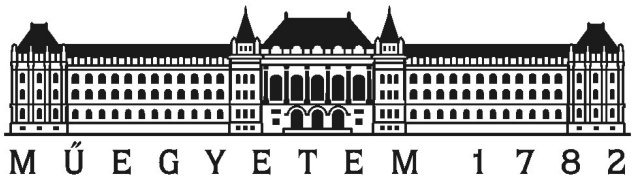


DEPARTMENT OF TELECOMMUNICATIONS AND MEDIA INFORMATICS
BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS

NOVEL ALGORITHMS FOR IP FAST REROUTE

Ph.D. Theses

By

Gábor Enyedi

Research Supervisor:

Dr. Gábor Rétvári
Department of Telecommunications and Media Informatics

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS
BUDAPEST, HUNGARY
FEBRUARY 2011

BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS

Date: **February 2011**

Author: **Gábor Enyedi**

Title: **Novel Algorithms for IP Fast ReRoute**

Department: **Department of Telecommunications and Media Informatics**

Degree: **Ph.D.** Convocation: **February** Year: **2011**

Permission is herewith granted to Budapest University of Technology and Economics to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

The reviews and the records of the department debate are available at the Dean's Office.

Signature of Author

Table of Contents

Abstract

The popularity of Internet and IP networks have risen dramatically in the last few decades. Unfortunately, this increasing popularity has brought serious problems as well. Currently, IP networks transport not only elastic traffic, as they did traditionally, but also real time traffic, like Voice over IP (e.g., in 3G or 4G mobile networks), IPTV, on-line gaming or stock exchange transactions. Long service disruption is not acceptable for these applications.

Recovery in current IP networks is based exclusively on reactive restoration techniques like OSPF and IS-IS. When restoration is applied, the network starts dealing with the way of bypassing the failed resource, after the failure occurred. Naturally, such mechanism needs some time to restore connectivity.

Faster recovery can be realized by protection techniques. These techniques are proactive, and compute detours long before the failure occurs. Thus, when a failure actually happens, traffic can be quickly switched to these precomputed detours. Unfortunately, there exists no native protection technique for IP networks, which is an important shortcoming nowadays. Therefore, serious efforts are made in order to endow IP with such capability. This native protection is called IP Fast ReRoute (IPFRR).

Unfortunately, although several IPFRR proposals do already exist, none of them was able to meet all the needs. Some of them are not able to cover all the failures, others may create long lasting forwarding loops. There are few, which could be applied, however, their extra administrative burden is unacceptable for operators.

This dissertation shows a possible way for overcoming these problems by applying special directed spanning trees called *redundant trees*. Since a pair of redundant trees has the capability that no single failure can disrupt the connection with the destination on both of the trees, they can be well applied in the field of IPFRR.

Therefore, the concept of redundant trees is improved in several ways in this dissertation. First, heuristics are proposed in order to decrease their costs and the length of paths along them. Second, a distributed algorithm is proposed, which

significantly reduce the complexity of finding redundant trees. Finally, the serious limitation of redundant trees that they can be found only in 2-connected graphs is lifted by generalizing the original concept; *maximally redundant trees* are introduced, which can be found in arbitrary connected graphs.

In order to utilize these results, I propose new IPFRR techniques as well. Loop-free Failure Insensitive Routing always avoids forming loops, in this way overcoming the drawback of IPFRR techniques using interface-based forwarding. Moreover, one of the most promising IPFRR proposals, Not-via, is improved by introducing its lightweight version having significantly decreased management and computational burden.

Kivonat

Az elmúlt évtizedekben az Internet és általánosságban az IP hálózatok népszerűsége drámai növekedést mutatott. Sajnos azonban ez a növekvő népszerűség komoly gondokat is hozott magával. Manapság az IP hálózatokat már nem csak elasztikus forgalom továbbítására használjuk – ellentétben azzal, ahogy ezt korábban tettük –, hanem olyan valós idejű átvitelekhez is, mint amilyen a hangátvitel (Voice over IP – VoIP, például 3G vagy 4G mobil hálózatokban), IPTV, on-line játékok vagy tőzsdei kereskedés. A hosszú szolgáltatáskiesés ezen alkalmazások számára nem elfogadható.

A mai IP hálózatokban a helyreállás kizárólag olyan reaktív újjáépítő (restoration) megoldásokra alapul, mint amilyen az OSPF vagy az IS-IS. Mikor azonban újjáépítést alkalmazunk, a hálózat csak az után kezd foglalkozni a meghibásodott erőforrás elkerülésének módjával, hogy a hiba maga bekövetkezett. Természetesen az ilyen megoldásoknak bizonyos időre van szükségük a kapcsolat helyreállításához.

Gyorsabb helyreállás érhető el védelmi technikák alkalmazásával. Ezek a technikák proaktívak, azaz jóval azelőtt kiszámítják az elkerülő utakat, hogy a hiba bekövetkezne. Így aztán – mikor a hiba ténylegesen bekövetkezik – a forgalom gyorsan ezekre az előkészített utakra irányítható. Sajnos azonban az IP nem rendelkezik saját védelmi módszerrel, ami egy fontos hátránnyá vált manapság, és komoly erőfeszítéseket váltott ki az IP-t védelmi képességekkel történő felruházására érdekében. Ezeket a védelmi módszereket nevezzük összefoglalóan IP alapú gyors hibajavításnak (IP Fast ReRoute – IPFRR).

Habár számos IPFRR módszer létezik, ezek egyike sem minden igényt kielégítő. Néhány nem képes minden hibát védeni, mások pedig továbbítási hurkokat képezhetnek. Van néhány, amely alkalmazható lenne, ám ezek extra adminisztrációs terhe elfogadhatatlan az operátorok számára.

Ez a disszertáció egy lehetséges utat mutat *redundáns fának* nevezett speciális feszítőfák segítségével ezen problémák orvoslására. A redundáns fák jól alkalmazhatóak az IPFRR területén, mivel rendelkeznek azzal a tulajdonsággal, hogy egyszeres hiba nem szakíthatja meg a céllal a kapcsolatot mindkét fán.

A redundáns fák koncepcióját számos ponton kiterjesztem ebben a disszertációban. Először is heurisztikát javaslok költségük, valamint a fák mentén található utak hosszának csökkentésére. Ezen kívül javaslok egy elosztott algoritmust, ami számottevően képes csökkenteni a redundáns fák kereséséhez szükséges számítások komplexitását. Végezetül a redundáns fák általánosításával azok egy komoly hiányosságát orvosolom, nevezetesen hogy ezek a fák csak 2-összefüggő hálózatokban találhatóak. Bevezetem a *maximálisan redundáns fákat*, amelyek már tetszőleges összefüggő hálózatban léteznek.

A kapott eredményeket új IPFRR technikákban alkalmazom. A Loop-free Failure Insensitive Routing mindig képes a továbbítási hurkok elkerülésére, így megoldja az interface-alapú IPFRR módszerek fontos hibáját. Továbbá az egyik legigéretesebb javaslat, a Not-via is kiterjesztésre kerül annak egyszerűsített verziójával. Ez a megoldás számottevően csökkenti a szükséges menedzsment terheket valamint számítási komplexitást.

