On the effectiveness of restoration path computation methods

Balázs Szviatovszki, Áron Szentesi, Alpár Jüttner
Ericsson Research, Traffic Analysis and Network Performance Laboratory,
POB 107, H-1300 Budapest, Hungary,
Tel: +36-1-437 7631, Fax: +36-1-437 7767, E-mail: Balazs.Szviatovszki@eth.ericsson.se

Abstract—In this piece of work, we compare the effectiveness of distributed restoration path computation methods that use head-end pre-computation and do not use backup path sharing. Our aim is to provide statistics on primary and secondary path lengths of simple disjoint path computation methods in real-world network topologies. Moreover, we propose a path computation method which can be used when paths are precomputed, but not pre-established. The benefit of this new method is that it intends to select strictly shortest paths for the working path and the restoration path as well. We allow common links between the two paths, therefore an additional restoration path is needed. By minimizing the number of common edges between the working and the shortest backup path, our method maximizes the probability that in case of a link failure the shortest backup path can be used for restoration. In case the restoration path is a shortest path, it is not necessary to revert to the original working path after the failed link is restored, instead, sub-optimal paths can be re-optimized in the network in order to balance traffic load.

I. INTRODUCTION

The Generalized Multiprotocol Label Switching (GMPLS) work of the Internet Engineering Task Force projects major simplification of protocol stacks and the re-evaluation of the used protection architectures [1]. To guide Internet Service Providers and network operators, detailed overall network cost analysis of different protection architectures has been carried out in [2]. The results show that mesh-based restoration provides lower overall cost then ring-based methods. While routing design of ring-based methods is quite straightforward, path computation in a meshed network is a challenging task.

Path optimality aspects of protection methods for meshed networks were studied by Benerjea in [3] where path-based (head-end), link-based (local reroute) and hybrid rerouting methods were compared. Not surprisingly in simulation studies path-based rerouting turned out to have better performance then local reroute.

Depending on how much service interruption is affordable, restoration path computation can be carried out in real time (i.e., after the failure), or paths can be precomputed. Furthermore, in case the restoration paths are precomputed, there is a choice whether they are pre-established or not.

In order to achieve high capacity utilization certain methods use centrally planned routing computation. These are generally path-based see e.g., [4]. The disadvantage of such methods is that they may not scale well. Alternatively, distributed restoration path computation can be used. Methods in this category span from simplest Penalty method or Suurballe’s method [5] to such methods where sharing of backup paths are optimized [6].

In this piece of work, we introduce a new distributed restoration path computation algorithm called ‘Shortest Backup’ method. We do not consider sharing of backup paths. The new method can be used when paths are precomputed, but not pre-established. Our basic idea is to select strictly shortest paths for the working (primary) and restoration (backup or secondary) paths as well. We allow common links between the two paths, to achieve higher efficiency than Suurballe’s method. In order to prepare for the failure of the common links between the primary path and the first (shortest) restoration path, we compute an additional restoration path for a working path. By minimizing the number of common edges between the working path and the shortest backup path, our method maximizes the probability that the shortest backup path can be used for restoration.

In case the restoration path is shortest, the used network resources are minimized. More importantly, in this case it is not necessary to revert to the original working path after the failed link is restored. Instead, sub-optimal paths can be re-optimized in the network in order to balance traffic load.

The rest of the paper is organized as follows. We start in Section II with an overview of the used MPLS technology background. The application of the studied algorithms are not limited to this technology, however, the selection of a well defined architecture eases the understanding of the paper. In Section III we survey restoration path computation methods from the literature and then describe the new Shortest Backup method in Section IV. In Section V we show the differences between different restoration path computation algorithms through an example network. In Section VI we present results of detailed numerical evaluations. Finally Section VII concludes the paper.

II. MPLS BACKGROUND

In this section we give a brief overview of the most important restoration related functionality of MPLS networks. We concentrate on the whole establishment, recovery, reversion and re-routing cycle as discussed in [7].

When configuring a Label Switched Path (LSP) at its head-end (ingress) router, its destination (egress) router, as well as its bandwidth requirement and optionally other constraints (hop-limit, explicit hops) are specified. The complete path of the LSP can be computed in a central traffic engineering tool, or alternatively by the head-end router using a Constrained Shortest Path First (CSPF) algorithm. In both cases the LSP setup is initiated with explicitly defined strict hops.

