On Finding Maximally Redundant Trees in Strictly Linear Time

Gábor Enyedi, Gábor Rétvári
Dept. of Telecommunications and Media Informatics
Budapest University of Technology and Economics
Email: {enyedi,retvari}@tmit.bme.hu

András Császár
TrafficLab, Ericsson Research
Email: Andras.Csaszar@ericsson.com

***Abstract*—Redundant trees are commonly used for protection and restoration in communications networks. Zhang *et al.* presented a linear time algorithm to compute node-redundant trees in 2-node-connected networks, which has become widely cited in the literature. In this paper, we show that it is difficult to implement this algorithm providing both correctness and linear complexity at the same time. Therefore, we present a revised algorithm with strict linear time complexity. Moreover, we generalize the concept of node-redundant trees from 2-node-connected networks to arbitrary topologies, a crucial development since real networks can not always satisfy 2-connectedness, especially after a failure.**

***Index Terms*—redundant trees; resilience**

I. Introduction

Nowadays, the proliferation of network-centric applications impose new requirements on telecommunications networks, perhaps most importantly amongst them the fast and reliable response to failures. An interesting idea to provide fault protection is the scheme of node-redundant trees (simply redundant trees in the sequel): A pair of redundant trees is a pair of directed spanning trees defined on an undirected graph, with edges directed towards a given root node. This node is reachable from any other node along both of the trees, but the paths in the two trees are always node-disjoint (see Fig. 1). Therefore, redundant trees readily lend themselves to be applied for resilience purposes: in a pair of redundant trees there always remains at least one path open to the root node, even after a single node or a link failure shows up.

Using two trees for failure protection was first proposed in [1], though, restricted to handle only edge failures. A way to construct edge-redundant trees was published earlier in [2]. The first algorithm for finding node-redundant trees in polynomial time was published by Medard *et al.* in [3] based on the notion of ear-decomposition of graphs. The complexity was later decreased to linear by Zhang *et al.* in [4]. They also show that a graph does not need to satisfy strong requirements to admit a pair of redundant trees with respect to any node: it turns out that 2-node-connectedness is both a sufficient and necessary condition. Redundant trees, therefore, have become popular in the literature to implement resilience schemes (see [3], [4], [5], [6] for optical protection and restoration in WDM networks, or [7] for an application to IP Fast ReRoute [8]), and the algorithm of Zhang *et al.* [4] has become the *de facto* algorithm for their computation.

This paper is devoted to answer the compelling problems we faced when we tried to deploy this algorithm in an operational IP Fast ReRoute testbed [9], [10]. First, we found that it is remarkably difficult to implement this algorithm correctly, while also retaining linear complexity. In Section II, we prove that a naive implementation necessarily turns out incorrect and we show that this adverse behavior arises even in graphs of no more than a few hundred nodes. Therefore, in Section III we propose a revised algorithm that features correct computation of redundant trees and strict linear time complexity. In Section IV, we generalize redundant trees from strictly 2-node-connected networks to arbitrary networks: we show that our algorithm always produces a pair of trees with maximum redundancy no matter the connectedness of the underlying graph. As far as we are aware of, this is the first time that such *maximally redundant trees* are addressed in the literature. Finally, in Section V we conclude our work.

II. Issues with a naive implementation

The prevalent linear time algorithm to compute redundant trees, given by Zhang *et al.* in [4], is based on a Depth First Search (DFS) traversal of the graph and the notion of lowpoints. The lowpoint number of a node is the minimum of the lowpoint numbers of its children and the DFS numbers of its other neighbors[1]. The algorithm walks down the DFS tree until it encounters a "jump" in the lowpoint number, at which point it has identified a new ear in the ear-decomposition. Consequently, the sequence of nodes specified by the freshly found ear is added to the first tree in one direction and to the second tree in the reverse direction. The right direction is decided based on a partial order, the so called *voltage*, built by the algorithm as it proceeds according to the following rule: voltages only increase along one tree and decrease along the other one. For a more detailed description, see the node-redundant tree algorithm in [4].

On the graph depicted in Fig. 1, the algorithm would proceed as follows. Suppose we have already walked through

The second author was supported by the Janos Bolyai Fellowship of the Hungarian Academy of Sciences.

[1]Note that this is only one of the two definitions used in [4]. In this paper we use this, since it can be applied on non-2-node-connected graphs as well.

