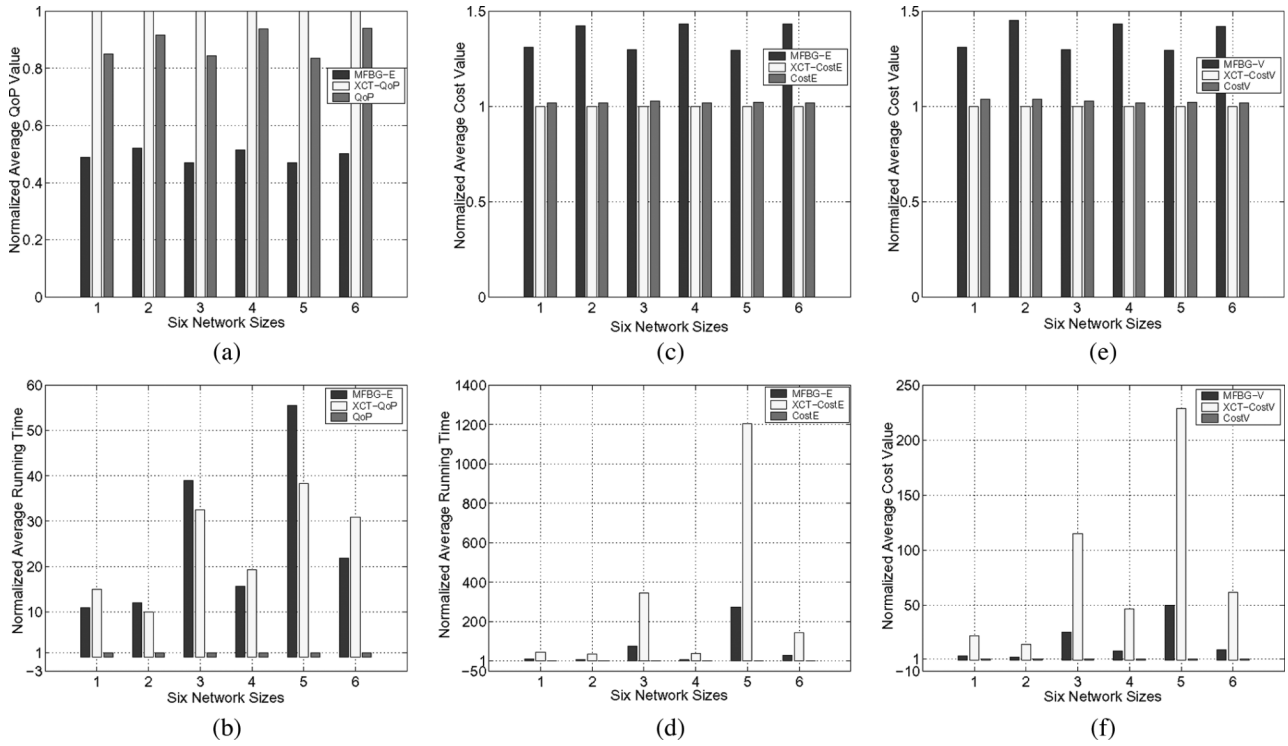


Fig. 4. (a), (c), (e) Performance and (b), (d), (f) running time comparison of various algorithms on the network topologies used in [28]. (a) Normalized QoP values. (b) Running time for QoP algorithms. (c) Normalized total cost values. (d) Running time for CostE algorithms. (e) Normalized total cost values. (f) Running time for CostV algorithms.

of protection and running time for single-link failure recovery trees, (2) total cost and running time for single-link failure recovery trees, and (3) total cost and running time for single-node failure recovery trees.

