Single backup table schemes for shortest-path routing

Hiro Ito\textsuperscript{a}, Kazuo Iwama\textsuperscript{a}, Yasuo Okabe\textsuperscript{b}, Takuya Yoshihiro\textsuperscript{c,}*

\textsuperscript{a}School of Informatics, Kyoto University, Yoshida-Honnachi, Kyoto 606-8501, Japan
\textsuperscript{b}Academic Center for Computing and Media Studies, Kyoto University, Yoshida-Honnachi, Kyoto 606-8501, Japan
\textsuperscript{c}Faculty of Systems Engineering, Wakayama University, 930 Sakaedani, Wakayama 640-8510, Japan

Received 3 November 2003; received in revised form 14 May 2004; accepted 10 June 2004

Abstract

We introduce a new recovery scheme that needs only one extra backup routing table for networks employing shortest-path routing. By precomputing this backup table, the network recovers from any single link failure immediately after the failure occurs. To compute the backup routing table for this scheme, we use an almost linear time algorithm to solve the $\{r, v\}$-problem, which is a variation of the best swap problem presented by Nardelli et al. We further show that the same solution can be computed in exactly linear time if the underlying graph is unweighted.

© 2004 Elsevier B.V. All rights reserved.

Keywords: Routing; Shortest paths; Fault tolerance; Backup tables; Graph algorithms; Swap edges

1. Introduction

Designing a system which can recover from a failure quickly and inexpensively is important. On 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 bypass the failed link. This adaptive routing scheme or shortest-path routing scheme is convenient because it can automatically recompute the new shortest path. But
the recomputation takes considerable time during which we may lose a lot of packets. To hasten the recovery process, one natural approach is to use dynamic algorithms \[1,7\] that use the information of the old shortest path tree to compute a new one. They are much quicker, in time \(O(n)\), than the original algorithm, but even a linear running time still appears unrealistic to use every time a link fails.

Drastic improvements are expected by precomputing and holding backup routing tables at each node and using them as soon as a link failure occurs. These extra routing tables enable us to forward packets from the failed link to their destinations using precomputed backup paths. This approach certainly achieves very fast recovery, but unfortunately creates a space problem. Note that since different backup paths basically require different routing tables at each node, we may need as many backup routing tables as whole links. Thus, there are several obstacles to our goal of a prompt and cheap recovery system.

Recently, Nardelli et al. \[4\] introduced swap edges for shortest path trees. (In the following we will use the standard graph terminology in stead of nodes, links, etc.) Let \(S(d)\) be a shortest path tree with its root \(d\), namely, a packet destined to \(d\) goes along this tree. Suppose that a single edge \(e = (u, v)\) in \(S(d)\) has failed, as in Fig. 1(a). Then \(S(d)\) is split into two trees, \(S_1(d)\) and \(S_2(d)\), let \(S_2(d)\) include the destination \(d\) and the vertex \(v\). Then an edge \(e'\) becomes a swap edge for \(e\) if \(e'\) connects \(S_1(d)\) and \(S_2(d)\). For example, in Fig. 1(a) \((u_1, u_2)\) is such a swap edge, and a packet can use this edge instead of the failed \((u, v)\) to get to \(d\) as shown. Since two or more such swap edges may exist, we wish to select the best one under an appropriate measure. In \[4\] several algorithms for several different measures are given for computing the set of best swap edges for all edges in \(S(d)\). Most are very efficient, running as fast as in time \(O(m \log^*m)\) where \(\log^*m\) is the inverse Ackermann function.

Swap edges are obviously useful to recover edge failures. However, incorporating them concretely into our routing system is not so obvious. Fig. 1(b) explains the reason: suppose that the best swap edge for \((u, v)\) is \((u_1, u_2)\) and that for \((w, u)\) it is \((v_1, v_2)\). (The latter may be different from the former since the edge \((u, v)\) can be used for the backup path for
the latter as shown in the figure.) Now one can see that we need at least two (maybe more) different routing tables at the single vertex $w$ corresponding to different failed edges. As described before, this causes a major difficulty for the recovery system.

