Polynomial-Time Computable
Backup Tables for Shortest-Path Routing

H. ITO
Kyoto University, Japan

K. IWAMA
Kyoto University, Japan

Y. OKABE
Kyoto University, Japan

T. YOSHIHIRO
Wakayama University, Japan

Abstract
For networks employing shortest-path routing, we introduce a new recovery scheme which needs only one backup routing table. By precomputing this backup table, the network recovers from any single link-failure immediately after the failure occurs. It is shown that this scheme always works if the network is given as a two-edge-connected symmetric digraph and that an optimal backup table can be computed in time $O(|V||E|)$. Furthermore, this time complexity can be reduced to linear if all edges of the graph have the same cost.

Keywords
routing, shortest paths, fault tolerance, backup tables, graph algorithms

1 Introduction
It is definitely important to design a system which can recover from a failure quickly and inexpensively. In the Internet, a typical failure is a disconnection of a link between two nodes. If such a failure occurs, our current system recomputes the routing table of each node so that packets will bypass the failed link. Of course we do not have to do this recomputation from scratch (several dynamic algorithms have been reported as described later). Nevertheless the cost for the recomputation does not seem too small. Moreover, the information of the failure has to be transmitted from one node to another, which might be even more costly.
As a result, it is said to be hard to avoid a lot of packet loss during the recovering phases.

One obvious approach to make such a recovering process quicker is to pre-compute and hold several different routing tables in the event of a link failure. For this purpose, one can use the routing scheme based on $k$ independent spanning trees, which will also be described in more detail later. (A simple extension of the previous approach, i.e., to hold all the different routing tables according to failures of each link, is obviously not realistic since the number of such tables can be as large as the number of entire links.) Because the set of $k$ independent spanning trees yields $k$ disjoint paths between any pair of nodes, this approach apparently tolerates up to $k$ link losses. Unfortunately, it also has a serious drawback, namely, none of the $k$ independent spanning trees may be a shortest-path tree. Since the Internet has been developed under the assumption of shortest-path routing (at least under the normal, fault-free condition), it is quite hard to convince people with this approach.

Our Contribution. In this paper we present a new scheme which uses the standard shortest-path routing under the normal condition and needs only one (fixed) backup table at every node to tolerate any single link-failure. This might seem impossible. See Fig. 1(a) for example: Note that the shortest-path tree uses both links from $x$ to $d$ and $y$ to $d$. Therefore our (single) backup table cannot use either of them in the event of the failure of each: a packet can never reach its destination $d$ if it follows the backup table! Our small trick is to use another table called a switching table, which is actually not an independent table but just places “marks” on some of the backup links.

See Fig. 1(a) again. The solid arrows indicate primary (shortest-path) routes for the destination $d$. The broken and chain arrows indicate backup routes for the same destination $d$, where the chain arrows constitute our switching table. Our
scheme works as follows: Suppose that the link from node $x$ to $d$ fails as shown in Fig. 1(b). Then a packet which is at node $x$ changes its mode from the normal mode to the backup mode by setting a special bit in its header. Now the backup-mode packet moves using the backup table until it passes a switching link denoted by a chain in the figure. Once it passes the switching link, the packet changes its mode back to the normal mode and it then travels along the normal route until it reaches the destination. This is shown in Fig. 1(b) by bold arrows. Note that the same backup and switching tables can also be used for another link failure, e.g., for the failure of the link from $y$ to $d$ as shown in Fig. 1(c).

In this paper we show the following results:

1. Such a pair of backup and switching tables always exists if the underlying network is given as a two-edge-connected, symmetric, directed graph.

