Fast reroute paths algorithms

Jarry, Aubin

How to cite

JARRY, Aubin. Fast reroute paths algorithms. In: Telecommunication Systems, 2013, vol. 52, p. 881–888. doi: 10.1007/s11235-011-9582-5

This publication URL: https://archive-ouverte.unige.ch//unige:114846
Publication DOI: 10.1007/s11235-011-9582-5

Telecommun Syst (2013) 52:881–888
DOI 10.1007/s11235-011-9582-5

Fast reroute paths algorithms

Aubin Jarry

Abstract In order to keep services running despite link or node failure in MPLS networks, RSVP-TE fast reroute (FRR) schemes use precomputed backup label-switched path tunnels for local repair of LSP tunnels. In the event of failure, the redirection of traffic occurs onto backup LSP tunnels that have the same quality of service constraints as original paths. Local repair of LSP tunnels notably differ from traditional (1:1) dedicated path protection schemes in that traffic is diverted near the point of failure which speeds up the protection process by not having to notify the source and then resend the lost traffic. This gain in protection delay is crucial for MPLS networks which would otherwise suffer from an important recovery latency.

In this paper, we investigate the algorithmic aspects of computing original paths along with their back-up so that they satisfy quality-of-service constraints (namely, delay) for single link or multiple link failure. In the case of single link failure, we propose an algorithm in $O(nm + n^2\log(n))$ that computes shortest guaranteed paths with their backup towards a single destination. In the case of directed graphs, we show that this algorithm is optimal by proving that computing shortest guaranteed paths is as hard as to compute multiple source shortest paths in directed graphs. In the case of undirected graphs, we propose a faster algorithm with time complexity $O(m\log(n) + n^2)$. We also provide a distributed algorithm based on Bellman-Ford distance computation which converges in $3n$ rounds at worst.

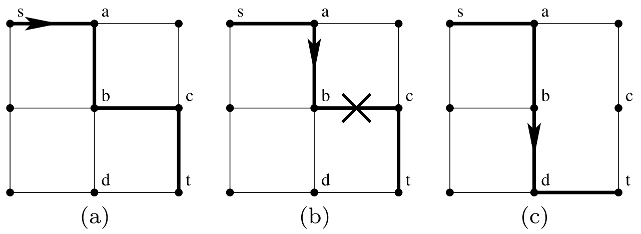
Keywords Shortest paths · Fast reroute · MPLS

A. Jarry (✉)
CUI, University of Geneva, Bat. A, route de Drize 7,
1227 Carouge, Switzerland
e-mail: aubin.jarry@unige.ch

1 Introduction

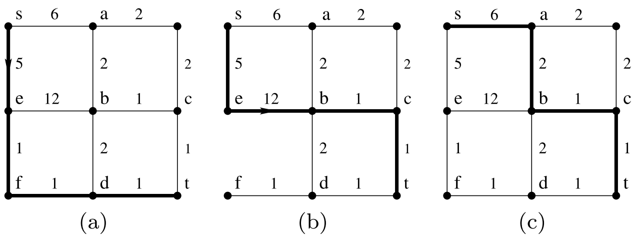
Network survivability has become of key importance to service providers due to the growth of real-time business applications over the Internet. The ability of keeping services running despite link or node failure and without apparent service disruption are nonetheless still a challenge for real time services such as high-quality video. For multicast applications such as real-time streaming or teleconferencing, dedicated fast reroute schemes have been proposed in [4, 9] for publish/subscribe internetworking architectures. In IP and multiprotocol label-switched (MPLS) networks, this demand is met through fast reroute schemes that allow networks to divert traffic to precomputed backup paths while they repair automatically (see [7] for a survey). In MPLS networks, resource reservation protocol—traffic engineering (RSVP-TE) fast reroute (FRR) schemes (see RFC4090 [6]) use two methods to redirect the traffic in case of failure within 10s of milliseconds: the one-to-one backup method creates detour LSPs at each potential point of local repair, whereas the facility backup method creates a bypass tunnel to protect a potential failure point. Both methods require the computation of backup paths along with the original LSPs that provide the same quality-of-service level. Therefore, the quality-of-service metric that is used for the paths along which data is normally routed must encompass the quality of backup paths that will be used in fast reroute schemes, in the event of node or link failure.

The problem under study has some marked differences with protection mechanisms developed for WDM optical networks, such as the p-cycle protection method [5], or even for multidomain optical networks [2]. First of all, recovery latencies are in general much lower when dealt with at the physical layer [8], and therefore the added lengths of recovery paths do not affect too much the overall quality of



(a) (b) (c)

Fig. 1 Online recovery on a 3 × 3 grid: (**a**) the original path goes through a, b, c; (**b**) edge bc is broken; (**c**) recovery through node d



(a) (b) (c)

Fig. 2 Recovery example: (**a**) shortest path with length 8; (**b**) recovery length 19; (**c**) best path recovery-wise

service of the network. Secondly, these protection schemes focus mainly on local link protection, and more generally on protection of the whole network, whereas LSP schemes have more user-oriented protection policies.