We used two kinds of test data for the evaluation. We first used the randomly generated graphs used in [28] to ensure fair comparison with the current best algorithms for enhancing quality of protection and reducing total cost [28]. These graphs have three different values of the number of nodes n: 50, 100 and 200. For each value of n, the graphs in [28] have two different numbers of edges m: $m = 3n$ and $m = n \log n$. This leads to the following six network sizes (given by $n \times m$): 50×150, 50×282, 100×300, 100×664, 200×600, 200×1529. In [28], the authors randomly generated, for each of the six network sizes, 100 2-connected graphs. For our first comparison, we have used those 600 network topologies [28]. Fig. 4 shows the average over 100 runs for each of the performance metrics. Each of the six sub-figures in this figure contains six groups of bars where the first group of bars (with x axis label 1) shows the average results for the 100 topologies with size 50×150, the second group of bars (with x axis label 2) shows the average results for the 100 topologies with size 50×282, the third group of bars (with x axis label 3) shows the average results for the 100 topologies with size 100×300, and so on.

Fig. 4(a) and (b) compares the QoP performances and running times of the algorithms MFBG-E, XCT-QoP and QoP on the randomly generated graphs. For better viewing, we normalized the QoP values of all algorithms by the corresponding values produced by XCT-QoP in Fig. 4(a). Similarly, we normalized the running times of all algorithms by the corresponding values produced by QoP in Fig. 4(b). From Fig. 4(a), we observe that algorithm XCT-QoP has the best QoP performance for all six different network sizes. The QoP performances of algorithm QoP are around 90% of those of algorithm XCT-QoP. From Fig. 4(b), we observe that on these randomly generated graphs, algorithm QoP runs much faster than both XCT-QoP and MFBG-E.

Fig. 4(c) and (d) compare the total cost values and running times of the algorithms MFBG-E, XCT-CostE and CostE on these randomly generated graphs. We normalized the total cost values of all algorithms by the corresponding values produced by XCT-CostE, and normalized the running times of all algorithms by the corresponding values produced by CostE. From Fig. 4(c), we observe that algorithm XCT-CostE has the lowest total cost for all six different network sizes. The total costs of algorithm CostE are within 1% of those of algorithm XCT-CostE. From Fig. 4(d), we observe that algorithm QoP runs much faster than both XCT-CostE and MFBG-E. Fig. 4(e) and (f) compare the total cost values and running times of the algorithms MFBG-V, XCT-CostV and CostV on these randomly generated graphs. We have similar observations as in the case of Fig. 4(c) and (d). The results shown in Fig. 4 confirm that, for the test cases used in [28], the algorithms presented in this paper run much faster than the corresponding current best algorithms [28] while having performances comparable to those of the current best.

Next, we evaluate the algorithms on network topologies generated by BRITE, a well-known network topology generator

