Linear Time Construction of Redundant Trees for Recovery Schemes Enhancing QoP and QoS

Weiyi Zhang, Guoliang Xue, Jian Tang
Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-8809
{weiyi.zhang, xue, jian.tang}@asu.edu

Krishnaiyan Thulasiraman
School of Computer Science
University of Oklahoma
Norman, OK 73019
thulasi@ou.edu

***Abstract*— Medard, Finn, Barry and Gallager proposed an elegant recovery scheme (known as the MFBG scheme) using redundant trees. Xue, Chen and Thulasiraman extended the MFBG scheme and introduced the concept of quality of protection (QoP) as a metric of multifailure recovery capabilities for single failure recovery schemes. In this paper, we present three linear time algorithms for constructing redundant trees for single link failure recovery in 2-edge connected graphs and for single node failure recovery in 2-connected graphs.**

Our first algorithm aims at high QoP for single link recovery schemes in 2-edge connected graphs. The previous best algorithm has a running time of $O(n^2(m+n))$, where n and m are the number of nodes and links in the network. Our algorithm has a running time of $O(m+n)$, with comparable performance. Our second algorithm aims at high QoS for single link recovery schemes in 2-edge connected graphs. Our algorithm improves the previous best algorithm with $O(n^2(m+n))$ time complexity to $O(m+n)$ time complexity with comparable performance. Our third algorithm aims at high QoS for single node recovery schemes in 2-connected graphs. Again, our algorithm improves the previous best algorithm with $O(n^2(m+n))$ time complexity to $O(m+n)$ time complexity with comparable performance.

Simulation results show that our new algorithms outperform previously known linear time algorithms significantly in terms of QoP or QoS, and outperform other known algorithms in terms of running time, with comparable QoP of QoS performance.

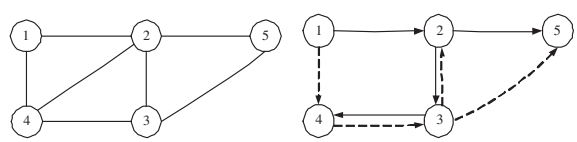
Keywords: Protection and restoration, redundant trees, blue/red trees, quality of protection, quality of service, linear time algorithms.

I. Introduction

Protection and restoration in high-speed networks are important issues that have been studied extensively [1], [6], [8], [13], [16], [17], [18]. They have important applications in SONET and WDM networks [11], [12], [22]. Path protection and link protection schemes are used in [1], [16], [17], [18]. The authors of [3], [22] discussed the use of protection cycles and self-healing rings. P-cycle protection schemes was proposed and implemented in [4]. The authors of [5] proposed using multi-trees to achieve link or node failure restoration. In [8], [9], [10], the authors extended the multi-tree scheme by creating preplanned redundant trees on arbitrary node-redundant or link-redundant networks. The authors of [25] introduced the concept of *Path-Based Spanner* and present two linear time algorithms for computing preplanned redundant trees. In [23], [24], the authors introduced the concept of *Quality of Protection* (QoP) and *Quality of Service* (QoS) of redundant trees and provided several heuristics to achieve better recovery performances. Those schemes are applicable to IP, WDM, SONET and ATM to provide protection and restoration from link or node failure [11], [13], [17], [18], [19].

In [9], Medard, Finn, Barry and Gallager presented an elegant scheme to construct two directed spanning trees from a root s in such a fashion that the failure of any node or edge in the graph (other than the root node) leaves each vertex connected to the root by at least one of the directed trees, provided that the network is 2-edge connected or 2-connected. They named one of the trees the *blue tree* and the other the *red tree*. We will use T^B to denote the blue tree and use T^R to denote the red tree.

The concept of blue/red trees is illustrated in Figure 1. Figure 1(a) shows a 2-connected network with 5 nodes and 7 links. Figure 1(b) illustrates two directed trees rooted at the root node 1, spanning all the other nodes in the network. The tree with solid edges is T^B and the tree with dashed edges is T^R.



(a) A sample network (b) Blue/Red trees of the sample network

Fig. 1. A Simple network illustrating redundant trees.

Before any failure, every node in the network is reachable via both the blue tree and the red tree. If a single link failure occurs, say at link $[1,2]$, nodes 3, 4, 5 are no longer connected to the root node 1 via T^B. However, they are still connected to the root node via T^R. Similarly, if a single node

The research of Guoliang Xue, Weiyi Zhang, Jian Tang was supported in part by NSF grants ANI-0312635 and CCF-0431167, ARO grant DAAD19-00-1-0377 and W911NF-04-1-0385, and a seed grant from CEINT. The research of Krishnaiyan Thulasiraman was supported in part by NSF grant ANI-0312435.

failure happens, say at node 2, none of the remaining nodes is connected to the root node via T^B. They are, however, still connected to the root node via T^R. The pair of trees T^B and T^R enables fast recovery from single link or node failure using automatic protection switching.

In [9], the authors showed that for any 2-connected (2-edge connected) graph, there exists a pair of recovery trees which provide fast recovery from single node (edge) failure. They also presented $O(n^3)$ time algorithms for computing such a pair of blue/red trees T^B and T^R for any 2-vertex or 2-edge connected graph, where n is the number of nodes in the network. In [25], Xue *et al.* extended the notion of redundant recovery trees to connected graphs. They also presented two linear time algorithms to construct a pair of blue/red trees based on depth first search (**DFS**).

In [23], [24], Xue, Chen and Thulasiraman introduced the concept of *quality of protection (QoP)* and *quality of service (QoS)* of recovery schemes using redundant trees. The quality of protection of a pair of blue/red trees T^B and T^R is defined as the maximum integer k such that there exists an instance of k simultaneous link failures that T^B and T^R can survive. For example, in Figure 1(b), if links $[1, 2]$ and $[2, 5]$ are broken at the same time, every node is still connected to the root node 1 via either T^B or T^R. However, the pair of blue/red trees in Figure 1(b) cannot handle more than two link failures simultaneously. By definition, the QoP of this pair of blue/red trees is 2. Xue *et al.* [24] provided an $O(n^2(m + n))$ time heuristic to compute a pair of blue/red trees with enhanced QoP, where n and m are the number of nodes and links in the network.