To provide restoration for an LSP, both path-based and link-based methods are defined in IETF. The MPLS terminology for
the former is backup or secondary LSP, while the latter is usually referred to as detour or fast reroute. To protect the working path of an LSP one or more backup paths can be configured. If the path of the primary LSP is determined by CSPF, it is possible to extend the simple bandwidth constrained path computation algorithm [8] to find two disjoint paths. Algorithms that can be used for this purpose will be discussed in Section III.

In MPLS link-based protection is imagined to be used only to eliminate service interruption for the time needed to notify the head-end about the failure. A detour provides the immediate upstream route (from the failure) with an alternate path that can route around the failed downstream node, link, or shared risk link group [9], [10]. Detour paths are determined at each hop of the working path by a local path computation algorithm of the label switching routers. A single bypass tunnel (detour) can be established for all uninterrupted LSPs between two routers [11]. In the rest of this work we concentrate only on path-based restoration, i.e., we do not consider detours in our study.

Once computed, a backup path can be established together with the working path. In this case the backup path is called hot-standby. Alternatively, backup LSPs can be established only after the failure of the primary path. If there are more backup LSPs configured, they can be tried one after the other, or if the failed link/node can be identified, such backup LSP can be chosen which does not include the failed link. As a third possibility, LSP paths can be calculated on-demand after the failure. However, this can take quite a long time in case there are lots of LSPs, since the CSPF algorithm has to calculate a path for them one-by-one. According to [12], the best possible strategy is to pre-compute the backup path, but not pre-establish it.

LSPs are established using the CR-LDP and RSVP-TE signaling protocols to allocate labels and reserve resources along the precomputed path. There are proposals for crankback routing extensions to CR-LDP and RSVP-TE to indicate the location of the failed link or node to the ingress router [13]. This can be viewed as a fast topology update, enabling the ingress router to select such a restoration path that does not include the failed link. In case only one totally disjoint restoration path is computed, of course, head-end based rerouting can be done without signaling the point of failure. Therefore crankback extensions are only meaningful if there are more (not totally disjoint) precomputed paths, or when restoration paths are computed after the failure.

Since restoration paths may not be optimal, after a hold-down time, reoptimization of paths may be triggered. Reoptimization should be performed only in such cases when significant bandwidth utilization gain can be achieved. As Villamizar pointed out [14], the hold-down time after a failure should be long enough to allow feedback of resource consumption changes caused by all LSP rerouting attempts.

When failed resources are put back to service, traffic can be restored and the traffic layout in the network can be balanced. There are different ways of achieving a healthy traffic routing in the network. If LSPs are established frequently, one possibility is to let new traffic fill the gap on the restored link. Alternatively, when restoration is configured to be revertive, traffic from backup paths can be moved back to the preferred primary paths. When LSPs are configured to be non-revertive, LSP re-optimization can be used to utilize the free capacity of the restored link. A specific implementation of the non-revertive case is when primary and secondary paths change roles; once traffic is redirected to the backup path, the former active path becomes the backup of the new active path. In this case, it is important that the two paths are equal quality paths, i.e., both are shortest paths. The non-revertive mode with re-optimization may result in less path change in the network.

### III. Backup Path Computation Methods

Calculation of working and restoration paths, as it was mentioned before, can be done in a centralized or a distributed way. In this section we provide an overview of proposed path computation methods of the literature. All discussed algorithms provide protection for single link or node failures.

Kodialam and Lakshman [6] propose both a central and a distributed method in which sharing of backup paths is allowed among certain active paths. The basic idea in both cases is the following. If two active paths do not have a common link, then their backup paths can share bandwidth on the entire path, since no link failure can cause the activation of both backup paths.

In the central method the authors suppose that complete path information is available and give an integer programming formulation for the problem. Moreover, they show that by maintaining aggregated and not per-path information, a distributed method can perform almost as well as the complete information scenario. Since implementation of any backup path sharing algorithm needs modification to currently proposed traffic engineering extensions of interior routing protocols, the authors propose extensions for OSPF and ISIS [10].

There are a number of basic backup path selection algorithms that do not require protocol changes. Without global view of LSP paths, distributed algorithms prefer shortest or close to shortest paths in order to provide good resource utilization (close to the global optimum). Since working paths are those that really reserve resources, it is important to have a shortest working path (e.g., in terms of hop count). The restriction for the backup path usually is to be disjoint from the active path, and to be as short as possible. Unfortunately, the problem of finding a shortest path having a shortest disjoint backup path is NP-hard. In the next paragraphs weSurvey simple restoration path selection algorithms.

