Finding Multiple Maximally Redundant Trees in Linear Time

Gábor Enyedi and Gábor Rétvári
Dept. of Telecommunications and Media Informatics
Budapest University of Technology and Economics
Magyar tudósok körútja 2., Budapest, Hungary, H-1117
Email: {enyedi,retvari}@tmit.bme.hu

***Abstract*—Redundant trees are directed spanning trees, which provide disjoint paths towards their roots. Therefore, this concept is widely applied in the literature both for providing protection and load sharing. The fastest algorithm can find multiple redundant trees, a pair of them rooted at each vertex, in linear time.**

Unfortunately, edge- or vertex-redundant trees can only be found in 2-edge- or 2-vertex-connected graphs respectively. Therefore, the concept of maximally redundant trees was introduced, which can overcome this problem, and provides maximally disjoint paths towards the common root. In this paper, we propose the first linear time algorithm, which can compute a pair of maximally redundant trees rooted at not only one, but at each vertex.

***Index Terms*—redundant trees, maximally redundant trees, independent trees, colored trees, recovery trees, linear, recovery, load sharing**

I. Introduction

Communication has changed our life in the last few decades. Nowadays, people are reachable almost everywhere and it is possible to find almost any information in no time. All these new possibilities are provided by the communication networks, which influence our life more and more significantly. Moreover, it seems that this trend will not change; developments like Google Chrome OS or Microsoft Windows Azure will bring us cloud computing in some years, making the whole economy completely dependent on these networks.

Naturally, directly connecting all the resources in a communication network is impossible, therefore it is always needed to find decent path from the source to the destination(s). Obviously, it does matter, which paths are found. Finding link- or node-disjoint paths is a common desire for multiple reasons. Mostly, these disjoint paths are used for resilience, for providing connectivity even after a failure (e.g. [1], [2], [3]), but some proposals were taken, where disjoint paths are used for distributing the load in the network(e.g. [4]).

An important and widely studied possibility for finding disjoint paths is the concept of redundant trees. A pair of edge- or vertex-redundant trees rooted at a given root vertex of an undirected connected graph is a pair of directed spanning trees, directed in such a way that there is a path from each vertex to

G. Rétvári was supported by the János Bolyai Fellowship of the Hungarian Academy of Sciences.

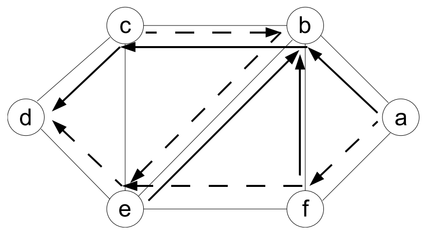


Figure 1: A pair of vertex-redundant trees rooted at vertex d.

the root on both trees and the two paths on these two trees are edge- or node-disjoint respectively. A pair of vertex-redundant trees rooted at d is depicted on Figure 1.

Redundant trees (also known as colored trees, independent trees and recovery trees) are well studied in the literature. It was first proven by Edmonds [5] that it is possible to find a pair of edge-disjoint directed spanning trees for a 2-edge-connected digraph. Later, Itai and Rodeh gave a linear time algorithm for finding both edge- and vertex-redundant trees in [6] for avoiding failures in computers with multiple CPUs. This concept was later improved by minimizing the path lengths [7], [8] and by algorithms for finding three and four trees in 3- and 4-vertex-connected graphs [9], [10], [11], [12], [13].

Médard *et. al.* applied this concept first on the field of communication [1]. Moreover, in their work they generalized the the way of computation. Based on this generalization, Xue *et. al.* endowed redundant trees with various QoS capabilities [14], [15], [16], [3], [17], [18]. Other approaches gave the possibility of computing redundant trees based on only local information [19], [20], [21], [22], [23], [24].

Even the first technique, proposed by Itai and Rodeh, computes redundant trees in linear, $O(|E(G)|)$ time, where $|E(G)|$ is the number of edges. In telecommunications, however, the task is given somewhat differently: usually a pair of redundant trees rooted at each node is needed. This is because a node usually needs to communicate with *all* the other nodes in the network. Therefore, computing all the trees is not linear, have $O(|V(G)||E(G)|)$ complexity, where $|V(G)|$ denotes the number of vertices, the nodes in the network.

On the other hand, observe that several networks base on hop-by-hop forwarding paradigm, thus knowing the whole redundant trees is not needed for these networks. In this special

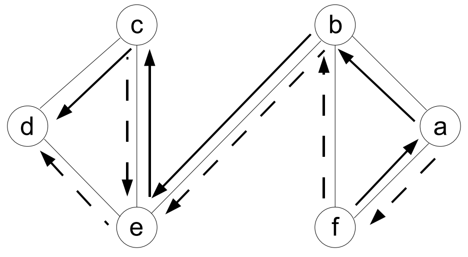


Figure 2: A pair of maximally redundant trees rooted at vertex d.

case, even a faster distributed algorithm is proposed in [25], which computes only these next hops along the redundant trees, but for all the trees rooted at each node.

Note that distributed manner in the field of redundant trees typically means token coordinated distributed computation, based on only local information. Hence, these algorithms make communication an essential part of the computation itself. In contrast, the technique presented in [25] supposes that the complete topology of the network is already explored (there is a link state routing protocol, like OSPF or IS-IS in the background), and computations in different nodes are made asynchronously without the coordination of potentially perishing tokens. This algorithm is distributed in the way that the nodes know only the edges going out from them, the next hops, but none of them knows any of the trees completely; this information is distributed in the network.

Unfortunately, edge- or vertex-redundant trees have a serious drawback: since these trees provide two edge-disjoint or vertex-disjoint paths respectively, the network must be 2-edge-connected or 2-vertex-connected in order to find such trees with an arbitrary root. Since networks are usually designed with a redundant manner, fulfilling this requirement seems possible at first, albeit redundancy can be easily lost when a failure occurs. Moreover, several real networks does not have 2-vertex-connected topology, even when they are intact (see e.g. Abeline, AT&T in [26] or Italian backbone in [27]).

Therefore, the concept of maximally redundant trees was introduced [28]. A pair of maximally redundant trees rooted at a given root vertex of an undirected graph is a pair of directed spanning trees directed in such a way that there is a directed path from each vertex to the root on both trees and the two paths on these trees has the minimum number of edges and vertices in common. This means that only the unavoidable cut-edges and cut-vertices are on both paths, therefore maximally redundant trees provide maximum redundancy in arbitrary connected graph.

A pair of maximally redundant trees rooted at d is depicted on Figure 2. As it can be observed, vertices b and e together with the edge between them is unavoidable, so both paths from a or f contain them.

The main contribution of this paper is that we first present a distributed linear time algorithm for finding a pair of maximally redundant trees rooted at not only one, but each vertex. This algorithm is an extension of the one presented in [25]. We suppose that there are $|V(G)|$ processors (these are typically the nodes of the network, $|V(G)|$ denotes the number of vertices again), all the processors have exactly the same graph as input (e.g. the topology of the network, with vertices and edges given in the same order) and each processor computes only the edges of the trees going out from its vertex. If the input graph is not the same for all the processors, some pre-computation may be needed, which is not in the scope of this paper.

Moreover, we present some heuristics as well, which do not improve the complexity of our algorithm, but significantly decrease the lengths of paths along the maximally redundant trees towards their roots. Furthermore, by improving IP fast reroute technique Lightweight Not-via, we present a potential applicability of distributed maximally redundant tree computation.

Since in this paper we describe a graph algorithm, we need some notations, which we define here. We deal only with simple graphs, where no multiple edges or loops exist. Thus, a simple graph G is a pair (V, E), where V is the set of vertices and E is the set of edges. If graph G is undirected, then $E \subseteq \{\{v_1, v_2\} : v_1, v_2 \in V\}$, so elements are unordered pairs, denoted by $\{v_1, v_2\}$ $(v_1, v_2 \in V)$. Otherwise, if G is directed, $E \subseteq V \times V$ ($\times$ denotes the Cartesian product), so elements are ordered pairs, denoted by (v_1, v_2) $(v_1, v_2 \in V)$, where v_1 is the source and v_2 is the target. Moreover, $V(G)$ and $E(G)$ denotes the set of vertices and edges of graph G. The number of elements of a given set S is denoted by $|S|$.