On an added note, MPLS networks are often composed of virtual links, that themselves represent physical network paths. In such settings, a physical failure will cascade down to multiple virtual link failure, where every virtual link that uses the physical impaired component will fail. A Shared Risk Link Group (SRLG) is the set of edges that share a common physical resource that may fail. In (1:1) protection, the problem of computing SRLG-diverse routing, that is to find a main path and a protection path that avoids every SRLG that may involve the main path has been shown to be NP-hard [3]. In contrast, the problem of computing fast reroute paths under quality-of-service constraints in the context of shared-link groups is tractable, as we will see in Sect. 4.

In this paper, we study the algorithmic aspects of computing original and back-up paths under quality of service constraints. We consider a network modeled by a communication graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges. Whenever a source node s needs to send message to node t, a path from s to t is chosen in the network, and the message is carried over this path unless a node or link fails during the transmission. A small example on a 3×3 grid is illustrated in Fig. 1. We shall make two remarks illustrated by the example. First, the message is not re-emitted from the source node: we have a launch-and-forget protocol. The traffic control occurs locally. Then, the recovery route goes directly towards node t without going through node c: the recovery is not local for the edge bc, but global. When a failure detection happens, the recovery route starts from the last reachable node (b in the example above).

We consider a cost function on the network denoted $c : E \to \mathbb{R}_+$ (that measures link latencies, for instance). The *length* of a path P is defined as the sum of the costs of all the edges on the path: $c(P) = \sum_{e \in P} c(e)$. A path from s to t which minimizes the length is called a *shortest* path. The length of a shortest path between s and t is called *distance* between s and t. To compute efficient shortest paths, the well known Dijkstra's algorithm may be used. However, a shortest path may well lead to an expensive (long) recovery path if one of its edges fails. In the example of Fig. 2, the shortest path from s to t goes through e, f, and d and costs 8; unfortunately, if the edge ef fails, the length of the recovery path from s to t is 19; in order to minimize the recovery part, it would have been better to go through a, b, and c; this latter path length is 10, but has worst recovery length 13 (attained if edge ct fails).

Since the length of a path may affect the overall quality of service provided by the network, an edge failure will remain unnoticed for the end user only if the length of the recovery path is under a certain limit. Therefore, we are interested in the worst case recovery scenario for a path: the *guaranteed length* $g(P)$ of a path P is defined as the length of the actual path followed by a message that was originally cast over P in the worst case recovery scenario. A path P from s to t that has the smallest guaranteed length is called *shortest guaranteed path*, and $g(P)$ is called *guaranteed distance* from s to t thereafter.

In this paper, we propose a generic algorithm in $O(nm + n^2 \log(n))$ that computes shortest guaranteed paths with their backup towards a single destination. We argue that this algorithm is optimal in the case of directed graphs by proving that computing shortest guaranteed paths is as hard as to compute MSSP (Multiple Source Shortest Paths) in directed graphs (see [1, 10] for shortest path algorithms and their complexity). In the case of undirected graphs, we propose a faster algorithm with time complexity $O(m \log(n) + n^2)$. This paper is organized as follows. In Sect. 2 we propose a generic algorithm to compute shortest guaranteed paths when recovery knowledge is available. Then, in Sect. 3, we propose an algorithm in $O(nm + n^2 \log(n))$ to compute shortest recovery paths, and we show in Sect. 5 that it is optimal for directed graphs. In Sect. 4 we discuss the impact of shared risk link groups on the recovery computations. In Sect. 6, we propose a faster computation in undirected graphs with a time complexity of $O(m \log(n) + n^2)$. Finally, in Sect. 7 we discuss centralized and distributed implementations of fast reroute path computation.

2 Single destination routing algorithm with recovery knowledge

In this section, we present the computation of path with guaranteed length while assuming that a recovery route towards the destination t has already been computed for each possible edge failure. In other words the computation of recovery costs must be done beforehand in the algorithm stack (see Sects. 3, 4 and 6). Therefore, for each vertex u and each edge uv we have the minimum distance from u to t in $G - uv$[1], which is denoted $r(u, uv)$. Note that if the edge uv fails on a path from s to t, the total length of the recovery path will be the sum of the distance between s and u and the value $r(u, uv)$. We will now prove the following lemma:

Lemma 1 (Postfix property) *Let P be a path from u to t beginning with edge uv and let P' be the subpath of P from v to t. Then we have $g(P) = \max\{c(uv) + g(P'), r(u, uv)\}$.*

Proof Let P be a path with from u_0 to u_k with k edges, going through $u_1, \dots, u_{k-1}$. Let $u = u_0$, $v = u_1$, and $t = u_k$. For all $i \in \{0, \dots, k\}$ let P_0^i be the subpath going from u_0 to u_i and for all $i \in \{1, \dots, k\}$ let P_1^i be the subpath going from u_1 to u_i, with $P_1^k = P'$. The guaranteed length of P corresponds to a worst-case scenario: either $g(P) = c(P)$ or there is $i \in \{0, \dots, k-1\}$ such that $g(P) = c(P_0^i) + r(u_i, u_i u_{i+1})$. The developed formula is

$$g(P) = \max\left\{c(P), \max_{i \in \{0,\dots,k-1\}} \{c(P_0^i) + r(u_i, u_i u_{i+1})\}\right\}.$$