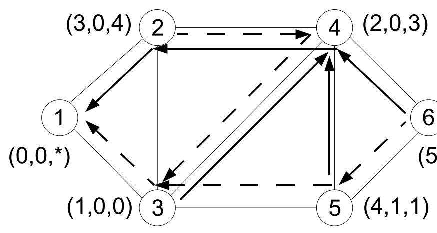

Figure 1: A sample graph and a pair of redundant trees, marked with slick and dashed arrows, for root node 1. The DFS traversal order is 1, 3, 4, 2, 5, 6. The DFS number, the lowpoint number and voltage values are given in parentheses for each node.

nodes 1, 3, 4 and 2, and we have already added the path $2 \to 4 \to 3 \to 1$ to the first tree (which we choose as the tree along which voltages would decrease) and path $3 \to 4 \to 2 \to 1$ to the second (increasing) tree. Next, we proceed to node 5 along the DFS tree, where a leap in the lowpoint number is encountered (because the subsequent node in the DFS tree, node 6, has lowpoint number 2 instead of 1), so we "tie the knot" back to node 3 (the lowpoint of 5). But we need to decide in which direction to add the new ear to the trees, so we compare the voltage of the endpoints, node 4 and 3, to discover that node 3 has smaller voltage, so we add the path $5 \to 3$ to the first tree and $5 \to 4$ to the second one. Additionally, we set the voltage of node 5 larger than that of node 3 but smaller than node 4. In the final step, at node 6, the situation just reverses as now the starting point has smaller voltage, so the newly found paths are added accordingly.

At the surface, this algorithm seems obviously implementable in linear time. Though, a naive implementation might easily prove incorrect. The problem stems from the need for a proper data structure to maintain node voltages, which supports both $O(1)$ insertion (so that new nodes can be introduced into the partial order) and $O(1)$ comparison (so that we can know in which direction to attach ears to the trees). Such data structures are, however, notoriously hard to implement in contemporary computer architectures. For instance, linked lists or binary trees are immediately ruled out, since—although insertion is $O(1)$ provided that the position of the new element is known—comparison is $O(n)$ (or $O(\log n)$ for binary trees). Additionally, ordinary arrays and explicit ordering of the elements after each insertion would require $O(n)$ steps, though, this would deteriorate overall linearity. A naive implementation would, therefore, assign platform-native numbers as voltages. However, as computers encode numbers in a finite number of bits (e.g., an IEEE double precision float uses 64 bits, so it can not represent more than 2^{64} different values), the algorithm might easily run out of assignable distinct voltage values, rendering the result incorrect.

Theorem 1: Suppose that voltages take their values from a finite, totally ordered set S. Then, there is a graph of at most $3\log_2(|S|) + 7$ nodes and a possible DFS traversal, on which an implementation of the algorithm gives incorrect answer.

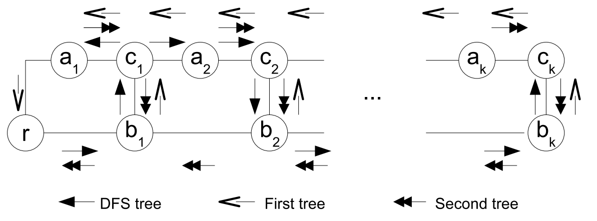

Figure 2: Illustration for Theorem 1.

Proof: Suppose that the elements of S are identified by a unique natural number, which we shall use to assign voltages, and by $v_i < v_j$ we shall mean that the voltage of node v_i is less than that of v_j. Consider the graph depicted in Fig. 2 and let r be the root node. The arrows show a possible DFS traversal of the graph: $r, b_1, c_1, a_1, a_2, c_2, b_2, \ldots, b_k, c_k, a_k$. Note that the last three items can vary if k is even, but the proof remains essentially the same.

