ISTITUTO DI ANALISI DEI SISTEMI ED INFORMATICA
CONSIGLIO NAZIONALE DELLE RICERCHE

A. Galluccio, G. Proietti

POLYNOMIAL TIME ALGORITHMS FOR 2-EDGE-CONNECTIVITY AUGMENTATION PROBLEMS

R. 556 Ottobre 2001

Anna Galluccio – Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, 00185 Roma, Italy. Email:galluccio@iasi.rm.cnr.it.

Guido Proietti – Dipartimento di Matematica Pura e Applicata, Università di L'Aquila, Via Vetoio, 67010 L'Aquila, Italy, and Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, 00185 Roma, Italy. Email:proietti@univaq.it.

This work has been partially supported by the CNR-Agenzia 2000 Program, under Grants No. CNRC00CAB8 and CNRG003EF8, and by the Research Project REACTION, partially funded by the Italian Ministry of University.

ISSN: 1128–3378

Collana dei Rapporti dell'Istituto di Analisi dei Sistemi ed Informatica, CNR

viale Manzoni 30, 00185 ROMA, Italy

tel. ++39-06-77161
fax ++39-06-7716461
email: iasi@iasi.rm.cnr.it
URL: http://www.iasi.rm.cnr.it

Abstract

Given a graph G with n vertices and m edges, the 2-edge-connectivity augmentation problem is that of finding a minimum weight set of edges of G to be added to a spanning subgraph H of G to make it 2-edge-connected. While the general problem is NP-hard and 2-approximable, in this paper we prove that it becomes polynomial time solvable if H is a depth first search tree of G. More precisely, we provide an efficient algorithm for solving this special case which runs in $\mathcal{O}\big(M \cdot \alpha(M, n)\big)$ time, where α is the classic inverse of the Ackermann's function and $M = m \cdot \alpha(m, n)$. This algorithm has two main consequences: first, it provides a faster 2-approximation algorithm for the general 2-edge-connectivity augmentation problem; second, it solves in $\mathcal{O}(m \cdot \alpha(m, n))$ time the problem of recovering the 2-edge-connectivity of a communication network after a link failure.

1. Introduction

Let $G = (V, E)$ be a connected, undirected graph, of n vertices and m edges, where an edge $e = (u, v) \in E$ represents a potential connection between vertices u and v. Let us assume that a real weight is associated with each edge $e \in E$, expressing some cost for activating the edge, and consider the problem of building a network in G which allows all the sites to communicate. For this kind of networks, it is generally important to be *economically attractive*, i.e., they should be as sparse as possible to reduce set-up costs. Then, the cheapest solution consists of designing a *minimum weight spanning tree* T of G, namely a connected spanning subgraph of G such that the sum over all the edge weights is minimum. Unfortunately, this structure does not survive a single link or site failure, and given the increasing attention towards the *reliability* of communication networks, this represents a severe drawback. Hence, in the case of link failures, which is of interest in this paper, a general strategy to strengthen the network reliability without affecting the *status quo* (i.e., without removing the already existing links), is that of adding a minimum weight set of edges to T so that its edge-connectivity is increased to 2. This optimization problem is NP-hard and 2-approximable for general graphs. But, as we will show in this paper, it may become considerably easier if the given spanning tree has a special structure. In fact, we prove that it is polynomial time solvable if the given spanning tree is a depth first search tree of G.

A classical formulation of an *edge-connectivity augmentation problem* is the following: given a weighted graph G, an h-edge-connected spanning subgraph H of G, with $h \geq 1$, and an integer $\lambda > h$, find a minimum weight set of edges of G, say $\textsc{Aug}_\lambda(H, G)$, whose addition to H increases its edge-connectivity to λ. Such a problem turns out to be NP-hard [2], and thus most of the research in the past focused on the design of approximation algorithms for solving it. In particular, for the special case $\lambda = 2$, which is of interest for this paper, efficient approximation algorithms for finding $\textsc{Aug}_2(H, G)$ are known. More precisely, for the weighted case, the best performance ratio is 2 [4, 8], while for the unweighted case, Nagamochi and Ibaraki developed a $(51/26 + \epsilon)$-approximation algorithm, for any constant $\epsilon > 0$ [10]. Analogous versions of augmentation problems for vertex-connectivity and for directed graphs have been widely studied, and we refer the interested reader to the following comprehensive papers [3, 9].

Apart from designing approximated solutions, researchers have also investigated the problem of characterizing polynomial time solvable cases of the problem. This can be done by giving additional constraints on the structure of H and/or G. First, Eswaran and Tarjan[2] proved that $\textsc{Aug}_2(H, G)$ can be found in polynomial time if G is complete and all edges have weight 1, i.e., all potential links between sites may be activated at the same cost. Afterwards, Watanabe and Nakamura extended this result to any desired edge-connectivity value [13], and faster algorithms in this scenario have been proposed in [5].

