ELSEVIER

Theoretical Computer Science 296 (2003) 167–177

Theoretical Computer Science

www.elsevier.com/locate/tcs

Finding the most vital node of a shortest path☆

Enrico Nardelli[a,b], Guido Proietti[a,b,*], Peter Widmayer[c]

[a] *Dipartimento di Informatica, Università di L'Aquila, Via Vetoio, 67010 L'Aquila, Italy*
[b] *Istituto di Analisi dei Sistemi ed Informatica "A. Ruberti", CNR, Viale Manzoni 30, 00185 Roma, Italy*
[c] *Institut für Theoretische Informatik, ETH Zentrum, CLW C 2, Clausiusstrasse 49, 8092 Zürich, Switzerland*

Abstract

In an undirected, 2-node connected graph $G = (V, E)$ with positive real edge lengths, the distance between any two nodes r and s is the length of a shortest path between r and s in G. The removal of a node and its incident edges from G may increase the distance from r to s. A *most vital node* of a given shortest path from r to s is a node (other than r and s) whose removal from G results in the largest increase of the distance from r to s. In the past, the problem of finding a most vital node of a given shortest path has been studied because of its implications in network management, where it is important to know in advance which component failure will affect network efficiency the most. In this paper, we show that this problem can be solved in $O(m+n\log n)$ time and $O(m)$ space, where m and n denote the number of edges and the number of nodes in G.

Keywords: Network survivability; Shortest path; Node vitality; Mechanism design

1. Introduction

The computational infrastructure throughout society is becoming increasingly large and complex. Networks of workstations are vulnerable to attack and failure, and it is

☆ This work has been partially supported by the Research Training Network contract No. HPRN-CT-1999-00104 funded by the European Union, by the CNR-Agenzia 2000 Program, under Grants No. CNRC00CAB8 and CNRG003EF8, and by the Research Project REAL-WINE, partially funded by the Italian Ministry of Education, University and Research.

* Corresponding author.
E-mail addresses: nardelli@di.univaq.it (E. Nardelli), proietti@di.univaq.it (G. Proietti), widmayer@inf.ethz.ch (P. Widmayer).

0304-3975/03/$ - see front matter
PII: S0304-3975(02)00438-3

generally recognized that the survivability of our computing systems is a critical issue. We are interested in a particular type of survivability: How is a communication network affected by the failure of a component? In this paper, we consider the effect that a node failure will have on a shortest path between two nodes. Our scenario assumes that each message is routed along a shortest path in a communication graph from its source to its destination. When a node on that path fails, we need to replace the old route by a new one, preferably by a shortest path in the graph that does not contain the failed node. Let us call this new route a *replacement* (*shortest*) *path*; it will in general be longer than the path it replaces, but it certainly will never be shorter. From a network management point of view, it is desirable to know for a shortest path ahead of time which node failure will result in the longest replacement path. Such a node is called a *most vital node*, because its failure degrades the transmission from the source to the destination most strongly.

The problem of finding a most vital node of a shortest path has been defined and motivated by Corley and Sha [3]. More precisely, they considered the more general problem of finding the k most vital nodes of a shortest path, that is the k nodes whose removal will increase the distance between the source and the destination node the most, and they gave an exponential algorithm for solving the problem. For the case $k=1$, an efficient implementation of their algorithm requires $O(mn + n^2 \log n)$ time, where m and n denote the number of edges and the number of nodes in the underlying graph. Note that this is not better than the trivial bound that we get by recomputing from scratch the replacement shortest path for every node along the given shortest path. Later on, Bar-Noy et al. [2] proved that, for arbitrary k, the problem is strongly NP-hard. Finally, Venema et al. [14] studied the problem for $k=1$ in a parallel computing environment, providing a polynomial algorithm.

In a related scenario, nodes are reliable, but edges can fail. The problem of finding a *most vital edge* on a shortest path has been studied extensively in the past: We look for an edge whose failure leads to the longest replacement path [1–3]. Now, naturally a replacement path is just a path avoiding the failed edge. Let us assume that the source and destination nodes lie in a 2-edge connected component of the given graph; otherwise, a most vital edge is just a bridge between these nodes, and that is easy to find. The fastest algorithm to compute most vital edges on a pointer machine runs in $O(m+n \log n)$ time and $O(m)$ space [7]; a recent paper [6] rediscovered this algorithm, in the mechanism design framework. On a RAM, there is an algorithm that solves the problem in $O(m\alpha(m,n))$ time and $O(m)$ space [9], where $\alpha(m,n)$ denotes the functional inverse of the Ackermann function defined in [13]. Notice that with the same time and space complexity, it is also possible to solve an interesting variant of the problem, in which the vitality of an edge $e=(u,v)$ is measured with respect to the length of a shortest *detour* (i.e., a path not using e) from u to the destination node [8].