Our contribution in this paper is twofold: (i) We present a recovery scheme that needs only one backup table. Namely, when an arbitrary single edge fails, we replace the original table with this backup table. We also need only one extra bit in a header of a packet to show which (the usual or the backup) table will be used for that packet. We present an algorithm that computes such a backup table that is optimal in the sense that the worst-case cost of a backup path cannot be improved under the scheme. By using the algorithm for computing best swap edges in [4], our algorithm is equally fast. (ii) The running time of our algorithm can be exactly linear (i.e., the inverse Ackermann function is removed) if the underlying graph is unweighted.

2. Single backup-table recovery

Recall that our goal is to develop a backup scheme that reduces the need for backup tables to only one. Suppose that we usually use the shortest path routing table and that an edge on this shortest path tree (SPT) fails. Then: (i) A packet moves along the SPT until it reaches the failed edge. (ii) Now the packet is routed by using the (unique) backup table until it reaches the swap edge. (iii) After the packet passes the swap edge, it again uses the normal SPT table. It should be noted that to implement this scheme is quite easy, i.e., by simply introducing one extra bit, called a mode bit, in a header of each packet. The packet usually has value 0 in that bit (during the period (i) above), value 1 in that bit if the packet hits the failed edge, i.e., during (ii) above, and value 0 again after crossing the swap edge. Obviously, if a packet has value 0 (value 1, resp.) in its mode bit, then it uses the normal (the backup, resp.) routing table.

Let us establish some precise definitions: let $G = (V, E)$ be a directed graph, where $V$ is a set of vertices and $E$ is a set of (directed) edges. 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 $G$ is symmetric, i.e., if an edge $(u, v)$ is in $E$, the reverse edge $(v, u)$ is also in $E$ and $l(u, v) = l(v, u)$. (Namely, we assume that each link of our network is full-duplex.) We also 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 impossible 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 shortest length among all the paths from $u$ to $v$. A shortest path tree (SPT) $T_d$ for a destination $d$ is a spanning tree on $G$ if 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 an SPT $T_d$. Then our problem is to obtain two sets, $B$ and $S$, called a backup table and a swap table, respectively, which satisfy the following conditions. $B$ and $S$ are also called a set of backup edges and a set of swap edges, respectively.
(1) \( B \subseteq E - E[T_d] \) and \( S \subseteq B \), namely, \( B \) never includes edges in \( T_d \).

(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 \( e_1 e_2 \cdots e_h \cdots e_k \) 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 \( v \) if it does not include the \( T_d \)-edge from \( v \) (which is supposed to fail). Thus the above path \( e_1 \cdots e_k \) is obviously a backup path from \( v \), unique according to condition (2).

Since several solutions might exist for this problem, we introduce a cost function, the longest backup path length, to obtain the best one. Then this problem is called the SHORTEST-BACKUP.

**Problem. SHORTEST-BACKUP**

We are given a 2-edge-connected symmetric digraph \( G = (V, E) \), an edge length function \( l \), a destination vertex \( d \in V \), and an SPT \( T_d \). Then output two sets \( B \) and \( S \) that minimize the longest backup path under the rules (i)–(iii) of the backup scheme described in the beginning of this section.

**Theorem 1.** The SHORTEST-BACKUP can be solved in time \( O(|E| \log(|E|, |V|)) \).

**Proof.** Refer to Fig. 1 again. Recall that the best backup edge for (failed) \( (u, v) \) is \( (u_1, u_2) \) and the one for \( (w, u) \) is \( (v_1, v_2) \). Thus to make both available, we need more than one backup table at vertex \( w \), but that is not allowed now. Our simple idea is to use \( (u_1, u_2) \) for \( (w, u) \) as shown in Fig.1(c). This backup path, from \( w \) to \( u_1 \), then to \( u_2 \), and then to \( d \), might be longer than the original one, but it obviously does not harm our optimization measure since the shortest backup paths from \( u \) are apparently longer than the one from \( w \).