In this paper we show that the structure of H is equally relevant in order to have polynomial time algorithms; in fact, we prove that $\textsc{Aug}_2(H, G)$ can be found in polynomial time if G is any weighted graph and H is a depth first search tree of G. In this case, we can compute $\textsc{Aug}_2(H, G)$ in $\mathcal{O}\big(m \cdot \alpha(m, n) \cdot \alpha(m \cdot \alpha(m, n), n)\big)$ time and $\mathcal{O}(m \cdot \alpha(m, n))$ space, where α is the classic inverse of the Ackermann's function defined in [11]. Results of similar flavour can be found in [6].

It is worth noticing that our algorithm can be used to design a 2-approximation algorithm – with the same time complexity – for the general 2-edge-connectivity augmentation problem [7], thus improving the previous $\mathcal{O}(m + n \log n)$ time bound due to Khuller and Thurimella [8].

We also prove that the complexity of the algorithm decreases to $\mathcal{O}(m \cdot \alpha(m, n))$ if the DFS-tree is a Hamiltonian path of G, and further improved to $\mathcal{O}(m)$ if G is unweighted. As we

will show in the paper, this finds immediate applications for recovering 2-edge-connectivity in a 2-edge-connected communication network undergoing a link failure. In such a case, as soon as a link fails, we can efficiently provide an optimal set of replacement links, both in the weighted and in the unweighted case, so that the *replacement network*, as obtained by removing from the original network the failed link and by adding the corresponding replacement links, is again 2-edge-connected, while at the same time it keeps on to maintain all the old, still working, links.

The paper is organized as follows: in Section 2 we give some basic definitions that will be used throughout the paper; in Section 3 we show how to augment a depth first search tree in unweighted graphs; in Section 4 we present the more general weighted case, while in Section 5, we present the technique to efficiently implement such an algorithm, and we give a detailed space and time complexity analysis; in Section 6, we apply these results to handle transient edge failures in 2-edge-connected networks, and finally, in Section 7, we present conclusions and list some open problems.

2. Basic definitions

Let $G = (V, E)$ be an undirected graph, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. G is said to be *weighted* if there exists a real function $w : E \mapsto \mathbb{R}$, otherwise G is *unweighted*. If parallel edges are allowed, then the graph is said to be a *multigraph*. A graph $H = (V(H), E(H))$ is called a *subgraph* of G if $V(H) \subseteq V$ and $E(H) \subseteq E$. If $V(H) = V$, then H is called a *spanning subgraph* of G. The weight of H is defined as $w(H) = \sum_{e \in E(H)} w(e)$.

A *simple path* (or a *path* for short) in G is a subgraph H of G with $V(H) = \{v_0, \ldots, v_k | v_i \neq v_j$ for $i \neq j\}$ and $E(H) = \{(v_i, v_{i+1}) | 0 \leq i < k\}$, also denoted as $P(v_0, v_k) = v_0 \rightsquigarrow v_k$. A *cycle* is a path whose endvertices v_0 and v_k coincide. A spanning path of G is called a *Hamiltonian path* of G. A graph G is *connected* if, for any $u, v \in V$, there exists a path $P(u, v)$ in G.

A *rooted tree* is a connected acyclic graph with a privileged vertex distinguished from the others. Let T denote a spanning tree of G rooted at $r \in V$. Edges in T are called *tree edges*, while the remaining edges of G are called *non-tree edges*. A non-tree edge (u, v) *covers* all the tree edges along the (unique) path from u to v in T. Let $P(r, x)$ denote the unique path in T between r and $x \in V(T)$. Any vertex y in $P(r, x)$ is called an *ancestor* of x in T. Symmetrically, x is called a *descendant* of any vertex y in $P(r, x)$. A *depth first search* tree of G, *DFS-tree* for short, is a rooted spanning tree T of G such that, for any non-tree edge (u, v), either u is an ancestor of v in T, or v is an ancestor of u in T.

A graph G is said to be *k-edge-connected*, where k is a positive integer, if the removal of any $k - 1$ distinct edges from G leaves G connected. Given an h-edge-connected spanning subgraph H of a k-edge-connected graph G, and a positive integer $h < \lambda \leq k$, finding a λ-*augmentation* of H in G means to select a minimum weight set of edges in $E \setminus E(H)$, denoted as $\textsc{Aug}_\lambda(H, G)$, such that the spanning subgraph $H' = (V, E(H) \cup \textsc{Aug}_\lambda(H, G))$ of G is λ-edge-connected.

3. Augmenting DFS-trees in unweighted graphs

Let $G = (V, E)$ be a 2-edge-connected and unweighted graph. First, we recall the notion of *tree-carving* $\Gamma(G)$ of G [9]. Let $\{V_1, V_2, \ldots, V_k\}$ be a partition of the vertex set V of G; $\Gamma(G)$ is a tree with vertex set $\{\nu_1, \nu_2, \ldots, \nu_k\}$, where vertex ν_i is associated with vertex set V_i, such that for every vertex $v \in V_i$, all the neighbours of v in G belong either to V_i itself, or to V_j, where V_j is adjacent to V_i in the tree $\Gamma(G)$.