In this paper, we show that the problem of finding a most vital node for a shortest path in a 2-node connected, undirected and positively weighted graph can be solved on a pointer machine in $O(m + n \log n)$ time and $O(m)$ space. The efficiency of our algorithm is based on two considerations. First, we make use of specific structural properties of replacement paths in the computation. This is realized by means of a priority queue that stores certain distance values for certain nodes. Unfortunately, the priority

queue contains both distance values that would lead to an incorrect result if they ever would be used and the desired distance values leading to the correct result. The reason for this mix is algorithmic performance: We have no way of efficiently distinguishing between both, but we make sure that the algorithm never uses the undesired values. Second, we perform this computation incrementally as we visit the nodes along the shortest path.

The paper is organized as follows: In Section 2, we define the problem formally and give the required basic definitions. In Section 3, we present the structural properties of replacement paths, along with our algorithm. In Section 4, we show how to explicitly compute all the replacement shortest paths, without any additional space and time overhead. Finally, Section 4 discusses modifications and further applications, and lists some open problems.

2. Basic definitions

Let $G=(V,E)$ be an undirected graph, where V is the set of nodes and $E\subseteq V\times V$ is the set of edges. Let n and m denote the number of nodes and the number of edges, respectively, and, for each $e\in E$, let $w(e)$ be a positive real length. 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.

A *simple path* (or a *path* for short) in G is a subgraph P with $V(P)=\{v_0,v_1,\ldots,v_k|v_i\neq v_j \text{ for } i\neq j\}$ and $E(P)=\{(v_i,v_{i+1})|0\leqslant i<k\}$, also denoted as $P(v_0,v_k)=\langle v_0,\ldots,v_k\rangle$. Path $P(v_0,v_k)$ is said to go from v_0 to v_k or, alternatively, to be between v_0 and v_k. Its *length* is the sum of the lengths of the path edges, and will be denoted as $|P(v_0,v_k)|$. A graph G is *connected* if, for any two nodes $u,v\in V$, there exists a path in G going from u to v. A connected acyclic spanning subgraph of G is called a *spanning tree* of G. Let $G-v$ denote the graph obtained by removing from G the node v and its incident edges. A graph G is *2-node connected* if for any $v\in V$, $G-v$ is connected.

A path between two nodes r and s is *shortest* in G if it has minimum length among all the paths in G between r and s. In this case, we denote the path by $P_G(r,s)$, and its length, also known as the *distance* in G between r and s, by $d_G(r,s)$. For a distinguished node $r\in V$, called the *source*, and all the nodes $v\in V\setminus\{r\}$, a *single-source shortest paths tree* (SPT) $S_G(r)$ in G is a spanning tree of G rooted in r and formed by the union of shortest paths, with one shortest path from r to v for each $v\in V\setminus\{r\}$.

Let $P_{G-v}(r,s)$ be a shortest path between r and s in $G-v$, named a *replacement shortest path* for v, and let $d_{G-v}(r,s)$ denote its length. The *most vital node* (*MVN*) *problem* with respect to $P_G(r,s)$ asks for finding a node $v^*\in V\setminus\{r,s\}$ such that $d_{G-v^*}(r,s)\geqslant d_{G-v}(r,s)$, for any $v\in V\setminus\{r,s\}$.

3. An efficient solution of the MVN problem

Let $P_G(r,s)=\langle v_0,v_1,\ldots,v_k\rangle$ be a shortest path between $r=v_0$ and $s=v_k$ in G. First of all, notice that a node (other than r and s) whose removal increases the distance