Acknowledgements

In the first place, I would like to thank all the help and support I got from Gábor Rétvári; no student could even wish a better supervisor. Furthermore, special thanks go to András Császár, who also helped me countless times, and who spoke me first about IP Fast ReRouting.

I would like to thank the support of my family, who always made me possible to study, and who always believed in me, even when I did not. Without them I would have no chance to even become a PhD student.

Last, but not least, I would like to thank Róbert Szabó, Tamás Henk, Tibor Cinkler, Erzsébet Győri and everyone else from the Department of Telecommunications and Media Informatics, who made me possible to focus exclusively on my research. I know how unique this possibility was.

Chapter 1

Introduction

1.1 Principles

In the last few decades, communication has changed our world. Thanks to the massive improvement of both Internet and mobile telephony, now it takes almost no time to find the decent information or reach somebody almost anywhere. Moreover, corporations have changed and nowadays the whole economy depends more or less on communication networks.

Communication networks have one responsibility: transporting information from one to some other points. Naturally, since it is impossible to directly connect all the devices, it is needed to find path(s) in the network from the source to the destination(s). Mechanisms for finding these paths are called *routing* mechanisms.

Since communication networks mesh the world, they need to be quite huge in size. Naturally, in a system huge enough, sooner or later a failure occurs. It is a natural desire that if after the failure of some resources transporting information is still possible, network should remain operable. Mechanisms providing this self-healing aspect are called *recovery* mechanisms.

There are two fundamentally different types of recovery [VPD04, MP06, RSM03]. One approach, called *restoration*, *reactively* deals with the failure after it occurred. Although some precomputation can take place, the way of bypassing the failed resource is computed only after the failure.

The main advantage of this approach is its simplicity and robustness. Since the failed resource is exactly known, it can well adapt to all the situations. Unfortunately, since significant part of the operation is done only after the failure, this approach can become slow in some cases.

In order to overcome the problems of restoration, *protection* techniques are applied. Protection techniques are *proactive*, since they find the way of bypassing some failures long before they happen. Naturally, since preparing to arbitrary number of failures is next to impossible, these techniques prepare to only a given number and given type of failures.

The advantage of protection techniques is their speed. Although, some signaling can be needed after the failure, the most important tasks are done long before it. Obviously, protection techniques, even with limiting the number of failures which can be bypassed, are much more complex than restoration ones are. Moreover, they cannot adapt to the new situation as well as restoration can, so they are commonly used together with restoration: as a first aid, protection recovers the service instantly, then restoration optimize the configuration and gives the possibility to prepare to new failures using protection.

Before further discussing recovery techniques, it is important to deal first with the two most common ways of routing. In a *(virtual) circuit switched* network a path for transporting is established, which is the basic object of routing. Therefore, this type of routing is called *connection oriented*. In these networks, all the paths are managed separately and it is possible to establish two paths between the same two endpoints, which were computed in fundamentally different ways. This approach provides quite high control on forwarding, making it relatively easy to bypass a failed resource. In connection oriented networks typically both protection and restoration techniques are applied.

On the other hand, it is possible to send data in a *connectionless* manner, without explicitly establishing the paths. This scheme can be applied mostly in *packet switched* networks, where the transported information is split into small pieces called packets. Each packet has a header with the information needed for forwarding it to the proper destination. Typically, packets contain a destination address, and they are forwarded based on this address. Therefore, I assume in the sequel that the next hop is determined by the destination.

Any routing, where the next hop is selected based on the destination address, defines a partial order of nodes per destination, where each node, except the destination, has at least one lower neighbour. Say that node a is lower than node b, if a packet heading to destination d from a can never reach b, but there is a possible packet flight from b to d through a. Observe that this order has a lower bound, destination d. The inverse of this claim is also true; if there is a partial order of nodes, where each node has at least one lower neighbour, expect exactly one node d, always forwarding packets to a lower node eventually makes up a routing towards d. In this way, computing

a routing is the same problem as computing a proper partial order for each node as a destination. In the common case, when links have lengths and packets are forwarded along the shortest path, this partial order is a total order; packets are forwarded to a node with smaller distance to the destination.

If a failure occurs, closer nodes may become unavailable. Therefore, restoration in packet switched networks typically means recomputing the partial orders. Protection in these networks is usually not used, although it would mean to switch to another, precomputed order.

This lack of protection is a growing problem nowadays due to the development of Internet. Current Internet is based on Internet Protocol (IP) [Pos81], a typical connectionless, packet switched protocol, with forwarding (typically) based on the destination address, and with extremely robust restoration but no protection. Thus, the recovery provided by IP is quite slow, it can take several seconds even in the simplest but very common case, when a single failure occurs [ICM+02, MIB+04]. This slow recovery is acceptable for the traditional elastic traffic, which IP was designed for. Unfortunately, current IP networks are used to transport real-time traffic too, such as the traffic of (video)telephoning (e.g., 3G, 4G mobile networks), on-line gaming, TV broadcasting or even business critical stock exchange transactions. These types of traffic need to avoid seconds of service disruption.

Moreover, currently several companies use Virtual Private Networks (VPN) in order to interconnect their geographically separated divisions. The quality of service of these VPNs is typically defined in a contract called Service Level Agreement (SLA). Since some of these companies use several real-time or delay sensitive applications (like continuous database connection or remote desktop), these SLAs can be quite strict with serious consequences, if the service provider fails to fulfil their requirements. In order to provide such VPN connections on pure IP, a native protection scheme is indispensable.

Currently, there is only one possibility for providing fast recovery in IP networks: operators need to use the protection capabilities of another network layer below IP. Fortunately, there is usually some connection oriented layer under IP, e.g., MultiProtocol Label Switching (MPLS) [PSA05] or some optical layer [SRM02]. Naturally, in order to use the protection capabilities of this layer, intensively using its routing techniques is needed, which brings up management issues. Since Internet is based on IP, configuring IP routing cannot be avoided, so configuring an additional layer means extra management cost. Moreover, some operators completely rely on IP routing, ignoring protection capabilities [ICM+02, MIB+04], thus, native IP protection techniques would be desirable alternatives for them.

Hence, serious efforts are being made, in order to endow IP with protection, known as IP Fast ReRoute (IPFRR). Internet Engineering Task Force (IETF) has already standardized the IPFRR framework [SB10b], and several proposals were made. Some of them are already on their ways to be applied in real networks [AZ08, BSP10].

The rest of this dissertation is organized as follows. In this chapter, I introduce the concept and possibilities of IPFRR, and I briefly review current solutions. In Chapter 2, the problems of IPFRR methods using interface-based forwarding are discussed. There, I show a concept, which can be applied more generally. In Chapter 3 and Chapter 4, I generalize this concept and give significant new results on a well studied area of graph theory. Finally, in Chapter 5, I use this concept in order to improve Not-via addresses, an IPFRR technique, which may have the most significant IETF and industrial backing currently. Finally, the results are summarized in Chapter 6.