In the ith step, the algorithm finds the node sequence a_i, c_i, b_i between nodes b_{i-1} and c_{i-1}. Without loss of generality, suppose that $b_{i-1} < c_{i-1}$ (otherwise, the proof proceeds similarly, only the relations are the other way around). So, to the tree along which voltages increase we need to add the path b_i, c_i, a_i, so we set $b_i < c_i < a_i < c_{i-1}$. Similarly, along the decreasing tree we write $a_i > c_i > b_i > b_{i-1}$, so we have $b_{i-1} < b_i < c_i < a_i < c_{i-1}$. We choose the voltage of c_i pessimistically to $\lfloor \frac{a_i+b_i}{2} \rfloor$ (otherwise, if $c_i - b_i > a_i - c_i$ held, we could reconstruct the graph by connecting a_{i+1} to a_i instead of c_i, and b_{i+1} to c_i instead of b_i, which would yield a suboptimal subdivision of S). In consequence, we have that $a_i - b_i < \frac{a_{i-1}-b_{i-1}}{2}$ and, for a general k, we have $a_k - b_k < \frac{|S|}{2^{k-1}}$. Finally, we observe that the algorithm fails if we run out of distinct voltage values in S, that is, if $a_k = b_k$, which occurs when $\frac{|S|}{2^{k-1}} < 1$. Therefore, for arbitrary finite S, we can show a graph of $k = \log_2(|S|) + 2$ ears (or $3(\log_2(|S|) + 2) + 1 = 3\log_2(|S|) + 7$ nodes), for which the algorithm fails to find correct redundant trees. ■

The smaller the set S of voltages, the sooner this pathologic behavior emerges. For 32 bit integers, we only need 103 nodes for the algorithm to fail, and for 64 bit integers we need 199 nodes. Floating point arithmetic does not come to rescue here either: double precision floats of 64 bits run short again at 199 nodes at the most. A solution would be to use arbitrary precision arithmetics, however, such arithmetics does not provide $O(1)$ insertion and/or comparison, rendering the implementation worse than linear.

III. A REVISED ALGORITHM

In this section, we present a novel algorithm for finding a pair of redundant trees in linear time. Our algorithm is divided into 3 distinct phases: in the first phase, we perform a DFS traversal of the graph, then we compute an intermediate graph representation based on which in the final phase we obtain the redundant trees. Since we take special care to ensure that each phase terminates in linear time, the resultant algorithm will still be linear. Moreover, we do not use voltages, so the traps discussed in the previous section are avoided. In this section, we assume 2-node-connectivity; the extension to arbitrary graphs is dealt with only in the next section.

A. DFS traversal

In the first phase of our algorithm, we compute a DFS traversal of the graph, which will be used in the subsequent phases to govern the search for the redundant trees. In the rest of the paper, the root node will be marked by r and the DFS number of some node n will be denoted by D_n (naturally, $D_r = 0$).

First, we present a simple technical lemma for characterizing DFS numbers.

Lemma 1: Let G be an undirected, 2-node-connected graph, let n be a node in G and let T be the DFS successor tree of n (n is in T). There is a node in T with a neighbor x outside T, such that at least one of the following claims holds true

- x is an ancestor of n but not its immediate parent and/or
- $x = r$.

Proof: First, suppose $D_n \geq 2$. The nodes of the successor tree T are a subset of the nodes of G, $N(T) \subset N(G)$. As G is 2-node-connected, there are at least two edges between $N(T)$ and $N(G) \setminus N(T)$, whose endpoints in $N(G) \setminus N(T)$ are different (since the nodes with DFS number 0 and 1 are not members of $N(T)$, so $|N(G) \setminus N(T)| \geq 2$). One of these edges is formed by n and its immediate parent p. Consider the other edge(s) $\{m, x\}$, with $m \in N(T)$ and $x \in N(G) \setminus N(T)$. Now, either $m = n$ or m is a successor of n. Additionally, because DFS traversals have the property that a neighbor of some node is either an ancestor or a successor, x must be an ancestor of m. Hence, x is an ancestor of n too. Furthermore, because G is 2-node-connected, there is an edge amongst these $\{m, x\}$ edges with $x \neq p$, which coincides with the first claim of the Lemma.

Next, suppose $D_n = 0$ or $D_n = 1$. Since G is 2-node-connected, one can always find a cycle traversing any two nodes (Dirac's Theorem). Consider the cycle through the root r (DFS number 0) and the node with DFS number 1. They have a common DFS successor, in particular, the other neighbor of r in this cycle, which coincides with the second claim of the Lemma. ■

Next, we present an algorithm to compute the DFS numbers and the lowpoint numbers (see Algorithm 1). This algorithm is essentially the same as the one presented in [4], with the only difference being that we also add an edge marking where the value of lowpoint number came from.

Note that Algorithm 1 is implementable with a slight modification of the standard DFS traversal algorithm [4], and thus its complexity is linear in the number of edges.