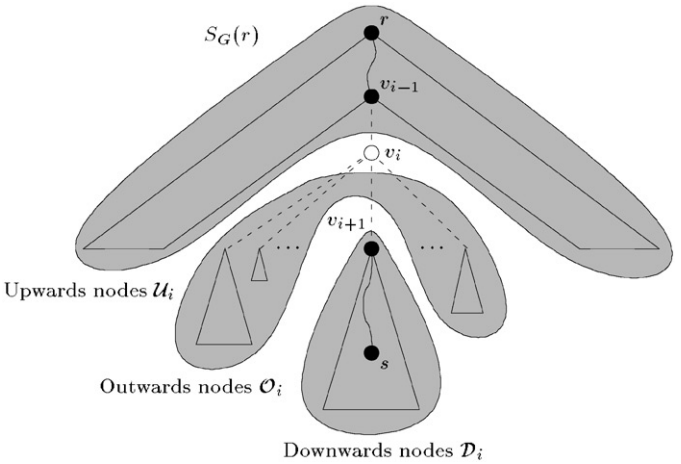

Fig. 1. Node v_i is removed from G: $S_G(r)$ is partitioned into a set of subtrees, with node sets $\mathcal{U}_i$, $\mathcal{O}_i$ and $\mathcal{D}_i$.

between r and s must belong to the node set $\{v_1,\ldots,v_{k-1}\}$. Therefore, in the following, we will consider only the removal of the nodes along the path.

3.1. The structural properties of replacement shortest paths

In this section, we present the structural properties of replacement shortest paths that will form the basis for the efficiency of our algorithm.

Let $S_G(r)$ denote a SPT in G rooted at r and containing $P_G(r,s)$, and let v_i, $1\leqslant i\leqslant k-1$, be a node on $P_G(r,s)$. When node v_i and its incident edges are removed from G, $S_G(r)$ is partitioned into a set of subtrees, that we classify as follows (see Fig. 1):

1. The subtree of $S_G(r)$ containing the parent v_{i-1} of v_i; we call the nodes of this subtree the *upwards nodes* of v_i, and we denote them as $\mathcal{U}_i$.
2. The subtree of $S_G(r)$ containing the child v_{i+1} of v_i; we call the nodes of this subtree the *downwards nodes* of v_i, and we denote them as $\mathcal{D}_i$.
3. All the remaining subtrees of $S_G(r)$; we call the nodes of the union of all these subtrees the *outwards nodes* of v_i, and we denote them as $\mathcal{O}_i$.

In the rest of the paper, we will make use of the following properties, that hold for $i,j=1,\ldots,k-1$ and $j\neq i$:

(P1) $\mathcal{U}_i\cup\mathcal{O}_i\cup\mathcal{D}_i=V\setminus\{v_i\}$;

(P2) $\mathcal{U}_i,\mathcal{O}_i$ and $\mathcal{D}_i$ are pairwise disjoint;

(P3) $\mathcal{U}_i\subset\mathcal{U}_{i+1}$;

(P4) $\mathcal{D}_{i+1}\subset\mathcal{D}_i$;

(P5) $\mathcal{O}_i\cap\mathcal{O}_j=\emptyset$.

We start by observing that for nodes in $\mathscr{U}_i$, the shortest path to r does not contain v_i. This immediately implies the following:

Lemma 1. *For each node* $u \in \mathscr{U}_i$, $d_G(r,u) = d_{G-v_i}(r,u)$.

On the other hand, for nodes in $\mathscr{D}_i$, we have that the distance to s does not change when v_i is removed.

Lemma 2. *For each node* $u \in \mathscr{D}_i$, $d_G(s,u) = d_{G-v_i}(s,u)$.

Proof. Suppose, for the sake of contradiction, that $d_G(s,u) \neq d_{G-v_i}(s,u)$. Let $S_G(s)$ be a SPT in G rooted in s containing $P_G(r,s)$. From $d_G(s,u) \neq d_{G-v_i}(s,u)$, it follows that every shortest path in G between s and u, in particular $P_G(s,u)$ in $S_G(s)$, contains v_i. Therefore, $P_G(s,u)$ has to contain its parent v_{i+1} in $S_G(s)$. Hence, the edge $e_i = (v_i, v_{i+1})$ belongs to $P_G(s,u)$, and since subpaths of shortest paths are shortest paths, we have that

$$d_G(v_{i+1},u) = w(e_i) + d_G(v_i,u) > d_G(v_i,u).$$

On the other hand, by the fact that $u \in \mathscr{D}_i$, we have that $P_G(r,u)$ in $S_G(r)$ contains v_i and v_{i+1}. Hence, since subpaths of shortest paths are shortest paths, we have that