The rest of this paper is organized as follows. Since our algorithm is divided into three phase, we deal with the first phase, which is special DFS traversal, in the next section. In Section III, using this DFS traversal, an intermediate digraph is computed. Maximally redundant trees themselves are computed in Section IV. In Section V, some heuristics are presented for minimizing the lengths of paths on the maximally redundant trees found. The quality of this optimization is discussed in Section VI. In Section VII, we present a possible way of applying these trees for IP fast reroute. Finally, we conclude our results.

II. Phase I – DFS

As it was discussed above, our algorithm is divided into three phases. The first phase is a special Depth First Search (DFS) traversal for computing DFS and lowpoint numbers. The DFS number of a given vertex v (denoted by D_v) is the number of vertices visited by the DFS traversal before v. Therefore, the starting vertex has 0 as a DFS number. The lowpoint number of a given vertex v (denoted by L_v), which is not the starting point of the traversal, is the minimum of the lowpoint numbers of its children in the DFS tree and the DFS numbers of its neighbors. The vertex, where the DFS was started from, has no lowpoint number.

Algorithm 1 presents this modified DFS traversal, needed for computing the maximally redundant trees. A sample graph and a possible procession of Algorithm 1 is depicted on Figure 3. Observe that vertex b got the lowpoint number from its immediate parent, since the edge between e and b

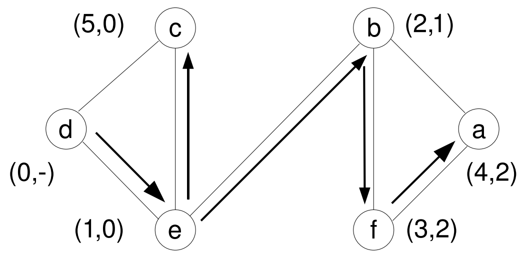


Figure 3: A possible DFS, the DFS and the lowpoint numbers.

is a cut-edge. Note that this algorithm can be implemented by slightly modifying the standard DFS traversal algorithm, thus its complexity is $O(|V(G)| + |E(G)|) = O(|E(G)|)$ (in connected graphs $|V(G)| - 1 \leq |E(G)|$).

Algorithm 1 Revised DFS for graph G and root vertex r

```
1: Start a DFS traversal from root $r$ on the graph. Set DFS number $D_v$ at each vertex $v$, so that $D_v$ denotes the number of vertices visited before $v$.
2: Recursively compute the lowpoint number for each vertex $v$ as $\min(L, D)$, where $L$ is the smallest lowpoint number of $v$'s children and $D$ is smallest DFS number among $v$'s neighbors.
3: For each vertex $v$, associate a directed edge $(v, x)$, where $x$ is the vertex from $v$ received its lowpoint number. If it is possible, choose an arbitrary child as $x$
```

Now, we define a technical lemma, which will be necessary in the sequel. Note that there is a similar lemma presented in [29]. Observe that this lemma basically tells us that walking down on the DFS tree by always selecting the child with the maximum lowpoint number leads to a neighbor of an ancestor.

Lemma 1: Let x be a vertex of an undirected connected graph. Do a DFS traversal and start it at $r \neq x$. Let the DFS parent of x be p. Than, $L_x \leq D_p$. If x is in a 2-vertex-connected component, which contains an ancestor of p, then $L_x < D_p$. Moreover, walking down as long as possible along the DFS tree from x by always selecting a child c, such that $L_x = L_c$, leads to a successor with such a neighbor y in G that

- if $L_x < D_p$, y is a DFS ancestor of p or
- if $L_x = D_p$, $y = p$.

Remark: Note that it is possible that x has no child c with $L_x = L_c$. Than we "walk down" zero hops along the DFS tree and y is a neighbor of x.

Proof: Since p is a neighbor of x, x gets its lowpoint number from p, if there is no better choice, so $L_x \leq D_p$. Now, suppose that x is in a 2-vertex-connected component, which contains an ancestor of p. Consider only this 2-vertex-connected component, a subgraph of G, let it be G'. G' is 2-vertex-connected. Let an ancestor of p in G' be a. There are two node-disjoint paths from x to a, so one of them does not contain p. Naturally, there must be a path from a to p, not containing x (the path on the DFS tree). Combining these two paths yields a walk from x to p not containing the edge between x and p. Thus, p is in G'.

Let the DFS subtree in G' rooted at x be T (so x and its successors in G' are in T). The vertices of T makes up a subset of the vertices of G'. Since there are at least 2 vertices outside T (p and a) and G' is 2-vertex-connected, there must be two $\{m, y\}$ edges, where $m \in V(T)$ and $y \in V(G') \setminus V(T)$, and the vertices in $V(G') \setminus V(T)$ of these two edges are not the same. Therefore, let $\{m, y\}$ be an edge, where $y \neq p$. Since DFS traversal has the property that the neighbor of a vertex is either an ancestor or a successor, and y is not a successor of x, y must be an ancestor of both m and x. Moreover, since $y \neq p$, y is an ancestor of p too. Thus, $L_x \leq L_m \leq D_y < D_p$.

Walking down along the DFS tree, and always selecing a child with lowpoint number L_x, leads to a successor s with a neighbor n, such that $D_n = L_s = L_x$ (the lowpoint number L_x came from n). Since n must be an ancestor of s (DFS traversal), n must be an ancestor of x too. If $L_x < D_p$, $n \neq p$, so n must be an ancestor of p. Naturally, if $D_p = L_x = D_n$, $n = p$. ■

III. Phase II – Generalized ADAG

In the second intermediate phase, a spanning digraph named Generalized Almost Directed Acyclic Graph (GADAG) is computed. This graph is a generalized version of the Almost Directed Acyclic Graph (ADAG) [28], and can be found in not only 2-vertex-connected, but arbitrary connected graphs. The naming comes from the fact that there is always a single vertex r in an ADAG, such that removing r transforms the graph into a Directed Acyclic Graph. In this section, first we give a formal definition of the Generalized ADAG, than we discuss its aspects and finally we present a linear time algorithm computing a spanning GADAG in a connected graph.

Definition 1: Let a digraph be *weakly n-vertex-connected*, if replacing its directed edges with undirected edges produces an n-vertex-connected undirected graph. Let a vertex v of a digraph be a *weak cut-vertex*, if the digraph is not weakly connected without v. Let an edge e of a digraph be a *weak cut-edge*, if the digraph is not weakly connected without e.

Remark: Note that a weak cut-edge is a directed edge with two weak cut-vertices as endpoints.

Definition 2: Let D be a strongly connected digraph with vertex r. Let the first weak cut-vertex r_x along the paths from vertex $x \neq r_x$, $x \neq r$ to r be the *local root* of x. If there is no cut-vertex between x and r (so x and r are neighbors or are in the same weakly 2-vertex-connected component), then $r_x = r$. Vertex r has no local root. Let C be the set of the maximum (here means inextensible) weakly 2-vertex-connected components of D. For all vertices $x \in V(D) \setminus \{r\}$, add x and r_x with the edges between them to C as a component, if there is no $A \in C$, so that $x, r_x \in V(A)$. Let $r_A \in V(D)$ be the local root of component $A \in C$, if $r_A = r_x$ for all $x \in V(A) \setminus \{r_A\}$. (Note that for all paths from A to r, r_A is the last vertex in A.)

D is a *Generalized ADAG (GADAG)* with r as a root, if for all $x \in V(D)$ there is a directed cycle in D containing both x and r_x, and $A \in C$ is a DAG without r_A. The *set of components of GADAG D* is set C.