2. Among those feasible solutions, we can compute an optimal pair in polynomial time.

3. Furthermore, this time complexity can be reduced to linear if all edges of the graph have the same cost.

Previous Work. As mentioned before, there are two categories of previous work. One is dynamic algorithms to recompute a shortest-path tree under a link failure. The dynamic algorithms [2] and [3] compute a new shortest-path tree quickly when topology of a network changes (sometimes resulted from link failures), using the information of the old shortest-path tree. As another result of the dynamic algorithms, Nardelli, et al. [9] presented an algorithm that computes a substitutive edge to combine the shortest-path tree divided by an edge failure. Both methods are fairly quick like $O(n)$, but this computation time is still far from negligible.

The other is on $k$ independent spanning trees. Itai, et al. [6] presented a routing scheme with multiple routing trees. In this scheme, $k$ distinct independent spanning trees are computed for each destination vertex, and the same copies of packets are sent along each tree. They also showed that a $k$-edge-connected graph always has $k$ distinct independent spanning trees. It is worth noted that on independent spanning trees, there is a great deal of literature. Several kind of independent spanning trees have been considered and the number of independent spanning trees that can coexist on a graph of various classes has also been studied [5]. Also, several algorithms for computing independent spanning trees on various graphs have also been studied [1][8].

Finally, a severe congestion of a link may be regarded as a kind of failure, for which there is also a large literature including [7] that shows how to avoid routing loops, when bypassing the busy link, only by adjusting the cost of each link.
2 Backup and Switching Tables

Our problem can be formulated as an optimization problem for directed graphs. A directed graph (or simply a graph) is denoted by $G = (V, E)$, where $V$ is a set of vertices and $E$ is a set of (directed) edges. (We use the standard graph terminology instead of nodes, links, etc.) For an edge $(u, v) \in E$, $l((u, v))$ (or simply written as $l(u, v)$) denotes the length of the edge $(u, v)$. We assume that a graph $G$ is always two-edge connected, i.e., there are at least two edge-disjoint (directed) paths between every pair of vertices. Note that if the connectivity is less than two, then it is generally not possible to recover a single-edge failure. A sequence of edges $p = (u_1, u_2) (u_2, u_3) \cdots (u_{n-1}, u_n)$ ($u_i \neq u_j$ for any $i \neq j$) is called a path and its length, denoted by $l(p)$, is defined as the sum of the length of the edges, i.e., $l(p) = \sum_{i=1}^{n-1} l(u_i, u_{i+1})$. A path $p$ from $u$ to $v$ is a shortest path if $p$ has the smallest length among all the paths from $u$ to $v$. A shortest-path tree $T_d$ for a destination $d$ is a spanning tree on $G$ such that the outdegree (the number of leaving edges) of $d$ is zero, the outdegree of a vertex except for $d$ is one, and the unique path from each vertex to $d$ is always a shortest path.

Suppose that we are given a graph $G = (V, E)$, an edge-length function $l$, a destination vertex $d \in V$, and a shortest-path tree $T_d$. Then our problem is to obtain two sets, $B$ and $S$ called a backup table and a switching table, respectively, which satisfy the following conditions. ($B$ and $S$ are also called a set of backup edges and a set of switching edges, respectively.)

1. $B \subseteq E - E[T_d]$ and $S \subseteq B$, namely, $B$ never includes edges in the shortest-path tree.

2. For every vertex $v \in V - \{d\}$, there is exactly one edge $e$ in $B$ such that $t(e) = v$, where $t(e)$ denotes the tail of $e$, namely, $e$ outgoes from vertex $t(e)$.

3. From every vertex $v \in V - \{d\}$, there is a path $e_1 e_2 \cdots e_h$ from $v$ to $d$, such that $e_1, \ldots, e_{h-1} \in B$, $e_h \in S$, and $e_{h+1}, \ldots, e_k \in E[T_d]$. Any path from $v$ to $d$ is said to be a backup path from $v$ (which is supposed to fail) if it does not include the $T_d$-edge from $v$. Thus the above path $e_1, \ldots, e_k$ is obviously a backup path from $v$, which is unique by the condition (2).