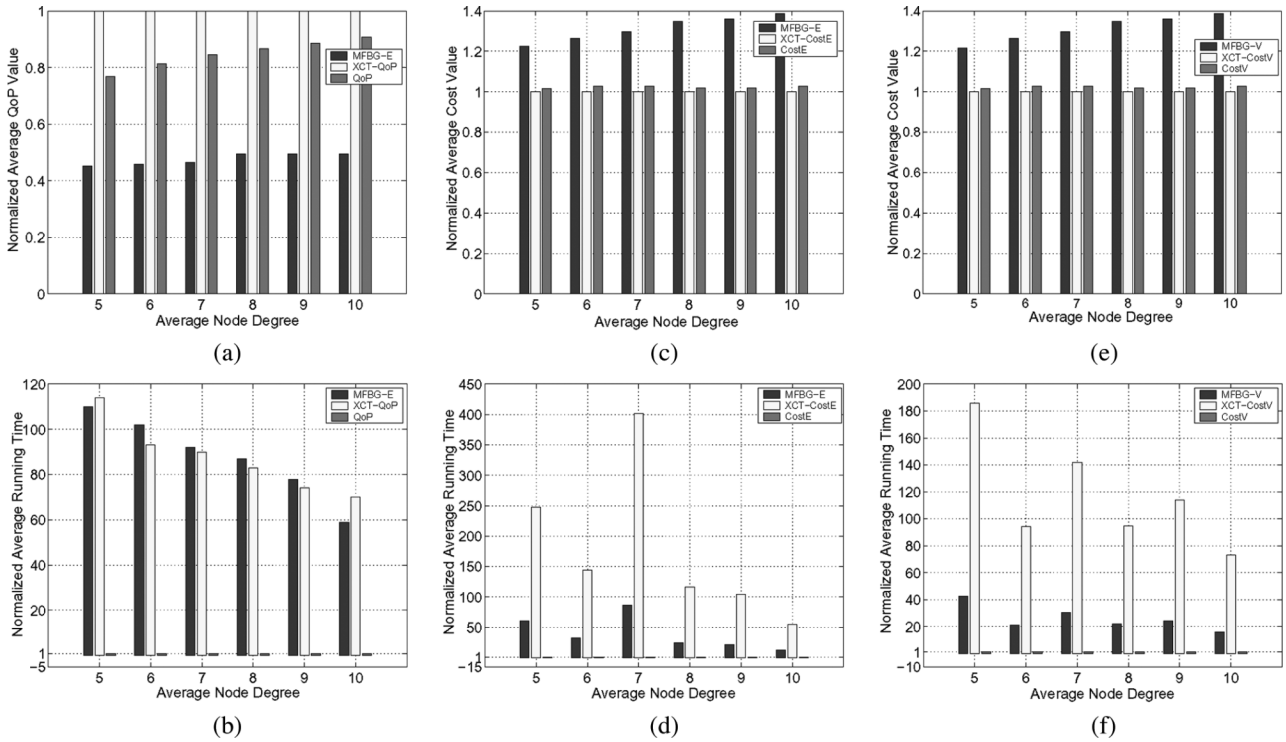


Fig. 5. Performance and running time comparison of various algorithms on network topologies generated by BRITE with $n = 100$ and $(2m/n) \in [5, 10]$. (a) Normalized QoP values; (b) running time for QoP algorithms; (c) normalized total cost values; (d) running time for CostE algorithms; (e) normalized total cost values; (f) running time for CostV algorithms.

[2]. BRITE provides several well-known models (including the Waxman model [24]) for generating reasonable network topologies. We adopted the Waxman model (with default parameters provided by BRITE) to generate random networks.

We first evaluate the algorithms with n fixed at 100 and let $2m/n$ take the values 5, 6, 7, 8, 9, 10. For each of these six network sizes, we randomly generated 100 network topologies using BRITE. Fig. 5 shows the average over 100 runs for each of the performance metrics. Each of the six subfigures in this figure contains six groups of bars where the first group of bars (with x axis label 5) shows the average results for the 100 topologies with average node degree set to 5, the second group of bars (with x axis label 6) shows the average results for the 100 topologies with average node degree set to 6, and so on.

As in the case of Fig. 4, we have shown the normalized QoP and total cost values, as well as the normalized running times. Here we observe that the results are very similar to those presented in Fig. 4: Algorithm QoP is significantly faster than MFBG-E and XCT-QoP, while having QoP performance comparable to that of XCT-QoP; Algorithm CostE is significantly faster than MFBG-E and XCT-CostE, while having average total cost comparable to that of XCT-CostE; Algorithm CostV is significantly faster than MFBG-V and XCT-CostV, while having average total cost comparable to that of XCT-CostV.

Finally, we evaluate the performances and running times of the algorithms on network topologies where the number of nodes n takes on the following values: 100, 200, 300, 400, 500, 600. For each value of n, we randomly generated 100 network topologies using BRITE, with the average node degree set to $2 \log n$ (Similar results were obtained with the average node degree set to 6, and hence are not shown here). Fig. 6 shows the average over 100 runs for each of the performance metrics.

Fig. 6(a) and (b) compare the QoP performances and running times of the algorithms MFBG-E, XCT-QoP and QoP on these randomly generated graphs. As in the cases of Fig. 4(a), (c), and (e), we have normalized the QoP and total cost values. Unlike in the cases of Fig. 4(b), (d), and (f), we did not normalize the running times in Fig. 6(b), (d), and (f). From Fig. 6(a) and 6(b), we observe that the QoP performance of algorithm QoP is around 90% of that of algorithm XCT-QoP, while our algorithm runs significantly faster than XCT-QoP. Similar observations can be obtained from Fig. 6(c) and (d) for low cost single-link failure recovery trees, and from Fig. 6(e) and (f) for low cost single-node failure recovery trees. In addition, we observe from Fig. 6(b), (d), and (e) that algorithms QoP, CostE and CostV all have running times almost linear in n, while their corresponding counterparts all have running times that grow much faster with n. This again confirms our theoretical analysis.