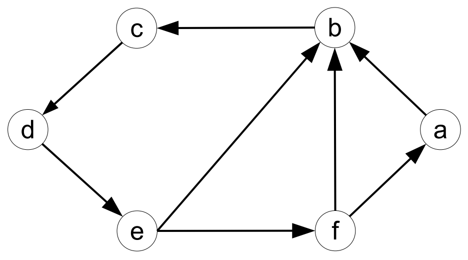


Figure 4: A GADAG with one component rooted at vertex d

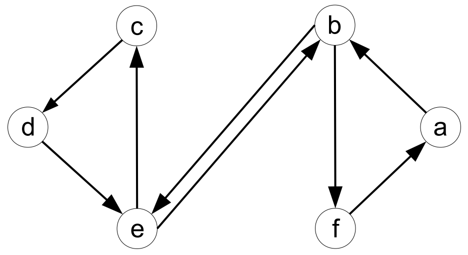


Figure 5: A GADAG with three components rooted at vertex d

Although one may find this definition a bit complicated at the first time, it is not so difficult to understand.[1] As the first example, consider the GADAG depicted in Figure 4. Since this digraph is 2-vertex-connected, set C has only one element, the complete GADAG itself. Since there is a directed cycle for each vertex, and all these cycles contains d, this digraph is definitely a GADAG.

Second, in Figure 5 a bit more complicated situation is presented. This graph is not 2-vertex-connected any more, but it is made up by two weakly 2-vertex-connected components, a, b, f (let it be component X) and c, d, e (let it be component Y). Since there is no weakly 2-vertex-connected component, which contains b and its local root e, so C also contains b and e with the two edges between them as a component (let it be component Z). It is easy to see, that $r_c = r_e = d$, $r_a = r_f = b$, $r_b = e$, $r_X = d$, $r_Y = b$ and $r_Z = e$. Trivially, for each vertex there is a directed cycle containing the vertex and its local root. Moreover, without the local root, any of the three elements of C is a DAG, so the graph depicted in Figure 5 is a GADAG with d as a root.[2]

Algorithm 2 computes the spanning GADAG of an arbitrary connected undirected graph. Before turning to deal with the specifics of this algorithm, let us discuss how it produces spanning GADAG depicted in Figure 5 using DFS traversal depicted in Figure 3. The algorithm starts from a given vertex, which is now vertex d, the root of the generated spanning GADAG. First, computes the DFS tree, the DFS numbers and the lowpoint numbers using Algorithm 1. Next, since d has a child which is not ready, the algorithm gets to branch at Line 7. By walking down along the DFS tree (Line 9), the ear (see Definition 3) containing e, c is found. Therefore, (d, e), (e, c) and (c, d) are added to D. The vertices of this ear are pushed on the top of the stack, so now it contains e, c. Moreover, $c.ready$ and $e.ready$ are set to true, $c.localRoot = d$ and $e.localRoot = d$. Since d has no more neighbor, which is not $ready$, the next vertex is removed from the top of stack S, which is e. Vertex e has a child, which is not ready, so the next ear found is b alone (b got its lowpoint number from e) and edges (e, b) and (b, e) are added to D. Now, $b.ready = true$, $b.localRoot = e$ and S contains b, c. The next element processed is b, ear f, a is found, $f.ready$ and $a.ready$ are set to true, (b, f), (f, a) and (a, b) are added to D. Although stack contains f, a, c, all the vertices are ready, so the algorithm terminates.

Definition 3: Let an *ear* be a sequence of vertices we push to the stack at the same time (Line 12 or Line 27).

Now, we prove that Algorithm 2 terminates, computes a spanning GADAG, computes the local roots and its complexity is linear. The algorithm terminates, when both branches at Line 7 and 22 terminate.

Lemma 2: The branches at Line 7 and 22 always terminate.

Proof: First, we use mathematical induction to show all DFS ancestors of an arbitrary $ready$ vertex are always marked $ready$. Initially, this is true, since only r is $ready$. Than, after finding an ear either at line 7 or at Line 22, the claim remains true, since all the ancestors of a vertex in the ear became $ready$ too.

At the end of the branch at Line 7, we always arrive to $current$ or to an ancestor of $current$, thanks to Lemma 1, so the branch at Line 7 indeed terminates. On the other hand, in the branch at Line 22 we always move upwards in the DFS tree, heading towards r. Since r is $ready$, a $ready$ vertex is always reached finally, so the branch at Line 22 also terminates. ■

Lemma 3: The output graph of Algorithm 2 is a spanning GADAG of G rooted at r.

Proof: Let the output graph be D, and create C the set of components of D as described in Definition 2 (it is possible even if D is not a GADAG). First, we deal the most complicated part of the proof, namely that for all $A \in C$, without r_A, A is a DAG. If A has only two vertices, it is trivial. Now, suppose that $|V(A)| > 2$, which means that A is weakly 2-vertex-connected.

Remove r_A from A and let this new graph be A'. Observe that in both cases when Algorithm 2 adds edges to A', the endpoints of the edges in the ear appear exactly in the same order both in the edge and in the stack. Consider an ear the algorithm finds either at Line 12 or Line 27. This ear starts at $current$ and terminates at another vertex, say, x. Since $r_A \notin V(A')$, claims about $current$, where $current = r_A$ or claims about x, where $x = r_A$ are not important (and not always true). Otherwise, the following claims hold for $current$ and x:

- $current \neq x$ (at branch 7, this is true due to Lemma 1, and at branch 22 because all the children have been made $ready$ by branch 7)

[1]Ones, who are familiar with the concept of ADAG, may think on a GADAG as several ADAGs "glued" together at the weak cut-vertices which are the roots of these components.

[2]Note that this is a very special case, since all the vertices of this graph could be the root.

Algorithm 2 Finding a spanning GADAG for graph G and root vertex r. The algorithm also computes the local root of each vertex.

```
1: Compute a DFS tree using Algorithm 1. Initialize the GADAG $D$ with the vertices of $G$ and an empty edge set. Create an empty stack $S$. Set the $ready$ bit at each vertex to $false$.
2: Set $localRoot$ at each vertex to $NULL$
3: push $r$ to $S$ and set $ready$ bit at $r$
4: while $S$ is not empty
5:     $current \leftarrow$ pop $S$
6:     for each children $n$ of $current$
7:         if $n$ is not $ready$ then
8:             while $n$ is not $ready$
9:                 let $e$ be the vertex, where $n$ got its lowpoint number from
10:                $n = e$
11:            end while
12:            Let the found vertices be $x_0 \rightarrow x_1 \rightarrow ... \rightarrow x_k$, where $x_k$ is $ready$, and $x_0$ is the neighbor of $current$. Set the $ready$ bit at $x_0, x_1, ..., x_{k-1}$ and push them to $S$ in reverse order, so eventually the top of the stack will be $x_0, x_1, ..., x_{k-1}$
13:            Add edges in the path $current \rightarrow x_0 \rightarrow x_1 \rightarrow ... \rightarrow x_k$ to $D$.
14:            if $current = x_k$ then
15:                Set $localRoot$ to $current$ at $x_0, x_1, ..., x_{k-1}$
16:            else
17:                Set $localRoot$ to $current.localRoot$ at $x_0, x_1, ..., x_{k-1}$
18:            end if
19:        end if
20:    end for
21:    for each neighbor $n$ of $current$ which is not a child
22:        if $n$ is not $ready$ then
23:            while $n$ is not $ready$
24:                let $e$ be the parent of $n$ in the DFS tree
25:                $n = e$
26:            end while
27:            Let the found vertices be $x_0 \rightarrow x_1 \rightarrow ... \rightarrow x_k$, where $x_k$ is $ready$ and $x_0$ is the neighbor of $current$. Set the $ready$ bit at $x_0, x_1, ..., x_{k-1}$ and push them to $S$ in reverse order, so eventually the top of the stack will be $x_0, x_1, ..., x_{k-1}$.
28:            Add edges in the path $current \rightarrow x_0 \rightarrow x_1 \rightarrow. ... \rightarrow x_k$ to $D$
29:            Set $localRoot$ to $x_k.localRoot$ at $x_0, x_1, ..., x_{k-1}$.
30:        end if
31:    end for
32: end while
```

- $current$ has already left the stack and
- x is still on the stack (since it has a neighbor, the last vertex of the ear, which is not ready, which is either a child or which got the lowpoint number from x).