B. Finding an ADAG

As mentioned previously, our algorithm is divided into three phases. Below, we discuss the second, intermediate phase, when a special directed spanning graph is computed, which we call an ADAG (Almost Directed-Acyclic-Graph) for reasons that will be made clear soon. This intermediate step is important for two reasons: First, it facilitates a cleaner, modular implementation. Second, an intermediate step makes it possible to completely eliminate voltages, hence avoiding the difficulties we pointed out in Section II.

Algorithm 1 Revised DFS for graph G and root node r

```
1: Execute a DFS traversal of the graph
2: Set DFS number $D_n$ at each node $n$, so that $D_n$ denotes
   the number of nodes visited before $n$
3: Compute the lowpoint number for each node $n$ as
   $\min(L, D)$, where $L$ is the smallest lowpoint number of
   $n$'s children and $D$ is smallest DFS number among $n$'s
   neighbors
4: For each node $n$, associate a directed edge $(n, x)$, where
   $x$ is the node from which $n$ received its lowpoint number.
   If there are more possibilities, choose an arbitrary child
   as $x$
```

The main idea is to ensure that we always proceed from lower voltage nodes towards higher voltage nodes, so we always know in which direction to attach new paths (recall that voltages were used in the first place to help us decide on the order of nodes as they are added to the trees). To maintain this invariant, we need to ensure that we only leave a node when we have found not only all those ears *emanating* from it (as the original algorithm of Zhang *et al.* does), but also those ones that *terminate* in it, by sometimes traversing the DFS tree upwards instead of moving always downwards. This idea is implemented in Algorithm 2. Note that Algorithm 2 does not compute the redundant trees right away, it instead builds an intermediate graph representation D.

Definition 1: Let an *ear* be a sequence of nodes we push to the stack at the same time (line 11 or line 21).

Before we turn to discuss the specifics of Algorithm 2, we first provide a short example of the algorithm's procession. Considering the same network and the same DFS traversal as before, the graph D calculated by Algorithm 2 is given in Fig. 3. We start from node 1 and the first ear we find is $3 \to 4 \to 2$, so edges $(1, 3)$, $(3, 4)$, $(4, 2)$ and $(2, 1)$ are added to D. Now, stack S contains "342", so the next node we pop from the top is node 3. Node 3 has no child, so we do not enter the branch at line 6. However, we observe that there is a neighboring node still not marked $ready$, node 5, so we take the branch at line 16 and we move upwards along the DFS tree until we arrive to a $ready$ node, node 4. Therefore, the next ear is made up by node 5 alone, and consequently edges $(3, 5)$ and $(5, 4)$ are added to D. Now, the stack contains "542". We pop 5, whose only child is node 6, so next we find the ear consisting of the sole node 6, and $(5, 6)$ and $(6, 4)$ are added to D. At this point all nodes are marked $ready$, so the the algorithm terminates (after popping the remaining entries "642" from the stack).

Next, we show that Algorithm 2 always terminates and reaches all nodes. For this, we only need to show that the two main branches of the algorithm (line 6 and line 16) terminate.

Lemma 2: The branches at line 6 and 16 always terminate.

Algorithm 2 Finding an ADAG for graph G and root node r

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

Proof: First, we show by mathematical induction that all DFS ancestors of an arbitrary $ready$ node are always marked $ready$. Initially, this is true, since only r is $ready$. Then, after finding an ear either at line 6 or at line 16, the claim remains true, since all the ancestors of a node in the ear became $ready$ too.

At the end of the branch at line 6, we always arrive to r or to an ancestor of the starting node, thanks to Lemma 1. From r and its immediate successor we arrive to r, and from any other node we eventually reach an ancestor (which is $ready$ at this point as we have shown above), so the branch at line 6 indeed terminates. On the other hand, in the branch at line 16 we always move upwards in the DFS tree, heading towards r. Since r is $ready$, a $ready$ node is always reached finally, so the branch at line 16 also terminates. ■

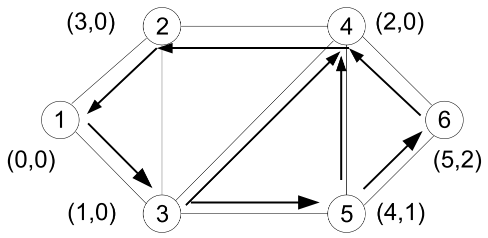

Figure 3: A sample graph and the computed ADAG, for root node 1. The DFS traversal order is 1, 3, 4, 2, 5, 6. The DFS number and the lowpoint number are given in parentheses for each node.