To study the scalability of the algorithms, we evaluated the algorithms on larger graphs generated by BRITE, with n taking on additional values 1000, 1500, 2000, 2500, 3000, 3500, 4000. Fig. 7 shows the running times of various algorithms on these test cases. For better viewing, we used a cutoff point for the algorithms MFBG-E, MFBG-V, XCT-QoP, XCT-CostE, and XCT-CostV when their corresponding running times become 10 times the running times of the corresponding algorithms reported in this paper. From this figure, we can clearly observe that the algorithms presented in this paper are significantly faster than the corresponding algorithms presented in [28] and and [11].

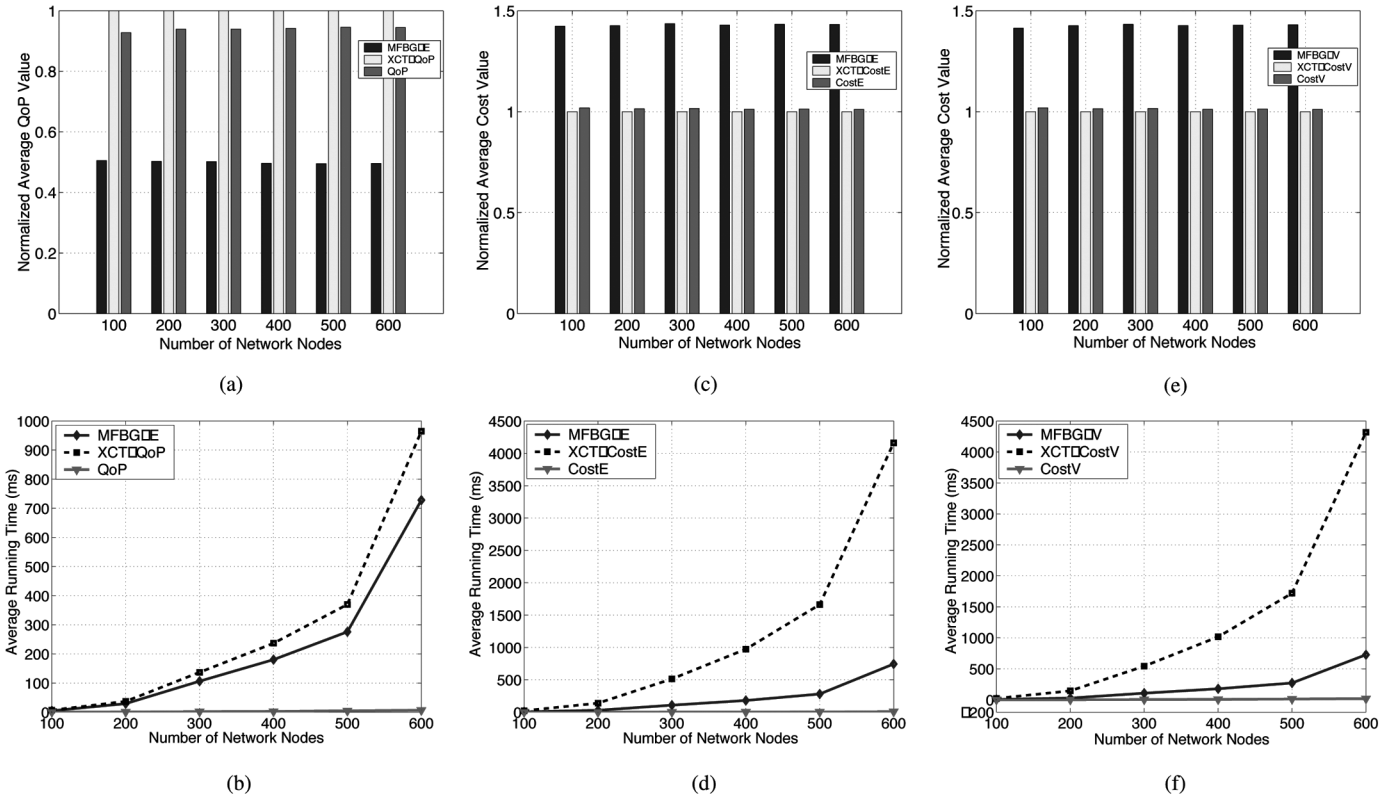


Fig. 6. Performance and running time comparison of various algorithms on network topologies generated by BRITE with $n \in [100, 600]$ and $m = n \log n$. (a) Normalized QoP values. (b) Running time for QoP algorithms. (c) Normalized total cost values. (d) Running time for CostE algorithms. (e) Normalized total cost values. (f) Running time for CostV algorithms.