1.2 IP Fast ReRoute – principles

Previously, the recovery problems of current IP networks were discussed. In this section, I introduce the concept of possible solutions, the IPFRR techniques. First, we discuss the requirements and then focus on the realization in general.

The basic requirement IP Fast ReRoute techniques are needed to meet, is providing fast protection capability inside an autonomous system [SB10b]. Although there is no theoretical limitation, which would avoid us making inter-domain IPFRR mechanisms, but, as it will turn out, native protection in a pure IP network raises numerous problems even without policy routing or security issues.

Second, it is also very important to retain the forwarding system of IP. Naturally, some changes are needed, but these changes must be as moderate, as it is possible.

Third, the time needed for recovery must be significantly decreased to a level, which is acceptable even for real-time traffic. Going by a rule of thumb, it is usually said that the recovery must be done in about 50 milliseconds [VPD04, Gjo07], since it is tolerable even for telephone calls in SDH/SONET networks.

Forth, it is needed to recognize that preparing to arbitrary number of failures is impossible. Moreover, multiple unrelated failures are extremely rare [MIB+04]. Therefore, complete protection against multiple unrelated failures is not in the scope of IPFRR [SB10b], albeit protection against as many single failure cases as it is possible is needed.

Finally, since fast rerouting using only pure IP is complicated, protection paths

cannot meet such capacity guarantees like the ones in a connection oriented network. In this way currently, only a "best effort" congestion mitigation is required by possibly decreasing the length of detours.

To fulfil these requirements, first, fine tuning of current IP restoration techniques was attempted. For the most common Interior Gateway Protocols (IGP), namely for Open Shortest Path First (OSPF) [Moy98] and Intermediate System to Intermediate System (IS-IS) [fS02], it was proven that theoretically, it is possible to reach some 100s of milliseconds convergence [FFEB05, AJY00, ST08], but by fine tuning of real routers, reaching only some seconds is more likely [ICM+02].

In order to overcome the problems of current solutions, it is needed to discuss the main reasons why IP networks provide slow recovery after a failure. In current IP networks typically a Link State Routing mechanism, OSPF or IS-IS, is responsible for computing the paths. This means that the routers in the network advertise the current state of their links, and in this way in a non-transient state all the nodes have the same complete topology of the network. Using this topology, distributed computation of the shortest paths is possible.

When a failure occurs, routers have three main tasks[1]: first detecting the failure, second advertising the fact of the failure and finally recomputing and installing the new forwarding information base.

Detecting a failure can be done basically in two ways: either the physical layer can detect it (e.g., the loss of voltage can be detected) or some kind of fast hello like protocol can be used. A proper candidate could be Bidirectional Failure Detection (BFD) [KW08]. Naturally, the speed of physical detection depends on the hardware configuration, but it takes typically some milliseconds at most. Detecting a failure with BFD takes more time, but it is possible to reach stable failure detection in about 9ms at most [ST08].

Unfortunately, there is no time for advertising the fact that a failure occurred. Since broadcasting a message takes time depending on the size of the network, IPFRR techniques must be able to reroute packets without advertising any information. This means that the rerouting must be done *locally*, so only the routers neighbouring the failed resource change their state, and packets bypass the failed resource based on their local actions.

This means that most of the network do not "know" anything about the failure,

[1]These are the unavoidable tasks, which are needed to be done by any router using arbitrary distributed restoration technique. However, for real routers some additional time is needed (e.g., waiting for timers in order to avoid CPU overload) for completing restoration; more detailed description is presented in [ICM+02, VPD04]

while only IPFRR reroutes the packets. Since other routers need to handle packets on detour differently for providing 100% failure cover, the packet itself must contain some information about the failure, it must be marked somehow. This marking can be implicit or explicit.

Implicit packet marking can be done by using some extra information the router has. Such an information can be the direction, the incoming interface from where the packet has arrived. In contrast, explicit marking modifies the header somehow. The simplest way is to use some bits in the IP header, however, finding free bits in IPv4 header is impossible. On the other hand, it is possible to add a completely new header to the packet by using IP-in-IP tunneling. In this extra header the best place for the marking is the destination address. Special destination address can mark the packet, since this is the field, which is usually processed by a forwarding engine. However, when tunneling is used, special care is needed for the Maximum Transfer Unit (MTU). Since the additional header increases the size of the packet, it is possible that fragmentation will be needed, which should be avoided in some networks.[2]

Note that sometimes, it is needed to send packets back where they came from, in order to bypass a failure. In these cases, packets may visit some nodes more than once. As an illustration, consider the transport network depicted in Figure 1.1 without node g, and suppose that ingress router a got some packets for egress d. Let the default path be the shortest one, namely $a-b-c-d$. When the link between c and d is down, c reroutes packets locally, marks them somehow, sends them back to b and b sends them to a. Thus, all the packets will use path $a-b-c-b-a-f-e-d$ till some restoration technique reconfigures the network. Observe that this is a natural behaviour, which stems from *local* rerouting; till only the neighbours of the failed resource have changed their states, all the packets need to get to the failure. Moreover, observe that packets must be marked, since both a and b must handle packets on detour differently.

As a final task, IP restoration techniques need to compute the new paths with respect to the topology change. In contrast, since IPFRR mechanisms are protection and not restoration techniques, IPFRR is *proactive*. This means that these techniques compute the way of bypassing a failure long before any failure occurs.

Before turning to discuss current IPFRR proposals, it is very important to discuss their typical usage. As it was mentioned above, protection can be considered as a first aid in order to keep up service, but while the service is still available some restoration technique (like OSPF or IS-IS) is needed in order to optimize the routing with respect

[2]Fragmentation doubles the number of packets. This is usually not a problem, it does not mean that the network load is doubled, since routers are usually designed to reach their maximum speed even when 64 byte packets are forwarded, but undoubtedly brings significant extra complexity.

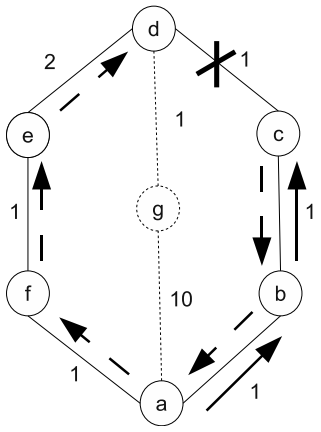


Figure 1.1. Example for local rerouting and FRR loop

to the new situation. In this way, a network using IPFRR should locally reroute packets, when a failure occurs. Since about 50% of failures is transient [ICM+02] (e.g., the layer below IP has its own recovery, router can reboot) and link may return after some time, it is needed to wait at this point. When the failure proved itself to be permanent, the node starts to advertise the fact of the topology change, but till then other nodes are not informed. Recall that the service is still available, so there is no need to expedite the recovery. Naturally, while restoration is not started, the routing is stable; however, conserving this stability during the restoration can become a challenge. Traditionally, routers reconfigure themselves with no sync, which may easily cause short-lived loops called microloops. Since microloops cause some service outage again, restoration must be done with a loop-free manner, which problem is well studied, and several solutions do exist [SB10a]. Thus, microloop prevention is not in the scope of this dissertation. After restoration, the new topology is explored, and there is time for recomputing the protection paths of IPFRR with respect to the new state. Naturally, since downloading alternatives does not change packet forwarding, the system is stable during this operation.