Now, let $V = v_1, v_2, ..., v_n$ be the sequence of vertices as they leave the stack S. Observe that if there is an (v_i, v_j) edge in A', then v_i and v_j was either in the same ear or (v_i, v_j) was an end of the ear (one of the vertices was $current$ or x). According to the argumentation above, when we add edge (v_i, v_j) to A' one of the following two cases hold

- v_i has already left the stack when we push v_j or
- v_i appears above v_j in the stack.

Thus, v_i will leave the stack before v_j, which means $i < j$. Therefore, we have that for each (v_i, v_j) in A', $i < j$ holds, so V is a topological ordering, hence A' is a DAG.

Next, we use mathematical induction in order to prove that D is strongly connected, and for each $A \in C$, $v \in V(A)$, there is a directed cycle, which contains both r_A and v. Initially, when D contains only r, the claim is true. Suppose that after adding some ears it is still true.

Now, we add a new ear from $current$ to x. There must be a path from r to $current$ and one from x to r thanks to strong connectivity, so strong connectivity is conserved. Moreover, the path from v to r_v and the path from r_v to v must be vertex-disjoint, since otherwise there would be a directed cycle in A not containing $r_v = r_A$. Therefore, combining these two paths makes up a cycle containing both v and r_v.

Now, we have seen that D is a GADAG. In order to prove that this is a spanning GADAG of G, it is needed to observe that all the vertices of G becomes $ready$. Since a $ready$ vertex leaves the stack sooner or later, a DFS child of a $ready$ vertex must be also $ready$ when Algorithm 2 terminates. Since the root of the DFS tree is $ready$, and since the graph and the DFS tree is connected, all the vertices must be $ready$, when the algorithm terminates. ∎

Lemma 4: Algorithm 2 computes the local roots for all vertices correctly.

Proof: Observe that the first vertex of a component A leaving S is r_A. Moreover, each neighbor $n \in V(A)$ of r_A has lowpoint number $L_n = D_{r_A}$, since either $r_A = r$ or all path to a DFS ancestor of r_A contains r_A as a cut-vertex. Moreover, $current$ can not be the same as x_k for ears, which are found at Line 22, since those were already found at Line 7. Thus, the $localRoot$ is set properly in the case of entering into a new component.

Inside a component, $current$ and x_k cannot be the same, since

- if the ear is found at Line 7, x_k is an ancestor of $current$ thanks to Lemma 1 and
- if the ear is found at Line 22, all of the children of x_k is already $ready$.

Therefore, at most one of the endpoints can be r_A. At Line 12 x_k is an ancestor of $current$ (Lemma 1), at Line 27 $current$ is an ancestor of x_k (since we get to a successor and walk up), so the local root is set properly both at Line 14 and at Line 29.

Naturally, there is no ear between two components, since there are a cut-vertex between them. ∎

Lemma 5: The computational complexity of Algorithm 2 is $O(|E(G)|)$.

Proof: Each vertex is pushed to S and popped from S once, so the most important part of the algorithm is at Line 7 and at Line 22, where the ears are found. Either walking down along the DFS tree, and always selecting the pre-computed vertex, which the lowpoint number came from, or walking upwards selecting the parent takes $O(|V(G)|)$ time all together.

Since a DFS traversal is needed, and the graph is supposed to be connected, the overall complexity is $O(|E(G)|)$. ∎

IV. Phase III – Computing maximally redundant trees

Previously, an intermediate graph representation called GADAG was discussed. In this section we use this digraph in order to compute the maximally redundant trees themselves.

As it was discussed previously, our algorithm is distributed in such a way, that it does not compute all the maximally redundant trees, but only the edges belonging to the trees going out from a given vertex. Although interleaving these edges makes up all the maximally redundant trees, it is not necessary for most of the networks, since usually only the next hops are needed.

Our algorithm is on the traces of [25], where an algorithm computing redundant trees for a 2-vertex-connected graphs was proposed. In contrast, this new algorithm can be considered as an extension of the one in [25]; in the first phase we compute the edges belonging to the trees rooted at vertices, which are in the same 2-vertex-connected component (this is almost the same as in [25]), than we find the cut-vertices, which the other vertices can be reached throw, and use the previously computed edges for the remaining trees. This idea is presented in Algorithm 4.

Before turning to discuss the issues of this algorithm, let us present a simple example. Consider the previous graph depicted in Figure 3. As we know, the GADAG rooted at d is depicted in Figure 5. Note that since G and the global root (which in this special case is vertex d) are the same for each node (processor), each node computes exactly the same GADAG. Next, consider the processor of f and compute the edges going out from f. We split r_f, which is vertex b to b^+ and b^-, so that edges only enter to b^+ and only leave b^-.

Than, in the first phase we do a Breadth First Search (BFS) traversal started from f taking the edges in normal direction, and visit all the vertices in the same component, these are a and b^+. These are the vertices *greater* than f (see Definition 4), so $a.V^+$ and $b^+.V^+$ are set to $true$. The BFS taking the edges in reverse direction finds b^-, which is the only vertex *less* than f, so $b^-.V^- = true$. Finally, Phase 1 computes the edges going out from f belonging to the trees rooted at a and b. Now, $h_f^P(b) = (f, a)$, $h_f^S(b) = (f, b)$, $h_f^P(a) = (f, a)$, $h_f^S(a) = h_f^S(b) = (f, b)$.

Phase 2 computes the edges for the remaining vertices. First, $h_f^P(d) = h_f^P(b) = (f, a)$ and $h_f^S(d) = h_f^S(b) = (f, b)$ is set.

Procedure 3 SetEdge(vertex x)

```
1: # Both, or neither is $NULL$
2: if $h_u^P(x) = NULL \wedge h_u^S(x) = NULL$ then
3:     SetEdge($r_x$)
4:     $h_u^P(x) = h_u^P(r_x)$
5:     $h_u^S(x) = h_u^S(r_x)$
6: end if
```

Algorithm 4 Computing the primary and secondary edges for all root d, ($h_u^P(d)$, $h_u^S(d)$) going out from vertex u.

```
1: For all $d \in V(G)$ set $h_u^P(d) = NULL$ and $h_u^S(d) = NULL$. Use Algorithm 2 for computing a spanning GADAG $D$ with a given $r$ as root ($G$ and $r$ are exactly the same for each node, so the found GADAG is the same). Create digraph $D'$ by splitting the local root $r_u$ into two vertices, so that edges only enter to vertex $r_u^+$ and only leave $r_u^-$. For each vertex $x$ set $x.V^+ = false$ and $x.V^- = false$. If $u = r$ ($r$ has no local root), do not split any of the vertices.
2:
3:         # Phase 1: vertices in the same component
4:
5: Do a BFS traversal on $D'$ from $u$ taking the edges in normal direction. Do not visit vertex $x$, if $x \neq r_u^+ \wedge x.localRoot \neq u \wedge x.localRoot \neq u.localRoot$. At visited vertex $x$ set $x.V^+ = true$, and set $h_u^P(x)$ to the first edge along the path to $x$ computed by the BFS.
6: Do a BFS traversal on $D'$ from $u$ taking the edges in reverse direction. Do not visit vertex $x$, if $x \neq r_u^- \wedge x.localRoot \neq u \wedge x.localRoot \neq u.localRoot$. At visited vertex $x$ set $x.V^- = true$, and set $h_u^S(x)$ to the first edge along the path to $x$ computed by the BFS.
7: set $h_u^P(r_u) = h_u^P(r_u^+)$
8: set $h_u^S(r_u) = h_u^S(r_u^-)$
9: for all vertex $x \neq u$, $x.localRoot = u.localRoot$
10:     if $x.V^+ = true$ then
11:         set $h_u^S(x) = h_u^S(r_u)$
12:     else if $x.V^- = true$ then
13:         set $h_u^P(x) = h_u^P(r_u)$
14:     else
15:         set $h_u^P(x) = h_u^S(r_u)$
16:         set $h_u^S(x) = h_u^P(r_u)$
17:     end if
18: end for
19:
20:         # Phase 2: other components
21:
22: set $h_u^P(r) = h_u^P(r_u)$
23: set $h_u^S(r) = h_u^S(r_u)$
24: for all vertex $x \neq r \wedge x \neq u$
25:     SetEdge($x$)
26: end for
```

Than, suppose that next c is processed. Procedure 3 is called, which sets $h_f^P(c) = h_f^P(d) = (f,a)$ and $h_f^S(c) = h_f^S(d) = (f,b)$. Finally, $h_f^P(e) = h_f^P(d) = (f,a)$ and $h_f^S(e) = h_f^S(d) = (f,b)$ are set. All the computed edges are presented on Table I, but note that any given node computes only a *single row* of this table.