$$d_G(v_i,u) = w(e_i) + d_G(v_{i+1},u) > d_G(v_{i+1},u),$$

that is, we have a contradiction. □

Let $E(\mathscr{U}_i \cup \mathscr{O}_i, \mathscr{D}_i) \subset E$ be the *cut* induced by $\mathscr{D}_i$ in $G - v_i$, i.e., the set of edges of $G - v_i$ with one end node in $\mathscr{U}_i \cup \mathscr{O}_i$ and the other one in $\mathscr{D}_i$. In the following, an edge $f = (x,y) \in E(\mathscr{U}_i \cup \mathscr{O}_i, \mathscr{D}_i)$ will be considered as having node y in $\mathscr{D}_i$. Edges in $E(\mathscr{U}_i \cup \mathscr{O}_i, \mathscr{D}_i)$ will be named the *crossing edges* associated with v_i. Since any replacement shortest path for node v_i must use a crossing edge, the length of the path can be expressed as follows:

$$d_{G-v_i}(r,s) = \min_{f=(x,y) \in E(\mathscr{U}_i \cup \mathscr{O}_i, \mathscr{D}_i)} \{d_{G-v_i}(r,x) + w(f) + d_{G-v_i}(y,s)\}. \tag{1}$$

This immediately suggests an algorithm to solve the MVN problem, but unfortunately we do not know how to compute all the $d_{G-v_i}(r,x)$ distances sufficiently fast, since this might require the computation, for each node v_i on $P_G(r,s)$, of the SPT rooted in r in $G - v_i$. Therefore, let us look at replacement shortest paths more closely. In a path in $G - v_i$ from r to s, let us call the path node $y \in \mathscr{D}_i$ closest to r the *entry node* (into $\mathscr{D}_i$) of the path. We can prove that, to minimize (1), not all the distances in $G - v_i$ between r and nodes in $\mathscr{U}_i \cup \mathscr{O}_i$ need to be computed:

Lemma 3. *Any replacement shortest path for node* v_i *can be expressed as a concatenation of* $P_{G-v_i-\mathscr{D}_i}(r,x)$, *edge* (x,y) *and* $P_G(y,s)$, *where* y *is the entry node into* $\mathscr{D}_i$, $P_{G-v_i-\mathscr{D}_i}(r,x)$ *is a shortest path from* r *to* x *in* $G - v_i - \mathscr{D}_i$, *and* $P_G(y,s)$ *is a shortest path from* y *to* s *in* G.

Proof. Since r is not contained in $\mathscr{D}_i$, but s is, there is a first node on the path traced from r towards s that belongs to $\mathscr{D}_i$. Call that node y, and call its predecessor on the path x. This proves the first and second part of the claim. Part three is due to Lemma 2. □

3.2. Computing components of distances

Lemma 3 allows us to compute replacement shortest paths as follows. First, we compute in G a SPT rooted in s. This gives us all distances $d_{G-v_i}(y,s)$ for $y \in \mathscr{D}_i$, $i=1,\ldots,k-1$. Second, we compute all paths $P_{G-v_i-\mathscr{D}_i}(r,x)$ from r to $x \in \mathscr{U}_i \cup \mathscr{O}_i$, $i=1,\ldots,k-1$. This is described in more detail in the following paragraph. Third, we compose these distance components by inspecting all crossing edges as we go along the nodes v_i, $i=1,\ldots,k-1$. This is described in more detail in Section 3.3.

For $x \in \mathscr{U}_i$, from Lemma 1 we have $d_{G-v_i-\mathscr{D}_i}(r,x)=d_G(r,x)$, and therefore the SPT $S_G(r)$ gives us all these values. The remaining more interesting task is the computation of $d_{G-v_i-\mathscr{D}_i}(r,x)$ for $x \in \mathscr{O}_i$. We propose to do this as follows, making use of $S_G(r)$. When node v_i, $1 \leqslant i \leqslant k-1$, is removed, we consider the subtree of $S_G(r)$ induced by $\mathscr{U}_i$ – which is of course a SPT rooted in r of the subgraph of G induced by the node set $\mathscr{U}_i$. Then we compute the distance from r to all the nodes in $\mathscr{O}_i$ in the subgraph of G induced by $\mathscr{U}_i \cup \mathscr{O}_i$. We do this by applying Dijkstra's algorithm [4] in the following way to the nodes in $\mathscr{O}_i$, starting from the precomputed distances for $\mathscr{U}_i$. Let $E(\mathscr{U}_i,\mathscr{O}_i)$ be the subset of edges in E having one end node in $\mathscr{U}_i$ and the other one in $\mathscr{O}_i$, let $E(\mathscr{U}_i,x)$ be the subset of edges in $E(\mathscr{U}_i,\mathscr{O}_i)$ having one end node in $\mathscr{U}_i$ and the other one in $x \in \mathscr{O}_i$, and let $E(\mathscr{O}_i,\mathscr{O}_i)$ be the subset of edges in E having both end nodes in $\mathscr{O}_i$. We create an initially empty heap $\mathscr{H}_i$, inserting into it all the nodes $x \in \mathscr{O}_i$, with key