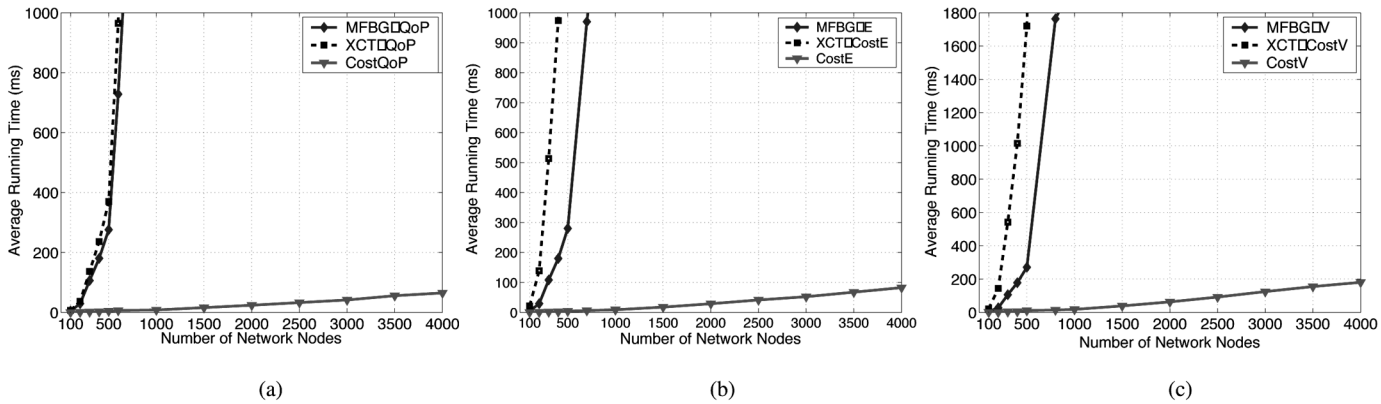


Fig. 7. Performance and running time comparison of various algorithms on network topologies generated by BRITE with $n \in [100, 4000]$ and $m = n \log n$. (a) Running time for QoP algorithm. (b) Running time for CostE algorithms. (c) Running time for CostV algorithms.

VII. Conclusion

In this paper, we have presented faster algorithms for constructing a pair of single-link (single-node) failure recovery trees with enhanced quality of protection, reduced cost, or maximum bottleneck bandwidth. For enhancement of quality of protection, our new algorithm has a linear time complexity and exhibits a performance comparable to that of the previously best $O(n^2(n+m))$ time algorithm. The same can be said for the reduction of cost. For bottleneck bandwidth maximization, our new algorithms improve the previously best in terms of worst-case running time, from $O(mn)$ to $O(m \log n)$. We wish to emphasize that the use of the concept of ear decomposition [25] combined with the concept of Depth First Tree [23] has made the design of these improved algorithms possible.