Definition 4: Let D be a spanning GADAG of graph G with the component set C, and let $A \in C$. Split the root vertex r_A in A into two vertices, r_A^+ and r_A^-, in such a way that edges only enter to r_A^+ and only leave r_A^-. Let this new graph be A'. Define a relation $(\prec)$ on $V(A')$ as follows: $u \prec v$ if and only if there is a directed path from u to v in A' $(u, v \in V(A'))$.

Generalize this relation; for given vertex x and y let $x \preceq y$ be true, if $x \prec y$ or $x \equiv y$. Let V_u^+ and V_u^- be the set of vertices definitely greater and definitely less than u.

Remark: It is easy to see that $(V(A'), (\preceq))$ makes up a bounded partially ordered set (poset); since A' is a DAG, $(\preceq)$ is reflexive, transitive and antisymmetric. Additionally, since edges only leave r_A^-, the minimum element is exactly r_A^-. Similarly, r_A^+ is the maximum element.

For proving the correctness and completeness of Algorithm 4, one more simple lemma is needed.

Lemma 6: In a spanning GADAG with component set C found by Algorithm 2, there is exactly one edge entering r_A in each component $A \in C$.

Proof: If $|V(A)| = 2$, the claim is trivial. If $|V(A)| > 2$, A is weakly 2-vertex-connected, so all the vertices of A can be reached in the original undirected graph without r_A. Thus, when the DFS enters to A throw r_A, it gets back to r_A only when all the vertices of A are visited, so r_A has only one child. Moreover, when $current = r_A$, Algorithm 2 finds all the ears in A with r_A as endpoint. Since there is only one child, there is only one among these ears, which has r_A as both endpoints. Therefore, there must be only one edge, which enters r_A. ■

Theorem 1: Let an undirected connected graph G and vertex d be given. For all $u \in V(G)$, interleaving the edges $h_u^P(d)$ and $h_u^S(d)$ computed by Algorithm 4 makes up a pair of maximally redundant trees rooted at d.

Proof: Let D be the computed GADAG, and let its global root be r. Let the set of components of D be C. In this proof we will use the ordering in Definition 4. Since this proof is complicated we divided it into three parts: the algorithm terminates, the computed edges inside a component make up two vertex-disjoint paths and the computed edges make up maximally redundant paths to other vertices.

The algorithm terminates: First, we prove that Algorithm 4 always terminates and computes two edges, $h_u^P(d)$ and $h_u^S(d)$, for any given vertex d. It is trivial that Phase 1 always terminates. Suppose that there is vertex d, such that there is $A \in C$, $d, u \in A$ and either $h_u^P(d)$ or $h_u^S(d)$ is still $NULL$ after Phase 1. If $u = r_d$, both traversals reach d, so $h_u^P(d)$ and $h_u^S(d)$ are set. Otherwise, if $d.localRoot = u.localRoot$, both $d.V^+$ and $d.V^-$ cannot be $true$, since in this case both BFS traversals would reach d, which is impossible, since all the cycles in A contains $r_u = r_A$. If only one of $d.V^+$ and $d.V^-$ is $true$, then one of the edges is computed by the BFS traversals, the other one is set at Line 11 or Line 13. Since if none of them is $true$, the edges are set at Line 15, the only possibility is $d = r_u$. However, $r_u^+.V^+ = true$ and $r_u^-.V^- = true$, so both $h_u^P(r_u)$ and $h_u^P(r_u)$ are set at Line 7.

Phase 2 terminates if Procedure 3 terminates. Since the recursion gets always to the local root, sooner or later r is reached. Since r is already computed, Phase 2 terminates. Trivially, both $h_u^P(d)$ and $h_u^S(d)$ are computed.

Interleaving the edges makes up two vertex-disjoint paths inside a component: First suppose that there is $A \in C$, such that $u, d \in A$. In this case interleaving the edges must make up a pair of vertex-disjoint paths towards the root.

We show that for two vertices $v, w : v \prec w$, what we obtain by following the primary outgoing edges $h^P(w)$ is a loop-free $v \to w$ path. Let $h_v^P(w) = (v, x)$. Using this edge, we either get to w, when $x = w$, or get to a vertex x where $v \prec x$. Moreover, $x \prec w$, since (v, x) is the first edge along a path from v to w, so there is a path from x to w too. Therefore, if $x \neq w$, we can repeat the same reasoning till we eventually arrive to w. Along the similar lines, following $h^S(w)$ yields a loop-free $v \to w$ path for $v, w : v \succ w$.

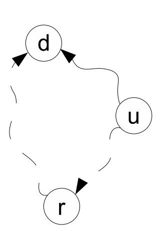
If $d = r_A$ or $u = r_A$, the two paths are trivially disjoint. Suppose $d \neq r_A$, $u \neq r_A$ and there is an ordering between u and d, say $u \prec d$. Now, following $h^P(d)$ yields an $u \to d$ path p_p (the path marked by solid arrow in Figure 6a), and following $h^S(d)$ yields first a $u \to r_A^-$ path p_s^1 and then an $r_A^+ \to d$ path p_s^2 (dashed arrow in Figure 6a). Based on the observation above, these subpaths are indeed paths and they are loop-free. The concatenation of p_s^1 and p_s^2 gives the secondary path p_s. Finally, p_p and p_s are vertex-disjoint: vertices along p_p belong to the interval $[u, d]$, p_s^1 to $[r_A^-, u]$ and p_s^2 to $[d, r_A^+]$, and these intervals are disjunct except the endpoints.

If there is no ordering between u and d, the situation is slightly more difficult: following $h^P(d)$ first yields an $u \to x$ path p_p^1 and then a $x \to d$ path p_p^2, where x is the first vertex, which $u \succ x$ and $x \prec d$ holds for (see the dashed arrows in Figure 6b). Similarly, $h^S(d)$ yields first a $u \to y$ path p_s^1 and then an $y \to d$ path p_s^2 for the first $y : u \prec y$ and $y \succ d$ (solid arrows in Figure 6b). Again, concatenation of the corresponding subpaths yields two vertex-disjoint paths: first, p_p^1 and p_s^1 are vertex-disjoint because $p_p^1 \in V_u^-$, $p_s^1 \in V_u^+$ and $V_u^- \cap V_u^+ = \emptyset$; second, p_p^1 and p_s^2 are also vertex-disjoint because the vertices of p_p^1 are not ordered with respect to d but those of p_s^2 are; third, p_p and p_s cannot both traverse r, because $r^+ \succ y$ (since only one edge goes into r (Lemma 6), we have a vertex m for which $m \succ v : v \in V \setminus \{r^+, m\}$, so the secondary path turns back in m at the very latest). Similar reasoning applies to see that the rest of the subpaths are mutually vertex-disjoint too.