$$k(x) = \begin{cases} \min\limits_{f=(u,x)\in E(\mathscr{U}_i,x)} \{d_{G-v_i}(r,u) + w(f)\} & \text{if } E(\mathscr{U}_i,x) \neq \emptyset; \\ +\infty & \text{otherwise.} \end{cases} \tag{2}$$

Afterwards, we extract the minimum $k(x)$ from $\mathscr{H}_i$, corresponding to $d_{G-v_i-\mathscr{D}_i}(r,x)$. Then, we update the keys of adjacent nodes still appearing in $\mathscr{H}_i$, by making use of edges in $E(\mathscr{O}_i,\mathscr{O}_i)$, exactly as in Dijkstra's algorithm. The algorithm iterates until $\mathscr{H}_i$ is empty.

This algorithm has an efficient implementation, as expressed in the following lemma:

Lemma 4. *The values $d_{G-v_i-\mathscr{D}_i}(r,x)$ for all nodes $x \in \mathscr{O}_i$, $i=1,\ldots,k-1$, can be computed in $O(m+n\log n)$ time and $O(m)$ space.*

Proof. The initial computation of $S_G(r)$ takes $O(m+n\log n)$ time and $O(m)$ space [5]. Throughout the $k-1$ iterations in our algorithm, $k-1=O(n)$ heaps are created. Let n_i denote the number of nodes of $\mathscr{O}_i$, and let $m_i=|E(\mathscr{U}_i,\mathscr{O}_i) \cup E(\mathscr{O}_i,\mathscr{O}_i)|$. On heap $\mathscr{H}_i$, we perform $O(n_i)$ *Insert* operations, and from Lemma 1, key initialization can be performed in $O(m_i)$ time once $S_G(r)$ has been computed. Moreover, we perform $O(n_i)$

ExtractMin and $O(m_i)$ *DecreaseKey* operations. By using Fibonacci heaps [5], for all the nodes $x \in \mathcal{O}_i$, $d_{G-v_i-\mathcal{D}_i}(r,x)$ can then be computed in $O(m_i + n_i \log n_i)$ time.

Since each *DecreaseKey* operation is associated with an edge of G, and each edge of G is considered at most twice, and given that sets $\mathcal{O}_i$ are disjoint, we finally have that the total time is

$$\sum_{i=1}^{k-1} O(m_i + n_i \log n_i) = O(m + n \log n). \qquad \square$$

3.3. Combining components of distances

We are now ready to combine the distance components computed so far. We first give a description of the algorithm, and we then analyze its correctness and its time and space complexity.

We consider the nodes $v_1, \ldots, v_{k-1}$ in this order along $P_G(r,s)$, and when the node v_i is considered, we maintain in a heap $\mathcal{H}$ the set of nodes $\mathcal{D}_i$ associated with it. For each node $y \in \mathcal{D}_i$, we consider the subset of edges in $E(\mathcal{U}_i \cup \mathcal{O}_i, \mathcal{D}_i)$ incident to y, denoted as $E(\mathcal{U}_i \cup \mathcal{O}_i, y)$. In the heap, with node y a key $k(y)$ is associated that satisfies the following condition immediately before a *FindMin* operation on $\mathcal{H}$ is performed:

$$k(y) = \min_{f=(x,y)\in E(\mathcal{U}_i \cup \mathcal{O}_i, y)} \{d_{G-v_i-\mathcal{D}_i}(r,x) + w(f) + d_G(y,s)\}. \tag{3}$$

Notice that in general, this key value is not the length of a shortest path in $G - v_i$ from r to s through y, but, as we explained in Section 3.1, we cannot afford to maintain these latter values. We will show later that these keys, however, give us sufficient information to solve the problem.