Next, we show that the output of the algorithm, graph D, "almost" qualifies as a Directed Acyclic Graph (DAG). More precisely, we show that with the removal of r, D becomes a DAG, if the original graph was 2-node-connected. Due to this property, we call D an Almost DAG (ADAG).

Lemma 3: Let D' be the digraph produced by Algorithm 2 with the root node r removed. Then, D' is a DAG.

Proof: We observe that in both cases when we add edges to D', 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 11 or line 21. This ear starts at $current$ and terminates at another node, say, x. The following claims hold for $current$ and x:

- $current \neq x$ (at branch 6, this is true due to Lemma 1, and at branch 16 because all the children have been made $ready$ by branch 6)
- $current$ has already left the stack and
- x is still on the stack.

Now, let $A = a_1, a_2, ..., a_n$ be the sequence of nodes as they leave the stack S. According to the argumentation above, when we add edge (a_i, a_j) to D' one of the following two cases hold

- a_i has already left the stack when we push a_j or
- a_i appears above a_j in the stack.

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

Finally, we observe that Algorithm 2 visits each edge at most once, therefore its running time is linear in the number of edges.

C. Constructing the redundant trees

In the final phase, we construct a pair of redundant trees from the ADAG produced by Algorithm 2.

Theorem 2: Perform a Breadth First Search (BFS) traversal from node r on the ADAG D, yielding a directed tree R. Perform a second BFS traversal from r, but now taking the edges of D in reverse direction, yielding another tree B. Now, R and B are a pair of redundant trees, rooted at r.

Proof: Create a new graph D' from D, by splitting node r into two nodes, r^+ and r^-, in such a way that edges only enter $r+$ and only leave r^-. We define a partial order on the nodes of graph D': let $a \prec b$ if there is a directed path from node a to node b in D'. Note that $(\prec)$ is well-defined, because D' is a DAG due to Lemma 3. Since there is a maximum and a minimum element (r^+ and r^-), the nodes of D' with the order $(\prec)$ make up a bounded partially ordered set (poset). Observe that moving from n towards r in R equals traversing the nodes in increasing direction. Conversely, moving towards r along the other tree, B, means moving in decreasing direction. This ensures that what we obtain by taking the paths $n \rightarrow r$ in R and B, respectively, are two node-disjoint paths, which concludes the proof of the Theorem. ■

Note that this final phase again can be performed in a linear number of steps (both BFS traversals are linear in the number of edges in D). Since each of the three phases turned out to be linear, the overall complexity of our algorithm is linear too.

Getting back to our example, it is easy to construct the redundant trees from the ADAG depicted in Fig. 3 using Theorem 2. These redundant trees coincide with the ones given in Fig. 1; the tree marked by solid lines in Fig. 1 coincides with R, while the tree marked by dashed lines is exactly B.

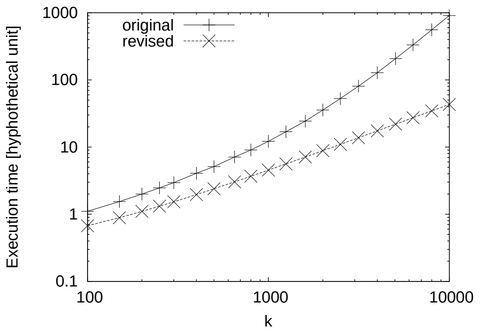

Figure 4: Comparison of the execution times of the original and the revised algorithm, on graphs of the type of Fig. 2 for increasing number of components (k). Note the log-log scale.

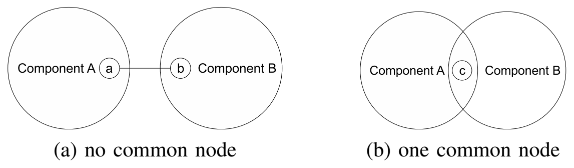

Figure 5: Relations of 2-node-connected components.