Note that $\forall i \in \{1, \dots, k\}, c(P_0^i) = c(uv) + c(P_1^i)$, so this gives

$$g(P) = \max\Big\{c(uv) + c(P'), \max_{i \in \{1,\dots,k-1\}} \{c(uv) + c(P_1^i) + r(u_i, u_i u_{i+1})\}, r(u, uv)\Big\},$$

$$g(P) = \max\Big\{c(uv) + \max\{c(P'), \max_{i \in \{1,\dots,k\}} \{c(P_1^i) + r(u_i, u_i u_{i+1})\}\}, r(u, uv)\Big\}.$$

From this, we conclude that indeed

$$g(P) = \max\{c(uv) + g(P'), r(u, uv)\}. \qquad \square$$

Lemma 1 has a direct implication: it never hurts to have the guaranteed length of P' as low as possible. Thereof, we introduce Algorithm 1 which incrementally constructs shortest guaranteed paths to a single destination, using already computed paths as postfixes. We can already deduce that the shape of the solution is a tree rooted in the destination.

Algorithm 1 (Guaranteed paths with knowledge)

```
Complexity: $O(m + n \times \log(n))$
Input:
  – a destination node $t \in V$
  – a function $c : E \to \mathbb{R}_+$
  – a function $r : V \times E \to \mathbb{R}_+$ (defined only on the pairs $(u, uv)$ such that $uv \in E$)
Output:
  – an array $g$ of reals which gives the guaranteed distance from every vertex to $t$
  – an array father of vertices which gives for every vertex (except $t$) its father in the guaranteed shortest paths tree
Variables:
  – a min-heap priority queue $Q$ of vertices sorted by the value of $g$. The queue admits three operations, insert to insert a vertex in the queue, update to update the queue after a guaranteed distance $g$ has been lowered, and extract which gives the vertex in its root (with minimum $g$) and deletes it from the queue.
  – an array state which gives the state (open or closed) of each vertex.
  – two vertices $u, v \in V$ and a real $r \in \mathbb{R}_+$
Instruction sequence:
1. initialization:
   (a) set the state of every vertex to open
   (b) $g[t] \leftarrow 0$
   (c) initialize the min heap queue $Q$ sorted according to $g$, with only $t$ in its root.
2. while $Q$ is not empty, do
   (a) $v \leftarrow$ extract(Q),
   (b) for every open $u$ such that $uv \in E$ do
       i. compute $r \leftarrow \max\{r(u, uv), (c(uv) + g[v])\}$
       ii. if $u$ was not in the queue, then
           . $g[u] \leftarrow r$
           . father$[u] \leftarrow v$
           . insert $u$ in the queue
       iii. else if $r < g[u]$
           . $g[u] \leftarrow r$
           . father$[u] \leftarrow v$
           . update(Q)
   (c) state$[v] \leftarrow$closed
```

Algorithm 1 is a simple variation of Dijkstra's, and as such completes in $O(m + n \times \log(n))$ steps. Note that its complexity is not more than Dijkstra's (except there is a max, + operation where there is only a + operation in Dijkstra's), and that it will only be added once to the complexity of computing recovery costs.

3 Recovery distance computation

In this section, we focus in computing shortest recovery paths. Given a vertex u and an edge uv we want to know what is the distance from u to t in $G - uv$, and optionally we want to know what is the actual shortest path from u to t in $G - uv$. An obvious way to compute a recovery distance is to remove the edge uv from G and apply Dijkstra's

---

[1] The graph $G - uv$ is defined as $G - uv = (V, E \backslash \{uv\})$.

algorithm on the resulting graph. This is presented as Algorithm 2:

Algorithm 2 (On Demand Recovery Computation)

Complexity: $O(m + n \times \log(n))$
Input:
- vertex $t \in V$
- a function $c : E \rightarrow \mathbb{R}_+$
- a vertex $u \in V$
- an edge $uv \in E$

Output:
- a real r which gives the recovery distance for the couple (u, uv)
- **(optional)** a path *path* which is the shortest recovery path for (u, uv)

Variables:
- none

Instruction sequence:
1. apply Dijkstra's shortest path algorithm on $G - uv$ with destination t
2. retrieve the distance between u to t and store it in r
3. **(optional)** retrieve the path from u to t and store it in *path*

This algorithm has the same complexity as Dijkstra's, $O(m + n \times \log(n))$. If it is applied for each edge, it will take $O(m^2 + mn \times \log(n))$. We may yet do something smarter than to apply it m times. Given a vertex u and an edge uv, you may observe that the shortest path from u to t in $(G - uv)$ is usually the same as in G. In fact, it differs only if uv is part of the shortest path from u to t in G.