If hardware usage is a major concern, one would like to construct a pair of single failure recovery trees with minimum total cost. Xue *et al.* [24] presented two $O(n^2(m + n))$ time heuristics for constructing a pair of single failure recovery trees with small total cost (one for single link recovery, one for single node recovery).

In this paper, we present three linear time algorithms for constructing redundant trees for single link failure recovery in 2-edge connected graphs and for single node failure recovery in 2-connected graphs. Our first algorithm aims at high QoP for single link recovery schemes in 2-edge connected graphs. The previous best algorithm has a running time of $O(n^2(m+n))$, where n and m are the number of nodes and links in the network. Our algorithm has a running time of $O(m+n)$, with comparable performance. Our second algorithm aims at high QoS for single link recovery schemes in 2-edge connected graphs. Our algorithm improves the previous best algorithm with $O(n^2(m + n))$ time complexity to $O(m + n)$ time complexity with comparable performance. Our third algorithm aims at high QoS for single node recovery schemes in 2-connected graphs. Again, our algorithm improves the previous best algorithm with $O(n^2(m+n))$ time complexity to $O(m+n)$ time complexity with comparable performance.

Our algorithms are good both in terms of running time and in terms of QoP or QoS. Results from previous work do not achieve both goals at the same time. The authors of [9] did not consider the QoP and QoS issues. The authors of [25] presented linear time algorithms for constructing a pair of blue/red trees, without consideration of QoP or QoS. The authors of [24] provided algorithms to construct T^B and T^R with superior QoP or QoS. However, the running times of those algorithms are $O(n^2)$ times longer than the running times of the algorithms presented in this paper.

The rest of this paper is organized as follows. In Section II, we present algorithms to construct blue/red trees with enhanced QoP and QoS performance. In Section III, we present simulation results. We conclude this paper in Section IV.

II. Fast Computation of Blue/Red Trees

Our algorithms are based on the *Depth First Search* (DFS) technique [20], which is briefly described in the following as Algorithm 1. We assume that the graph has n vertices and m edges and is given by its adjacency lists.

Algorithm 1 DFS(G, v, u)

v is the new vertex to be visited. u is the parent of v in the DFS spanning tree if v is not the root node s. It is assumed that the global array $D[]$ is initialized to zero and that the global variable N is initialized to 1. There is also a global array $L[]$. The set of (directed) *tree edges* is initialized to $\emptyset$. The set of (directed) *back edges* is initialized to $\emptyset$.

```
D[v] := N; L[v] := D[v]; N := N + 1;
for each vertex w adjacent from v
  if (D[w] = 0)
    add the directed edge (v, w) to the set of tree edges;
    parent[w] = v;
    DFS(G, w, v);
    L[v] := min(L[v], L[w]);
  else if (w ≠ u)
    add the directed edge (v, w) to the set of back edges;
    L[v] := min(L[v], D[w]);
  endif
endfor
```

In the description of the algorithm, v is the current new vertex to be visited, u is the parent of v in the DFS tree if v is not the root node s. $D[v]$, the DFS *number* of v, indicates the order that node v is visited. The *lowpoint number* of each node v, denoted by $L[v]$, is the lowest DFS number of a vertex u that can be reached from v by a sequence of tree edges followed by an optional back edge. If the graph G is connected, Algorithm 1 requires $O(m + n)$ time to construct a DFS tree of graph G [20].

A. Redundant Trees with Enhanced QoP

In this section, we present a linear time algorithm for constructing a pair of single-link recovery trees with comparable QoP performance to the result in [24]. In [24], the authors noted that the construction of blue/red trees is closely related to the *ear decomposition* of a graph [21]. They also noted that

the QoP of a pair of blue/red trees equals the number of ears in the corresponding ear decomposition. Consequently, it is desirable to use more ears in the construction of the blue/red recovery trees.

Definition 2.1: Given a **DFS** tree of graph G, and a pair of blue/red trees T^R and T^B spanning a subset of the network nodes, a back edge (u, w) is called an *acceptable back edge* if node u is not on T^B and T^R while node w is on both T^B and T^R. An acceptable back edge (u, w) is called a *maximal back edge* if for any child v of u in the **DFS** tree, it is impossible to reach a node on T^B and T^R from v by a sequence of tree edges followed by an optional back edge.

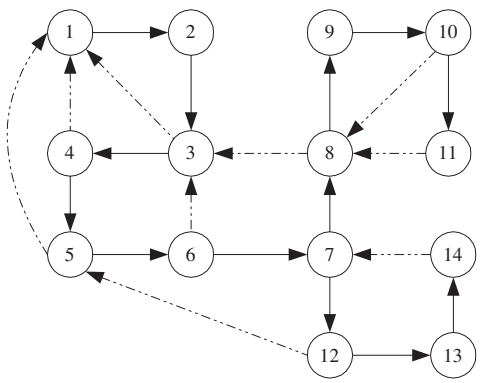


Fig. 2. DFS tree of a sample network.

Figure 2 shows a DFS tree of a graph G, where the solid links represent tree edges and the dashed links represent back edges. Suppose at present that only the root node 1 is on the existing T^B and T^R. Then back edges $(3, 1)$, $(4, 1)$ and $(5, 1)$ are acceptable back edges. Other back edges, such as edge $(6, 3)$, are not acceptable back edges. Among all acceptable back edges, only $(5, 1)$ is a maximal back edge. $(3, 1)$ is not a maximal back edge because from node 3, we can follow the tree edges $(3, 4)$, $(4, 5)$ and back edge $(5, 1)$ to a node on T^R and T^B. Similarly, $(4, 1)$ is not a maximal back edge either.

It is worthy of 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 that a back edge which is not an *acceptable back edge* at present, but will become an *acceptable back edge* later. For example, if node 3 had been added into the current T^B and T^R, then back edges $(6, 3)$ and $(8, 3)$ will become *acceptable back edges*, and $(8, 3)$ will become a *maximal back edge*.

Our algorithm for computing a pair of blue/red trees with enhanced QoP is presented in Algorithm 2.