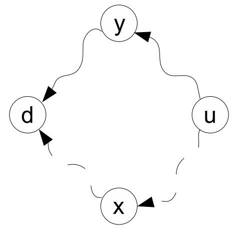
Interleaving edges makes up maximally redundant paths: Suppose that u and d are not in the same component. Create digraph T, and let $V(T) = C \cup \{r\}$. For all $A \in C$, $r \in A$ add edge (A, r) to T. Moreover, for all $A, B \in C$, $r_A \neq r_B \wedge r_B \in A$ add edge (B, A) to T. Since a local root is always on the path to r, r is reachable on a directed path from any vertex in T, so T is weakly connected. Moreover, T is a directed tree, since every component has only one local root, so there is only one edge going out from each $x \in V(T) \setminus \{r\}$ and no edge leaves r, so $|E(T)| = |V(T)| - 1$.

Vertex	$h^P(a)$	$h^S(a)$	$h^P(b)$	$h^S(b)$	$h^P(c)$	$h^S(c)$	$h^P(d)$	$h^S(d)$	$h^P(e)$	$h^S(e)$	$h^P(f)$	$h^S(f)$
a	–	–	(a,b)	(a,f)	(a,b)	(a,f)	(a,b)	(a,f)	(a,b)	(a,f)	(a,b)	(a,f)
b	(b,f)	(b,a)	–	–	(b,e)	(b,e)	(b,e)	(b,e)	(b,e)	(b,e)	(b,f)	(b,a)
c	(c,d)	(c,e)	(c,d)	(c,e)	–	–	(c,d)	(c,e)	(c,d)	(c,e)	(c,d)	(c,e)
d	(d,e)	(d,c)	(d,e)	(d,c)	(d,e)	(d,c)	–	–	(d,e)	(d,c)	(d,e)	(d,c)
e	(e,b)	(e,b)	(e,b)	(e,b)	(e,c)	(e,d)	(e,c)	(e,d)	–	–	(e,b)	(e,b)
f	(f,a)	(f,b)	(f,a)	(f,b)	(f,a)	(f,b)	(f,a)	(f,b)	(f,a)	(f,b)	–	–

Table I: The edges of maximally redundant trees computed using GADAG depicted in Figure 5.



(a) ordered case



(b) unordered case

Figure 6: Illustration for Theorem 1.

Let the component closest to r in T containing u be U, and similarly let the closest component containing d be D (here closest means that the path from U or D to r has minimum number of vertices). If U is on the path from D to r in T, Procedure 3 finds cut-vertex x in the component closest to D containing u, and sets $h_u^P(d) = h_u^P(x)$ and $h_u^S(d) = h_u^S(x)$. Since any path from u to d contains x, the walks leave each component at the right vertex, so both walks are paths and reach d. If U is not on the path from D to r, then Procedure 3 sets $h_u^P(d) = h_u^P(r)$ and $h_u^S(d) = h_u^S(r)$. In this way, the walks go up towards r in T until it reaches the first vertex X, which is on the path from D to r, so there is no cycle again, and the walks are paths. Since the paths inside a component are vertex-disjoint, the two paths are maximally vertex-disjoint. ■

Finally, we only need to show that Algorithm 4 is linear in the number of edges.

Theorem 2: The computational complexity of Algorithm 4 is $O(|E(G)|)$ for any connected graph.

Proof: Computing GADAG D and doing the BFS traversals need $O(|E(G)|)$ time. The main question is the complexity of Procedure 3. Each time Procedure 3 is called recursively (from the procedure), a vertex x is needed with $h_u^P(x) = NULL$ and $h_u^S(x) = NULL$, so it can be called recursively at most $O(|V(G)|)$ times altogether. Since it is called from Algorithm 4 $|V(G)| - 2$ times, the overall complexity of the algorithm is $O(|V(G)| + |E(G)|) = O(|E(G)|)$ (the graph is connected). ■

V. Optimization

Previously, an algorithm finding a pair of maximally redundant trees rooted at each of the vertices was discussed. There, the attributes of the trees found were not considered important. Unfortunately, this is usually not true.

A very common requirement, paths in networks must meet, is to minimize the length of paths with respect to some edge length function. Probably, the most important networks, where this this kind of optimization is needed are the IP networks. Unfortunately, minimizing paths with arbitrary length function can not be done in linear time but only in $O(|V(G)|\log|V(G)| + |E(G)|)$ with Dijkstra's algorithm. In contrast, if edges have uniform lengths, BFS traversal can also find the shortest paths in linear, $O(|E(G)|)$, time; these are the paths containing the minimum number of vertices.

Observe that the situation is the same for maximally redundant trees as well. When the spanning GADAG is computed, Algorithm 4 computes the paths to greater and lesser vertices at Line 5 and Line 6 using BFS. In this way, optimal always increasing and always decreasing paths are found in the GADAG for uniform edge length, but for arbitrary lengths these paths are suboptimal.

Fortunately, the exact way of finding increasing and decreasing paths is not important, therefore the BFS traversals can be exchanged to two runs of Dijkstra's algorithm. However, this is a trade-off, since using Dijkstra's algorithm would ruin linearity. In this paper we choose conserving linear complexity even with path minimization, thus we assume uniformly 1 edge lengths in the sequel (we minimize the number of vertices along the paths). Nevertheless, all the following techniques can be applied with simply exchanging the BFS traversals to Dijkstra's algorithm, but in this case the overall complexity becomes $O(|V(G)|\log|V(G)| + |E(G)|)$.

Assuming uniform edge lengths, the most important aspect influencing the number of vertices along the paths of maximally redundant trees is the spanning GADAG; using a "better" GADAG, BFS traversals can find better paths. Observe that when a GADAG is found, there can be some edges in the original graph, which are not used in either direction. Adding these edges in a direction, which keeps up the GADAG property may reduce the length of paths.

Moreover, observe that for any vertex v, optimizing the whole spanning GADAG is not necessary; it is enough to add some edges to the components of the GADAG, which contain v. Since paths towards vertices in different components are just paths towards a decent cut-vertex, optimizing the paths in the local components optimizes all the paths.

Unfortunately, note that simply keeping up the GADAG property is not enough, since Algorithm 4 needs special spanning GADAG, which fulfills Lemma 6 too. Therefore, adding edge in the direction entering the local root must also be avoided.

Considering these observations, it is possible to construct some simple linear time heuristics for some vertex v:

- Compute GADAG D of graph G with set of components C.
- For all $A \in C$, where $v \in V(A)$, remove the single edge entering into r_A. Make a topological ordering where r_A is the minimum element. Edges of G, used in D in neither

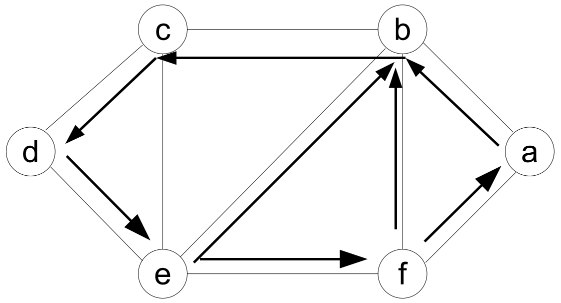


Figure 7: A sample graph and a spanning GADAG.

direction, can be added in a direction such that the source is the lower, the target is the higher vertex with respect to the topological ordering.

Trivially, in this way the GADAG property is kept up, and no new edge entering a local root is added. Moreover, since topological ordering is linear, these heuristics do not increase the complexity of the algorithm.

As a simple example, consider the graph depicted on Figure 7. Observe that edge $\{c, e\}$ is used in neither direction, so use the optimization. Since there is only one weakly 2-vertex-connected component, the optimization is the same for all the vertices. With removing the single edge entering to root d, there can be one topological ordering with d as the minimum element: d, e, f, a, b, c. Since $e \prec c$ with respect to this ordering, edge (e, c) can be added to the GADAG.

Note that these heuristics can not only decrease but sometimes increase the length of paths. Trivially, when two vertices in the same component are ordered, these heuristics definitely cannot increase the path length. On the other hand, when vertices are not ordered, the turning point, vertex x and y on Figure 6b, may get closer to the local root and farther from d. In the next section we prove by extensive simulations that in average these heuristics decrease the lengths of paths significantly.