However, there are some IPFRR proposals, which may form loops, if they are not able to handle a failure case (e.g. there are multiple failures). Consider the network depicted in Figure 1.1 again (without node g), and now suppose that not only link $c-d$, but also link $d-e$ is down. Packets would get to node e, as previously, where e must detect that a detour has failed. If e is able to detect this fact, some restoration may be started immediately, which can be as fast as restoration is in current networks, since avoiding microloops is pointless, it does not keep up an already down service. If, however, e is not able to detect that there is more than one failure, it reroutes the

packets again to c along path $e-f-a-b-c$, and a loop is formed[3]. Observe that in contrast to microloops, which are short term results of transient misconfiguration, this kind of loops, named FRR loop in the literature, is a result of inadequate protection capability. Furthermore, observe that FRR loop is not a result of losing the connection with the destination, since the same loop can be formed, if there is a path from a to d through g, since a may select always the shortest detour, which is not the one leading through g[4].

Moreover, observe that FRR loops may have devastating effects. Since an IPFRR technique waits for spontaneous restoration of the failed resource, this waiting time, which can be even in the order of minute, delays service restoration. Furthermore, FRR loops are long term phenomenons thanks to the same waiting, so they can cause significant congestion as well. Thus, if an IPFRR technique is applied, which may create FRR loop, restoration must be started immediately after a failure occurred. This still means that IPFRR restores the connection, however, the fact of the failure is started to be advertised without any additional wait.

Observe that IPFRR techniques, which may form loops, have several disadvantages. First, with respect to the reasoning above, these techniques cannot be used for overcoming transient failures. Thus, in the case of a transient failure, a second reconfiguration is needed, when the resource finally comes back. Second, in order to avoid link flapping, special care is needed: when a resource recovers ("good news") some extra wait is needed before the node could notify other routers in the network. Third, after IPFRR protection was invoked, either microloop-free restoration cannot be provided (which causes short term service disruptions again), or a slower restoration is present, which may be awkward in the case of multiple failures, when IPFRR cannot help and the speed of restoration is critical.

According to the concept discussed above, one may find that the most important aspects defining an IPFRR technique are the ways of solving the problem of local rerouting and proactive computation. Therefore, the discussion of proposals in the next section focuses on these problems.

1.3 IP Fast ReRoute – proposals

Previously, the main concept of IP Fast ReRoute was discussed. It was found that the main aspect identifying an IPFRR technique has two parts, namely the way of

[3]Recall that there is only local rerouting, so both c and e know only the failure of the local link.

[4]Recall that a do not know any failure, it just do, what is dictated by its forwarding table, and this is the optimal behaviour, when there is only a single failure.

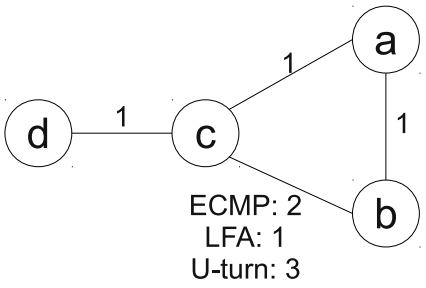


Figure 1.2. Example for ECMP, LFA and U-turn Alternates

local rerouting and the way of marking packets. In this section we briefly discuss current IPFRR solutions in the light of this claim.

1.3.1 Simple techniques with no marking

The first IPFRR techniques try to use the current infrastructure of IP. This means that these techniques do not mark the packets in any way, but simply forward it to an available neighbour. Naturally, this means that there are some failures, which cannot be covered.

The simplest case of this packet rerouting is when multiple shortest paths to the destination exist. Naturally, if there are multiple shortest paths from the node, which cannot forward the packet on the default path, forwarding the packet on the other shortest path solves the problem. One may observe that in the network depicted in Figure 1.2 node b could forward the packet heading to d either to a or to c, if the length of the link between b and c is 2.

Observe that it is already possible to forward packets on this Equal Cost MultiPath (ECMP) [TH00] in IP networks, albeit it is used for dividing the traffic in order to balance the load in the network. Now, the situation is almost the same, except that if one of the paths fails, packets should be forwarded only on the remaining paths.

Naturally, it is quite rare that multiple shortest paths exist, so covering all the failures is not possible in this way. Therefore, a natural generalization of ECMP, called Loop-Free Alternates (LFA) [AZ08] was proposed. Although this generalization still does not provide 100% protection, but it can increase the number of covered failures. The main observation leading to the idea of LFA is that equal cost paths are not necessary for loop-free fast reroute; node a can send a packet with destination d to neighbour b, if a is not on any of the shortest paths from b to d (Figure 1.2, length of $b - c$ is 1).

Unfortunately, only using this simple idea may produce FRR loops in special cases. As it was presented in [AZ08], multiple failures or a single node failure, if protection was computed for link failure, can cause loops. When loops must always be avoided, only the neighbours strictly closer can be used for rerouting (these paths are called "downstream" paths). Observe that there is a trade-off: in this way loops can be avoided, but the number of protected failures is decreased. Back to our example, suppose that either both link $b-c$ and link $a-c$ go down (multiple failures) or node c is failed. In this case, applying LFA for both a and b would cause forwarding loops; since c is unavailable, both a and b would try to reroute using its LFA. On the other hand, using only the downstream paths would mean that there would be no LFA, neither for a nor for b.

As these techniques cannot cover all the possible failures, a very important question is their efficiency, which was studied in [Gjo07, FB05]. According to these works, ECMP gives very limited protection, at most 30% of the potential single failure cases can be covered in very special networks, and LFA has usually about 50-80% coverage with respect to the network topology and the type of protected failures (link or node). According to these results, despite the simplicity of these mechanisms, it is possible to bypass most of the failures with applying only LFA. However, it can be observed that further techniques are needed for covering the remaining cases.

1.3.2 Techniques based on incoming interface

As it was discussed previously, IPFRR mechanisms need to mark packets in order to cover all the single failure cases. Techniques in this part use implicit marking, and benefit from the extra information supplied by the incoming interface. Forwarding, which takes both the incoming interface and the destination address into consideration, is known as interface-based forwarding.

The first technique, which uses this idea is U-turn Alternates [Atl06]. U-turn Alternates is some extension of LFA. A given neighbour b of a can be used as a U-turn alternate with respect to destination d, if there is a loop-free alternate from b to d avoiding a, and a is the next hop on one of the shortest paths from b to d. In order to keep this definition simple, one may consider U-turn Alternates as a possibility to send a packet back one hop, if there is an LFA from that neighbour. In the network depicted in Figure 1.2 (length of $b-c$ is 3) b should recognize that the packet heading to d was received from a. In this case, it should be forwarded to c.

Naturally, U-turn raises the problem of identifying the traffic sent back from the next hop. According to [Atl06], this can be done by marking the packet somehow or

by identifying the incoming interface. Since, as it was discussed previously, finding extra bits in IPv4 header for this purpose is impossible, the later possibility is the realizable one. This means that U-turn needs a forwarding, which depends on the incoming interface.