All the pairs of a failed edge and its best swap edge can be computed in time \( O(|E| \log(|E|, |V|)) \) using the algorithm for the \( \{r, v\} \)-problem given in [4]. Although details are omitted, it is straightforward to modify backup paths, as described above, so that a longer backup path has priority, in a linear time. Thus we can minimize the length of the longest backup path which starts from a tail vertex, \( v \), of a failed edge. Note that all packets being initiated at vertices other than the tail vertex \( v \) are routed to \( v \) or the final destination \( d \) on the shortest paths in the current scheme. Therefore, even though such a packet takes the longest backup path, its length is minimized if we follow our routing scheme. \( \square \)

3. **Linear time algorithms**

This section gives a linear time algorithm for solving the SHORTEST-BACKUP if the given graph is unweighted. For this result, it is enough to give a linear time algorithm for the \( \{r, v\} \)-problem, seen from the proof of Theorem 1.

Our linear time algorithm consists of three parts. First, we sort edges \( (u_1, u_2) \in E - E[T_d] \) according to height in \( T_d \): this is done by sorting the edges by their tail vertex height \( h(u_1) \) in \( T_d \), and if some vertices have the same height, we further sort them by their head-vertex height \( h(u_2) \) in \( T_d \). Second, for each edge \( (u_1, u_2) \) in ascending order by height, we find
vertices whose best swap edge may possibly be \((u_1, u_2)\). (In the original definition, the best swap edge is determined for each edge \((u, v)\), but we also denote it as the best swap edge for vertex \(u\).) The vertices to find is shown in Fig. 2. Let \(w\) be the Nearest Common Ancestor (NCA) of \(u_1\) and \(u_2\) in \(T_d\). A starting path is the \(T_d\)-path from \(u_1\) to the previous vertex of \(w\), shown as \(u\) in the figure. Then the vertices to be found are the vertices in these starting paths because if a path goes from \(u\) to \(u_1\) along \(T_d\) (reversely), crosses \((u_1, u_2)\), and goes to \(d\) along \(T_d\), the path is the shortest backup path from \(u\) to \(d\). (Consider that the underlying graph is symmetric.) Finally, we set the best swap edges of each vertex as follows: while processing edges \((u_1, u_2)\), if we find the vertex \(x\) in its starting path whose best swap edge are not set, we set \((u_1, u_2)\) as the best swap edge of \(x\). Namely, for each vertex \(v \in V - \{d\}\), the first-found edge becomes its best swap edge. As a result, we obtain a set of best swap edges for each vertex \(V - \{d\}\). It is the solution of the \(\{r, v\}\)-problem.

The key idea of our algorithm is the sorting of edges in \(E - E[T_d]\) by height. The following two lemmas show how this idea contributes to correctness. For the proof of the lemmas, see the preliminary version of this paper [3].

**Lemma 1.** Let \(p_u\) and \(p_v\) be the two shortest backup paths determined by the common starting vertex \(s\) and the swap edges \((u_1, u_2)\) and \((v_1, v_2)\), respectively. Then, \(h(u_1) > h(v_1)\) implies that \(l(p_u) \geq l(p_v)\).

**Lemma 2.** For every edge \((u_1, u_2) \in E - E[T_d]\), \(h(u_1) - h(u_2) \in \{-1, 0, 1\}\).

Let us concentrate on the backup paths that have the same starting vertex \(s\). Then Lemma 1 shows that sorting swap edges by their tail vertex height is also sorting roughly the backup paths induced by \(s\) and the swap edges by length. Also, Lemma 2 shows that the sorting swap edges by head vertex height precisely completes the sorting of the backup paths, since the three values of head vertex height mean the three values of the backup path length. Consequently, if we focus on a single starting vertex \(s\), the sequence of swap edges sorted in our algorithm also means the sequence of backup paths starting from \(s\) in order by
length. Thus, the collection of the first-found swap edges of each vertex, which is what our algorithm outputs, is also the collection of swap edges used in the shortest backup paths from each vertex. This is the solution of the \{r, v\}-problem.