Recall our recovery scheme for a failure of a single edge $e$ whose outline was given in the previous section. One can see that the third condition guarantees that the scheme works. Let $t(e) = v$ and suppose that a packet whose destination is $d$ is now at the vertex $v$ (it may start from $v$ or may have traveled on $E[T_d]$ from another vertex to $v$). Then what we do is the following:

1. Make the mode of the packet the backup mode (for which we prepare a single bit, called the mode bit, in the header of the packet and set that bit).
(ii) A backup-mode packet uses the backup table; in other words, it travels on edges in \( B \).

(iii) Once the packet passes an edge in \( S \), then its mode is again changed into the normal mode by resetting the mode bit.

(iv) A normal-mode packet uses the original routing table, i.e., it travels on \( T_d \) and finally goes to \( d \).

Thus our scheme needs only one extra table and the original routing table \( T_d \) can be a standard shortest-path routing table. Also the overhead is moderate (only one extra bit in the packet header is necessary); this approach appears to have a lot of merit. Unfortunately, such tables \( B \) and \( S \) do not exist in general. Here is a simple example: See Fig. 2(a), which shows a graph in which the link between vertices \( x \) and \( y \) is unidirectional. Suppose that the edge from \( x \) to \( d \) fails. Then one can see easily that any backup path from \( x \) has to pass the \( T_d \)-edge from \( z \) to \( y \) (because we have no edge from \( x \) to \( y \)). The condition (3) requires that once the packet passes a \( T_d \)-edge then it must continue doing so until it finally gets to \( d \). This means that the packet along the above backup path has to move from \( y \) to \( x \), a contradiction.

3 Basic Algorithms

Now we are ready to give our first result: Our scheme given in the previous section always works for two-edge-connected symmetric graphs. A graph \( G = (V, E) \) is said to be symmetric if for any edge \((u, v)\) in \( E \), its reverse edge \((v, u)\) is also in \( E \) and \( l(u, v) = l(v, u) \). Since a communication link is usually bidirectional, this symmetry condition is obviously not too strong.
Theorem 1  There always exists a pair $(B, S)$ of a backup table and a switching table for a two-edge-connected symmetric graph.

Proof.  For our proof, we need a few more definitions. See Fig. 3. Let us fix an arbitrary vertex $u (\neq d)$ and consider a backup path from $u$. An edge $e$ is called an escaping edge if $e$ goes from a vertex in the subtree of $T_d$ rooted at $u$ to a vertex outside the subtree. A backup path from $u$ is said to be fundamental if it passes along $T_d$ reversely, then passes some escaping edge, and after that moves to $d$ along $T_d$. In Fig. 3, the path denoted by a thick line is fundamental and the path denoted by a dotted line is not.

Note that a fundamental path from a vertex $v$ is not unique in general. Our basic idea is to select a single fundamental path $F_v$ from each vertex $v \in V - \{d\}$ and make it the backup path from $v$ when the outgoing edge from $v$ fails. As such $F_v$, the shortest fundamental path from $v$ might be desirable. Unfortunately, this is not always possible for the following reason. See Fig. 4. The shortest fundamental path from a vertex $u$ and the one from another vertex $v$ fork at vertex $w$. (This is possible since a fundamental path from $v$ can use the $T_{d, u}$-edge from $u$ but a fundamental path from $u$ cannot. Also, such a branch can happen only if $u$ is an ancestor of $v$ or vice versa.) Since we can include only one outgoing edge from each vertex in $B$, we cannot select these two fundamental paths as $F_u$ and $F_v$. What we do in such a situation is to change the fundamental path from $v$ (farther from the root than $u$) to the subpath of the shortest fundamental path from $u$ as shown in Fig. 4(b), and we let it be $F_v$. More formally we compute $F_v$ as follows:

Let $v_1, v_2, \ldots, v_{n-1} \in V - \{d\}$ be the breadth-first order of $T_d$ vertices. Then compute the shortest fundamental path from $v_1$ and we let this path be $F_{v_1}$. We next look at $v_2$ and if $v_2$ is on some of the previously computed paths (only $F_{v_1}$ at present), then we let its subpath starting from $v_2$ be $F_{v_2}$. Otherwise, we compute a
new shortest fundamental path from $v_2$ and let it be $F_{v_2}$. Repeat the same operation for $v_3, v_4, \ldots, v_{n-1}$. Now let

$$B = \{ e \mid e \in F_v \text{ for some } v \} - E[T_d],$$

$$S = \{ e \mid e \text{ is in } B \text{ and is an escaping edge of some subtree of } T_d \}.$$

It should be noted that if an edge is an escaping edge of some subtree of $T_d$, then it is not a reversal edge of any $e'$ in $E[T_d]$. It then follows that these $B$ and $S$ satisfy all the conditions (1) to (3) for the backup and switching tables.

Once we know that there is always a pair of backup and switching tables for a symmetric graph, we in turn wish to select an optimal one. One natural measure of the optimality is the length of the longest backup path determined by $B$ and $S$. The problem of obtaining such an optimal pair of $B$ and $S$ is denoted by SHORTEST-BACKUP. This optimization problem can be solved in polynomial time.

**Theorem 2** SHORTEST-BACKUP can be solved in time $O(|V||E|)$.

**Proof.** We can use exactly the same algorithm as described in the proof of the previous theorem. As for its complexity, it is enough to show that we can compute the shortest fundamental path from each vertex in time $O(|E|)$: we search all the escaping edges for each subtree rooted at each vertex and compare the length of the fundamental paths including them. (Note that a starting vertex and an escaping edge determine a unique fundamental path.) Let $u$ be a starting vertex and $(v, w)$ be an escaping edge of some fundamental path, then its length is computed from the distances from $u$, $v$, and $w$ to $d$ in constant time. Thus, although searching edges takes $O(|E|)$ time, this can be done also in time $O(|E|)$ if we previously compute the distances from each vertex to $d$ in time $O(|E|)$. 

\[ \text{Figure 4: Proof of Theorem 1} \]
As for the correctness of the algorithm, we only give the proof ideas: (i) consider two backup paths $p_1$ and $p_2$ from a vertex $u$ such that $p_1$ is fundamental but $p_2$ is not and both $p_1$ and $p_2$ use the same escaping edge (as shown in Fig. 3). Then $p_1$ is not longer than $p_2$. (Otherwise, the reverse of $p_1$ from the tail of the escaping edge to $u$ is longer than the same portion of $p_2$ because of the symmetry condition, which contradicts the fact that $T_d$ is a shortest-path tree.) Thus using a shortest fundamental path is suitable for our purpose. (ii) If two shortest fundamental paths from $u$ and $v$ fork at some vertex, we give a priority to the (longer) path which begins from the node closer to the root. This also contributes to reduce the length of the longest backup path.

4 Linear-Time Algorithms

In this section, we consider the case that all edges have the same length, i.e., we assume the length is 1.0. In this case we can reduce the running time of the algorithm to linear. On introducing the new algorithm, we give an additional definition: let $h(v)$ be the height of a vertex $v$ defined as the length of the shortest path from $v$ to $d$.

4.1 Basic Strategies

Our basic strategy for the linear-time algorithm is to look at the edges in $E - E[T_d]$ in the increasing order of the height of their tail vertices, and for each of them, find fundamental paths including the edge as their escaping edge and fix them. To show the correctness of this strategy, the following two lemmas are essential.

Lemma 1 For every edge $(u, v) \notin E[T_d]$, the difference of $h(u)$ and $h(v)$ is at most one.

Proof. We assume $h(u) > h(v)$ without loss of generality. Let $p_u$ (resp., $p_v$) be a path from $u$ (resp., $v$) to $d$ along $T_d$. Both are shortest paths between their terminals and their length are $h(u)$ and $h(v)$, respectively. Now assume that $h(u) > h(v) + 1$, then the length of the path $(u, v) \cap p_v$ is $h(v) + 1$, contradicting to the fact that $p_u$ is a shortest path. □

Lemma 2 Let $p_1$ and $p_2$ be two fundamental paths which depart from the same vertex $s$. Let $u_1$ and $u_2$ be the tail vertices of their escaping edges, respectively. Then, $h(u_1) > h(u_2)$ implies $l(p_1) \geq l(p_2)$.