Lemma 2.1: Given a 2-edge connected graph G, each node in G will be marked exactly once in Algorithm 2, and each marked node will be added into T^B and T^R. Thus, when Algorithm 2 terminates, all nodes are inserted into T^B and T^R, and a pair of single-link failure recovery trees are created.

PROOF. In Step_3, we mark node w only if it is not marked. Therefore a node will not be marked more than once. On the other hand, each node in G will be marked at least once. To show this, assume that node u is not marked after running

Algorithm 2 $EnhancedQoPE(G, s)$

```
step_1 apply DFS from root vertex s, compute the DFS tree
       of G, and DFS number D[v], lowpoint number L[v],
       parent node parent[v] for every vertex v; initialize T^R
       and T^B to the empty tree.
step_2 mark node s, insert it into queue marked_nodes and
       add it into T^B and T^R; set v^B(s) and v^R(s) such that
       v^B(s) > v^R(s) = 0.
step_3 while (marked_nodes is NOT EMPTY)
         dequeue a marked node u;
         for (each adjacent node w of u)
           if ((w,u) is a back edge) then
             while (w is not marked)
               mark node w;
               push w onto stack temp_tree;
               w = parent[w];
             endwhile
             pop off all the elements on temp_tree
             and insert them into marked_nodes
             while popping;
           else if ((u,w) is an acceptable back edge)
             let v be the nearest ancestor of node
             u on existing T^B and T^R;
             if (v^B(w) ≥ v^B(v)) then
               p_s = w; p_t = v;
             else
               p_s = v; p_t = w;
             endif
             the tree path from v to u concatenated
             with the acceptable back edge (u,w)
             form a path or cycle (in case v = w)
             connecting w and v;
             Let this path or cycle be
             (p_s, x_1, ...x_k, p_t);
             add p_s → x_1 → ... → x_k to T^B;
             add p_t → x_k → ... → x_1 to T^R;
             assign v^B(p_s) > v^B(x_1) > v^R(x_1) >
             ... > v^B(x_k) > v^R(x_k) > v_max;
           endif
         endfor
       endwhile
       where v_max=max{v^B(y), v^R(y)|y is on existing T^B and
       T^R and v^B(y) < v^B(p_s), v^R(y) < v^B(p_s)};
```

Algorithm 2. This means that it is impossible to go from node u to one of its marked ancestors by going through a sequence of tree edges followed by an optional back edge. For otherwise u will be marked in Step_3.

We have two cases here:

1) It is impossible to go from u to one of its ancestors via a sequence of tree edges followed by an optional back edge.
2) It is possible to go from u to one of its ancestors via a sequence of tree edges followed by an optional back

edge, but impossible to go from u to one of its marked ancestors via a sequence of tree edges followed by an optional back edge.

If the first case occurs, we conclude that the graph is not 2-edge connected. This contradicts our assumption that G is 2-edge connected. If the second case occurs, there must be an ancestor (say w) of u that is not marked. This in turn implies that there is an ancestor of w that is not marked. Following this argument, we conclude that the root node s is not marked. This is a contradiction.

For each marked node v which is not on the current T^B and T^R, if there is an outgoing acceptable back edge connecting v to an ancestor of v that is on T^B and T^R, v will be inserted into the recovery trees when it is processed. Otherwise, since graph G is 2-edge connected, v can reach an ancestor on the current T^B and T^R by following a path consisting of several tree edges and a back edge (m, n), where m is not on T^B or T^R, and n is m's ancestor on T^B and T^R. In other words, (m, n) is an *acceptable back edge*. When node m is processed, we start to add a new path (cycle) connecting node n and m's nearest ancestor which is on the current T^B and T^R. Since v is m's ancestor and v is not on T^B and T^R yet, it must be included in the new path (cycle) and inserted into T^B and T^R. Therefore, each marked node v will be added to T^B and T^R. Therefore when the $While$ loop in Step_3 terminates, T^B and T^R must both span all the nodes in the network. □

Theorem 2.1: Assume that G is a 2-edge connected graph with n vertices and m edges. Algorithm 2 computes a pair of single-link recovery trees correctly. Furthermore, the worst-case running time of the algorithm is $O(m + n)$.

PROOF. It follows from Lemma 2.1 that when Algorithm 2 terminates, all nodes are inserted into T^B and T^R. Each time a path (cycle) is found, we grow T^R and T^B according to the voltage rules [9]. As proved in [9], the voltage rules guarantee the correctness of the algorithm.

Step_1 of the algorithm constructs the DFS tree and computes corresponding information in $O(m + n)$ time. Step_2 only needs constant time to mark root s and add it into T^B and T^R. Since we have proved in *Lemma* 2.1 that each node will be marked exactly once in the *While* loop, and each edge will be checked at most twice. Thus finding a new path (cycle) to be inserted only needs $O(m + n)$ time in Step_3. Furthermore, when an acceptable back edge (u, w) is found, a new path (cycle) needs to be added into the T^B and T^R. From node w, node u will be added into the new path (cycle) through the acceptable back edge. Then we add u's parent node $parent[u]$ into the path (cycle) if $parent[u]$ is not in the current recovery trees. We repeat adding new nodes into the path (cycle) until some node's parent node v is already on T^B and T^R. Now a new path (cycle) is found and we grow the recovery trees and assign voltages to each newly added node. Because each node will be inserted into the recovery trees only once, and each edge will only be added to T^B and T^R at most once, adding all new paths (cycles) to T^B and T^R only needs $O(m+n)$ time. Therefore, the total running time of Step_3 is $O(m+n)$. This shows that the worst case time complexity of the algorithm is $O(m + n)$. □