We can prove the following:

Theorem 3.1. *Let $G = (V, E)$ be a 2-edge-connected and unweighted graph with n vertices and m edges. Let T be a DFS-tree of G. Then, $\textsc{Aug}_2(T, G)$ can be computed in $\mathcal{O}(m)$ time and space.*

Proof. Since T is a DFS-tree of G, the *DFS-tree partition* of the vertices of G induced by T yields a tree-carving of G and can be computed in $\mathcal{O}(m)$ time [9]. Let us briefly recall how this partition works. First of all, a set of cycle edges is added to T, in the following way: perform a depth first visit of T, and before withdrawing from a vertex v for the last time, check whether the edge joining v and its parent in T is currently still not covered; if so, cover it by adding a cycle edge which leads to the ancestor of v in T closest to r. After this first phase, all the edges of T which caused the insertion of a cycle edge are removed, and the resulting connected components in T provides the DFS-tree partition.

Let $k > 1$ be the number of vertices of the tree-carving $\Gamma(G)$ induced by this DFS-tree partition. The following holds:

Claim. $|\textsc{Aug}_2(T, G)| \geq k - 1$.

Proof. We know that a lower bound on the number of edges of any 2-edge-connected spanning subgraph H of G is $2(k-1)$ [9]. More precisely, each edge (ν_i, ν_j) in $\Gamma(G)$ implies that at least two edges between V_i and V_j are needed in H. Since T contains just one edge between V_i and V_j (otherwise we would have a cycle in T), it follows that any 2-edge-connected spanning subgraph of G contains at least $k-1$ edges which do not belong to T. From this, the claim follows. ∎

To complete the proof, we observe that when the $k-1$ cycle edges selected by the DFS-tree partition are added to T, its connectivity is augmented to 2. Moreover, $\mathcal{O}(m)$ time and space are trivially enough to perform all the operations. ∎

4. Augmenting DFS-trees in weighted graphs

Let $G = (V, E)$ be a 2-edge-connected graph with a real weight function w on the edges and let T be a DFS-tree of G rooted at $r \in V$. Let $v_0, v_1, \ldots, v_{n-1} = r$ be a numbering of the vertices of G as obtained by any fixed postorder visit of T. In the following, for a given non-tree edge $f = (u, v)$, the endvertex v will always be farther from r than u, and will be called the *tail* of f. Moreover, $\texttt{par}(v)$ will denote the parent of v in T.

Let $e_k = (\texttt{par}(v_{k-1}), v_{k-1})$, and let T_k denote the subgraph of T induced by the edge set $\{e_1, \ldots, e_k\}$. Notice that T_k is not necessarily a subtree of T; in fact, the post-order numbering $\{e_1, \ldots, e_k\}$ of $E(T)$, induces a forest in T. A *covering* of T_k is a set of edges in $E \setminus E(T)$ which cover all the edges of T_k. Figure 1 illustrates the notation we use.

Let $V(v_i \rightsquigarrow v_j)$ and $E(v_i \rightsquigarrow v_j)$ denote the set of vertices and the set of edges on the (unique) path in T between v_i and v_j, respectively, and let $C(e_k)$ denote the set of non-tree edges covering the tree edge e_k. Let $D(v_i)$ denote the (possibly empty) set of tree edges joining v_i with its children in T. Finally, given a tree edge e_k and a non-tree edge $f = (u, v) \in C(e_k)$, let

$$F(e_k, f) = \bigcup_{v_i \in V(v_{k-1} \rightsquigarrow v)} D(v_i) \setminus E(v_{k-1} \rightsquigarrow v). \tag{1}$$

In the next subsections, we first give a high-level description of the algorithm, and we then analyze its correctness.

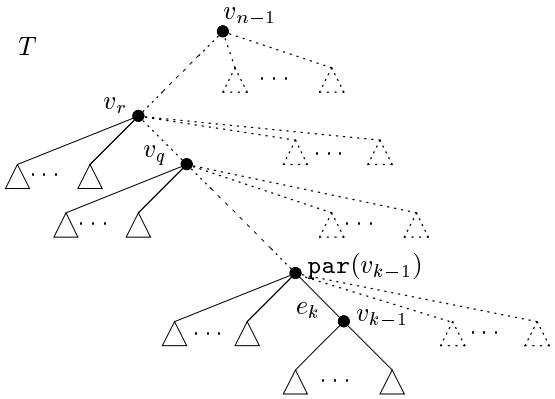

Figure 1: The subgraph T_k of T (solid) induced by $\{e_1, \ldots, e_k\}$ (triangles denote subtrees).

4.1. High-level description of the algorithm

The algorithm consists of $n-1$ iterations. At the k-th iteration, the algorithm considers the tree edge e_k and associates with it a *selected edge* $\epsilon(e_k) \in C(e_k)$. The edge $\epsilon(e_k)$, together with the selected edges computed at previous iterations, induces a covering of T_k, as we will see shortly. The edge $\epsilon(e_k)$ is defined as follows:

DEFINITION 4.1. *Let* $e_k = (\mathtt{par}(v_{k-1}), v_{k-1})$ *be a tree edge, and let* $f \in C(e_k)$*; let* ϕ *be defined recursively as*

$$\phi(e_k, f) = w(f) + \sum_{e_j \in F(e_k, f)} \min_{f_i \in C(e_j)} \{\phi(e_j, f_i)\}. \tag{2}$$