D. Performance evaluation

In the foregoing discussions, we have shown that even though the original algorithm by Zhang *et al.* works reasonably well on practical networks, it might provide erroneous result on certain pathological graphs, like the one depicted in Fig. 2. However, our algorithm gives correct results even in such pathological cases, and, what is more, it does that in strict linear time. In this section, we complete these theoretical results with practical insights: we compare the running times of the original and the revised algorithm.

To obtain valid benchmark results, we needed a correct implementation of both algorithms. Therefore, we implemented the original algorithm over arbitrary precision rational arithmetics to prevent the adverse exhaustion of the voltage space we pointed out as its main problem in Theorem 1, and we compared its raw execution time, as measured by the `valgrind(1)` tool, with that of our revised algorithm.

For generic graphs, we did not notice significant difference in the performance of the algorithms. However, benchmarks on a series of increasingly sized pathological graphs indicate a substantial performance penalty for the original algorithm (see Fig 4). We observe that while our revised algorithm exhibits strictly linear execution time, the original algorithm gradually diverges from this trend. This is attributed to the fact that, as the graph increases, the assigned voltage values decrease to a level where the performance of arbitrary precision rational arithmetics really begins to dominate execution time. Recall, however, that without arbitrary precision arithmetics the original algorithm might provide incorrect result.

IV. Maximally redundant trees on arbitrary graphs

So far, we have dealt with 2-node-connected graphs exclusively. However, in practice the 2-connectedness of an operational network can not always be guaranteed, which violates a fundamental assumption that underlies redundant tree algorithms available in the literature. In this section, we lift this artificial limitation and we turn our attention to generalize redundant trees to arbitrary graphs.

Naturally, if a graph is not 2-node-connected, it is impossible to find a pair of redundant trees on it. Nevertheless, it is still possible to find a pair of directed trees with maximum redundancy: If there exists a pair of node-disjoint paths between any two nodes, our trees will include them. Otherwise, the paths in the trees will include the minimum number of common nodes and edges. We shall call trees possessing this property as *maximally redundant trees* in the sequel.

A non-2-node-connected graph consists of two or more 2-node-connected disjunct components (we allow for components consisting of a single node). Any two components can be connected in two ways: they either have no, or only one node in common (two components having more than one common node qualify as a single 2-node-connected component).

Our algorithm treats both cases correctly. First, we observe that *inside* the 2-node-connected components, perfectly valid redundant trees will be computed, with the root node set to the node through which we entered the component. Thus, we only need to see what happens *between* the components.

Consider the case of no common nodes between neighboring components (see Fig. 5a). Now, Algorithm 1 sets the lowpoint number of node b to the DFS number of node a, so when Algorithm 2 reaches b, it finds an ear containing only node b (in line 6) and adds both edge (a, b) and (b, a) to D. Note that edge (b, a) will eventually appear both in tree R and B, but this can not be avoided due to $\{a, b\}$ being a bridge edge. From this point on, our algorithm proceeds in component B as if node b was the root node.

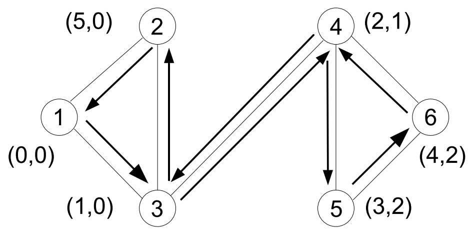

Figure 6: A sample graph and the computed ADAG, for root node 1. The DFS traversal order is 1, 3, 4, 5, 6, 2. The DFS number and the lowpoint number are given in parentheses for each node.

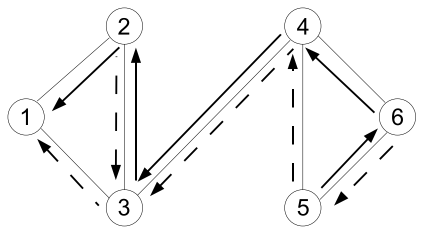

Figure 7: The maximally redundant trees found using the ADAG depicted on Fig 6. The edges of tree R and B are depicted with solid and dashed arrows.

For the second case, when the neighboring components have one node in common (see Fig. 5b), the situation is quite similar, with the only difference that it is now node c which becomes the single point of failure (instead of edge $\{a, b\}$ as occurred in the previous case). This ensures that what we eventually obtain by executing our algorithm on non-2-node-connected graphs is a pair of maximally redundant trees.