Observe that U-turn alternate neighbours not necessarily exist, so even this solution gives only cover for a part of the failures. On the other hand, by using LFA and U-turn together, it is possible to protect a very significant part of the resources. According to [Gjo07, FB05] it is quite common that 90% of the failures can be covered.

The concept of using the extra information of the incoming interface can be generalized. The main idea of Failure Inferencing based Fast Rerouting (FIFR) [ZNY+05, NLYZ03, NLY+07, LYN+04, WN07][5] is that not only the next hop can indicate a failure by sending a packets back, but basically any node. Here, it is not some special bits in the IP header that mark the packet as being on detour, but rather the fact that it has been received on an interface usually not applied in failure-free case. As it was proven in [NLY+07, ZNY+05], it is possible to bypass any single link or node failure using this simple idea.

Unfortunately, there are some problems with techniques using interface-based forwarding. As it is discussed in Chapter 2, they are prone to form loops in the case of a failure they have not prepared for. Namely, the version capable to bypass single link failure may form loops in the case of multiple link failures or single node failure and the version capable to bypass single node failure may form loops in the case of multiple node failures.

In order to overcome loops, I have proposed a new interface-based IPFRR mechanism named Loop-free Failure Insensitive Routing (LFIR) [C2, C3]. LFIR is able to bypass any single link failure. Moreover, it can never create loops, albeit it uses interface based forwarding. However, there is a trade-off: LFIR does not always use the shortest paths, when the network is intact, but these paths are only slightly longer than the shortest ones. Further details are discussed in Chapter 2.

One important issue remained unanswered: the implementation impact of interface-base forwarding. Theoretically, realizing this forwarding scheme is possible with slight modification of current router architectures. In modern routers there are linecards at each interface. Due to speed issues, each linecard has its own memory, where the forwarding information is downloaded. If the same information is downloaded, we get the traditional IP forwarding. If there is different information, interface-based forwarding is realized.

[5] A version of FIFR is also called as Failure Insensitive Routing (FIR) in Chapter 2.

Unfortunately, there are some difficulties with this principle, albeit it is undoubtedly realizable. Since usually more than one interface belongs to a given linecard, simply downloading different forwarding information base cannot provide forwarding, which fully depends on the incoming interface. The processor of a linecard could take care of the incoming interface, but it would need extra effort. Moreover, changing the forwarding is not simple either. Since traditional IP forwarding was supposed during each phase of the development of a router, changing this scheme necessarily raises serious implementation problems too.

1.3.3 Techniques using tunneled detours

Marking packets with additional IP header is popular in the field of IPFRR, since finding extra bits in the header is very difficult. In this section we discuss the techniques using this additional header of an IP-in-IP tunnel. Mechanisms using multiple different routing configurations are discussed in the next section.

First, IPFRR tunnels [BFPS05] were proposed. This technique is based on the idea that node s can locally reroute, if there is a node a, reachable using the normal forwarding even after a failure, with a path or at least an LFA to d, which bypasses the failed resource. If there is such node, then push the packet into an IP-in-IP tunnel with address of a.

Unfortunately, there are several serious problems with this scheme. First, it needs a mechanism called "directed forwarding", which means that s can force a to select LFA instead of the shortest path. Unfortunately, it is not clear, which mechanism could provide directed forwarding in IP networks. Moreover, even with directed forwarding, it is not always possible to bypass a failed node, so IPFRR tunnels are just another partial solution, although it can protect numerous failures (according to [Gjo07, FB05]: all link failures can be protected, but bypassing nodes has about 60%-80% chance).

An example is depicted in Figure 1.3. Suppose that the destination is node d and the link between node e and node d is down. In this case node e could put the packet into an IP-in-IP tunnel in order to send it to b. Now, the packet is decapsulated, and directed forwarding tells b to send the packet to c and in this way it reaches d.

100% failure coverage can be reached by Not-via [BSP10]. The idea behind Not-via is to encode in the outer IP address of the IP-in-IP tunnel not only the endpoint of the tunnel, but also the identifier of the failed resource. Since in Chapter 5 this technique is improved, a more detailed description can be found there, and here I give only a brief picture.

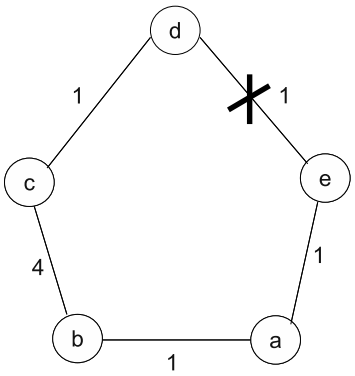


Figure 1.3. Example for IP tunnels

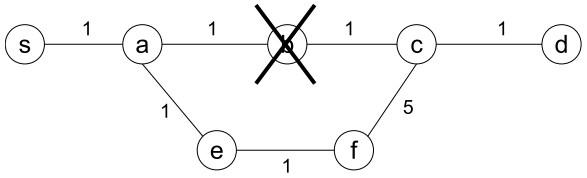


Figure 1.4. Example for Not-via

In order to understand the way Not-via handles a failure, suppose that node a cannot forward the packet (Figure 1.4), heading to destination d, to node b. Node a assumes node failure (Not-via always assumes node failure, if it is possible to reroute without the next hop, since in this way link failures are also handled), and selects its next-next hop, the next hop of b, let it be c. Now, a encapsulates the packet into an IP-in-IP tunnel, selects a destination address for the outer header with the meaning "forward the packet to c, but not-via b" and forwards the packet to e. Although both the shortest paths from e and from f are through a, packets do not return to a, thanks to the special address. In this way the packet reaches the next-next hop c, where it is decapsulated, and the packet can reach the destination using the default routing.

Not-via computes a detour for each possibly failing resource. However, there are other possibilities using "redundant trees". The first technique, which used redundant trees [IR84, MBFG99] for rerouting in IP networks is IP Redundant Trees (IPRT) [CHA07]. A pair of redundant trees is a pair of directed spanning trees of an undirected graph with a common root vertex, where the root can be reached on both trees, but the two paths on the two trees are node-disjoint. Redundant trees are well studied in this dissertation, for further details reader is referred to Chapter 3

and Chapter 4.

IPRT computes a pair of redundant trees rooted at each node. If there is no failure in the network, the shortest paths can be used as usual. On the other hand, if there is a failure, one of the redundant trees rooted at the destination is used, the one which bypasses the failure.

IPRT has some desirable attribute in contrast to Not-via. If it is implemented using tunneling, each node requires only 3 IP addresses. On the other hand, the number of IP addresses needed by Not-via scales quadratic with the number of nodes in Local Area Networks (LAN). Unfortunately, redundant trees can be found only in 2-vertex-connected networks, which criterion cannot always be fulfilled by real networks (see e.g., Abeline, AT&T in [SND] or Italian backbone in [GO05]). Moreover, even if a network is 2-vertex-connected, it can easily lose this property, when a failure occurs. Therefore, redundant trees are needed to be improved.