We can make another observation: the recovery distance for (u, uv) where uv is not on the Dijkstra tree will be the distance between u to t, and thus always ignored (because it is too small) by Algorithm 1. Therefore, we need only to compute the recovery for edges on the Dijkstra tree, and arbitrarily set to zero (for easy recognition) the recovery distance of the other pairs (u, uv). It is done in the following Algorithm 3:

Algorithm 3 (Global Recovery Computation)

Complexity: $O(n \times m + n^2 \times log(n))$
Input:
- a vertex $t \in V$
- function $c : E \rightarrow \mathbb{R}_+$

Output:
- an array r of reals which gives the recovery distance (or zero) for every couple (u, uv) with $uv \in E$
- **(optional)** an array *path* of paths which gives the shortest path for every vertex towards t
- **(optional)** an array *alternate* of paths which gives a shortest recovery path for every vertex except t

Variables:
- an array *father* of vertices which gives for every vertex (except t) its father in the Dijkstra tree
- two vertices $u, v \in V$

Instruction sequence:
1. for every edge $uv \in E$, set $r[u, uv] = 0$ and $r[v, uv] = 0$
2. apply Dijkstra's shortest path algorithm on G with destination t
   (a) store the result in the array *father*
   (b) **(optional)** store the result in the array *path*
3. for every node $u \neq t$ do
   (a) set $v \leftarrow father[u]$
   (b) apply Dijkstra's algorithm to $G - uv$ with destination t
   (c) retrieve the distance between u to t and store it in $r[(u, uv)]$
   (d) **(optional)** retrieve the path from u to t and store it in *alternate*$[u]$

Algorithm 3 runs $n - 1$ times Dijkstra's and its complexity is $O(nm + n^2 \times log(n))$.

4 Shared risk link groups

A Shared Risk Link Group (SRLG) is a set of edges $SRLG_i \subset E$ that may simultaneously fail. This may happen due to the proximity of physical network components (several optical fibers in a same underground pipe may be severed at the same time), or due to the virtual nature of many MPLS networks (where two virtual edges may represent two physical paths that share a common physical resource). Once identified, SRLG are used by the network administrator to foresee simultaneous failures and enforce sensible recovery strategies.

In the case of fast reroute schemes, only one failure is detected at a time, that is, the failure of the next LSP component, which may belong to several SRLGs. Thus, if the edge uv fails in a path from s to t in the graph G, a recovery path from u to t must avoid all the edges that may have failed at the same time as edge uv. In other words, the recovery path from u to t must be computed in the graph $G - E_{uv}$, where E_{uv} is the union of all the shared risk link groups that contain uv ($E_{uv} = \bigcup_{uv \in SRLG_i} SRLG_i$).

Using an identical argument as in the previous section, we can see that the recovery distance from u to t in $G - E_{uv}$ needs only to be computed in the case where E_{uv} intersects the shortest path from u to t. This can happen at worst for every edge uv, which means that the computation of recovery costs may take up to $m(rk + m + n \times \log(n))$ time steps, where r is the number of shared risk link groups, k their size, and rk is the time taken mark the edges in E_{uv}.

5 Recovery distances in directed graphs

In this section, we show in Theorem 1 that computing single destination recovery paths in a directed graph G' with $2n + 1$ vertices and $m + 3n$ edges is as hard as to compute shortest paths between n arbitrary pairs of vertices in a directed graph G with n vertices and m edges. Therefore, the complexity $O(nm + n^2 \times \log(n))$ for finding single destination recovery paths is unbeatable, unless a major breakthrough occurs in the field of Multiple Source Shortest Paths (MSSP) algorithms.

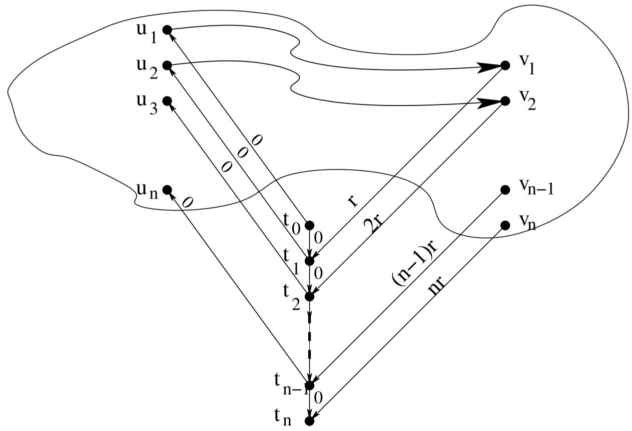


Fig. 3 The graph constructed by Algorithm 4

Theorem 1 *Given a directed graph* $G = (V, E)$ *with* n *vertices and* m *edges, given a function* $c : E \to \mathbb{R}_+$ *and given* n *pairs of vertices, we can construct in linear time a graph* $G' = (V', E')$ *with* $2n + 1$ *vertices and* $m + 3n$ *edges, compute a function* $c' : E' \to \mathbb{R}_+$ *such that* $V \subset V'$, $E \subset E'$, c' *equals* c *on* E, *and such that computing recovery paths towards a certain vertex* t *in* G' *gives us the shortest paths between the* n *pairs of vertices in* G.

In order to prove Theorem 1, we will first give the linear algorithm for the reduction (Algorithm 4), illustrated in Fig. 3, and then prove that this algorithm produces the required graph for Theorem 1.