VI. Evaluation

In the previous section some linear time heuristics were proposed for decreasing the path lengths in maximally redundant trees (recall, that uniform link cost is supposed). Unfortunatelly, in some special cases using these heursitics the lengths of paths can increase. In this section we use extensive simulations in order to prove that in average our heuristics singnificantly shorten the paths.

Since redundant trees can be applied in communication networks, we used the topology of real and randomly generated artificial networks. The selected real networks are the Abilene, NSF, AT&T and 50 node German backbone network from [26], and the Italian, German and European Cost266 backbone network from [27]. For each of these networks, we computed the maximally redundant trees with respect to each vertex as root, and we averaged the length of the resultant paths.

Random network topologies were generated by Boston university Representative Internet Topology gEnerator (BRITE) [30], using Waxman's algorithm, with random node placement and parameters $\alpha = 0.15$ and $\beta = 0.2$. The root of the generated GADAG was selected randomly in each case. The number of node varied between 20 and 50 and the number of neighbors was 2 and 3. In each case we made 250 000 random experiments in order to well approximate the expected value of the lengths of paths with the mean of the results.

Since several real topologies are 2-vertex-connected, when no failure exists, for these topologies we computed two optimal vertex disjoint paths using Suurballe's algorithm. Moreover, we also implemented the heuristics proposed by Xue *et. al.* in [15], [3] for minimizing the path lengths of redundant trees. The mean of the lengths of path pairs computed by these two algorithms and the lengths of paths computed by Algorithm 4 with and without heuristics are presented on Table II and Table III.

One may observe that paths get significantly shorter when the heuristics poposed in Section V are applied. Unfortunately, these paths are significantly longer than the optimal ones are. Thus, we can identify an interesting trade-off here: using our maximally redundant tree algorithm instead of Suurballe's algorithm or Xue's heuristics is clearly advantageous in performance-sensitive applications, because its complexity is much smaller (linear, $O(|E|)$) than that of Suurballe's algorithm (for all the vertex pairs $O(|V(G)|^3 \log |V(G)|)$) or that of Xue's heuristics (a tree rooted at each vertex is $O(|V(G)|^3(|E(G)|+|V(G)| \log |V(G)|))$). On the other hand, our technique gives suboptimal protection paths, whose length may be significantly larger than the optimal path length. Our simulations reveal that the increase is at most two-fold, which is not necessarily poses difficulties if these paths are only used for protection in out-of-order situations, which, supposedly, only last a couple of seconds, and the default paths can still be optimal shortest paths. But perhaps most importantly, our algorithm is much better suited to certain applications, namely those based on the hop-by-hop forwarding paradigm like IP, because in these applications we only need the next-hops along the recovery trees instead of the entire protection paths as returned by Suurballe's or Xue's algorithm. In the next section, we present such an application.

VII. Lightweight Not-via

Previously, the way of computing a pair of maximally redundant trees rooted at each node was discussed. In this section we present an application of this concept. Observe, that maximally redundant trees can also be applied, where redundant trees are used, so the concept proposed in this paper is not limited to this example. Moreover, note that maximally redundant trees can be especially useful for providing 1+1 protection or load sharing.

Lightweight Not-via [25] is an advanced variant of the IP Fast ReRoute (IPFRR) [31] mechanism named Not-via addresses [32]. Since IPFRR, Not-via and Lightweight Not-via is not in the main scope of this paper, here we only briefly discuss them.

Nowadays, significant efforts are taken in order to endow traditional IP with protection capabilities. Since traditional IP is based only restoration techniques like OSPF [33] or

Network	Node number	Suurballe	Xue	Prim. path without heur.	Sec. path without heur.	Prim. path with heur.	Sec. path with heur.
Abilene	12	–	–	210%	212%	168%	171%
Germany	17	135%	136%	231%	230%	191%	190%
AT&T	22	–	–	221%	224%	166%	167%
NSF	26	121%	124%	224%	222%	178%	174%
Italy	33	–	–	248%	247%	175%	174%
Cost266	37	129%	154%	250%	253%	190%	194%
Germany50	50	118%	160%	304%	309%	212%	214%

Table II: Average number of vertices on paths of maximally redundant trees in real word networks (100% is the path with minimum number of vertices).

Node number	Neighbors	Suurballe	Xue	Prim. path without heur.	Sec. path without heur.	Prim. path with heur.	Sec. path with heur.
20	2	120%	147%	217%	224%	173%	174%
20	3	116%	155%	298%	313%	180%	181%
30	2	120%	147%	235%	243%	182%	182%
30	3	115%	152%	332%	352%	190%	190%
40	2	119%	148%	250%	259%	190%	189%
40	3	114%	151%	361%	385%	198%	197%
50	2	118%	148%	263%	273%	197%	195%
50	3	113%	150%	388%	415%	205%	203%

Table III: Average number of vertices on paths of maximally redundant trees in artificial networks (100% is the path with minimum number of vertices).

IS-IS [34], its recovery capabilities prove themselves to be insufficient more and more often with the spreading of real time traffic, like IPTV, VoIP, stock exchange transactions or on-line gaming. Therefore, IPFRR techniques are expected to provide recovery fast enough even for these applications; usually it is said that IPFRR must provide recovery in 50 ms at most, like SDH/SONET [35] does.

In contrast to traditional IP recovery, IPFRR mechanisms are always *proactive* and reroute packets *locally*. Proactive manner means that the way of avoiding a given resource is computed long before any failure shows up. Local rerouting describes the manner that routers using IPFRR do not need to advertise the fact of the failure (since it needs some time), and packets can be rerouted, when only the neighbors of the failed resource know the presence of the failure.

Not-via uses special IP addresses, called not-via addresses, in order to provide local rerouting. When a failure occurs, the neighbor of the failed resource puts the packets in an IP-in-IP tunnel with a special destination address. This address describes not only the endpoint of the tunnel, where packets are needed to be decapsulated, but also the failed resource. Therefore, Not-via needs significant number of protection addresses. Fortunately, since these addresses are not used globally, local IP address domains (like 10.x.x.x or 192.168.x.x) can be used. On the other hand, this high number of IP addresses rises significant management problems.

In order to mitigate these management problems, Lightweight Not-via was proposed. Lightweight Not-via uses vertex-redundant trees for recovery. If there is no failure in the network, packets are forwarded along the shortest paths, like Not-via does. Moreover, if a failure occurs, packets are encapsulated into an IP-in-IP tunnel with a special destination address. In contrast to traditional Not-via, the recovery addresses of Lightweight Not-via do not describe exactly the failed resource, instead it describes a vertex-redundant tree. Since in the case of a single link or node failure, the root of a pair of vertex-redundant trees can be reached on at least one of the trees, Lightweight Not-via can protect all the single failures.

As an example, consider the graph depicted on Figure 1 as a network, and suppose that node a tries to send some packets to d. Moreover, suppose that the shortest path from a to d is $a \to b \to c \to d$ and node c is down. Node a does not know anything about the failure, since we have local rerouting, therefore it sends packets to b as usual. Node b is the neighbor of the failed resource, so it reroutes the packets locally. Since it knows that both the shortest path and the redundant tree depicted by solid arrows are failed, it encapsulates the packets into an IP-in-IP tunnel with a special IP address telling all the nodes to forward the packet to d along the dashed redundant tree rooted at d. Since the trees are vertex-redundant, packets reach d along path $a \to b \to e \to d$. If link $\{c, d\}$ is the failed resource, c is the one rerouting, and packets are forwarded along path $a \to b \to c \to b \to e \to d$. Note that not always the destination, but the next-next hop is the endpoint of the tunnel, since this behavior makes the failure as local as possible.

Naturally, this technique can be applied only in 2-vertex-connected networks, since vertex-redundant trees are needed. In contrast, if we simply change redundant trees to maximally redundant trees this limitation is lifted. Although finding the 2-vertex-connected components and sending packets their exit points could help for plain redundant trees, observe that this is exactly what is done by the algorithm finding maximally redundant trees. On the other hand, using the concept of maximally redundant trees, gives not only a cleaner solution, which is easier to debug, but makes it unnecessary to differentiate between normal and cut-vertices, which problem is known as "bridge problem", and rises several special cases.