A heuristic solution referred to as the Penalty method in the literature [15], first obtains an arbitrary shortest path between the source and the destination node using Dijkstra’s method. In order to find a disjoint backup path from the selected shortest path, Dijkstra’s algorithm is executed on a modified graph where link weights of the working path are increased. If the weights are significantly increased, the second shortest path
computation will find a shortest disjoint path if there is any. There is a weak point of this heuristic algorithm; depending on the selection of the shortest path, secondary paths with different quality might be found. There is no way to optimize how long this secondary path will be.

If we relax the constraint that the primary path has to be shortest, we can get the problem of finding two disjoint paths for which the sum weights are minimal. An alternative perception of this problem is to send two units of flow as cheaply as possible from node \( s \) to node \( t \) in a unit capacity network. Therefore, this problem is generally referred in the literature as the two-valued unit capacity minimum cost integer flow problem. The book of Ahuja, Magnanti and Orlin [16] provides and excellent overview of general minimum cost flow algorithms. For the special case of this problem when we are interested in finding the two-valued minimum cost integer flow between a source node and all other nodes in a graph, Suurballe and Tarjan provided a fast method [5]. This is widely referred to in the literature as Suurballe’s method.

Although the problem of finding a shortest path having a shortest disjoint backup path is NP-hard, an integer linear programming problem can be easily formulated and more importantly, it provides very good practical results as e.g., reported by Cinkler, Laborczy and Horváth in [4]. Considering the quality of the found paths (length of secondary path), this algorithm is more efficient than the Penalty method, on the other hand, it has worse running time because of the complexity of the Integer Linear Programming problem.

All above methods compute one restoration path for an active path. If the crankback feature is used and if we can store more restoration paths, failure dependent path pre-computation methods can be used. One solution is to compute one restoration path for each link in the working path. These paths can be calculated by the Penalty method, by increasing only one link’s weight at a time. The disadvantage of this algorithm is that many secondary paths have to be stored.

### IV. Proposed method

In this section we propose a restoration path computation method called ‘Shortest Backup’ that pre-computes and stores two backup paths for a working path. Since the backup paths are not totally disjoint from the working path, we should know which link has failed when selecting the backup path from among the stored two paths.

We use pre-computation to provide fast restoration. In our method the simple lookup operation to select a stored backup path is much faster than a Dijkstra-based or an even more complicated algorithm used in on-demand schemes. In real time methods, after a link failure CSPF should be performed for every affected LSP, since the LSP’s bandwidth is treated as a constraint. If real time path computation is used, this sequential computation can take a significant amount of time depending on the link size and the number of LSPs on it.

We search for a shortest backup path, since this minimizes the used resources. Moreover — as it was described in Section II — when non-revertive restoration paths are used, working and backup paths may change roles after the failure. This can only be done efficiently when both paths are optimal, i.e., both are shortest. When the failed link is restored, it can be utilized to reoptimize other sub-optimal paths in the network.

Let us formulate a path computation problem in which we seek two shortest paths having minimal number of common edges. Imagine that we have an algorithm that provides for example two four-hop shortest paths having only one common edge. If we configure one as the primary path and the other as a non-revertive secondary path, we achieve a 75% protection against single link failures (three out of four links are protected). To achieve full protection we need one more backup path to protect against the failure of the common edge of the previously found two shortest paths. Since this second backup path may be sub-optimal we may configure it to be revertive.

### A. Problem formulation

The key step in our algorithm is to determine two shortest paths having minimal number of common edges. More precisely: given a directed acyclic graph \( D = (N, E) \) with a weight function \( w : E \rightarrow \mathbb{R}^+ \) associated with each edge \( e \in E \), a source and a destination node \( s, t \in N \), the task is to find two paths \( P_1, P_2 \) between \( s \) and \( t \), such that:

\[
\sum_{e \in P_1} w(e) = d_{\min}(s, t) \quad \text{and} \quad \sum_{e \in P_2} w(e) = d_{\min}(s, t) \quad \text{min} | P_1 \cap P_2 |
\]

where, \( d_{\min}(s, t) \) is the sum of edge weights of the shortest path between \( s \) and \( t \) in \( D \).