Let us illustrate Algorithm 2 with the sample network shown in Figure 2. Node 1 is the root node. In Step_1, we construct the DFS tree, which is shown in Figure 2. In Step_2, we mark node 1 and insert it into T^B and T^R. In Step_3, we check all adjacent nodes of node 1, and find that there are 3 incoming back edges $(3, 1)$, $(4, 1)$ and $(5, 1)$. While processing back edge $(3, 1)$, we find that node 3 is not marked yet. So we mark it and push it onto stack *temp_tree*. Then node 2, being 3's parent node, will be marked and pushed onto *temp_tree*. Since node 2's parent node 1 has been marked, we stop marking nodes and pop nodes 2 and 3 from *temp_tree* and insert them into queue *marked_nodes* in that order. Next, back edges $(4, 1)$ and $(5, 1)$ will be processed and nodes 4 and 5 will be marked and inserted into *marked_nodes*. At this point, the nodes in *marked_nodes* are $2, 3, 4, 5$. Next we will process node 2. It has neither incoming back edges nor outgoing back edges incident with it. Therefore it will be skipped and node 3 is processed, which has an outgoing *acceptable back edge* $(3, 1)$ going back to the current T^B and T^R. Now a new cycle needs to be added. From node 1, node 3 is added into the new cycle through the *acceptable back edge*. Then node 2, as node 3's parent node, will be added. The parent node of node 2 is node 1, which is already on the current T^B and T^R. Thus we get the first cycle $(1, 2, 3, 1)$. We insert nodes $2, 3$ into the recovery trees, add edges $(1, 2)$ and $(2, 3)$ to T^B, $(1, 3)$ and $(3, 2)$ to T^R, and assign a voltage to each newly added node according to the voltage rule [9]. Meanwhile, node 3 has incoming back edges $(6, 3)$, $(8, 3)$ incident with it. So nodes $6, 7, 8$ will be marked and inserted into *marked_nodes*. By now we have finished the operations on node 3. The next node dequeued from *marked_nodes* is 4, and it has an outgoing acceptable back edge $(4, 1)$ incident with it. A new path starting from node 1 needs to be added. Node 4 will be added into the new path from node 1 through the acceptable back edge. Node 4's parent node 3 is on the current T^B and T^R. Thus we get a path $(1, 4, 3)$. We insert node 4 to the current recovery trees, and add $(1, 4)$ to T^B, $(3, 4)$ to T^R. Then we assign a voltage to node 4. Following this way, we can construct T^R and T^B as shown in Figure 3, where solid blue edges are on T^B and dashed red edges are on T^R.

B. Redundant trees with enhanced QoS

We also consider cost as the primal criterion of *Quality of Service* and deal with single link or single node failure in a 2-edge or 2-vertex connected graph G. We assume that each link in G has the same constant cost, and the total cost of a pair of redundant trees is the summation of the costs of all links on T^B and T^R. In this section, we will present two algorithms for constructing a pair of single link and single node recovery trees with low total cost in case of single link failure and single node failure, respectively.

1) ***Redundant trees for 2-edge connected graph****:* Given a 2-edge connected graph G, we want to construct a pair of single link recovery trees with small cost. Such an algorithm is listed as Algorithm 3.

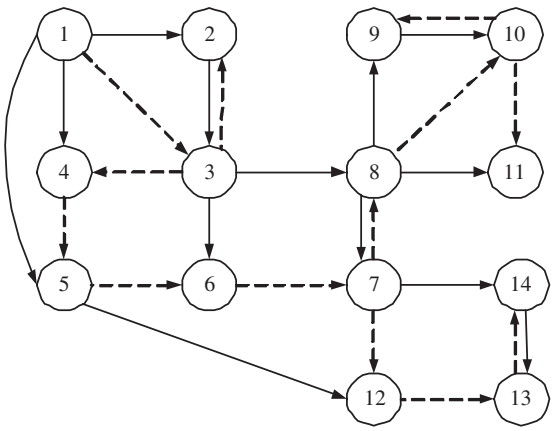


Fig. 3. Illustration for Algorithm EnhancedQoPE.

Lemma 2.2: Assume that G is a 2-edge connected graph. Algorithm 3 takes only $O(n)$ time to check all maximal back edges.

PROOF. After each node u is marked, it will be appended into the queue *marked_nodes* and added to the current T^B and T^R. Therefore, when node u is dequeued from *marked_nodes*, it is already on the current T^B and T^R. Therefore, the incoming back edge (w, u) is an acceptable back edge. We need to check whether it is a *maximal back edge*. To check this, we need to see the lowpoint numbers of all children of node w. If none of them can go back to the current T^B and T^R through several tree edges followed by an optional back edge, we know that (w, u) is a *maximal back edge*. Otherwise, we record node w to be a visited node and will not check it again. The reason is that since w has a child which can go back to the current T^B and T^R by following several tree edges and a back edge, node w will not have any *maximal back edge* incident with it no matter how the recovery trees are grown. Therefore w does not need to be checked again. In addition, all of w's children will also be checked only once because they must all be w's children in this DFS tree. Therefore, Algorithm 3 needs only $O(n)$ time to check all *maximal back edges* in G because each node will be checked at most once. □

Lemma 2.3: Given a 2-edge connected graph G, each node in G will be marked exactly once in Algorithm 3. When Algorithm 3 terminates, all marked nodes have been inserted into T^B and T^R.

PROOF. If a node is marked, it will be inserted to the current T^B and T^R in Step_3, and will not be marked again. For each node u which is unmarked, if it has an outgoing maximal back edge connecting to some node v on the current T^B and T^R, then it will be marked and inserted into T^B and T^R. Otherwise, node u will be recorded as a visited node, and cannot trigger a process of adding nodes to T^B and T^R. However, as long as the graph G is 2-edge connected, u must have a child that can go back to the current T^B and T^R, and there exist path(s) that are formed by several tree edges followed by a back edge connecting u to some node on the current T^B and T^R. From those paths, we can find one path which is composed of several tree edges followed by a *maximal back edge*. Assume this maximal back edge is (w, t), where node w is a descendant of node u and node t is on the current T^B and T^R. Since node t is marked, when it is dequeued, node w will be checked and a new path (cycle) from w's nearest ancestor v being on the the current T^B and T^R to node t will be added into T^B and T^R. As an ancestor of node w, and being not on T^B and T^R, node u must be on this path (cycle), and be marked and inserted into T^B and T^R. Thus any node in G will be marked and inserted into the recovery trees exactly once. □

Algorithm 3 $ReducedCostE(G, s)$

step_1 apply DFS from root vertex s, compute the DFS tree of G, and DFS number $D[v]$, lowpoint number $L[v]$, parent node $parent[v]$ for every vertex v;initialize T^R and T^B to the empty tree.