Then, the selected edge $\epsilon(e_k)$ *is such that*

$$\phi(e_k, \epsilon(e_k)) = \min_{f \in C(e_k)} \{\phi(e_k, f)\}. \tag{3}$$

Notice that if v_{k-1} is a leaf, then $\phi(e_k, f) = w(f)$ and $\phi(e_k, \epsilon(e_k)) = \min_{f \in C(e_k)}\{w(f)\}$. Moreover, by plugging (3) into (2), we have that

$$\phi(e_k, f) = w(f) + \sum_{e_j \in F(e_k, f)} \phi(e_j, \epsilon(e_j)), \tag{4}$$

and therefore, (3) can be rewritten as

$$\phi(e_k, \epsilon(e_k)) = \min_{f \in C(e_k)} \left\{ w(f) + \sum_{e_j \in F(e_k, f)} \phi(e_j, \epsilon(e_j)) \right\}. \tag{5}$$

Notice also that if v_{k-1} coincides with the tail of f, then from (1), we have that (2) reduces to

$$\phi(e_k, f) = w(f) + \sum_{e_j \in D(v_{k-1})} \phi(e_j, \epsilon(e_j)). \tag{6}$$

We call this value the *basic weight* of f, and we indicate it by $\phi_0(f)$.

Let S_k denote the subtree of T rooted at v_{k-1} plus the edge e_k. The following set of selected edges defines a covering of S_k

$$X(e_k) = \begin{cases} \epsilon(e_k) & \text{if } v_{k-1} \text{ is a leaf,} \\ \epsilon(e_k) \cup \bigcup_{e_j \in F(e_k, \epsilon(e_k))} X(e_j) & \text{otherwise.} \end{cases} \tag{7}$$

Then, let $E_k = \{e_j \in D(v_i) \mid v_i \in V(v_{n-1} \rightsquigarrow \mathtt{par}(v_{k-1})) \ \wedge \ j \leq k\}$. The solution provided by the algorithm at iteration k, say $\textsc{Sol}(k)$, is defined as follows

$$\textsc{Sol}(k) = \bigcup_{e_j \in E_k} X(e_j). \tag{8}$$

To guarantee efficiency, $\textsc{Sol}(k)$ is not computed explicitly at each iteration. In fact, from (5), it follows that to select the edge $\epsilon(e_k)$ properly, it suffices to maintain $\phi(e_j, \epsilon(e_j))$, for every $j < k$. Thus, only after the $n-1$ iterations have been completed, the algorithm returns the set

$$\textsc{Sol}(n-1) = \bigcup_{e_j \in E_{n-1}} X(e_j). \tag{9}$$

In the following, we shall prove that $\textsc{Sol}(n-1)$ contains a minimum weight set of edges of G covering T.

4.2. Correctness of the algorithm

The algorithm clearly returns a set of edges covering T. Hence, in order to prove that the algorithm is correct, it remains to show that the weight of the solution found by the algorithm equals $w(\textsc{Aug}_2(T, G))$, as stated in the following:

Lemma 4.2. $w(\textsc{Sol}(n-1)) = w(\textsc{Aug}_2(T, G))$.

Proof. Let $\textsc{Opt}(k)$ denote a minimum weight set of edges of G covering $T_k, 1 \leq k \leq n-1$. Notice that $w(\textsc{Opt}(n-1)) = w(\textsc{Aug}_2(T, G))$. We shall prove that $w(\textsc{Sol}(k)) = w(\textsc{Opt}(k))$, for $k = 1, \ldots, n-1$. The proof is by induction on k. For $k = 1$, the thesis follows trivially. Assume the thesis is true up to $k-1 < n-1$, i.e., $w(\textsc{Sol}(i)) = w(\textsc{Opt}(i))$ for $i = 1, \ldots, k-1$. We shall prove that $w(\textsc{Opt}(k)) = w(\textsc{Sol}(k))$.

Clearly, $\textsc{Opt}(k)$ has to contain a non-tree edge covering $e_k = (\mathtt{par}(v_{k-1}), v_{k-1})$, say $f_k = (u, v)$. Since T is a DFS-tree, the insertion of f_k in $\textsc{Opt}(k)$ can only affect the edges of $\textsc{Opt}(k-1)$ covering S_k. Thus, if $X^*(e_j)$ is the set of edges of $\textsc{Opt}(k)$ covering the subtree S_j, then

$$\textsc{Opt}(k) = X^*(e_k) \cup \bigcup_{e_j \in E_k \setminus \{e_k\}} X^*(e_j).$$

Since $\bigcup_{e_j \in E_k \setminus \{e_k\}} X^*(e_j) = \textsc{Opt}(h)$, for some $h < k$, we have, by induction, that

$$w(\textsc{Opt}(h)) = w(\textsc{Sol}(h)) = \sum_{e_j \in E_k \setminus \{e_k\}} w(X(e_j)).$$

Hence, it suffices to show that $w(X^*(e_k)) = w(X(e_k))$. Let $Y(e_k)$ denote any covering of S_k. From (7) and (4), it is easy to see that $w(X(e_k)) = \phi(e_k, \epsilon(e_k))$, and then, by definition, we have that $w(X(e_k)) \leq w(Y(e_k))$, from which the thesis follows. ∎

5. Implementing efficiently the algorithm

In this section, we describe an efficient implementation of our algorithm, and we finally give a detailed space and time complexity analysis.