The formal description of our algorithm follows.

**Algorithm.** \{r, v\}-Linear\((G, d, T_d)\):
1. Sort edges \(e \in E - E[T_d]\) by height.
2. **For each** edge \((u_1, u_2)\) in ascending order by height,
3. Compute the NCA of \(u_1\) and \(u_2\).
4. **For each** vertex \(x\) in the starting path whose best swap edge has not been set,
5. Set \((u_1, u_2)\) as the best swap edge of \(x\).

**Theorem 2.** The \{r, v\}-problem can be solved in \(O(|V| + |E|)\) time for unweighted graphs.

**Proof.** The correctness proof has already been completed because the set of best swap edges obtained by \{r, v\}-Linear is obviously the solution of \{r, v\}-problem. Thus, in the following part, we prove that our algorithm runs in linear time.

We estimate the time complexity of the three parts of the algorithm. The first is the sorting part. Consider that the head vertex height takes only three values for each tail vertex height. Thus we can use a bucket sort (we prepare three buckets for each height of the tree vertices) so that this takes only \(O(|V| + |E|)\) time. In the second part, computing the NCA of the two vertices \(u_1\) and \(u_2\) is time-consuming. We can, however, use the algorithm of Schieber et al. [5] that computes a single NCA in \(O(1)\) time if we have executed \(O(n)\) (\(n\): the number of vertices in the tree) time preprocess beforehand. Since we compute NCAs only \(|E|\) times throughout our algorithms, this occurs in linear time.

The third part is the setting of the best swap edges. For each edge in \(E - E[T_d]\), we have to check each vertex in its starting path whether its best swap edge has already been set or not. However, since the procedure for different edges might repeatedly check the same vertex, the total number of checks might exceed linear times. To prevent such redundancy, we use the algorithm presented by Gabow et al. [2] to solve the disjoint set union problem for tree-type data. Consider a tree \(T = (V, E)\) and a family of singleton sets \(\{A_1 = \{v_1\}, A_2 = \{v_2\}, \ldots, A_n = \{v_n\}\}\) where \(V = \{v_1, v_2, \ldots, v_n\}\). The **disjoint set union problem** is to carry out a sequence of operations on the following two types of sets: the \textit{union}(x, y) operation, which is allowed only when \((x, y) \in E\), combines a pair of sets \(A_i\) and \(A_j\) \((x \in A_i, y \in A_j, i \neq j)\) then makes a new set \(A_i\) (here, \(i\) must be the smallest index of vertices included in the new set). The \textit{find}(x) operation returns the index \(i\) such that \(x \in A_i\). Gabow et al. ’s algorithm executes \(m\) intermixed operations of \textit{union} and \textit{find} in \(O(n + m)\) time.

We use this algorithm simply: every time the best swap edge of a vertex is set, we apply an \textit{union} operation and combine the vertex with its parent. Then, during the checking phase of vertices, if we find a vertex whose best swap edge has already set, we use \textit{find} operation and skip the vertices which have already located their best swap edges. For example, Fig. 3(a) is the situation that the procedure for \((u_1, u_2)\) is finished. Vertices from \(u_1\) to \(w\) have been combined, and the set is named \(w\). Next, when we process \((v_1, v_2)\), we check \(v_1\) first, and then go to lower vertices, and if we find \(u_1\) that has the best swap edge, we apply \textit{find}
operation and jump to \( w \). (At this time, the best swap edge of \( w \) has not been set.) In this way we can skip vertices that have once set the best swap edges, reducing the number of checking vertices to linear times. □

4. Concluding remarks

This research has several possibilities for extension. For example, such measures to optimize backup and switching tables as trying the total length of backup paths instead of the longest one are promising. Another simple but much harder to analyze extension of our scheme is to allow a packet to pass through more than one switching edge. (The mode of the packet changes back and forth between normal and backup modes.) One may try to recover the failure of nodes instead of edges.

References