step_2 mark node s, insert it into queue *marked_nodes* and add it into T^B and T^R; set $v^B(s)$ and $v^R(s)$ such that $v^B(s) > v^R(s) = 0$.

step_3 while (*marked_nodes* is NOT EMPTY)
  dequeue a marked node u;
  for (each adjacent node w of u)
    if ((w is not marked OR visited) AND
    ((w, u) is a *maximal back edge*))
      let v be the nearest ancestor of node w on existed T^B and T^R;
      if $v^B(u) \geq v^B(v)$
        then $p_s = u$; $p_t = v$;
      else $p_s = v$; $p_t = u$;
      endif
      the tree path from v to w concatenated with the *maximal back edge* (w, u) form a path or cycle (in case $v = u$) connecting nodes u and v;
      from v to w, mark each node if it is unmarked, and insert it to the end of queue *marked_nodes*;
      assume this path or cycle be $(p_s, x_1, ..., x_k, p_t)$;
      add $p_s \rightarrow x_1 \rightarrow ... \rightarrow x_k$ to T^B;
      add $p_t \rightarrow x_k \rightarrow ... \rightarrow x_1$ to T^R;
      $v^B(p_s) > v^B(x_1) > v^R(x_1) > ... > v^B(x_k) > v^R(x_k) > v_{max}$;
    else if (node w is not marked or visited)
      record it as visited;
    endif
  endfor
endwhile

where $v_{max} = max\{v^B(y), v^R(y) | y$ is on existing T^B and T^R and $v^B(y) < v^B(p_s), v^R(y) < v^B(p_s)\}$;

Theorem 2.2: Assume that G is a 2-edge connected graph with n vertices and m edges. Algorithm 3 computes a pair of

recovery trees correctly. Furthermore, the running time of the algorithm is $O(m+n)$.

PROOF. The correctness of the algorithm follows from the proof of Theorem 2.1. From *Lemma* 2.2, we know that each node is checked at most once in Algorithm 3. As proved in *Lemma* 2.3, each node will be marked and inserted into the recovery trees exactly once, and each edge will be processed at most once. Therefore the running time of the algorithm is $O(m+n)$. □

Let us illustrate Algorithm 3 with the sample network shown in Figure 2. Node 1 is the root node. In Step_1, we construct the DFS tree, which is shown in Figure 2. In Step_2, we mark node 1 and insert it into T^B and T^R. In Step_3, we first check all adjacent nodes $2, 3, 4, 5$ of node 1. None of them is marked or visited. $(1, 2)$ is a tree edge. $(3, 1)$ and $(4, 1)$ are not *maximal back edges*. So we record nodes $2, 3$ and 4 as visited. The incoming back edge $(5, 1)$ is a *maximal back edge*, and a new cycle needs to be added. From node 1, through the *maximal back edge*, node 5 is added into the new cycle. Then node 5's parent, node 4, is added into the new cycle. Then similarly, node 3 and node 2 will be added into the new cycle. Node 2's parent, node 1, is on the current T^B and T^R, so we stop the process and obtain a new cycle, which is $(1, 2, 3, 4, 5, 1)$. We add all the nodes on this cycle to the current T^B and T^R. We add the edges $(1, 2), (2, 3), (3, 4), (4, 5)$ to T^B, and add the edges $(1, 5), (5, 4), (4, 3), (3, 2)$ to T^R. A voltage will be assigned to each node on T^B and T^R according to the voltage rule [9]. Nodes $2, 3, 4, 5$ will be marked and added into the queue *marked_nodes* in that order. Then node 2 is the next node dequeued from *marked_nodes*. It has no back edge incident with it and will be skipped. Next we dequeue the marked node 3. It has unmarked adjacent nodes 6 and 8. $(8, 3)$ is an incoming *maximal back edge*. So nodes $6, 7, 8$ will be marked and inserted into *marked_nodes*, and a new path $(3, 8, 7, 6, 5)$ will be added. We insert nodes $6, 7, 8$ into the current T^B and T^R, add edges $(3, 8), (8, 7), (7, 6)$ to T^B, add the $(5, 6), (6, 7), (7, 8)$ to T^R. Similarly, A voltage will be assigned to each node on T^B and T^R according to the voltage rule. Continuing in this way, we can construct T^R and T^B as shown in Figure 4.

*2) **Redundant trees for 2-vertex connected graph**:* Now we focus on the node failure case. Algorithm 4 can be used to construct a pair of single-node recovery trees in a 2-vertex connected graph. First of all, a new definition needs to be introduced.

*Definition 2.2: A node v is called the **oldest child** if among all of its parent node u's children, v has the minimum lowpoint number. In other words, for any child node w of node u, we have $L[v] \leq L[w]$. If there are several nodes having the same lowpoint number, we choose one of them to be the oldest child arbitrarily.*

Lemma 2.4: Given a 2-vertex connected graph G, for each node u which is not on the current recovery trees, assume its nearest ancestor on the current existing T^B and T^R is node v ($v \neq s$), we claim that u can go through several tree edges and a maximal back edge *to go back to some node w which is also on the current T^B and T^R with $D[w] < D[v]$.*

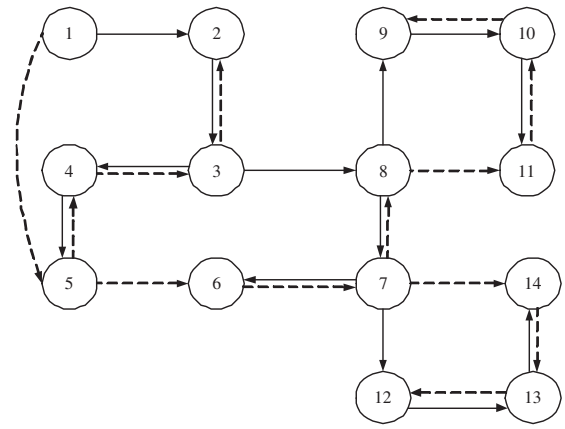


Fig. 4. Illustration for Algorithm ReducedCostE.

PROOF. It is obvious that each node in G can go back to the current T^B and T^R by tree edges and an optional back edge in a 2-connected graph. Let us assume that node u can go back to some node w which is on the current T^B and T^R. But $L[u] = D[w] = D[v]$ or $L[u] = D[w] > D[v]$.