In order to always provide the maximum possible redundancy, I introduced the concept of maximally redundant trees [C7, J4]. The first technique, which used maximally redundant trees for IPFRR is Lightweight Not-via [C5, C6, J4, P2]. Moreover, this technique uses a special algorithm for computing maximally redundant trees in a distributed way with significantly decreased computational complexity. Furthermore, Lightweight Not-via can completely avoid the use of extra IP addresses in several IP networks, thanks to utilizing interface addresses. Finally, as Lightweight Not-via is an improved version of Not-via, which uses the next-next hop as the endpoint of the tunnel, the suboptimal repairing paths are usually shorter. Further details of Lightweight Not-via are discussed in Chapter 5.

There is another IPFRR proposal [KRKH09] on the traces of Not-via using redundant trees. This technique can cover any two link failures but no node failure. It uses 4 IP addresses – one for default forwarding and 3 for the additional 3 detours. Unfortunately, simultaneous link failures are uncommon and this mechanism cannot provide node protection. Moreover, 3-edge-connected network is needed, which criterion can rarely be fulfilled. The technique has two versions called Red Tree First (RTF) and Shortest Tree First (STF). STF finds shorter paths, but can form loops in case of 3 simultaneous link failures or in the case of a single node failure.

1.3.4 Multiple Routing Configurations

In this section, I introduce Multiple Routing Configurations [KHC+06, KHv+09] and relaxed Multiple Routing Configurations [KHC+08, CHK+10]. These techniques use essentially the same concept for bypassing a failed resource.

The main idea of these techniques is creating multiple link length configurations. Naturally, since the shortest paths differs, in this way multiple routings are produced. If there is a configuration for each resource, where shortest paths do not contain that resource, switching among these configurations can provide protection. The configuration, the packet is needed to be forwarded on, is selected by either some bits in the IP header, or by the destination address in the same way as Not-via does. When a node fails to forward the packet on the shortest path, it simply switches to a topology (e.g., puts the packet into an IP-in-IP tunnel with a special destination address), where the next hop is not on the shortest path. The difference between MRC and rMRC is that relaxed MRC needs less configurations to cover all the resources.

It is easy to observe that the number of required routing configurations is a weak point of these techniques. Moreover, the most important problem is the almost complete lack of upper limit for this number. As it was presented in [Cic06], the number of needed configurations is less than the number of nodes in the *largest minimal cycle* of the graph of the network. The minimal cycle of an edge is the smallest cycle containing the edge; the largest minimal cycle is the largest among the minimal cycles for all the edges. This upper bound is strict, since it is always reached by a network with ring topology. Unfortunately, this means that the number of topologies can be equal even with the number of nodes. Since each configuration needs an extra IP address for *each of the nodes*, the high number of configurations means that the number of IP addresses in the network can scale quadratic with the number of nodes in the network in the worst case.

On the other hand, authors have shown that the number of topologies needed is usually between 2 and 7, so much less than the number of nodes in most of the networks. Unfortunately, if it is needed to mark packets using destination addresses, even this result means that each node needs 2 to 7 *extra* IP addresses, which is much higher than the number of IP addresses needed by e.g., Lightweight Not-via (it needs only 2).

1.3.5 Rerouting multicast packets

Although protection of multicast IP traffic is usually considered less important, recently this question was also studied. Currently, there is only one solution, known by the author, proposed in [LLW+09], which provides multicast fast reroute in case of single link failures. The main idea is based on the special way of path computation using Protocol Independent Multicast (PIM) [FHHK06]. Multicast trees built by PIM use the paths, which would be the shortest ones from the *destination* to the

source[6], while unicast traffic is forwarded differently on the shortest path from the *source* to the *destination*. Since setting asymmetric link costs is possible both by OSPF and IS-IS, it is possible to route multicast and unicast traffic on completely different paths. When a given link fails packet is encapsulated to an unicast IP-in-IP tunnel, and sent to the other side of the link. Although the authors of [LLW+09] did not recognize, they computed redundant trees by applying a version of the algorithm presented in [MBFG99].

1.4 Research Objectives

Previously, we have discussed the main concept of IPFRR and current techniques were briefly reviewed. In this section, my research objectives are introduced. As it was observed, almost all the previously discussed techniques suffer some serious shortcomings. In order to overcome these drawbacks and make better IPFRR mechanism, first we discuss the requirements a modern IPFRR technique must meet.

First, recall the basic requirements, discussed in Section 1.2, every IPFRR technique must fulfil: such a mechanism is needed to be applicable inside a single autonomous system, traditional IP forwarding can be only slightly modified, recovery time must be in at most 50ms and as many single failure cases must be protected as it is possible with best effort congestion mitigation.

Since the most important goal of rerouting is rebuilding the connection, we can immediately extend the last requirement and say that a modern IPFRR technique needs to provide 100% protection against single failure cases, which do not partition the network into two. This means that such a technique necessarily marks packets on detour either implicitly or explicitly.

Moreover, a proper IPFRR technique never makes a situation worse. This means that a modern IPFRR technique must never create FRR loops. In this way, overcoming the transient failures using fast reroute is possible.

Furthermore, IPFRR techniques must not increase the overhead significantly. Naturally, IPFRR always adds some complexity to routing, but this additional complexity must be as low as it is possible, and it must scale well with the size of the network. The requirement of keeping the additional complexity low applies to all kinds of complexity including e.g., the management (managing extra IP addresses).

[6]PIM builds the multicast tree using messages, which are sent to the source on the shortest path by the destination; this is the "reverse" shortest path of the source. Naturally, this is suboptimal, if the link lengths are asymmetric.

In this way my research objective is creating IPFRR techniques, which can provide fast reroute for unicast packets in the case of any single failure. It must be able to always avoid forming FRR loops and it needs very moderate additional computational and management complexity. As one may observe, none of the techniques fulfil these requirements. Mechanisms using interface-based forwarding, except LFIR (Chapter 2), are immediately ruled out, since they are prone to create FRR loops. Not-via and MRC need too much management overhead thanks to the high number of additional IP addresses. IPRT is not able to deal with non-2-vertex-connected networks, the technique presented in [KRKH09] cannot handle node failures. The last remaining technique, Lightweight Not-via, is discussed in Chapter 5.

As it was already discussed, after fast rerouting, there must be some restoration technique, which reconfigures the network with respect to the new topology. As it was mentioned in Section 1.2, this is a quite well solved problem (further details can be found in [SB10a]), thus they are not among my research objectives. Moreover, although the importance of multicast traffic is improving with the spreading of IPTV, I deal with unicast traffic, which is far the most important currently. Multicast IPFRR is out of the scope of this dissertation.

1.5 General Assumptions

In this dissertation, I deal with IP networks. Although there are IP networks, where the forwarding is based on much more information, I suppose that the forwarding engine determines the next hop based on only the destination address contained by the packet. No other information (e.g., source address) can be taken into consideration, albeit a router may have multiple forwarding engines, even one for each interface, and each of them can be configured in different ways (interface-based forwarding).