To describe our solution for the above problem, first we construct an auxiliary graph in which every path represents a shortest path in the original graph. We can obtain this auxiliary graph by running a simple shortest path calculation algorithm, determining a feasible potential for the graph nodes and marking edges as ‘tight’ or ‘not tight’. In the next section we describe these steps one after the other with necessary definitions.

### B. Solution

Our solution presented later in this section builds upon some graph theoretical definitions and theorems. Although these concepts are more or less well-known, we shortly summarize the essentials in order to support the reader in understanding the algorithm. For further details and proofs, see for example [16].

#### B.1 Prerequisites

**Definition 1:** Given a directed graph \( D = (N, E) \) and a weight function \( w : E \rightarrow \mathbb{R}^+ \), a function \( \pi : N \rightarrow \mathbb{R} \), called
potential is feasible if $\pi(v) - \pi(u) \leq w(u, v)$ for every edge $uv \in E$.

It can be proven that the output of Dijkstra’s classical algorithm — the length of the shortest paths from $s$ to each other node in the graph — defines a feasible potential. That is:

**Proposition 1:** If $w$ is non-negative, and $\pi_{\text{min}}(v)$ is defined to be the minimum cost of a path from a fixed $s$ to $v$, then $\pi_{\text{min}}(v)$ is a feasible potential, which actually majorizes all feasible potentials.

Now let us show why this potential is useful. Remember our aim is to construct such a graph in which whatever path is chosen, we get a shortest path in the original graph. Let us call an edge tight if its weight equals to the potential difference of its end nodes, i.e. $\pi(v) - \pi(u) = w(u, v)$.

**Proposition 2:** A path in the graph is shortest if and only if each of its edges are tight.

If we know a feasible potential of a graph, the sought auxiliary graph — or as we call the Graph of Tight Edges, $D_t$ — can be constructed.

### C. Proposed algorithm

We intend to find two paths having minimum number of common edges. If we seek these path in the graph of tight edges, it is guaranteed that both paths will be shortest. In this subsection we show how we can make further modifications on $D_t$ in order to be able to use simply the Suurballe’s method for our purposes. Let us duplicate each edge in $D_t$ and assign weight ‘0’ to the old edge and weight ‘1’ to the new edge, respectively. We denote this second auxiliary graph by $D_{t2}$. If Suurballe’s minimum cost disjoint paths algorithm is executed on this graph, the output will be surely two disjoint paths, since there are two distinct edges between each node. However, it may happen that these paths use both an edge with weight ‘0’ and an edge with weight ‘1’ between two nodes. This means that in the original graph, the two paths have a common edge. Suurballe’s algorithm minimizes the sum weight of the two paths, therefore the number of common edges will be minimized, due to the proposed special weighting.

### V. Example

To show the differences between the different backup path computation algorithms we have created a simple artificial example graph in which for each edge $w(e) = 1$. The paths found by the different methods are shown in Fig. 1. As we will see in Section VI the basic characteristics of the algorithms identified in this section can be spotted in the resulted path statistics as well.

### VI. Numerical Results

We have conducted numerical experiments to investigate the produced path lengths of different CSPF methods. In order to get practically relevant results, simulations were carried out on publicly available real-world backbone network topologies [17]. The first experiment aims at measuring path length without any statistical effect. Therefore, path lengths are counted between all ingress egress pairs. The second experiment uses random traffic to measure the path length statistics of backup LSPs. This more realistic setup will be described in Section VI-B.

#### A. Simple backup path statistics

In the first round of experiments we generated a full-mesh traffic matrix. At this experiment bandwidth reservations were neglected when doing path calculations. The purpose of this setup was to get simple backup path length statistics for the four path computation methods, namely Suurballe’s method, the Penalty method, the ILP method and our Shortest Backup method. Since the Penalty method always produces a shortest path as a working path, we could compare this with the length of the paths produced by Suurballe’s method. We say that Suurballe’s method produces long paths, if neither its working, nor its backup path is shortest. On the other hand, we say that Suurballe’s method produced short paths when both of its paths were shortest paths. We say that the Penalty method produced long paths, if the length of its backup path was longer than the length of the backup path generated by Suurballe’s method.

We carried out simulations on two network topologies, namely on the Cable & Wireless and on the AT&T network [17]. Suurballe’s method produced 4 long paths out of 870 and 600 cases for the C&W and AT&T network, respectively. This means that although Suurballe’s method aims at minimizing the total length of the two paths, almost always achieves it in such a way that one of the paths is shortest. Moreover the number of short Suurballe paths was 116 for both networks. This means that for around $15 - 20\%$ of the source destination pairs, both paths were shortest paths. The Penalty method produced long paths altogether 22 and 30 times for the C&W and AT&T network, respectively, which is around $5\%$ of the total cases.