If $L[u] = D[v]$, then node v must be an *articulation node* in the graph [21]. But in a 2-vertex connected graph, there is no articulation node. So the first case is impossible;

We assume that node w is an ancestor of node u and is on the current T^B and T^R. If $D[w] = L[u] > D[v]$, however, w cannot be on the current T^B and T^R because we know that v is the nearest ancestor of node u on the current T^B and T^R. This is a contradiction. Hence this case is also impossible. $L[u]$ cannot be larger than $D[v]$.

Therefore, node u can go back to a node w which is an ancestor of node v and also on the current T^B and T^R. □

Lemma 2.5: In Algorithm 4, each marked node u must be on the current T^B and T^R when it is popped from stack marked_nodes.

PROOF. In Step_4, root node s is the first marked node, and it is on T^B and T^R as well. From this node, several unmarked nodes will be marked until a maximal back edge is found. Then all the newly marked nodes will be added into the current T^B and T^R in Step_4. Next, another marked node (which is already on the current recovery trees) will be popped and some unmarked nodes will be marked and added into the current T^B and T^R. Following this way, each time we start to mark some new nodes, the old marked nodes have been pushed onto stack *marked_nodes*, and added into the current T^B and T^R. Therefore, every time when a marked node u is popped from *marked_nodes* to start marking some new nodes, u is already on the current T^B and T^R. □

Theorem 2.3: Assume that G is a 2-vertex connected graph with n vertices and m edges. Algorithm 4 computes a pair of single-node recovery trees correctly, and its running time is $O(m+n)$.

PROOF. For each node in graph G, it will be marked only if it has not been marked. So each node will not be marked

Algorithm 4 $ReducedCostV(G, s)$

```
step_1 apply DFS from root vertex s, compute the DFS tree
       of G, and DFS number D[v], lowpoint number L[v],
       parent node parent[v] for every vertex v; initialize T^R
       and T^B to the empty tree.
step_2 sort children of each node according to their lowpoint
       number L[];
step_3 mark node s, push it onto stack marked_nodes, and
       add it to T^B and T^R; set v(s) = V ≥ 0.
step_4 while (marked_nodes is NOT EMPTY)
         pop a marked node u;
         for (each child node w of u)
           while (w is not marked) (*)
             Let node n be the oldest child of w; (**)
             if (L[n] < D[u] OR L[n] == 1)
               mark node w;
               push w onto stack marked_nodes;
               w = n;
             else
               mark node w;
               push w onto marked_nodes;
               if v(L[w]) ≥ v(u)
                 then p_s = L[w]; p_t = u;
               else p_s = u; p_t = L[w];
               endif
               the tree edges from u to w concatenated
               with the maximal back edge (w, L[w])
               form a path or cycle (in case u is the
               root node) connecting u and L[w];
               assume this path or cycle (in case of
               L[w] = u = s) be (p_s, x_1, ...x_k, p_t);
               add p_s → x_1 → ... → x_k to T^B;
               add p_t → x_k → ... → x_1 to T^R;
               assign v(p_s) > v(x_1) > ... >
               v(x_k) > v_max;
             endif ;
           endwhile;
         endfor ;
       endwhile
       where v_max = max{v(y)|y is on existing T^B and
       T^R and v(y) < v(p_s)};
```

more than once. On the other hand, if a marked parent node is being processed, all of its children will be marked in Step_4. So all nodes will be marked since G is connected. Thus each node will be marked at least once. Therefore, each node in the graph will be marked exactly once.

In Step_4, Algorithm 4 will terminate when all marked nodes have been processed. Since all nodes in graph will be marked, all nodes have been processed when the algorithm terminates.

Algorithm 4 grows T^B and T^R gradually by finding a path connecting a node u to another node t (or $u = t =$ root node s in case that a cycle is found). Both nodes are on the current recovery trees. Some nodes on this path (cycle) which are not on the recovery trees yet will be added to the current T^B and T^R. From Lemma 2.5 we know that the popped marked node u is node w's nearest ancestor in the current recovery trees. In the proof of Lemma 2.4, we showed that there always exists a node t on the current recovery trees such that there is a new path ($t \neq u$) or a new cycle ($t = u = s$) connecting node t and node u via some nodes which are not on the current recovery trees. So each time when a marked node is processed, at least one new node will be added into the current T^B and T^R with the assignment of the voltages. The voltage rule guarantees the correctness of the algorithm [9]. When the algorithm terminates, all marked nodes (all nodes in the graph) have been processed, and all nodes in G will be on the constructed T^B and T^R. These recovery trees can survive a singe node (except root node s) failure.

In Step_1, we need $O(m + n)$ time to construct the DFS tree and the corresponding information. In Step_2, for each node, we sort its child nodes. Thus each node u will be sorted exactly once in this step because u has only one parent node in the DFS tree. We use bucket sort to perform sorting in linear time. Step_3 only takes constant time. In Step_4, In (*) and (**), for node w, only its *oldest child* needs to be checked. Thus, we only need to check each node exactly once before marking it or inserting it to the current recovery trees. Consequently, we can finish Step_4 in linear time as well. All in all, the time complexity of Algorithm 4 is $O(m + n)$. □