Algorithm 4 (Reduction)

```
Complexity: $O(n + m)$
Input:
  – a directed graph $G = (V, E)$ with $n$ vertices and $m$ edges
  – a function $c : E \to \mathbb{R}_+$
  – $n$ pairs $(u_1, v_1), (u_2, v_2) \dots (u_n, v_n)$ of vertices
Output:
  – a directed graph $G' = (V', E')$ with $2n + 1$ vertices and $m + 3n$ edges
  – a function $c' : E' \to \mathbb{R}_+$
Variables:
  – an edge $e$, an integer $i$, a real $r$
Instruction sequence:
1. set $r = 1 + \sum_{e \in E} c(e)$
2. construct $n + 1$ vertices $t_0, t_1, \dots t_n$
   (a) set $V' = V \cup \{t_0, t_1, \dots t_n\}$
3. construct $3n$ edges:
   (a) construct $n$ edges $t_0t_1, t_1t_2, \dots t_{n-1}t_n$
   (b) construct $n$ edges $t_0u_1, t_1u_2, \dots t_{n-1}u_n$
   (c) construct $n$ edges $v_1t_1, v_2t_2, \dots v_nt_n$
   (d) set $E' = E \cup \{t_0t_1, t_1t_2, \dots t_{n-1}t_n\} \cup$
       $\{t_0u_1, t_1u_2, \dots t_{n-1}u_n\} \cup \{v_1t_1, v_2t_2, \dots v_nt_n\}$
4. $\forall e \in E$ set $c'(e) = c(e)$
5. for $i$ from 1 to $n$
   (a) set $c'(t_{i-1}t_i) = 0$
   (b) set $c'(t_{i-1}u_i) = 0$
   (c) set $c'(v_it_i) = r \times i$
```

Proof It is self-evident that Algorithm 4 runs in time $O(n + m)$, that its output G' contains G as a subgraph, has $2n + 1$ vertices and $m + 3n$ edges, and that the two functions c and c' are equal on E. We will now prove that computing shortest recovery paths towards vertex t_n in G' gives us the shortest paths between the pairs $(u_1, v_1), (u_2, v_2), \dots (u_n, v_n)$ in G.

First, since $r = \sum_{e \in E} c(e)$, we know that any single edge v_it_i is longer that any path in G (provided it has no loops). Moreover, any single edge v_jt_j is longer that any path containing only one edge v_it_i, with $i < j$.

For any $i \in \{1, \dots, n\}$, consider the shortest path from t_{i-1} to t_n. It is obviously the zero cost path $t_{i-1}t_it_{i+1} \dots t_n$. The recovery path for the edge $t_{i-1}t_i$ has to go through u_i since t_{i-1} has only two outgoing edges. Then, the recovery path has to attain one of the vertices t_j with $j \geq i$ to finally reach t_n. The only way to do so is to use one of the edges v_jt_j with $j \geq i$. In order to have a length no greater than $(i + 1)r$, the path necessarily goes through v_it_i: the shortest recovery path for $t_{i-1}t_i$ goes through u_i and t_i. Since subpaths of shortest paths are shortest paths themselves, it follows that the shortest recovery path for $t_{i-1}t_i$ contains the shortest path between u_i and v_i. Moreover, the length of the recovery path is exactly $r \times i$ plus the length of the shortest path between u_i and v_i. □

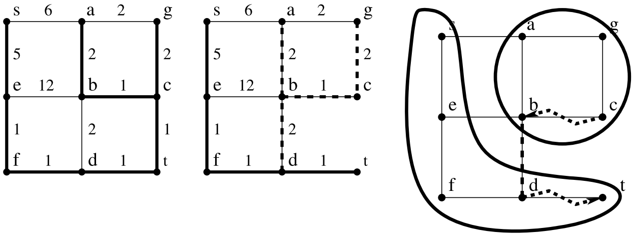
6 Recovery distances in undirected graphs

In this section, we present a more efficient algorithm to compute recovery paths by exploiting the properties of undirected graphs. Given an undirected graph $G = (V, E)$, a cost function $c : E \to \mathbb{R}_+$ and a destination vertex $t \in V$, we call *Dijkstra tree* the tree composed by the shortest paths in G towards t. In this section, we say that u is a *child* of v if the shortest path from u to t in the Dijkstra tree contains v (as such, u is a child of itself).

6.1 Properties of shortest recovery paths

Given a couple (u, uv), we have already seen in Sect. 3 that the recovery distance from u to t in $G - uv$ was virtually zero if v was not the father of u in the Dijkstra tree. Thus, we now suppose that v is the father of u. The shortest recovery path of (u, uv) is by definition, the shortest path from u to t on $G - uv$. Notice that we already know some of the shortest paths in $G - uv$: indeed, all the vertices y that are not children of u in the Dijkstra tree have the same shortest paths towards t in both G and $G - uv$. This is illustrated in Fig. 4.

Since a subpath of a shortest path is itself a shortest path, we deduce that a shortest path from u to t in $G - uv$ can be decomposed into three parts: a shortest path P_1 from u to x



(a) (b) (c)

Fig. 4 Recovery path properties: (**a**) Dijkstra tree rooted in t; (**b**) shortest paths towards t in $G - ct$; (**c**) decomposition of the recovery path from c to t after ct failed

where all the vertices are children of u (x included), an edge xy, and a shortest path P_2 from y to t in the Dijkstra tree (see Fig. 4c). In undirected graphs, shortest paths from u to its children are the same as from the children to u, so both P_1 and P_2 are part of the Dijkstra tree. x should be a child of u and y should not be a child of u.