In the literature, many schemes for recovery from single failure have been proposed. Many of those schemes, although originally designed for guaranteed single failure recovery, can also provide recovery from multiple failures, provided that the failures satisfy certain patterns. We intend to extend the redundant tree approach to provide guaranteed recovery from multiple failures [3], [4], [17], [18]. Also, it would be of interest

to apply the techniques presented in this paper to the domain of wireless networks.

ACKNOWLEDGMENT

We thank the associate editor and the anonymous reviewers whose comments on an earlier version of this paper have helped to significantly improve the presentation of this paper.

REFERENCES

[1] V. Anand and C. Qiao, "Dynamic establishment of protection paths in WDM networks, part I," in *Proc. IEEE ICCCN*, 2000, pp. 198–204.
[2] BRITE [Online]. Available: http://www.cs.bu.edu/brite/
[3] H. Choi, S. Subramaniam, and H.-A. Choi, "On double-link failure recovery in WDM optical networks," in *Proc. IEEE INFOCOM*, 2002, pp. 808–816.
[4] H. Choi, S. Subramaniam, and H.-A. Choi, "Loopback recovery from double-link failures in optical mesh networks," *IEEE/ACM Trans. Netw.*, vol. 12, no. 6, pp. 1119–1130, Dec. 2004.
[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA: MIT Press, 2001.
[6] G. Ellinas, A. G. Hailemariam, and T. E. Stern, "Protection cycles in mesh WDM networks," *IEEE J. Sel. Areas Commun.*, vol. 18, no. 10, pp. 1924–1937, Oct. 2000.
[7] W. D. Grover and D. Stamatelakis, "Cycle-oriented distributed preconfiguration: Ring-like speed with mesh-like capacity for self-planning network restoration," in *Proc. IEEE ICC*, 1998, pp. 537–543.
[8] A. Itai and M. Rodeh, "The multi-tree approach to reliablity in distributed networks," *Inf. Computat.*, vol. 79, pp. 43–59, 1988.
[9] A. Jukan, A. Monitzer, G. de Marchis, and R. Sabella, Eds., "Restoration methods for multi-service optical networks," in *Optical Networks: Design and Modelling*. Boston, MA: Kluwer, 1999, pp. 3–12.
[10] M. Médard, S. G. Finn, and R. A. Barry, "A novel approach to automatic protection switching using trees," in *Proc. IEEE ICC*, 1997, pp. 272–276.
[11] M. Médard, S. G. Finn, R. A. Barry, and R. G. Gallager, "Redundant trees for preplanned recovery in arbitrary vertex-redundant or edge-redundant graphs," *IEEE/ACM Trans. Netw.*, vol. 7, no. 5, pp. 641–652, Oct. 1999.
[12] M. Médard, R. A. Barry, S. G. Finn, W. He, and S. S. Lumetta, "Generalized loop-back recovery in optical mesh networks," *IEEE/ACM Trans. Netw.*, vol. 10, no. 1, pp. 153–164, Feb. 2002.
[13] K. Mehlhorn and S. Nher, *LEDA*. Cambridge, U.K.: Cambridge Univ. Press, 1999.
[14] G. Mohan, C. S. R. Murthy, and A. K. Somani, "Efficient algorithms for routing dependable connections in WDM optical networks," *IEEE/ACM Trans. Netw.*, vol. 9, no. 5, pp. 553–566, Oct. 2001.
[15] B. Mukherjee, *Optical Communication Networks*. New York: McGraw Hill, 1997.
[16] B. Mukherjee, "WDM optical networks: Progress and challenges," *IEEE J. Sel. Areas Commun.*, vol. 18, no. 10, pp. 1810–1824, Oct. 2000.
[17] I. Ouveysi and A. Wirth, "On design of a survivable network architecture for dynamic routing: Optimal solution strategy and an efficient heuristic," *Eur. J. Oper. Res.*, vol. 117, pp. 30–44, 1999.
[18] I. Ouveysi and A. Wirth, "Minimal complexity heuristics for robust network architecture for dynamic routing," *J. Oper. Res. Soc.*, vol. 50, pp. 262–267, 1999.
[19] C. Qiao and D. Xu, "Distributed partial information management (DPIM) schemes for survivable networks, part I," in *Proc. IEEE INFOCOM*, 2002, pp. 302–311.
[20] S. Ramamurthy and B. Mukherjee, "Survivable WDM mesh networks. Part I—Protection," in *Proc. IEEE INFOCOM*, 1999, pp. 744–751.
[21] S. Ramamurthy and B. Mukherjee, "Survivable WDM mesh networks. Part II—Restoration," in *Proc. IEEE ICC*, 1999, pp. 2023–2030.
[22] L. Sahasrabuddhe, S. Ramamurthy, and B. Mukherjee, "Fault management in IP-over-WDM networks: WDM protection versus IP restoration," *IEEE J. Sel. Areas Commun.*, vol. 20, no. 1, pp. 21–33, Jan. 2002.
[23] R. E. Tarjan, "Depth first search and linear graph algorithms," *SIAM J. Computing*, vol. 1, pp. 146–160, 1972.
[24] B. M. Waxman, "Routing of multipoint connections," *IEEE J. Sel. Areas Commun.*, vol. 6, no. 9, pp. 1617–1622, Dec. 1988.
[25] D. B. West, *Introduction to Graph Theory*. Upper Saddle River, NJ: Prentice-Hall, 2001.
[26] T. H. Wu and W. I. Way, "A novel passive protected SONET bidirectional self-healing ring architecture," *IEEE J. Lightw. Technol.*, vol. 10, no. 9, pp. 1314–1322, Sep. 1992.
[27] G. Xue, L. Chen, and K. Thulasiraman, "QoS issues in redundant trees for protection in vertex-redundant or edge-redundant graphs," in *Proc. IEEE ICC*, 2002, pp. 2766–2770.
[28] G. Xue, L. Chen, and K. Thulasiraman, "Quality of service and quality protection issues in preplanned recovery schemes using redundant trees," *IEEE J. Sel. Areas Commun.*, ser. Optical Communications and Networking series, vol. 21, no. 8, pp. 1332–1345, Oct. 2003.