Let us illustrate Algorithm 4 with the sample network shown in Figure 5. Node 1 is the root node. In Step_1, we construct the DFS tree, which is shown in Figure 5. The solid edges in the graph are the tree edges, and dashed edges are the back edges. In Step_2, we mark node 1 and insert it to the current T^B and T^R. In Step_3, we check the child node of node 1, which is node 2. Since node 2's oldest child 3 can go back to the current T^B and T^R by several tree edges and a back edge, we mark node 2, push it onto *marked_nodes*. Its oldest child, node 3 will be checked. Node 3's oldest child is node 4, which has lowpoint number $L[4] = 1$. Thus we do not use link $(3, 1)$ in the recovery trees since it is not a *maximal back edge*. We mark node 3, and push it onto *marked_nodes*. Node 4 will be checked now. Similarly, node 4 will be marked and its oldest child node 5 is to be checked. We find 5's oldest child 6 can only go back by several tree edges and a back edge as far as to node 2, which is not on the current T^B and T^R. Therefore we know the back edge $(5, 1)$ is a maximal back edge, and we need to add an cycle to the current T^B and T^R at this time. The nodes on cycle $(1, 2, 3, 4, 5, 1)$ will be added into T^B and T^R. Edges $(1, 2), (2, 3), (3, 4), (4, 5)$ will be added into T^B, and edges $(1, 5), (5, 4), (4, 3), (3, 2)$ will be added into T^R. Each node will be assigned a voltage according to the voltage rule[9]. Then we pop node 5, and its child node 6 will be checked. Node 6's oldest child node 7 has lowpoint number 2, which is on the current T^B and T^R. So we mark node 6, and push it onto *marked_nodes*. Its oldest child node 7 is the next node to be checked. Node 7 has 2 children, node 8 and node 12. Node 8 is the oldest child, and its lowpoint

number is 2. Thus node 7 will be marked and pushed onto *marked_nodes*. Next node 8, being node 7's oldest child, is to be checked. It will be marked and pushed onto *marked_nodes* because its oldest child node 9 has an incident outgoing back edge connecting to node 2. When we check the node 9, we find that its oldest child 10 cannot go back to current T^B and T^R by tree edges followed by a back edge, so we need to add all nodes on the path $(5, 6, 7, 8, 9, 2)$ to the current T^B and T^R. We can add edges $(2, 9), (9, 8), (8, 7), (7, 6)$ to T^B, add edges $(5, 6), (6, 7), (7, 8), (8, 9)$ to T^R with assignment of voltage to each node on the path. Following this way, we can construct T^R and T^B in Figure 6. The solid blue edges represent the edges on T^B, and the dashed red edges are T^R edges.

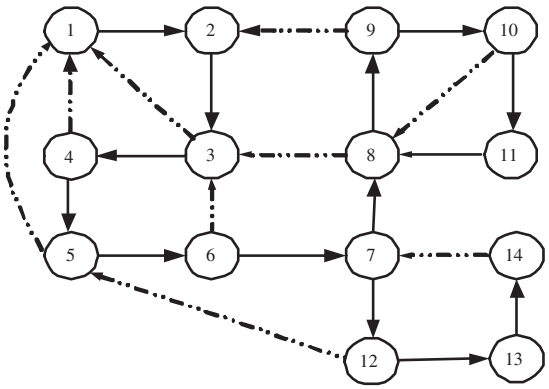


Fig. 5. DFS tree of a 2-connected graph.

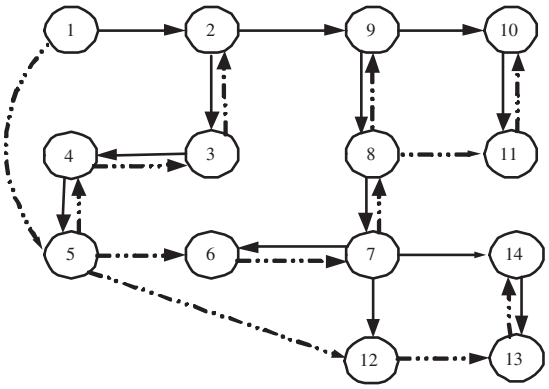


Fig. 6. Illustration for Algorithm ReducedCostV.

III. Computational Result

To evaluate the performance of our proposed algorithms, we implemented all the algorithms and tested them on randomly generated input data. A C++ class library, LEDA [7] is used in all of our implementations. We have used 50, 100 and 200 as the number of nodes in the networks. For each value of n, we have used $3n$ and $n \log n$ as the number of links. Therefore, there are six node-link combinations. For each given size (given by the node-link combination), we randomly generate 100 2-connected graphs (for node recovery) or 2-edge connected graphs (for link recovery). *Each entry in the tables reported is the average over 100 runs*. We use **RedBlueE**(**RedBlueV**) to denote the algorithms in [9] for edge (node) recovery, where the paths and cycles are computed without any of the QoP or QoS considerations.

Table I presents the QoP performance of the constructed single-link recovery trees. **MaxQoPE** and **DFSQoPE** denote the algorithms from [24] and [25], respectively. **MoreQoPE** represents Algorithm 2 in this paper. From the table, we observe that our algorithm has a comparable QoP performance with **MaxQoPE**. Its running time is as fast as that of **DFSQoPE** and is much faster than that of **MaxQoPE**.

The QoS performance of the single-link recovery trees are reported in Table II. Similarly, **MinCostE** and **DFSQoSE** denote the algorithms from [24] and [25], respectively. **LessQoSE** represents Algorithm 3 in this paper. **MinCostE** has a slightly lower cost than **LessQoSE** and **DFSQoSE**. We notice that the latter two algorithms have much faster running times than that of **MinCostE**.

In Table III, we show the QoS performance of the single-node recovery trees. **MinCostV** denotes the algorithm in [24]. **ReducedCostV** represents Algorithm 4 in this paper. **MinCostV** has a slightly lower cost than **ReducedCostV**, but **ReducedCostV** has a much faster running time. It is worth noting that we do not have the result from [25] in this table because the algorithms in [25] cannot handle the node failure case.

IV. Conclusions

In this paper, we have presented three linear time algorithms for computing a pair of redundant trees that can be used for single link/node failure recovery. Our algorithms not only are the fastest possible, but also have excellent performance in terms of either quality of protection or quality of service. In particular, the running times of our algorithms are $O(n^2)$ times faster than that proposed in [24].

Many recovery schemes for recovery from single failure are 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 continue to work along this line. We will also extend the redundant tree approach to provide guaranteed recovery from multiple failures [2], [14], [15].

References

[1] V. Anand and C. Qiao, Dynamic establishment of protection paths in WDM networks, Part I, *IEEE International Conference on Communications and Computer Networks'2000*, pp. 198–204.

[2] Hongsik Choi, S. Subramaniam, and Hyeong-Ah Choi, On double-link failure recovery in WDM optical networks, *IEEE INFOCOM2002*, pp. 808–816.