More detailed results are presented in Table I and II, in which distribution of successful and unsuccessful path computations are shown, classified by the length of the theoretical shortest path between the source and destination of traffic demands. The network diameter for the C&W network equals to 7, while for the AT&T network it equals to 5. The ILP method always produced optimal paths, i.e., its first path was shortest, and its second path was never longer then the backup path produced by Suurballe’s method. The drawback of the ILP solution is its slow speed. The running times of all other methods does not differ significantly. Both Suurballe’s methods and the Penalty methods execute two shortest path calculation and some additional graph modifications, while the Shortest Backup method produces its results with the help of three shortest path calculations.

The performance of the Shortest Backup method cannot be measured by the length of its paths, since it produces shortest paths for both of the paths. Therefore we measured the number of common edges between them. In Table III, we can notice
that this method produced 196 and 136 more shortest paths than the simple Suurballe’s method. Moreover, more than half of these paths had only one common edge. Altogether this method found two shortest paths for 36% (312 out of 870) and 42% (252 out of 600) of the total source destination pairs. This is a promising result.

![Fig. 1. (a) Suurballe’s method minimizes the total weight of the two paths. In our case, the working path has \( w = 4 \) while the backup path has \( w = 5 \). It may happen that none of the paths found by Suurballe’s method are shortest. Imagine that the third edge of the lower path in Fig. 1.a. has \( w = 2 \). In this case Suurballe’s algorithm would still find the same paths, but none of them would be shortest. The remaining two shortest paths would not be disjoint from the \( w = 5 \) long backup path, and for these a \( w = 7 \) long backup path would not result in such a path set that has minimum total weight. (b) The Penalty method selects a random shortest path and then selects a disjoint backup path by increasing the edge weights of the shortest path. We plotted a pessimistic case, in which such a shortest path is chosen that has a \( w = 7 \) long backup path. If the same shortest path were chosen as before, the Penalty method would output the same paths as Suurballe’s method. One can note that the probability of selecting a ‘bad shortest’ path is 66%, since there are three possible shortest paths, among which two has a long backup path, and only one has a short backup path. (c) The Shortest Backup method finds two shortest paths having minimum number of common edges. In this graph only tight edges are shown together with the feasible potentials determined by the shortest paths. In order to protect against the failure of the common edge of the two paths, a third path should be calculated. By increasing the weight of this edge, a subsequent shortest path calculation would find the same path as the \( w = 5 \) backup path of the Suurballe method. This five hop backup could then be used to protect the first link of the active path.

Table I

<table>
<thead>
<tr>
<th>Number of paths</th>
<th>104</th>
<th>204</th>
<th>286</th>
<th>170</th>
<th>104</th>
<th>46</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Suurballe</td>
<td>0</td>
<td>30</td>
<td>32</td>
<td>20</td>
<td>18</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Long Penalty</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table II

<table>
<thead>
<tr>
<th>Number of paths</th>
<th>152</th>
<th>184</th>
<th>192</th>
<th>118</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Suurballe</td>
<td>34</td>
<td>40</td>
<td>38</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Long Penalty</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>

Table III

<table>
<thead>
<tr>
<th>Number of common edges</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>C&amp;W network</td>
<td>116</td>
<td>102</td>
<td>52</td>
<td>30</td>
<td>12</td>
<td>312</td>
</tr>
<tr>
<td>AT&amp;T network</td>
<td>116</td>
<td>106</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>252</td>
</tr>
</tbody>
</table>
path, and to be able to use non-revertive mode efficiently we proposed to compute two shortest paths, such that the number of common edges are minimized. To protect for the failure of the common edge(s), a second backup path is computed with the Penalty method. Our method requires crankback extensions to notify the head-end router about which link was failed. The complexity of the proposed algorithm is still in the order of Dijkstra's shortest path algorithm. We compared the performance of the proposed new algorithm with known methods from the literature. We found that the probability that a shortest secondary path can be used at failure is doubled when the Shortest Backup method is used. We also showed that the average secondary path length is also smaller for our method than for existing methods such as Suurballe’s or Penalty method.

REFERENCES