6.2 Dijkstra with memory

In order to properly use the Dijkstra tree, we use Algorithm 5, a variant of Dijkstra's algorithm which keeps the memory of paths and children for each vertex. Its complexity is $O(n^2)$ (equal to the size of its output).

Algorithm 5 (Dijkstra's with Memory)

Complexity: $O(n^2)$
Input:

- an *undirected* graph $G = (V, E)$
- a destination node $t \in V$
- a function $c : E \to \mathbb{R}_+$

Output:

- an array *distance* of reals which gives for each vertex the distance from the vertex to t
- an array *path* of arrays which gives for each vertex the shortest path from this vertex to t, under the form of an array of vertices, starting with t. As an illustration, we would have $path[s] = [t, d, f, e, s]$ in our usual example (see Fig. 2).
- an array *children* of sets of vertices which gives for each vertex its children in the Dijkstra tree

Variables:

- an array *father* of vertices which gives for every vertex except t its father in the Dijkstra tree
- a min-heap priority queue Q of vertices sorted by *distance*. The queue admits three operations, *insert* to insert a vertex in the queue, *update* to update the queue after a *distance* has been lowered, and *extract* which gives the vertex in its root (with minimum *distance*) and deletes it from the queue.
- an array *state* which gives the state (*open* or *closed*) of each vertex.
- two vertices $u, v \in V$ and a real $p \in \mathbb{R}_+$

Instruction sequence:

```
1. initialization:
   (a) set the children of every vertex to $\emptyset$
   (b) set the state of every vertex to open
   (c) set the distance of $t$ to zero
   (d) set $path[t]$ to $[t]$
   (e) initialize the min heap queue $Q$ sorted by distance, with
       only $t$ in the root
2. while $Q$ is not empty, do
   (a) $v \leftarrow extract(Q)$,
   (b) for every open neighbor $u$ of $v$
       i. compute $p = distance[v] + c[uv]$
       ii. if $u$ was not in the queue, then
           . set $distance[u] = p$
           . set $father[u] = v$
           . insert $u$ in the queue
       iii. else if $p < distance[u]$
           . set $distance[u] = p$
           . set $father[u] = v$
           . update(Q)
   (c) for every vertex $u$ found in $path[father[v]]$ do
       i. update $children[u] \leftarrow children[u] \cup \{v\}$
   (d) set $path[v]$ to the concatenation of $path[father[v]]$ and $[v]$
   (e) set $state[v] \leftarrow closed$
```

6.3 Computing recovery data on the edges

Given a vertex u and the edge uv in the Dijkstra tree, our main problem is so to find a vertex x among the children of u and an edge xy such that a shortest path from u to x, the edge xy and a shortest path from y to t would make a short and correct recovery path towards t.

The length of this path is $c(P) = d(u, x) + c(xy) + d(y, t)$ where $d(u, x)$ is the distance between u and x. In another formulation, $c(P) = d(x, t) + d(y, t) + c(x, y) - d(u, t)$. This latter formulation is very interesting because the first part $d(x, t) + d(y, t) + c(x, y)$ is specific to the edge xy, whereas the second part $-d(u, t)$ is constant for the vertex u. Therefore we want to minimize $\mathbf{ed}(xy) = d(x, t) + d(y, t) + c(x, y)$ among the eligible edges xy. Algorithm 6 computes this value *ed* for every edge in G.

Given an edge xy, we will also compute the distance $d(z, t)$ where z is the common ancestor of x and y. This distance is named **depth**(xy). If x is a child of u and y not a child of u, then the common ancestor of x and y should also be an ancestor of u, and so $depth(xy) < d(u, t)$. If x and y are both children of u then $depth(xy) \geq d(u, t)$. Therefore, when considering vertex u, one could scrap right away the edges xy with $depth(xy) \geq d(u, t)$, and then be confident that remaining edges from its children will yield correct recovery paths. In Algorithm 6, we compare the arrays $path(x)$ and $path(y)$ to find the common ancestor z. This search takes at worst $O(\log(n))$ steps if done properly, so the overall computation takes $O(m \times \log(n))$.

Given a vertex u, and one of its children x, we will want to find the edge xy that minimizes $ed(xy)$, and with $depth(xy) < d(u, t)$. We order these edges by strictly increasing *ed* (so the minimum is first) and strictly decreasing *depth*; edges with greater *ed* and greater *depth* would be less effective and less likely to be eligible, so they are discarded.

The ordering of edges takes for each vertex $O(k \times \log(k))$ steps, where k is its degree, which can be upper bounded by $O(k \times \log(n))$. Summed on all the vertices, this gives an overall complexity of $O(m \times \log(n))$.