The algorithm works in stages. At the beginning, the heap $\mathcal{H}$ is created for $\mathcal{D}_0$, that is, all the nodes in the subtree $\mathcal{D}_0$ rooted at v_1 in $S_G(r)$ are inserted, with arbitrarily large keys associated. At the ith stage, we consider the node v_i on $P_G(r,s)$, and we update the heap in the following way:

Step 1: We remove from $\mathcal{H}$ the node v_i and the nodes $\mathcal{O}_i$ associated with it. (`Comment:` This leaves exactly the nodes in $\mathcal{D}_i$ in $\mathcal{H}$; we update their keys in Steps 2 and 3.)

Step 2: We consider all the nodes in $\mathcal{O}_i$; for each such node x, we inspect its incident edges, and we limit further actions on those crossing into $\mathcal{D}_i$. Let $f = (x, y)$ be one of these crossing edges, if any, and let

$$k' = d_{G-v_i-\mathcal{D}_i}(r,x) + w(f) + d_G(y,s), \tag{4}$$

where $d_{G-v_i-\mathcal{D}_i}(r,x)$ has been computed by means of the procedure described in Section 3.2. If $k' < k(y)$, we decrease the key of y in $\mathcal{H}$ to value k'. (`Comment:` When this step is completed, all the crossing edges associated with v_i and induced by its removal have been exhausted.)

Step 3: We then consider all the nodes in $v_{i-1} \cup \mathcal{O}_{i-1}$; for each node x in this set, we look at its incident edges, and we limit further actions on those crossing into $\mathcal{D}_i$.

Let $f=(x,y)$ be one of these crossing edges, if any, and let

$$k' = d_G(r,x) + w(f) + d_G(y,s). \tag{5}$$

If $k'<k(y)$, we decrease the key of y in $\mathscr{H}$ to value k'. (`Comment:` When this step is completed, all the crossing edges associated with v_i and induced by the reinsertion of v_{i-1} have been exhausted, and the corresponding key maintenance in the heap is complete.)

Step 4: We finally find the minimum of $\mathscr{H}$. (`Comment:` We will prove shortly that the key associated with this minimum is exactly the length of a replacement shortest path in $G-v_i$ between r and s, that is $d_{G-v_i}(r,s)$.)

When all stages $1,\ldots,k-1$ have been completed, a most vital node can then be determined as a node v^* on $P_G(r,s)$ such that

$$d_{G-v^*}(r,s) = \max_{i=1,\ldots,k-1} \{d_{G-v_i}(r,s)\}. \tag{6}$$

Let us now prove that our algorithm indeed computes at each stage the length of a corresponding replacement shortest path.

Lemma 5. *The minimum key found in $\mathscr{H}$ at the ith stage is the length of a replacement shortest path between r and s in $G-v_i$.*

Proof. We prove the lemma in two steps. First, we prove that each key in the heap $\mathscr{H}$, say $k(y)$ for node y, is the length of a shortest path in $G-v_i$ from r to s through the entry node y. The reason is that our algorithm inspects all crossing edges (x,y) incident to y, and keeps track of the best.

Second, we prove that at least one node in the heap has a key corresponding to the length of a replacement shortest path between r and s in $G-v_i$. In fact, for any replacement shortest path $P_{G-v_i}(r,s)$, the corresponding entry node y is in $\mathscr{H}$. Let x be its predecessor on $P_{G-v_i}(r,s)$. Then, $k(y)$ equals the length of $P_{G-v_i}(r,s)$, because the prefix of such a path from r to x is contained in $G-v_i-\mathscr{D}_i$, and then $d_{G-v_i-\mathscr{D}_i}(r,x)=d_{G-v_i}(r,x)$. Therefore, (1) and (3) are both minimized when edge (x,y) is considered, and $k(y)=d_{G-v_i}(r,s)$. □

The following theorem can finally be proved:

Theorem 1. *A most vital node on a shortest path $P_G(r,s)$ between two nodes r and s in a 2-node connected, undirected graph $G=(V,E)$ with n nodes and m edges, with positive real edge lengths, can be determined in $O(m+n\log n)$ time and $O(m)$ space.*