Proof. We denote the escaping edges of $p_1$ and $p_2$ by $(u_1, v_1)$ and $(u_2, v_2)$, respectively. Path $p_1$ consists of the path from $s$ to $u_1$, the escaping edge, and the
path from \( u_1 \) to \( d \). Hence,

\[
\ell(p_1) = h(u_1) - h(s) + h(v_1) + 1. \tag{1}
\]

From Lemma 1, the difference of \( h(u_1) \) and \( h(v_1) \) is at most one, thus,

\[
h(u_1) - 1 \leq h(v_1) \leq h(u_1) + 1. \tag{2}
\]

Let \( k = h(u_1) - h(u_2) \). Then from (1) and (2),

\[
\ell(p_1) \geq 2h(u_2) - h(s) + 2k. \tag{3}
\]

On the other hand, (1) and (2) also hold if we replace all the subscripts by “2”, namely

\[
\ell(p_2) \leq 2h(u_2) - h(s) + 2. \tag{4}
\]

From (3) and (4), we obtain

\[
\ell(p_2) \leq \ell(p_1). \tag{5}
\]

Those lemmas indicate how the length of fundamental paths differs according to the height of the tails and heads of their escaping edges. Lemma 2 implies that, as long as they start with the same vertex, their length can be compared by just seeing the height of the tails. Further, Lemma 1 implies that, even if the height of the tails is the same, the length is determined from the three different values according to the height of the heads. It is useful that those lemmas give us the way of selecting the shortest fundamental path starting from each vertex in \( V - \{d\} \). The operations are as follows: first, prepare three buckets on each height of vertices (i.e., for each height of heads and tails) and classify edges \( e \notin E[T_d] \) into the buckets accordingly. Next, look at edges sequentially in the increasing order of bucket height (handle buckets of lower tails faster, and if the height of tails are the same, handle buckets of lower heads faster), and for each of them find fundamental paths whose escaping edge is the edge. Because the order of the fundamental paths appearing through this process is the order of the length of them, if we select the firstly found fundamental path starting from \( v \in V - \{d\} \), it is a shortest fundamental path starting from \( v \).

The solution is obtained by fixing each shortest fundamental path found in the process, in other words, by setting all edges of the paths as backup edges. We do this as follows: for each edge \((u, v)\) in the buckets, we compute the NCA (nearest common ancestor) \( w \) of \( u \) and \( v \) (see Fig. 5). Note that, if we let \( w' \) be the child of \( w \) on \( u-w \) path, only the vertices on \( u-w' \) path can be the starting vertices of the fundamental paths whose escaping edge is \((u, v)\). Thus we call the \( u-w' \) path a governing path. Next, from the vertices on the governing path, we find the lowest vertex \( x \) whose backup edge has not been set and select all edges in \( x-v \) path as backup edges. (At the same time we set the escaping edge \((u, v)\) as a backup edge and also a switching edge.) Note that, after those operations, all vertices on the governing path (namely, all vertices on the fundamental paths) are given their backup edges.

The pseudo code of the algorithm is as follows.
Algorithm BackupLinear\((G, d, T_d)\)

1. Classify edges \(e \not\in T_d\) into buckets.
2. \textbf{foreach} edge \((u, v)\) in the low order of its bucket
3. \hspace{1em} Compute NCA of \(u\) and \(v\).
4. \hspace{1em} Find the lowest vertex \(x\) in the governing path whose backup edge has not been set before.
5. \hspace{1em} Fix the fundamental path starting from \(x\).