[3] G.Ellinas, A.G.Hailemariam, T.E. Stern, Protection cycles in mesh WDM networks, , *IEEE Journal on Selected Areas in Communications*, Vol. 18(2000), pp. 1924–1937.

TABLE I

COMPARISON OF QOP PERFORMANCE

network	RedBlueE		MaxQoPE		DFSQoPE		MoreQoPE	
$n \times m$	QoP	msec	QoP	msec	QoP	msec	QoP	msec
050x0150	23	1.10	47	1.50	13	0.00	40	0.10
050x0282	25	1.20	48	1.00	15	0.30	44	0.10
100x0300	45	7.80	96	6.50	22	0.10	81	0.20
100x0664	50	4.70	97	5.80	20	0.30	91	0.30
200x0600	91	55.5	194	38.3	39	0.20	162	1.00
200x1529	99	24.1	197	33.9	29	0.60	185	1.10

TABLE II

COMPARISON OF COST OF EDGE RECOVERY TREE.

network	RedBlueE		MinCostE		DFSQoSE		LessQoSE	
$n \times m$	Cost	msec	Cost	msec	Cost	msec	Cost	msec
050x0150	72	1.10	55	4.60	62	0.20	56	0.00
050x0282	74	0.70	52	3.40	64	0.70	53	0.10
100x0300	144	7.59	111	34.4	121	1.10	114	0.10
100x0664	149	3.90	104	18.7	119	2.60	106	0.50
200x0600	290	55.0	224	240.9	238	3.50	229	0.20
200x1529	298	22.9	208	114.7	228	10.2	212	0.80

TABLE III

COMPARISON OF COST OF NODE RECOVERY TREE

network	RedBlueV		MinCostV		ReducedCostV	
$n \times m$	Cost	msec	Cost	msec	Cost	msec
050x0150	72	1.10	55	6.60	57	0.30
050x0282	74	0.70	51	4.20	53	0.30
100x0300	144	7.60	111	34.5	114	0.30
100x0664	149	3.90	104	23.2	106	0.50
200x0600	290	55.0	224	251.4	229	1.10
200x1529	297	24.2	208	166.3	212	2.70

[4] W.D. Grover and D. Stamatelakis, Cycle-oriented distributed preconfiguration: ring-like speed with mesh-like capacity for self-planning network restoration, *IEEE International Conference on Communications'1998*, pp. 537–543.

[5] A. Itai and M.Rodeh, The multi-tree approach to reliablity in distributed networks, *Information and Computation*, Vol.79(1988), pp. 43-59;

[6] A. Jukan and A. Monitzer, Restoration methods for multi-service optical networks, G. de Marchis and R. Sabella (ed.), *Optical Networks: Design and Modelling*, Kluwer Academic Publishers, 1999, pp. 3–12.

[7] Algorithmic Solutions Software GmbH, The LEDA User Manual, Version 4.2.1, www.mpi-sb.mpg.de/LEDA/MANUAL.

[8] M. Médard, S.G. Finn and R.A. Barry, A novel approach to automatic protection switching using trees, *IEEE International Conference on Communications'1997*, pp. 272–276.

[9] 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 Transactions on Networking*, Vol. 7(1999), pp. 641-652;

[10] M. Medard, R.A. Barry, S.G. Finn, W. He, and S.S. Lumetta, Generalized loop-back recovery in optical mesh networks, *IEEE/ACM Transactions on Networking*, Vol. 10(2002), pp. 153–164.

[11] G. Mohan, C. S. Ram Murthy, A. K, Somani, Efficient Algorithms for Routing Dependable Connections in WDM Optical Networks, *IEEE/ACM Transactions on Networking*, Vol. 9(2001), pp. 553–566.

[12] B. Mukherjee, *Optical Communication Networks*, McGraw Hill, 1997.

[13] B. Mukherjee, WDM optical networks: progress and challenges, *IEEE Journal on Selected Areas in Communications*, Vol. 18(2000), pp. 1810–1824.

[14] I. Ouveysi, A. Wirth, On Design of a Survivable Network Architecture for Dynamic Routing: Optimal Solution Strategy and an Efficient Heuristic, *European Journal of Operational Research*, Vol. 117(1999), pp. 30-44.

[15] I. Ouveysi, A. Wirth, Minimal Complexity Heuristics for Robust Network Architecture for Dynamic Routing, *Journal of the Operational Research Society*, Vol. 50(1999), pp. 262-267.

[16] C. Qiao and D. Xu, Distributed partial information management (DPIM) schemes for survivable networks - Part I, *IEEE INFOCOM2002*, pp. 302–311.

[17] S. Ramamurthy and B. Mukherjee, Survivable WDM mesh networks. Part I-Protection *IEEE INFOCOM1999*, pp. 744–751.

[18] S. Ramamurthy and B. Mukherjee, Survivable WDM mesh networks. II. Restoration *IEEE International Conference on Communications'1999*, pp. 2023-2030.

[19] L. Sahasrabuddhe, S. Ramamurthy and B. Mukherjee, Fault management in IP-over-WDM networks: WDM protection versus IP restoration, *IEEE Journal on Selected Areas in Communications*, Vol. 20(2002), pp. 21–33.

[20] R.E. Tarjan, Depth first search and linear graph algorithms, *SIAM Journal on Computing*. Vol. 1(1972), pp. 146-160.

[21] D.B. West, *Introduction to Graph Theory*, Prentice Hall, 1996, 2001.

[22] T.H. Wu and W.I. Way, A novel passive protected SONET bidirectional self-healing ring architecture, *IEEE Journal of Lightwave Technology*, Vol. 10(1992), pp. 1314–1322

[23] G. Xue, L. Chen and K. Thulasiraman, QoS issues in redundant trees for protection in vertex-redundant or edge-redundant graphs, *IEEE International Conference on Communications'2002*, pp. 2766–2770.

[24] G. Xue, L. Chen and K. Thulasiraman, Quality of service and quality protection issues in preplanned recovery schemes using redundant trees, *IEEE JSAC series in Optical Communications and Networking*, Vol. 21(2003), pp. 1332–1345.

[25] G. Xue, R. Gottpu, and W. Zhang, Survivable Network Design using Path- based Spanners, submitted to *Computer Communications*.