5.1. Selecting the edge

To compute efficiently the selected edge $\epsilon(e_k)$ at each iteration $k = 1, \ldots, n-1$, we use an auxiliary graph called *transmuter* [12], representing the set of *fundamental cycles* of G with respect to T, i.e., the cycles of G containing only one non-tree edge. Basically, a transmuter $D_G(T)$ built on G with respect to T is a directed graph containing one *source* vertex $s(e_k)$ for each tree edge e_k in T, one *sink* vertex $t(f)$ for each non-tree edge f in G, plus a number of additional vertices of indegree 2. The fundamental property of a transmuter is that there is a directed path from a given source vertex $s(e_k)$ to a given sink vertex $t(f)$ if and only if edges e_k and f form a fundamental cycle in G.

Let us describe in detail how the transmuter can be used to compute the selected edges. With each vertex in the transmuter, we associate a *label*, consisting of a couple $\langle f, \phi(e_k, f)\rangle$, where f denotes a non-tree edge, while $\phi(e_k, f)$ is the *key* of the label and is obtained from (4); furthermore, a vertex can be marked as *unreached*, *active* or *reached*. At the beginning, all the vertices in the transmuter are marked *unreached*, while their labels are undefined.

At the first iteration, when the algorithm considers the edge e_1, the following operations are performed over $D_G(T)$:

1. Visit all the vertices of $D_G(T)$ which are reachable from $s(e_1)$, and mark them *active*.
2. Label each active sink vertex $t(f)$ of $D_G(T)$ with the corresponding couple $\langle f, \phi(e_1, f) = \phi_0(f) = w(f)\rangle$, and mark it as *reached*.
3. Process the active vertices of $D_G(T)$ in *reverse topological order*, by backtracking over the edges used during the descent from the source vertex; during this process, label a given active vertex with a minimum-key label among all the labels of its reached successors, and then mark it as *reached*. At the end of this step, vertex $s(e_1)$ receives a label $\big\langle \epsilon(e_1), \phi(e_1, \epsilon(e_1)) = w(\epsilon(e_1))\big\rangle$, where $\epsilon(e_1)$ is a minimum-weight edge in $C(e_1)$.

At the k-th iteration, the algorithm considers the edge e_k and $D_G(T)$ is processed as follows: if v_{k-1} is a leaf vertex, then the same steps as above are performed, with e_k replacing e_1 (no vertex marked *reached* can be encountered during the descent); otherwise the following operations are executed:

1. Visit all the vertices of $D_G(T)$ which are reachable from $s(e_k)$, by stopping the descent along a path as soon as either a vertex marked *reached*, or a sink vertex is encountered. Mark all the reached vertices as *active* (this means, a vertex already marked as *reached* might again be marked *active*).

2. If an active sink vertex $t(f)$ has an undefined label, then label it with the corresponding couple $\langle f, \phi(e_k, f) = \phi_0(f)\rangle$, where $\phi_0(f)$ is computed from (6) by using the labels associated with the edges in $D(v_{k-1})$ (which have already been processed), and mark it as *reached*.

3. If an active (either sink or internal) vertex has a defined label (namely, it was labelled $\langle f, \phi(e_h, f)\rangle$ and marked *reached* at a previous iteration $h < k$), then update its label to $\langle f, \phi(e_k, f)\rangle$ (where $\phi(e_k, f)$ is defined by (4) and is computed by using the efficient technique specified in the next section), and mark it as *reached*.

4. Process the active vertices of $D_G(T)$ in reverse topological order, as explained previously. At the end of this step, vertex $s(e_k)$ will receive a label $\langle \epsilon(e_k), \phi(e_k, \epsilon(e_k))\rangle$, where $\epsilon(e_k)$ coincides with an edge in $C(e_k)$ satisfying (3), as we will prove shortly.

In the following, we prove that by processing the transmuter as described, at the end of the $(n-1)$-th iteration, for each tree edge e_j, the corresponding selected edge $\epsilon(e_j)$ can be retrieved by the label associated with the source vertex $s(e_j)$ of $D_G(T)$. More precisely, the following holds:

Lemma 5.1. *At the end of the k-th iteration, the label associated with the source vertex $s(e_j), j \leq k$, contains an edge satisfying (5).*

Proof. The proof is by induction on k. For $k = 1$, the thesis follows trivially. Assume the thesis is true up to $k - 1 < n - 1$, and let $\langle f, \phi(e_k, f)\rangle$ be the label associated with $s(e_k)$ at the end of the k-th iteration. Then, it suffices to prove that f satisfies (5); in fact, since the source vertices have zero indegree, their labels, once assigned, cannot be modified.