Proof. The correctness of the above algorithm derives from Lemma 5. The time complexity follows from that of Lemma 4 for the initial phase. This allows us to compute (4) in O(1) time for each crossing edge. Clearly, (5) can be computed in O(1) time for each crossing edge as well, once $S_G(r)$ and $S_G(s)$ have been computed. Globally, we perform $O(m)$ computations of (4) and (5), since a crossing edge is checked at

most twice, once each in Steps 2 and 3. Then, we make use of a Fibonacci heap [5] for maintaining $\mathscr{H}$. Since each node of G is inserted into the heap and removed from it at most once, we have a single *MakeHeap*, $O(n)$ *Insert*, $k-1=O(n)$ *FindMin*, $O(n)$ *Delete* and $O(m)$ *DecreaseKey* operations (since a key may be decreased only when a new crossing edge is considered), and thus we obtain a total time of $O(m+n\log n)$ for heap operations. The time complexity for other tasks is, respectively, $O(m+n\log n)$ time for computing $S_G(r)$ and $S_G(s)$, $O(n)$ time for managing sets $\mathscr{O}_i$, $i=1,\ldots,k-1$, and $O(n)$ time for computing (6). Finally, $O(m)$ space is trivially enough to handle all the operations. Thus, the claim follows. □

4. Computing the replacement shortest paths

Implicitly, our algorithm computes not only the lengths of replacement paths, but also the paths themselves, and it can be easily modified to do so explicitly, without any additional space and time overhead. In fact, let $p_r(v)$ and $p_s(v)$ denote the parent of a node v in $S_G(r)$ and $S_G(s)$, respectively. Moreover, for a node $v\in\mathscr{O}_i$, let $p(v)$ denote its parent in the (partial) SPT $S_{G-v_i}(r)$ obtained from the incremental application of the Dijkstra's algorithm described in Section 3.2. Notice that for some of the nodes in V, this value will remain undefined, as a consequence of the above procedure. However, as we already know, these nodes do not belong to any replacement shortest path.

By making use of the above pointers, we have that a replacement shortest path $P_{G-v_i}(r,s)$ between r and s in $G-v_i$ can be computed as follows. First of all, we modify the heap $\mathscr{H}$ in such a way that it accommodates, along with each element y it contains, also the respective crossing edge minimizing the associated key. Hence, assume that $f_i=(x_i,y_i)$ denotes the edge associated with the minimum key found by the algorithm at the ith stage. The path $P_{G-v_i}(r,s)$ is computed by connecting paths $P_{G-v_i}(r,x_i)$ and $P_{G-v_i}(y_i,s)$ through the edge f_i. The latter path can be easily computed starting from y_i and by making use of the parents in $S_G(s)$. Concerning the former path, we build it starting from x_i, and by making use of parents from $S_{G-v_i}(r)$ and from $S_G(r)$, for the outwards and the upwards nodes of v_i, respectively. More precisely, let u_i be the node of $\mathscr{U}_i$ (not necessarily distinct from x_i) first encountered when moving from x_i towards r. Such a node can be easily detected in $O(1)$ time. Then, we have that

$$P_{G-v_i}(r,s)=\Big\langle \underbrace{r,\ldots,p_r(u_i)}_{P_G(r,p_r(u_i))},\underbrace{u_i,\ldots,p(x_i),x_i}_{P_{G-v_i}(u_i,x_i)},\underbrace{y_i,p_s(y_i),\ldots,s}_{P_G(y_i,s)}\Big\rangle.$$

We therefore have the following:

Corollary 1. *Given an undirected, 2-node connected graph $G=(V,E)$ with n nodes and m edges, with positive real edge lengths, and given a shortest path $P_G(r,s)$ between two nodes r and s in G, the set of replacement shortest paths between r and s for all the nodes of $P_G(r,s)$ can be computed in $O(m+n\log n)$ time and $O(m)$ space.*

5. Discussion

In this paper, we have presented a fast solution to the problem of finding a most vital node along a shortest path $P_G(r,s)$ between two nodes r and s in a graph G. It runs in $O(m + n \log n)$ time and $O(m)$ space, which, as far as we know, is the first improvement over the trivial bound of $O(nm + n^2 \log n)$ time and $O(m)$ space that we get by recomputing a replacement shortest path between r and s from scratch after the removal of each node along $P_G(r,s)$.

In some applications, such as transportation networks, it appears to be more realistic to associate costs with both, nodes and edges, instead of only one type of network components. Our approach also answers the corresponding more general question that suggests itself: In a graph where both edges and nodes have a positive cost, and where both edges and nodes can fail, what is a most vital edge or node on a shortest path? The algorithm can be modified slightly and still runs within the same asymptotic bounds for this more general question, for two reasons. First, edge failures can be modelled as node failures, when each edge is replaced by a path of length two with an extra node in the center of that path; then, the failure of the extra node represents the failure of the original edge. Second, Dijkstra's algorithm can be adapted easily to work also for shortest path computations in graphs with costs on edges and nodes, where the cost of a path is the sum of the costs of its edges and nodes. Obviously, both modifications do not change the asymptotic bounds for the runtime and the storage space.