Before considering the running time, we prove the correctness of this algorithm. It must be noted that the algorithm sometimes overwrites the backup edges that have been set once. It occurs when there are some vertices between \(x\) and \(u\) whose backup edges have not been set, during the above process. And this results in some backup edges being replaced by the new backup edges. It may be surprising that, we can nevertheless obtain the correct solution due to this “overwrite” operation.

**Theorem 3** Algorithm \(\text{BackupLinear}\) solves the problem \(\text{SHORTEST-BACKUP}\) if all edges have the same length.

**Proof.** We show that BackupLinear gives the same solution as the algorithm described in section 3 if all edges have the same length. Recall that the firstly tried fundamental path at each vertex is the shortest fundamental path starting with that vertex. Thus it is enough if we prove that the “overwrite” operation always gives priority (at the overwriting vertex) to the fundamental path whose starting vertex is closer to the root of \(T_d\).
Suppose that the fundamental path starting with \( x \) is just fixed in the algorithm, and this operation overwrites the backup edge of some vertex \( y \). This means that no fundamental path which includes \( y \) and starts from \( x \) or some another lower vertex has been fixed before, thus we can say that the current fundamental path (starting with \( x \)) has the lowest starting vertex ever and it is certainly given priority at \( y \). We can conclude that algorithm BackupLinear always gives priority to the fundamental path whose starting vertex is the lowest.

4.2 Details

Algorithm BackupLinear, in fact, does not run in linear time if we implement it naively, but we can reduce the time complexity to linear using some further techniques. There are two parts which need to be improved: (i) the part of finding fundamental paths to be fixed and (ii) the part of fixing them.

(i) Recall that, in the former part, we have to find the lowest vertex \( x \) whose backup edge has not been set yet among all vertices in a governing path. Thus, if a vertex is included in more than one governing paths, the vertex must be scanned more than once. The problem is that the number of the vertices scanned may be more than linear in total.

To prevent such a repeated scan of a vertex, we use the data structure called set union find tree [11]. It treats a set of singleton sets \( \{A_1 = \{a_1\}, A_2 = \{a_2\}, \ldots, A_n = \{a_n\}\} \) as primary data, and two operations are applied for it. The \( \text{union}(x, y) \) operation combines a pair of sets \( A_i \) and \( A_j(i \neq j) \), which include the elements \( x \) and \( y \), respectively, into a unified set \( A_i \). The \( \text{find}(x) \) operation returns the name of the set including the element \( x \). As a variant of this data structure, the algorithm on trees are presented by Gabow, et al. [4]. They operate vertices on trees as data elements and allow the only \( \text{union} \) operation whose outputs (i.e., a unified set of elements) induces a connected tree. And, their \( \text{find} \) operation returns the name of the lowest (closest to the root) vertex in the set as the name of the set. Their algorithm executes \( m \) intermixed operations of \( \text{union} \) and \( \text{find} \) in \( O(m) \) time. Throughout our algorithm, we maintain this data structure on the shortest path tree \( T_d \).

We apply the data structure simply: Every time the backup edge of some vertex is fixed, we execute an \( \text{union} \) operation to combine the vertex with its parent, and we scan a governing path from higher vertices to lower. Then, we can skip scanning the vertices whose backup edge has already been set; when we encounter such a vertex, execute \( \text{find} \) operation and jump to the vertex whose name is returned by the operation, and continue scanning again with the vertex. Figure 6 shows an example. Figure 6(a) illustrates the state just after the process for escaping edge \((u_1, v_1)\) is finished; backup edges of \( u_1 \) and \( x_1 \) are set and then \( u_1, x_1, \) and \( w_1 \) are combined by \( \text{union} \) operations. (The name of the set is then \( w_1 \) since it is the lowest vertex in the set.) At this state, we execute the process for escaping edge \((u_2, v_2)\). We begin with the highest vertex \( u_2 \) and go downward.
When we find a vertex $u_1$ whose backup edge has already been fixed, we execute $\text{find}(u_1)$, then obtain the name of $w_1$, and next start scanning again from $w_1$. Fig. 6(b) shows the state after the process for $(u_2, v_2)$.