Furthermore, I suppose that the topology of the network is explored. This means that there is some routing protocol in the background, like OSPF or IS-IS, which does this task. Moreover, I suppose that the network is connected, since an unconnected network can be considered as some connected networks. Therefore, I suppose that the graphs of networks used by the algorithms in this dissertation are always connected.

Moreover, since the traffic transported by current IP networks is almost exclusively unicast, I suppose that only unicast traffic is needed to be forwarded.

Since IPFRR is applied inside autonomous systems, I suppose that only paths towards interior destinations must be protected. This assumption is very realistic, since in several routers outer prefixes are resolved by a recursive lookup, so first only

the egress router is found and the next hop is calculated by a second lookup based on this information; hence IPFRR protecting interior paths protects outer prefixes as well. Moreover, even if there is no recursive lookup, routing can be originated in the same problem by considering outer IP addresses as addresses of egress routers. In this dissertation, I do not deal with the case, when a sole prefix can be reached through multiple egress routers.

As it was discussed previously, fast rerouting techniques need to prepare to failures before they actually occur. Since preparing to handle arbitrary number of simultaneous failures is next to impossible, IP fast reroute techniques prepare to bypass only single link or node failures. Naturally, this seems like an artificial assumption at first, since sooner or later another resource will fail. Fortunately, using some restoration technique, IPFRR can prepare to a new failure after the network was reconfigured, as it was discussed in Section 1.2. Therefore, I suppose that failures only happen one by one in normal operation, and although multiple failures can occur, they are very rare.

My graph algorithms commonly assign some values to the vertices and edges of some graph. In these cases, I always assume that getting these values can be done in $O(1)$ time, when the corresponding vertex/edge is given; in this way e.g., the length or the endpoints of an edge can be reached rapidly. Moreover, I also assume that these values can even be pointers to some linked lists, thus it is easy to enumerate e.g., the edges connected to a given vertex, or the children of a vertex in a tree. Finally, I assume that there are two linked lists for each graph, one contains all the vertices and the other contains all the edges.

1.6 Notations

In the sequel, graphs are commonly dealt with, which are usually simple graphs. A simple graph G is a pair (V, E), where V is the set of vertices and E is the set of edges. If graph G is undirected, then $E \subseteq \{\{v_1, v_2\} : v_1, v_2 \in V, v_1 \neq v_2\}$, so elements are unordered pairs, denoted by $\{v_1, v_2\}$ $(v_1, v_2 \in V)$. Otherwise, if G is directed, $E \subseteq V \times V \setminus \{(v, v) : v \in V\}$ ($\times$ denotes the Cartesian product), so elements are ordered pairs, denoted by (v_1, v_2) $(v_1, v_2 \in V)$, where v_1 is the source and v_2 is the target. Moreover, $V(G)$ and $E(G)$ denotes the set of vertices and edges of graph G. The number of elements (cardinality) of a given set S is denoted by $|S|$.

In Section 2.3.2, I use graphs with multiple edges. Therefore, simple graphs are generalized to multigraphs. The definitions above still hold for multigraphs as well,

expect for E. The set of edges, is not a simple set anymore, but a multiset. The multiset is a set, which can contain the same element multiple times. Formally defined, a multiset is a pair (A, m), where A is some set and $m : A \to \mathbb{Z}^+$, where $\mathbb{Z}^+$ is the set of positive integers; function m denotes the multiplicity of an element. In this way, the formal definition of E: $E = (E', f)$ where $E' \subseteq \{\{v_1, v_2\} : v_1, v_2 \in V\}$ or $E' \subseteq V \times V$ for undirected or digraphs respectively and $f : E' \to \mathbb{Z}^+$. Naturally, for multigraphs the number of edges is $|E| = \sum_{\forall e \in E'} f(e)$.

A graph is connected, if there is a (directed) path from any $u \in V(G)$ to any $v \in V(G)$. Connected directed graphs are also referred as strongly connected graphs. In contrast, a digraph is weakly connected, if replacing its directed edges with undirected ones produces a connected undirected graph. A graph is n-edge-connected or n-vertex-connected, if after removing any $n - 1$ edges or vertices respectively, the remaining graph is connected. A digraph is weakly n-edge-connected or n-vertex-connected, if after removing any $n - 1$ edges or vertices respectively, the remaining graph is weakly connected. Let $v \in V(G)$ and $e \in E(G)$. Vertex v is a cut-vertex, if without v the graph is not connected and edge e is a cut-edge, if without e the graph is not connected. Vertex v is a weak cut-vertex, if without v digraph G is not weakly connected and edge e is a weak cut-edge, if without e digraph G is not weakly connected. Observe that the two endpoints of a (weak) cut-edge are (weak) cut-vertices.

In this dissertation, directed spanning trees with a given root vertex, commonly denoted by r, are often dealt with. Therefore, it is essential to define some notations in connection with these trees. The *parent* of a vertex is the neighbour on the path towards r (even if this path is not a directed one). The *children* are the neighbours, which are not the single parent. The *ancestors* of a given vertex v are the vertices along the path from v to r. The *successors* of v are the vertices, which have v as an ancestor. Finally, the term *walking up* along a tree means walking towards r. Similarly, *walking down* denotes the opposite direction.

Since in this dissertation numerous algorithms are presented, it is needed to deal with their complexity. For upper approximation notation $f(x) = O(g(x)) \iff \exists M \in \mathbb{R}^+, \limsup_{x\to\infty} \frac{|f(x)|}{|g(x)|} \leq M$, for lower approximation $f(x) = \Omega(g(x)) \iff \exists M \in \mathbb{R}^+, \liminf_{x\to\infty} \frac{|f(x)|}{|g(x)|} \geq M$ is used, where $\mathbb{R}^+$ is the set of positive real numbers.

A brief enumeration of further notations used in this dissertation is presented in Table 1.1. Further details and exact definitions are presented before the first use of these notations.

Notation	**Comment**
$V(G)$	Set of vertices of graph G
$E(G)$	Set (or multiset) of edges of graph G
(a, b)	An edge of a digraph (ordered pair of vertices); a is the source, b is the target
$\{a, b\}$	An edge of an undirected graph (unordered pair of vertices)
$\|S\|$	Number of elements of set S
D_n	DFS number of vertex n
L_n	Lowpoint number of vertex n
$v(n)$	Voltage of vertex n
$h_u^P(d)$	Edge going out from u belonging to the primary (maximally) redundant tree rooted at d
$h_u^S(d)$	Edge going out from u belonging to the secondary (maximally) redundant tree rooted at d
r	Root of an ADAG, or global root of a GADAG
r_x	Local root of vertex x
r_A	Local root of cluster A
C	Set of clusters
V_u^+	Set of vertices greater than vertex u
V_u^-	Set of vertices less than vertex u
$\mathcal{D}_v$	Default address of node v
$\mathcal{P}_v$	Primary detour address of node v
$\mathcal{S}_v$	Secondary detour address of node v
$\text{nh}(\mathcal{A})$	Next hop node towards address $\mathcal{A}$
$\text{nnh}(\mathcal{A})$	Next-next hop node towards address $\mathcal{A}$

Table 1.1. Common notations used in this dissertation

Chapter 2

Loop-free Interface-based routing

2.1 Introduction

It was discussed previously that each fast rerouting technique, which provides 100% cover for single failure cases, needs to mark the packets on detours. Packets can be marked explicitly (using some bits or tunneling), or implicitly by using the extra information of incoming interface. In this chapter, I deal with the latter concept, with IPFRR techniques using interface-based forwarding.