Our algorithmic solution is also useful in quite a different application context. In large networks, components (nodes and edges) may be owned by different owners. The incentive of an owner of a component to forward a message, naturally, is to get some reward. In standard economic terms, that reward is the price of the service of forwarding the message. It is economically desirable that each owner declares the true price for the service that its component offers, so as to allocate the overall resources in a best possible way. Nevertheless, there is an incentive for owners to speculate and ask for a higher price, in the hope of getting a higher profit. This leads to economically suboptimal resource allocation and is therefore undesirable. A few studies in the computer science literature have devoted their attention to setting the boundary conditions in such a way that speculating with high prices does not pay off. This is known as *mechanism design* for selfish agents [10–12]. In [11], Nisan and Ronen are explicitly suggesting a rewarding model for forwarding messages on paths, based on microeconomic theory, that requires the computation of replacement path lengths for edges. This model assumes that only edges charge a price for forwarding a message; nodes perform their service for free. Here, again, it would be more realistic to have a price for both, edges and nodes, than a limitation of the pricing to the edges alone. A straightforward modification of the charging scheme from [11] to node and edge prices serves the desired purpose. Now, the modification of our algorithm for node and edge costs and failures is an efficient implementation of the required replacement path computations.

Our solution is efficient, but it is still open whether it is optimal. Notice that to improve our solution, a faster computation of a single source shortest paths tree must

be provided. For more general settings there are still many open problems; one of them deals with multiple edge or node failures on a shortest path.

References

[1] M.O. Ball, B.L. Golden, R.V. Vohra, Finding the most vital arcs in a network, Oper. Res. Lett. 8 (1989) 73–76.
[2] A. Bar-Noy, S. Khuller, B. Schieber, The complexity of finding most vital arcs and nodes, TR CS-TR-3539, Institute for Advanced Studies, University of Maryland, College Park, MD, 1995.
[3] H.W. Corley, D.Y. Sha, Most vital links and nodes in weighted networks, Oper. Res. Lett. 1 (1982) 157–160.
[4] E.W. Dijkstra, A note on two problems in connection with graphs, Numer. Math. 1 (1959) 269–271.
[5] M.L. Fredman, R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, J. ACM 34 (3) (1987) 596–615.
[6] J. Hershberger, S. Suri, Vickrey prices and shortest paths: what is an edge worth? Proc. 42nd Annu. IEEE Symp. on Foundations of Computer Science (FOCS'01), 2001, pp. 252–260.
[7] K. Malik, A.K. Mittal, S.K. Gupta, The k most vital arcs in the shortest path problem, Oper. Res. Lett. 8 (1989) 223–227.
[8] E. Nardelli, G. Proietti, P. Widmayer, Finding the detour-critical edge of a shortest path between two nodes, Inform. Process. Lett. 67 (1) (1998) 51–54.
[9] E. Nardelli, G. Proietti, P. Widmayer, A faster computation of the most vital edge of a shortest path between two nodes, Inform. Process. Lett. 79 (2) (2001) 81–85.
[10] N. Nisan, Algorithms for selfish agents, Proc. 16th Symp. on Theoretical Aspects of Computer Science (STACS'99), Lecture Notes in Comput. Sci., vol. 1563, Springer, Berlin, 1999, pp. 1–15.
[11] N. Nisan, A. Ronen, Algorithmic mechanism design, Proc. 31st Annu. ACM Symp. on Theory of Computing (STOC'99), 1999, pp. 129–140.
[12] J.S. Rosenschein, G. Zlotkin, Rules of Encounter: Designing Conventions for Automated Negotiation Among Computers, MIT Press, Cambridge, MA, 1994.
[13] R.E. Tarjan, Efficiency of a good but not linear set union algorithm, J. ACM 22 (1975) 215–225.
[14] S. Venema, H. Shen, F. Suraweera, A parallel algorithm for the single most vital vertex problem with respect to single source shortest paths, Online Proc. First Internat. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT'2000), Chapter 22, http://www2.comp.polyu.edu.hk/PDCAT2000/publish.html.