Weiyi Zhang (M'08) received the Ph.D. degree in computer science from Arizona State University, Tempe, in 2007.

He is an Assistant Professor with the Department of Computer Science at North Dakota State University, Fargo. His research interests include reliable communication in networking, protection and restoration in WDM networks, QoS provisioning in communication networks, and wireless sensor networks.

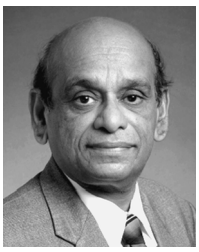
Guoliang (Larry) Xue (SM'99/ACM'93) received the B.S. degree in mathematics and the M.S. degree in operations research from Qufu Teachers University, Qufu, China, in 1981 and 1984, respectively, and the Ph.D. degree in computer science from the University of Minnesota, Minneapolis, in 1991.

He is a Full Professor with the Department of Computer Science and Engineering, Arizona State University, Tempe. He has held previous positions with Qufu Teachers University (Lecturer, 1984–1987), the Army High Performance Computing Research Center (Postdoctoral Research Fellow, 1991–1993), and the University of Vermont (Assistant Professor, 1993–1999; Associate Professor, 1999–2001). His research interests include efficient algorithms for optimization problems in networking, with applications to survivability, security, privacy, and energy efficiency issues in networks ranging from WDM optical networks to wireless ad hoc and sensor networks. He has published over 150 papers in these areas, including many papers in journals such as the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS, the IEEE TRANSACTIONS ON COMMUNICATIONS, the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY, the IEEE/ACM TRANSACTIONS ON NETWORKING, *SIAM Journal on Computing*, *SIAM Journal on Discrete Mathematics*, *SIAM Journal on Optimization*, and conferences such as ACM MobiHoc, ACM/SIAM SODA, and IEEE INFOCOM. His research has been continuously supported by federal agencies including the National Science Foundation (NSF) and the Army Research Office (ARO).