In the following, the pseudo code for this part is presented as procedure FindPaths. The procedure replaces the line 4 of algorithm BackupLinear. Now let $b(v)$ be the backup edge of $v \in V - \{d\}$. Notice that the procedure assumes the initialization process, thus $b(v) = \text{null}$ for every $v \in V - \{d\}$, meaning that backup edges for every vertex is unset.

\begin{verbatim}
Procedure FindPaths(u, v)
1    x := null (x is the output at the end)
2    foreach vertex y: in the order from u to the child of NCA(u, v)
3        if $b(y) = \text{null}$
4            then x := y
5            union(parent of y, y)
6        else
7            y := find(y) and return to line 2
\end{verbatim}

**Lemma 3** Procedure FindPaths runs in linear time.

**Proof.** We show that the number of steps relating to the data structure can be bounded by $O(|E|)$. It is enough if we count how many times vertices are visited and the operations $\text{union}$ and $\text{find}$ are executed. Once a vertex is combined with its parent by a $\text{union}$ operation, the vertex is basically omitted to visit after that, thus every vertex is basically visited once. The only exception is the vertex on which a $\text{find}$ operation is executed, but the number of this kind of redundant visit...
is the same as the number of \textit{find} operations. The number of \textit{union} operations is $|V| - 1$, since the number of vertices is $|V|$. The number of \textit{find} operations is counted by classifying them into two types. The number of the first type, which returns the vertex on the current governing path, is at most $|V| - 1$ because at least one \textit{union} operation are always executed after each of them. The number of the second type, which returns the vertex outside the governing path, is at most $|E|$ because the operation of this type finishes the procedure for the current edge. 

(ii) In the latter part we fix the fundamental paths that are found in part (i). Recall that if distinct governing paths overlap, the backup edges of the common vertices may be overwritten. The problem is that the number of overwrites might exceed linear times.

Our solution for this problem is to enumerate shortest fundamental paths to be fixed and fix them in the order that no overwrite occurs. Concretely, prepare a bucket for each height, put all fundamental paths into them by the height of their starting vertices, and pick them up one by one from the buckets in the increasing order of their height. (Here, we might express a fundamental path as a pair of a starting vertex $x$ and an escaping edge $(u, v)$. This pair $(x, (u, v))$ determines a fundamental path uniquely.) For each of the fundamental paths, set its backup edges sequentially in the decreasing order of the height of their tails, then if we find some tail vertex whose backup edge has already been set during this process, quit setting and begin the process for the next path.

Those operations compute the correct solution. There are two points which assure the correctness. We first note that the fixing order of the paths does not require overwrites to obtain the correct solution. That is because the order is just the order of priority of the fundamental paths in setting backup edges. We second note that there is no problem if we quit setting when we find a vertex whose backup edge has already been set. It is guaranteed that, after we find such a vertex, there is no vertex whose backup edge has not been set yet. That is because the existence of such a vertex means that at least one fundamental paths starting from lower vertices (than the current fundamental path) has been fixed before.

The pseudo code of this part is as follows. This procedure replaces line 5 of BackupLinear. Note that the procedure should be executed for each fundamental path $(x, (u, v))$ after all of them are put into buckets.

\begin{verbatim}
Procedure FixPaths()
1    foreach pair of $x$ and $(u, v)$: taking from the buckets in the low order
2        foreach vertex $y$: sequentially from $u$ to $x$
3            if $b(y) =$null
4                then set $b(y)$ as a backup edge
5                    if $b(y)$ is an escaping edge
6                        then set $b(y)$ as a switching edge
7                    else
8                return to line 1 and operate the next pair
\end{verbatim}
Lemma 4 Procedure FixPaths runs in linear time.