Algorithm 6 (Ordering edges)

```
Complexity: $O(m \times \log(n))$
Input:
  – an undirected graph $G = (V, E)$
  – a vertex $t \in V$
  – a function $c : E \to \mathbb{R}_+$
  – an array distance of reals which gives for each vertex the distance from the vertex to $t$
  – an array path of arrays (see Algorithm 5)
Output:
  – an array ed of reals which gives for every edge $xy \in E$ the value distance[x]+distance[y]+c(xy)
  – an array depth of reals which gives for every edge $xy \in E$ the distance of the closest common ancestor of $x$ and $y$ in the Dijkstra tree
  – an array neighbors of vertices that gives for every vertex a set of its neighbors ordered by increasing ed and decreasing depth.
Variables:
  – three vertices $x, y, z \in V$
Instruction sequence:
1. for every edge $xy \in E$ do
   (a) set ed[$xy$] =distance[$x$]+distance[$y$] + $c(xy)$
   (b) compute the vertex $z$ such that the arrays path[x] and path[y] are equal until $z$, and differ afterwards.
   (c) set depth[$xy$] =distance[$z$]
2. for every vertex $x \in V$ do
   (a) order the neighbors $y$ of $x$ by increasing ed[$xy$]
   (b) in the process, discard the neighbors that would have a greater or equal ed[$xy$] and a greater or equal depth[$xy$]
   (c) the remaining neighbors are now ordered by increasing ed[$xy$] and decreasing depth[$xy$]
   (d) store the result in the array neighbors[$x$]
```

6.4 Computing recovery distances and paths

Given all the data gathered by Algorithm 6, we are now able to compute shortest recovery paths. For each vertex u, we will need to get rid of the edges xy with $depth(xy) \geq d(u,t)$. In order to do this efficiently, we will order the vertices u by decreasing distance to t. At the start of Algorithm 7, we take the vertex u with maximum $d(u,t)$, so of course for any edge xy we have $depth(xy) \leq d(u,t)$. We need only to remove those with $depth(x,y) = d(u,t)$. Since the neighbors of every vertex x are ordered by strictly decreasing depth, it takes only one operation per vertex to remove all of these edges. At some point in the algorithm with a vertex u, assuming we have $depth(xy) < d(u,t)$ for every edge xy, we will again have $depth(xy) \leq d(u,t)$ when moving to the next vertex u, and so it will take again n operations to remove all the ineligible edges.

Once all the ineligible edges have been removed, we know that the edge we search is one with minimum $ed(xy)$, with x among the children of u. Since the edges are ordered by increasing ed, only the first neighbor of every children needs to be considered, so it takes only $O(n)$ operations. The complexity of ordering the vertices ($O(n \times \log(n))$) is smaller than the $O(n)$ operations per vertex we do afterwards, so the overall complexity of Algorithm 7 is $O(n^2)$.

Algorithm 7 (Computing recovery costs)

```
Complexity: $O(n^2)$
Input:
  – an undirected graph $G = (V, E)$
  – a vertex $t \in V$
  – an array distance of reals which gives for each vertex the distance from the vertex to $t$
  – an array depth of reals (see Algorithm 6)
  – an array ed of reals (see Algorithm 6)
  – an array neighbors of vertices (see Algorithm 6)
  – an array children of sets of vertices (see Algorithm 5)
Output:
  – an array $r$ of reals which gives the recovery distance (or zero) for every couple $(u, uv)$ with $uv \in E$
  – (optional) an array bridge of edges which gives the critical edge for an alternate route for every vertex except $t$
Variables:
  – four vertices $u, v, x, y$, a real $r$
Instruction sequence:
1. for every edge $uv \in E$, set $r[u, uv] = 0$ and $r[v, uv] = 0$
2. order the vertices of $V$ by decreasing distance
3. for vertex $u \neq t$ from the one with the greatest distance to the one with the lowest, do
   (a) for all vertex $x$ remove the first entry of neighbor[$x$] if the depth of the entry is greater or equal to distance[$u$]
   (b) let $v$ be the first entry of neighbor[$u$]
   (c) remove $v$ from neighbor $u$
   (d) set $r \leftarrow \infty$
   (e) for all vertices $x$ in children[$u$] do
       i. if neighbor[$x$] is not empty, do
          . let $y$ be the first entry of neighbor[$x$]
          . if ed[$xy$] $< r$ then
               set $r \leftarrow$ed[$xy$]
               (optional) set bridge[$u$] $= xy$
   (f) set $r[u, uv] = r -$distance[$u$]
```

7 Distributed implementation

Fast reroute path computation can be done within the frameworks of distance-vector routing protocols, link-state routing protocols, or centralized network administration.

In the case of centralized administration or link-state protocols (which duplicate the centralized computation on each router), the computation of shortest recovery paths can be simply done by using Algorithm 3, that runs with complexity $O(nm + n^2 \log(n))$. If the network has asymmetric links, this complexity can hardly be beaten as shown in Theorem 1. Otherwise it is possible to compute fast reroute paths with complexity $O(m \log(n) + n^2)$, by first running Dijkstra with memory (Algorithm 5) in $O(n^2)$ steps, by then ordering the edges of the graph with Algorithm 6 in $O(m\log(n))$ steps, and by eventually computing recovery distances and paths with Algorithm 7 in $O(n^2)$ steps. Once recovery paths and distances have been computed for each edge of the network, we may use the postfix property of guaranteed paths (Lemma 1) and compute shortest guaranteed paths with Algorithm 1 in $O(m+n\log(n))$ steps.