For the sake of contradiction, let us assume that there exists a non-tree edge $f' \in C(e_k)$ such that $f \neq f'$ and $\phi(e_k, f') < \phi(e_k, f)$. First of all, observe that edges in $C(e_k)$ can be partitioned into two sets: $C_{\text{new}}(e_k)$, containing all the non-tree edges whose tail coincides with v_{k-1}, and $C_{\text{old}}(e_k) = C(e_k) \setminus C_{\text{new}}(e_k)$. Clearly, f' cannot belong to $C_{\text{new}}(e_k)$, since in such a case it would be associated with a sink vertex $t(f')$ marked as *active* at iteration k; therefore, its label receives the key $\phi(e_k, f') = \phi_0(f') < \phi(e_k, f)$, and will eventually reach $s(e_k)$, contradicting the assumptions. Hence, it must be $f' \in C_{\text{old}}(e_k)$. In this case, f' must cover an edge in $D(v_{k-1})$, say $e_h = (v_{k-1}, v_{h-1})$. This means, paths $s(e_h) \rightsquigarrow t(f')$ and $s(e_k) \rightsquigarrow t(f')$ in $D_G(T)$ share one or more vertices. Among these vertices, let v be the one closest to $s(e_k)$ along $s(e_k) \rightsquigarrow t(f')$. Two cases are possible:

(1) $v = t(f')$: in this case, the label of $t(f')$ is updated to $\langle f', \phi(e_k, f')\rangle$, with $\phi(e_k, f') < \phi(e_k, f)$, and therefore it will eventually reach $s(e_k)$, contradicting the assumptions.

(2) v is an internal vertex: in this case, from the inductive step and from the fact that the indegree of the internal vertices of $D_G(T)$ is 2, we have that before the label updating at iteration k, v was labelled with an edge $f'' \in C(e_h) \cap C(e_k)$ such that $\phi(e_h, f'') \leq \phi(e_h, f')$. Hence, from (5) and from the fact that $F(e_k, f) = F(e_h, f) \cup \{D(v_{k-1}) \setminus \{e_h\}\}$, it follows that

10.

$$\phi\left(e_k, f''\right) = \phi\left(e_h, f''\right) + \sum_{e_j \in D(v_{k-1}) \setminus \{e_h\}} \phi(e_j, \epsilon(e_j)) \leq$$

$$\phi\left(e_h, f'\right) + \sum_{e_j \in D(v_{k-1}) \setminus \{e_h\}} \phi(e_j, \epsilon(e_j)) = \phi\left(e_k, f'\right) < \phi(e_k, f),$$

and therefore the label $\langle f'', \phi(e_k, f'')\rangle$ will eventually reach $s(e_k)$, contradicting the assumptions.

∎

5.2. Updating the labels in the transmuter

To perform the label updating in $D_G(T)$, we use an adaptation of the technique proposed in [12] to compute functions defined on paths in trees.

We start by creating a *forest* $\mathcal{F}(G)$ of trees. Initially, $\mathcal{F}(G)$ is composed of n singletons $\nu_1, \ldots, \nu_n$, where vertex ν_j is associated with vertex $v_j \in V$. With each vertex ν_j in $\mathcal{F}(G)$, we associate a key $\kappa(\nu_j)$, initially set equal to 0. The following instructions manipulate $\mathcal{F}(G)$:

- *Link*(ν_i, ν_j): combine the trees with roots ν_i and ν_j into a single tree rooted in ν_i, adding the edge $e = (\nu_i, \nu_j)$;
- *Update*(ν_j, x): if ν_j is the root of a tree, then set $\kappa(\nu_j) := \kappa(\nu_j) + x$;
- *Eval*(ν_j): find the root of the tree currently containing ν_j, say ν_i, and return the *sum* of all the keys on the path from ν_j to ν_i.

Note that *Eval*(ν_j) assumes that a pointer to element ν_j is obtained in constant time.

The sequence of operations in $\mathcal{F}(G)$ goes hand in hand with the postorder visit of T and with the processing of $D_G(T)$. More precisely, immediately before edge e_k is considered by the augmentation algorithm, we perform the following steps:

1. Compute $\Delta(v_{k-1}) = \sum_{e_j \in D(v_{k-1})} \phi(e_j, \epsilon(e_j))$, where these ϕ values are stored in the labels of the source vertices of the transmuter which have already been processed;
2. For each j such that $e_j \in D(v_{k-1})$, perform *Update*$\big(\nu_{j-1}, \Delta(v_{k-1}) - \phi(e_j, \epsilon(e_j))\big)$;
3. For each j such that $e_j \in D(v_{k-1})$, perform *Link*(ν_{k-1}, ν_{j-1}).

Furthermore, when edge e_k is considered, a label updating in $D_G(T)$ is computed as follows: Let $f = (u, v_d)$ be a non-tree edge covering e_k; then, we have that an *Eval*(ν_d) operation returns

$$\sum_{i | v_i \in V(v_{k-1} \rightsquigarrow v_d)} \kappa(\nu_i) = \kappa(\nu_{k-1}) + \sum_{i | v_i \in V(v_{k-1} \rightsquigarrow v_d) \setminus \{v_{k-1}\}} \kappa(\nu_i)$$

and being $\kappa(\nu_{k-1}) = 0$, this can be rewritten as

$$\sum_{i|v_i \in V(v_{k-1} \rightsquigarrow v_d) \setminus \{v_{k-1}\}} \Big(\Delta(\mathtt{par}(v_i)) - \phi(e_{i+1}, \epsilon(e_{i+1})) \Big) =$$

$$\sum_{e_j \in F(e_k, f)} \phi(e_j, \epsilon(e_j)) - \sum_{e_j \in D(v_d)} \phi(e_j, \epsilon(e_j)). \tag{10}$$

Hence, from (4) and (6), we have that

$$\phi\big(e_k, f\big) = \phi_0(f) + \mathit{Eval}(\nu_d), \tag{11}$$

and therefore, as soon as we associate with each non-tree edge f its basic weight, we have that a label updating can be performed through an *Eval* operation.