Perhaps a simple example is in order. Consider the network depicted on Fig. 6. One may observe that this network is not 2-node-connected any more; there are two 2-node-connected components: one consisting of nodes 1, 2 and 3, and another one of nodes 4, 5 and 6, with only one edge between them.

First, the DFS traversal is made and it reaches all the nodes setting both the DFS and the lowpoint numbers. One may observe that now node 4 gets its lowpoint number from node 3, since the DFS number of node 3 is the lowest, even if it is the parent of node 4.

Second, Algorithm 2 is executed to find an ADAG. We start from node 1, and since the node from where node 3 got its lowpoint number is node 2, the first ear we found is $3 \to 2$, so edges $(1, 3)$, $(3, 2)$ and $(2, 1)$ are added to D. Now, stack S contains "32", so node 3 is the next one we pop. Node 3 has one neighbor not $ready$ yet, node 4, which got its lowpoint number from node 3, so the next ear found is node 4 alone. Edge $(3, 4)$ and $(4, 3)$ are added to D and stack S contains "42". From node 4 ear $5 \to 6$ can be found and edge $(4, 5)$, $(5, 6)$ and $(6, 4)$ are added to D. At this point all the nodes are $ready$, so after popping the nodes in the stack the algorithm terminates.

In the third, final phase, the maximally redundant trees are computed through executing two BFS traversals on the ADAG D. The trees found are depicted on Fig. 7. One may observe that both of them contain edge $(4, 3)$, the bridge between the two 2-node-connected components. One may also observe that any two paths heading to node 1 in the two trees are node disjoint aside from node 3 and node 4, the two endpoints of the bridge.

V. Conclusion

In this paper, we dealt with the problem of finding maximally redundant trees in linear time. These trees compactly encode routings that are as fault-tolerant as possible, given the inherent redundancy of the underlying network. Such routings are applied for implementing protection and restoration schemes in diverse areas of telecommunications.

A linear time algorithm for 2-node-connected graphs was proposed by Zhang *at al.* in [4], which is now widely cited in the literature. We proved that an implementation of this algorithm fails at either correctness or linearity, and this behavior arises even in graphs with only some few hundred nodes. We presented a revised algorithm, which is on the one hand provably linear and, on the other hand, generalizes the notion of redundant trees from 2-node-connected networks to arbitrary networks.

The revised algorithm was successfully applied in an IP Fast ReRoute prototype deployed at BME-TMIT. For a report on our operational experiences with this algorithm, the reader is referred to [9] and [10].

References

[1] A. Itai and M. Rodeh, "The multi-tree approach to reliability in distributed networks," Inf. Comput., pp. 43–59, 1988.

[2] J. Edmonds, "Edge-disjoint branchings," *Combinatorial Algorithms*, pp. 91–96, 1973.

[3] M. Médard, R. G. Barry, and R. A. G. S. G. Finn, "Redundant trees for preplanned recovery in arbitary vertex-redundant or edge-redundant graphs." pp. 641–652, Oct 1999.

[4] 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.

[5] M. Médard, R. A. Barry, S. G. Finn, W. He, and S. S. Lumetta, "Generalized loop-back recovery in optical mesh networks," pp. 153–164, Feb 2002.

[6] G. Xue, L. Chen, and K. Thulasiraman, "Delay reduction in redundant trees for preplanned protection against single link/node failure in 2-connected graphs," Global Telecommunications Conference, 2002. GLOBECOM '02. IEEE, pp. 2691–2695, November 2002.

[7] T. Cicic, A. F. Hansen, and O. K. Apeland, "Redundant trees for fast IP recovery," in *Broadnets*, 2007, pp. 152–159.

[8] M. Shand and S. Bryant, "IP Fast Reroute framework," Internet Draft, available online: http://tools.ietf.org/html/draft-ietf-rtgwg-ipfrr-framework-08, Feb. 2008.

[9] G. Enyedi, P. Szilágyi, G. Rétvári, and A. Császár, "IP Fast ReRoute: Lightweight Not-Via without additional addresses," accepted to INFOCOM'09 Mini-Conference, available online: http://opti.tmit.bme.hu/~enyedi/papers/, April 2009.

[10] P. Szilágyi and Z. Tóth, "Design, implementation and evaluation of an IP Fast ReRoute prototype," BME, Technical Report, to appear at Scientific Student Conference'08, available online: http://opti.tmit.bme.hu/~enyedi/papers/.