It is worth noting that the computation of recovery data on the edges of undirected graphs (see Sect. 6.3) lends itself well to a distributed approach. We propose a three-layered adaptation of distance-vector computation. Each node u in the network has a traditional *distance* table of size $O(n^2)$ that may contain for each destination t known by u the computed distance from u to t along with a shortest path from u to t, a *recovery* table that may contain for each known node t a recovery edge xy as well as its edge distance $ed(xy)$, common ancestor z and *depth* (cf. Algorithm 6, and a fast reroute *routing* table that may contain for every destination t known by u a neighbor and a guaranteed distance. All tables are initially empty. The *distance* table is classically maintained using Bellman-Ford's distributed variant used in distance-vector algorithms. The shortest paths are remembered within the table, so as to avoid count-to-infinity problems in case of link failure, and to compute edge distance and depth of edge uv for every neighbor u and every possible destination t. The *recovery* table is maintained by forwarding recovery information towards each possible destination t using the *distance* table. When receiving recovery information, node u checks the validity of recovery edges xy towards t by looking if the common ancestor z is indeed in the shortest path from u to t. Valid recovery edges are forwarded towards t, and the one with shortest edge distance is inserted in the *recovery table*. Finally, the *routing* table proper is built by replacing the operation $c(u,v)+d(v,t)$ with $\max(r(u,v), c(u,v)+d(v,t))$ if and only if v is in the shortest path from u to t as recorded in the *distance* table. In case of synchronized simultaneous exchanges of messages between nodes, each layer of the fast reroute tables computation scheme converges at worst n rounds after the previous layer has converged.

8 Conclusion

In this paper, we have investigated the algorithmic aspects of computing original paths along with their back-up so that quality-of-service constraints are satisfied under the scenarios of a single link failure or of the failure of multiple links belonging to the same Shared Risk Link Group (SRLG). We have seen that to solve this problem, it is required to compute beforehand recovery distances for each link that may fail, and this may be as hard as computing multiple source shortest paths (Theorem 1) in directed graphs. The algorithm we propose (Algorithm 3), and which runs in $O(nm+n^2\log(n))$ steps, is therefore arguably optimal. Nonetheless, we also propose a faster algorithm in the case of undirected links with time-complexity $O(m\log(n)+n^2)$. Both algorithms can be implemented for centralized as well as distributed network administration; the convergence time for the distributed version is no greater than $3n$ rounds.

References

1. Cherkassky, B. V., Goldberg, A. V., & Radzik, T. (1994). Shortest paths algorithms: theory and experimental evaluation. In *SODA '94: proceedings of the fifth annual ACM-SIAM symposium on discrete algorithms* (pp. 516–525). Philadelphia: Society for Industrial and Applied Mathematics.
2. Drid, H., Cousin, B., Molnar, M., & Lahoud, S. (2010). A survey of survivability in multi-domain optical networks. *Computer Communications*, *33*(8), 1005–1012. http://dx.doi.org/10.1016/j.comcom.2010.02.003.
3. Hu, J. Q. (2003). Diverse routing in optical mesh networks. *IEEE Transactions on Communications*, *3*(51), 489–494.
4. Jokela, P., Zahemszky, A., Esteve, C., Arianfar, S., & Nikander, P. (2009). Lipsin: line speed publish/subscribe inter-networking. In *SIGCOMM '09: proceedings of the ACM SIGCOMM 2009 conference on data communication* (pp. 195–206). New York: ACM. http://doi.acm.org/10.1145/1592568.1592592.
5. Kiaei, M. S., Assi, C., & Jaumard, B. (2009). A survey on the *p*-cycle protection model. *IEEE Communications Surveys & Tutorials*, *11*(3).
6. Pan, P., Swallow, G., & Atlas, A. (2005). *Rfc-4090, fast reroute extensions to rsvp-te for lsp tunnels*.
7. Raj, A., & Ibe, O. C. (2007). A survey of ip and multiprotocol label switching fast reroute schemes. *Computer Networks*, *51*(8), 1882–1907. http://dx.doi.org/10.1016/j.comnet.2006.09.010.
8. Ramaswami, R., Sivarajan, K., & Sasaki, G. (2010). Network survivability. In *Optical networks: a practical perspective* (3rd edn.). San Francisco: Morgan Kaufmann.
9. Zahemszky, A., & Arianfar, S. (2009). Fast reroute for stateless multicast. In *Proceedings of ICUMT'09*, St. Petersburg (pp. 1–6). http://dx.doi.org/10.1109/ICUMT.2009.5345576.
10. Zwick, U. (2001). Exact and approximate distances in graphs—a survey. In *ESA '01: proceedings of the 9th annual European symposium on algorithms* (pp. 33–48). London: Springer.

Aubin Jarry is an Assistant Professor at the Centre Universitaire d'Informatique of the University of Geneva. He has been a postdoctoral researcher in Geneva since 2005 after obtaining his PhD at the French Institut de Recherche en Informatique et Automatique at Sophia Antipolis. His main research domain is graph theory, dynamic networks and energy balancing and obstacle avoidance in sensor networks. He also works on applications to sensor networks modeling and algorithms, game theory and the study of energy measures of computation. He has contributed to the conception and realization of several projects financed by the Swiss National Science Foundation and the European Union.