Proof. The number of the fundamental paths to be handled is at most $|V|$ because there are $|V|$ vertices whose backup edges are required to be set. At each vertex only one backup edge is set and each vertex is basically visited once. That is because the procedure for each fundamental path is terminated as soon as the vertex whose backup edge has been set is found. □

Using the techniques described above, we obtain the following theorem.

Theorem 4 If all edges have the same length, problem SHORTEST-BACKUP can be solved in $O(|V| + |E|)$ time.

Proof. We show that the rest part of algorithm BackupLinear runs in linear time. In the part of classifying edges into buckets (line 1), we compute the height of every vertex and classify edges according to the height of their head and tail vertices, takes $O(|V| + |E|)$ time. For computing NCAs (line 3), we use the fast algorithm of Schieber, et al. [10]. Their algorithm computes a single NCA in $O(1)$ time if we execute $O(n)$ (n: the number of the vertices in the tree) time preprocessor beforehand. Thus this part also runs in linear time. □

5 Concluding Remarks

This research has several possibilities for extension: For example, other measures for the optimality of backup and switching tables are also interesting such as the total length of backup paths instead of the longest one. Another simple (but much harder to analyze) extension of our scheme is to allow a packet to pass more than one switching edge. (The mode of the packet changes back and forth between the normal and backup modes.) One may notice that the example given in Fig. 2 can be recovered if we use two switching edges.

References


Hiro Ito is with the School of Informatics, Kyoto University, Yoshida-Honmachi, Kyoto, Japan, 606-8501. E-mail: itohiro@kuis.kyoto-u.ac.jp

Kazuo Iwama is with the School of Informatics, Kyoto University, Yoshida-Honmachi, Kyoto, Japan, 606-8501. E-mail: iwama@kuis.kyoto-u.ac.jp

Yasuo Okabe is with the Academic Center for Computing and Media Studies, Kyoto University, Yoshida-Honmachi, Kyoto, Japan, 606-8501. E-mail: okabe@i.kyoto-u.ac.jp

Takuya Yoshihiro is with the Department of Systems Engineering, Wakayama University, 930 Sakaedani, Wakayama, Japan, 640-8510. E-mail: tac@sys.wakayama-u.ac.jp
Instability of Networks with Quasi-Static Link Capacities*

DIMITRIOS KOUKOPOULOS
University of Patras and RACTI, Greece

MARIOS MAVRONICOLAS
University of Cyprus, Cyprus

PAUL SPIRAKIS
University of Patras and RACTI, Greece

Abstract

In this work, we continue the study of stability issues for packet-switched routing. More specifically, we adopt the Adversarial Queueing Theory framework, where an adversary controls rates of packet injections and determines packet paths. In addition, the power of the adversary is enhanced to include manipulation of link capacities. However, in doing so, the adversary may use only two possible (integer) values, namely 1 and \( C > 1 \); moreover, the capacity changes are not abrupt: once a link capacity is set to a value, it maintains this value for a period of time proportional to the number of packets in the system at the time of setting the link capacity to the value. We call this the Adversarial, Quasi-Static Queueing Theory model. Within this model, we obtain the following results:

- The protocol LIS (Longest-in-System) is unstable at rates \( r > \sqrt{2} - 1 \) for large enough values of \( C \). The proof uses a small network of just ten nodes.
  This represents the current record for the instability threshold of LIS over models of Adversarial Queueing Theory with dynamic capacities.

- The composition of LIS with any of SIS (Shortest-in-System), NTS (Nearest-to-Source) and FTG (Furthest-to-Go) is unstable at rates \( r > \sqrt{2} - 1 \) for large enough values of \( C \).
  These represent the first results on the instability thresholds of compositions of greedy protocols over models of Adversarial Queueing Theory with dynamic capacities.

---

*This work has been partially supported by the IST Program of the European Union under contract numbers IST-1999-14186 (ALCOM-FT) and IST-2001-33116 (FLAGS) and by funds from the Joint Program of Scientific and Technological Collaboration between Greece and Cyprus (research project “Efficiency and Performance of Distributed Systems: Capabilities and Limitations”).