VIII. Conclusions

In this paper we improved the concept of redundant trees. In contrast to redundant trees which can be found only in 2-edge- or 2-vertex-connected graphs, maximally redundant trees can be found in arbitrary connected graphs. Since maximally redundant trees can provide maximum redundancy for all networks, they can be applied in any connected network, even in ones, where redundant trees can not be found. Moreover, since maximally redundant trees are edge-redundant or vertex-redundant, when such trees exist, these trees can be applied with no modification in any network using traditional redundant trees. Moreover, we presented a linear time algorithm for computing the maximally redundant trees rooted at not only one, but all the vertices.

Using the results of this paper, we improved Lightweight Not-via, an advanced version of IPFRR mechanism Not-via, which has probably the most significant IETF and industrial backing currently. Naturally, our results are not limited to IPFRR; since redundant trees are applied in various environments such as sensor networks or optical networks, maximally redundant trees can be applied in the same fields too.

References

[1] M. Médard, R. A. Barry, S. G. Finn, and R. G. Galler, “Redundant trees for preplanned recovery in arbitary vertex-redundant or edge-redundant graphs.” *IEEE/ACM Transactions on Networking*, vol. 7, no. 5, pp. 641–652, Oct 1999.

[2] M. Médard, 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, no. 1, pp. 153–164, Feb 2002.

[3] G. Xue, L. Chen, and K. Thulasiraman, “Quality-of-service and quality-of-protection issues in preplanned recovery schemes using redundant trees,” *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 8, pp. 1332–1345, October 2003.

[4] R. D. D. Wang, G. Li, “Igp weight setting in multimedia ip networks,” in *IEEE Infocom Mini’07*, 2007.

[5] J. Edmonds, “Edge-disjoint branchings,” *Combinatorial Algorithms*, pp. 91–96, 1973.

[6] A. Itai and M. Rodeh, “The multi-tree approach to reliability in distributed networks,” in *SFCS ’84: Proceedings of the 25th Annual Symposium on Foundations of Computer Science, 1984*. Washington, DC, USA: IEEE Computer Society, 1984, pp. 137–147.

[7] D. Handke, “Independent tree spanners,” in *Graph-Theoretic Concepts in Computer Science*, 1998, pp. 203–214.

[8] F. Annexstein, K. Berman, and R. Swaminathan, “Independent spanning trees with small stretch factors,” Tech. Rep., 1996.

[9] A. Zehavi and A. Itai, “Three tree-paths,” *Journal of Graph Theory*, vol. 13, no. 2, pp. 175–188, 1989.

[10] A. Huck, “Independent trees in graphs,” *Graphs and Combinatorics*, vol. 10, no. 1, pp. 29–45, 1994.

[11] K. Miura, D. Takahashi, S.-I.Nakano, and T. Nishizeki, “A linear-time algorithm to find four independent spanning trees in four-connected planar graphs,” in *WG ’98: Proceedings of the 24th International Workshop on Graph-Theoretic Concepts in Computer Science*. London, UK: Springer-Verlag, 1998, pp. 310–323.

[12] S. Curran, O. Lee, and X. Yu, “Chain decompositions and independent trees in 4-connected graphs,” in *SODA ’03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003, pp. 186–191.

[13] S. Curran, O. Lee, and X. Yu, “Finding four independent trees,” *SIAM Journal on Computing*, vol. 35, no. 5, pp. 1023–1058, 2006.

[14] G. Xue, L. Chen, and K. Thulasiraman, “Qos issues in redundant trees for protection in vertex-redundant or edge-redundant graphs,” in *IEEE International Conference on Communications (ICC)*, vol. 5, 2002, pp. 2766–2770.

[15] G. Xue, L. Chen, and K. Thulasiraman, “Delay reduction in redundant trees for preplanned protection against single link/node failure in 2-connected graphs,” in *IEEE Globecom*, November 2002.

[16] G. Xue, L. Chen, and K. Thulasiraman, “Cost minimization in redundant trees for protection in vertex-redundant or edge-redundant graphs,” in *PCC ’02: Proceedings of the Performance, Computing, and Communications Conference, 2002. on 21st IEEE International*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 187–194.

[17] W. Zhang, G. Xue, J. Tang, and K. Thulasiraman, “Linear time construction of redundant trees for recovery schemes enhancing QoP and QoS,” INFOCOM 2005, pp. 2702–2710, March 2005.

[18] W. Zhang, G. Xue, J. Tang, and K. Thulasiraman, “Faster algorithms for construction of recovery trees enhancing qop and qos,” *IEEE/ACM Trans on Networking*, vol. 16, no. 3, pp. 642–655, 2008.

[19] S. Ramasubramanian, “Supporting multiple protection strategies in optical networks,” Department of Electrical and Computer Engineering, University of Arizona, Tech. Rep., November 2004.

[20] P. Thulasiraman, S. Ramasubramanian, and M. Krunz, “Disjoint multipath routing in dual homing networks using colored trees,” in *IEEE Globecom*, November/December 2006, pp. 1–5.

[21] R. Balasubramanian and S. Ramasubramanian, “Minimizing average path cost in colored trees for disjoint multipath routing,” in *15th International Conference on Computer Communications and Networks, ICCCN 2006*, October 2006, pp. 185–190.

[22] S. Ramasubramanian, H. Krishnamoorthy, and M. Krunz, “Disjoint multipath routing using colored trees,” *Computer Networks*, vol. 51, no. 8, pp. 2163–2180, 2007.

[23] S. Ramasubramanian, M. Harkara, and M. Krunz, “Linear time distributed construction of colored trees for disjoint multipath routing,” *Computer Networks*, vol. 51, no. 10, pp. 2854–2866, 2007.

[24] G. Jayavelu, S. Ramasubramanian, and O. Younis, “Maintaining colored trees for disjoint multipath routing under node failures,” *IEEE/ACM Transactions on Networking*, vol. 17, no. 1, pp. 346–359, 2009.

[25] G. Enyedi, P. Szilágyi, G. Rétvári, and A. Császár, “Ip fast reroute: Lightweight not-via,” in *IFIP Networking*, May 2009.

[26] “Survivable fixed telecommunication Network Design library (SNDlib),” http://sndlib.zib.de.

[27] M. L. Garcia-Osma, “TID scenarios for advanced resilience,” Tech. Rep., The NOBEL Project, Work Package 2, Activity A.2.1, Advanced Resilience Study Group, Sep 2005.

[28] G. Enyedi, G. Rétvári, and A. Császár, “On finding maximally redundant trees in strictly linear time,” in *IEEE Symposium on Computers and Communications*, July 2009.

[29] S. Even and R. E. Tarjan, “Computing an st-numbering,” *Theoretical Computer Science*, no. 2, 1976.

[30] A. Medina, A. Lakhina, I. Matta, and J. Byers, “BRITE: Boston university Representative Internet Topology gEnerator,” http://www.cs.bu.edu/brite, 2005.

[31] M. Shand and S. Bryant, “IP Fast Reroute framework,” Internet Draft, available online: http://tools.ietf.org/html/draft-ietf-rtgwg-ipfrr-framework-13, October 2009.

[32] S. Bryant, M. Shand, and S. Previdi, “IP fast reroute using Not-via addresses,” Internet Draft, available online: http://tools.ietf.org/html/draft-ietf-rtgwg-ipfrr-notvia-addresses-04, 2009.

[33] J. Moy, “OSPF version 2,” Internet Engineering Task Force: RFC 2328, April 1998.

[34] I. O. for Standardization, “OSI IS-IS intra-domain routing protocol,” ISO/IEC 10589:2002, 2002.

[35] J. Vasseur, M. Pickavet, and P. Demeester, *Network Recovery: Protection and Restoration of Optical, SONET-SDH, IP, and MPLS*. Elsevier, 2004.