5.3. Analysis of the algorithm

We can finally prove the following:

Theorem 5.2. *Let $G = (V, E)$ be a 2-edge-connected graph with n vertices, m edges and with real weights on the edges. Let T be a DFS-tree of G. Then, $\textsc{Aug}_2(T, G)$ can be computed in $\mathcal{O}\big(m \cdot \alpha(m, n) \cdot \alpha(m \cdot \alpha(m, n), n)\big)$ time and $\mathcal{O}(m \cdot \alpha(m, n))$ space.*

Proof. To compute $\textsc{Aug}_2(T, G)$, we use the algorithm presented in Section 4.1, implemented through the data structures presented in Sections 5.1 and 5.2.

The correctness of the algorithm derives from Lemmas 4.2 and 5.1, while the time complexity follows from the use of the transmuter and the corresponding label updating. Concerning the label updating, this is performed through the manipulation of $\mathcal{F}(G)$. First of all, notice that the computation of $\Delta(v_i)$, for all $v_i \in V$, costs $\mathcal{O}(n)$ time. Moreover, since the *Eval* instruction requires the sum of the keys associated with the vertices, which is an *associative* operation over the *group* of real numbers, we can apply the *path compression with balancing* technique described in [12] to modify the execution of the various operations occurring in $\mathcal{F}(G)$. Hence, a sequence of p *Eval* and n *Link* and *Update* operations can be performed in $\mathcal{O}((p + n) \cdot \alpha(p + n, n))$ time [12]. Since each edge of $D_G(T)$ corresponds at most to a single *Eval* instruction, and given that $D_G(T)$ has size $\mathcal{O}(m \cdot \alpha(m, n))$ [12], it follows that $p = \mathcal{O}(m \cdot \alpha(m, n))$, and therefore the total time needed to handle the label updating is $\mathcal{O}\big(m \cdot \alpha(m, n) \cdot \alpha(m \cdot \alpha(m, n), n)\big)$.

Since $D_G(T)$ can be built in $\mathcal{O}(m \cdot \alpha(m, n))$ time and each edge in $D_G(T)$ is visited a constant number of times by the algorithm, and given that all the remaining operations (more precisely those corresponding to (6) and (9)) can be performed by using linear time and space, the claim follows. ∎

6. Maintaining biconnectivity through augmentation

The results of the previous sections have an interesting application for solving a survivability problem on networks, that is the problem of adding to a given 2-edge-connected network undergoing a transient edge failure, the minimum weight set of edges needed to reestablish the

biconnectivity. In this way, extensive (in terms of both computational efforts and set-up costs) network restructuring is avoided.

First, we prove that the time complexity of the algorithm presented in the previous sections can be further decreased if the DFS-tree is actually a Hamiltonian path. In fact, the following holds:

Corollary 6.1. *Let $G = (V, E)$ be a 2-edge-connected graph with n vertices, m edges and with real weights on the edges. Let Π be a Hamiltonian path of G. Then, $\textsc{Aug}_2(\Pi, G)$ can be computed in $\mathcal{O}(m \cdot \alpha(m, n))$ time and space.*

Proof. To compute $\textsc{Aug}_2(\Pi, G)$, we use the algorithm presented in Section 4.1, but now we implement it just through the data structure presented in Section 5.1, thus avoiding the time overhead induced by the use of $\mathcal{F}(G)$. In fact, for any path edge e_k and any non-path edge $f = (v_i, v_j) \in C(e_k)$, we have that $F(e_k, f) = (v_j, v_{j-1})$, and therefore, from (4), it follows that $\phi(e_k, f)$ is constantly equal to its basic weight (6). In other words, label updating in the transmuter is not required. From this and from the analysis performed in Theorem 5.2, the thesis follows. ∎

Now, let H be a 2-edge-connected spanning subgraph of a 3-edge-connected graph G. Let $G - e$ denote the graph obtained from G by removing an edge $e \in E$. Given an edge $e \in E(H)$, if $H - e$ is not 2-edge-connected, then we say that e is *vital* for H. In the sequel, an edge e removed from H will always be considered as vital for H.