Dr. Xue was the recipient of the Graduate School Doctoral Dissertation Fellowship from the University of Minnesota in 1990, a Third Prize from the Ministry of Education of P.R. China in 1991, an NSF Research Initiation Award in 1994, and an NSF-ITR Award in 2003. He is an Editor of *Computer Networks* (COMNET), an Editor of *IEEE Network*, and the *Journal of Global Optimization*. He has served on the executive/program committees of many IEEE conferences, including INFOCOM, Secon, Icc, Globecom and QShine. He served as a TPC co-chair of IEEE IPCCC in 2003, a TPC co-chair of IEEE HPSR in 2004, the General Chair of IEEE IPCCC in 2005, a TPC co-chair of IEEE Globecom'2006 Symposium on Wireless Ad Hoc and Sensor Networks, a TPC co-chair of IEEE ICC'2007 Symposium on Wireless Ad Hoc and Sensor Networks, and a TPC co-chair of QShine in 2007. He also serves on many NSF grant panels and is a reviewer for ARO.

Jian Tang (M'08) received the Ph.D. degree in computer science from Arizona State University, Tempe, in 2006.

He is an Assistant Professor with the Department of Computer Science, Montana State University, Bozeman. His research interests are in the area of networking, with an emphasis on routing, scheduling, cross-layer design, and QoS provisioning in communication networks.



Krishnaiyan Thulasiraman (F'90/ACM'95) received the B.S. degree and M.S. degree in electrical engineering from the University of Madras, India, in 1963 and 1965, respectively, and the Ph.D. degree in electrical engineering from the Indian Institute of Technology (IIT), Madras, in 1968.

He holds the Hitachi Chair and is a Professor with the School of Computer Science, University of Oklahoma, Norman, which he joined in 1994. Prior to joining the University of Oklahoma, he was a Professor (1981–1994) and Chair (1993–1994) of the Electrical and Computer Engineering Department, Concordia University, Montreal, QC, Canada. He was on the faculty in the Electrical Engineering and Computer Science Departments of IIT Madras during 1965–1981. His research interests have been in graph theory, combinatorial optimization, algorithms and applications in a variety of areas in computer science and electrical engineering: electrical networks, VLSI physical design, systems-level testing, communication protocol testing, parallel/distributed computing, telecommunication network planning, fault tolerance in optical networks, and interconnection networks. He has published more than 100 papers in archival journals, coauthored with M. N. S. Swamy two text books, *Graphs, Networks, and Algorithms* (Wiley Inter–Science, 1981) and *Graphs: Theory and Algorithms* (Wiley Inter-Science, 1992), and authored two chapters in the *Handbook of Circuits and Filters* (CRC and IEEE, 1995) and a chapter on "Graphs and Vector Spaces" for the handbook *Graph Theory and Applications* (CRC, 2003). He has held visiting positions with the Tokyo Institute of Technology, the University of Karlsruhe, the University of Illinois at Urbana-Champaign, and Chuo University, Tokyo.

Dr. Thulasiraman was the recipient of several awards and honors: 2006 IEEE Circuits and Systems Society Technical Achievement Award, Endowed Gopalakrishnan Chair Professorship in Computer Science at IIT Madras (summer 2005), elected member of the European Academy of Sciences (2002), IEEE Circuits and Systems (CAS) Society Golden Jubilee Medal (1999), Fellow of the IEEE (1990) and Senior Research Fellowship of the Japan Society for Promotion of Science (1988). He has been Vice President (Administration) of the IEEE CAS Society (1998, 1999), Technical Program Chair of ISCAS (1993, 1999), Deputy Editor-in-Chief of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS I–REGULAR PAPERS (2004–2005), Co-Guest Editor of a special issue on "Computational Graph Theory: Algorithms and Applications" (IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS, March 1988), Associate Editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS (1989–91, 1999–2001), and Founding Regional Editor of the *Journal of Circuits, Systems, and Computers* and an editor of the *AKCE International Journal of Graphs and Combinatorics*. Recently, he founded the Technical Committee on "Graph theory and Computing" of the IEEE CAS Society.