Let $\textsc{Aug}_2(H - e, G - e)$ be a minimum weight set of edges in $E \setminus E(H - e)$ such that the spanning subgraph $H' = (V, E(H - e) \cup \textsc{Aug}_2(H - e, G - e))$ of $G - e$ is 2-edge-connected. Using the results of the previous sections, we prove that $\textsc{Aug}_2(H - e, G - e)$ can be computed efficiently. More precisely:

Theorem 6.2. *Let $G = (V, E)$ be a 3-edge-connected, real weighted graph with n vertices and m edges. Let H be a 2-edge-connected spanning subgraph of G. Then, for any vital edge $e \in E(H)$, we have that $\textsc{Aug}_2(H - e, G - e)$ can be computed in $\mathcal{O}(m \cdot \alpha(m, n))$ time and space. The running time and the used space can be lowered to $\mathcal{O}(m)$ if all edge weights are equal.*

Proof. After the removal of e from H, every 2-edge-connected component in $H - e$ can be contracted into a single vertex in $\mathcal{O}(m)$ time and space [1]. Let V_i denote the vertex set of the i-th 2-edge-connected component in $H - e$, and let $\{V_1, V_2, \ldots, V_k\}$ be the vertex partition of V induced by contracting all such components. Let $\{\nu_1, \nu_2, \ldots, \nu_k\}$ be the vertex set of the path Π resulting from the contraction, where vertex ν_i is associated with vertex set V_i. Let $\mathcal{G}$ be the multigraph with vertex set $V(\mathcal{G}) = V(\Pi)$ and edge set

$$E(\mathcal{G}) = E(\Pi) \cup \{(\nu_i, \nu_j) \mid (\exists\, u \in V_i) \wedge (\exists\, v \in V_j) \text{ such that } (u, v) \in E \setminus E(H - e)\}.$$

It is easy to realize that the algorithms presented in previous sections can be extended to the case where parallel edges in G are allowed. Therefore, since Π is a Hamiltonian path of $\mathcal{G}$, we can apply both Theorem 3.1 and Corollary 6.1. It follows that, for any given edge $e \in E(H)$, there exist polynomial time algorithms to compute $\textsc{Aug}_2(H - e, G - e)$. Their complexity is the same as in Theorem 3.1 and Corollary 6.1, respectively. ∎

7. Conclusions

In this paper we presented time and space efficient algorithms for solving special cases of the classic problem of finding a minimum weight set of edges that has to be added to a spanning subgraph of a given (either unweighted or real weighted) graph to make it 2-edge-connected. These techniques have been applied to solve efficiently an interesting survivability problem on 2-edge-connected networks, but we believe they are of independent interest and can be applied to a larger class of similar graph problems.

For the weighted case, our algorithm is efficient, but it is still open to establish whether its running time is optimal. Apart from that, many interesting problems remain open: (1) the extension to vertex-connectivity augmentation problems, which are of interest for managing vertex failures in 2-vertex-connected networks; (2) the extension of the results to the case in which all the possible edge failures in H are considered, aiming at providing a faster solution than that obtained by repeatedly applying our algorithms, once for the failure of each vital edge in H.

We consider the last one as the highest-priority open problem, and we plan to attack it by means of ad-hoc amortization techniques. In fact, from a network management point of view, computing *a priori* the augmentation set associated with every edge in the network is essential to know how the network will react in any possible link failure scenario.

Acknowledgements – The authors are very grateful to Peter Widmayer for inspiring discussions on the topic.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The design and analysis of computer algorithms*, Addison Wesley, (1974).

[2] K.P. Eswaran and R.E. Tarjan, Augmentation problems, *SIAM Journal on Computing*, **5** (1976) 653–665.

[3] A. Frank, Augmenting graphs to meet edge-connectivity requirements, *SIAM Journal on Discrete Mathematics*, **5** (1992) 25-53.

[4] G.N. Frederickson and J. Jájá, On the relationship between the biconnectivity augmentation problems, *SIAM Journal on Computing*, **10** (1981) 270–283.

[5] H.N. Gabow, Application of a poset representation to edge-connectivity and graph rigidity, *Proc. 32nd Ann. IEEE Symp. on Foundations of Computer Science (FOCS'91)*, IEEE Computer Society, 812–821.

[6] A. Galluccio and G. Proietti, Polynomial time algorithms for edge-connectivity augmentation of Hamiltonian paths, *12th Ann. Int. Symp. on Algorithms and Computation (ISAAC'01)*, December 19-21, 2001, Christchurch, New Zealand, accepted for presentation.

[7] A. Galluccio and G. Proietti, A faster approximation algorithm for connectivity augmentation problems. In preparation, 2001.

[8] S. Khuller and R. Thurimella, Approximations algorithms for graph augmentation, *Journal of Algorithms*, **14** (1993) 214–225.

[9] S. Khuller, Approximation algorithms for finding highly connected subgraphs, in *Approximation Algorithms for NP-Hard Problems*, Dorit S. Hochbaum Eds., PWS Publishing Company, Boston, MA, 1996.

[10] H. Nagamochi and T. Ibaraki, An approximation for finding a smallest 2-edge-connected subgraph containing a specified spanning tree, *Proc. 5th Annual International Computing and Combinatorics Conference (COCOON'99)*, Vol. 1627 of Lecture Notes in Computer Science, Springer, 31–40.

[11] R.E. Tarjan, Efficiency of a good but not linear set union algorithm, *Journal of the ACM*, **22** (1975) 215–225.

[12] R.E. Tarjan, Applications of path compression on balanced trees, *Journal of the ACM*, **26**(4) (1979) 690–715.

[13] T. Watanabe and A. Nakamura, Edge-connectivity augmentation problems, *Journal of Computer and System Sciences*, **35**(1) (1987) 96–144.