The main idea behind these techniques is that a failure must exist, if a packet arrives through an uncommon interface. In this case, it is possible to compute the possibly failed resources, and forward the packet to the destination on a path, which does not include them.

For realizing this concept *interface-based forwarding* is needed. Interface-based forwarding is an extension of IP forwarding. Traditionally, IP forwarding uses the destination address for determining the next hop. In contrast, if a router uses interface-based forwarding, then not only the destination address, but also the incoming interface is taken into consideration.

It is possible to realize interface-based forwarding with only moderate modification on modern router architectures. In modern routers, there are linecards at each interface, determining the outgoing interface of the incoming packets. For performance issues there is dedicated memory at each linecard, where the forwarding table is downloaded. If the same forwarding table is downloaded to each linecard, traditional IP forwarding is realized. On the other hand, if different forwarding tables are download, exactly the same hardware can realize interface-based forwarding.

The most important problems of this way of implementation were already mentioned in the previous chapter, namely that a linecard may have multiple interfaces and the serious implementation problems stemming from changing IP forwarding. However, taking everything into consideration, interface-based forwarding can be realized on current hardware with no doubt, albeit it is not easy.

As it was mentioned in Section 1.3.2, the first algorithm, which used the extra information of incoming interface is the U-turn Alternates [Atl06], which gives the possibility to a node a to send packets one hop back to a neighbour b with a as a default next hop and with a Loop-free Alternate [AZ08] to the destination. Unfortunately, one hop detours cannot provide 100% failure cover. Therefore, the concept of detecting the packet flight was generalized, so that detour can be longer and packets on detour may arrive on any uncommon interface. The first IPFRR mechanism, which used this generalized scheme was the *Failure Insensitive Routing (FIR)* [NLYZ03, LYN+04, NLY+07]. This technique can always bypass a single failed link, which is the most common type of failures [ICM+02, MIB+04]. Later, this technique was improved to *Failure Inferencing based Fast Rerouting (FIFR)* [ZNY+05] capable to reroute packets even in the case of a single node failure.[1] Unfortunately, FIFR needs 2-node-connected networks, thus cannot cover failures, when only a link of a cut-node fails (since it always supposes node failure, which would cut the network into two). Therefore, the two techniques were combined in [WN07], making FIFR capable to protect any resource which can be bypassed.

FIFR has several advantages. First, with interface-based forwarding, it is possible to provide IPFRR without changing IP itself, using extra addresses or dealing with the extra load and packet fragmentation of tunneling. Second, if all the interfaces have their own forwarding information bases, interface-based forwarding brings no overhead and easily realizable with current hardware. With considering linecards with several interfaces, the situation is a bit more complicated, but it is still undoubtedly realizable with updating only the software of these linecards.

Unfortunately, there is a significant drawback as well: FIR and FIFR may create FRR loops in case of multiple failures. Avoiding loops is an important task of fast rerouting algorithms. As it was discussed in Section 1.4, one of my main goals is to create IPFRR mechanism always capable to avoid loops.

[1] Although the authors later renamed FIR to FIFR_L, I refer on it as FIR in this dissertation. Thus, FIFR is the algorithm sometimes referred as FIFR_N in the literature. Moreover, there is an improved version of FIFR presented in [WN07]; I make it always clear, when I deal with this special version.

The authors of FIR and FIFR also recognized the problem of loops and proposed *Blacklist-based Interface-Specific Forwarding* (BIFS) [WZN06]. Unfortunately, this solution is inapplicable in backbone networks, because of not only the significant modification of IP forwarding, but also the overhead stemming from keeping up blacklists.

In this chapter, first, I show that the possibility of loops when multiple failures exist is not the result of some design problem, but a natural behavior of IPFRR methods using interface-based forwarding. Understanding the limitations makes possible to propose a new IPFRR technique, which can overcome this problem of FIR and FIFR; *Loop-free Failure Insensitive Routing* (LFIR) [C2, C3, P1] is capable to always handle any single link failure, and it can never create loops.[2] Moreover, since LFIR needs only interface-based forwarding, it can be applied in backbone networks. Unfortunately, as it turns out in Section 2.2, there is a trade-off: in some cases LFIR does not forward the packets on the shortest paths when the network is intact. However, as it is proven in Section 2.5 by extensive simulations, these default paths are only slightly longer than the shortest ones.

2.2 Loops using FIR and FIFR

It was mentioned previously and proved in [NLYZ03, ZNY+05] that it is possible to correct one link failure with FIR or a node failure with FIFR in a network using interface-based forwarding. First, we recall these algorithms and I prove that both can make loops in the case of multiple failures.

The base idea of FIR and FIFR is simple: if a node gets a packet from a neighbour which usually does not use this direction for forwarding, then there is a failure in the network. FIR calculates which links, called keylinks, could have been failed and a path to the destination without all these links. Using this information, FIR precomputes an alternative route for each incoming interface, which guarantees that the failed link will be bypassed. Similarly, FIFR computes the possibly failed nodes, called keynodes, and bypasses them. It is important that both techniques use shortest paths as default.

Perhaps a simple example is in order. Consider the network in Figure 2.1, and suppose that the lengths of links are uniformly 1 and FIR is applied in this network.

[2]Naturally, as all the other IPFRR techniques, LFIR prepares to protect against single failure cases. When, however, multiple failures occur, LFIR is capable to detect the situation, and to drop packets instead sending them into forwarding loops. Furthermore, detecting that there are multiple failures makes it possible to start restoration technique immediately, and to reach as fast restoration, as it would be possible without IPFRR.

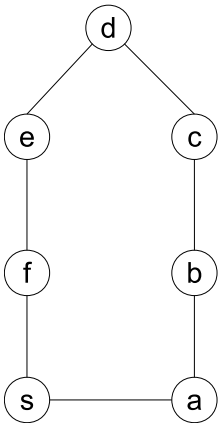


Figure 2.1. A network with ring topology

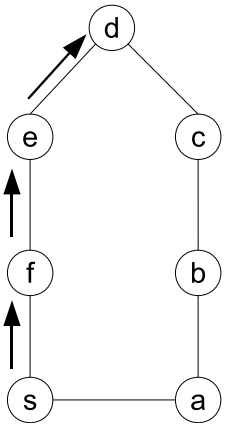


Figure 2.2. Default forwarding in the ring from s to d

If there is no failure the shortest path, depicted in Figure 2.2, is used from s to d. If link $\{e, d\}$ fails, node e locally reroutes the packet to d on the shortest path without $\{e, d\}$, and the packet is sent back to f. There is only one keylink for destination d and directed edge (e, f), namely link $\{e, d\}$, so the packet is forwarded to s on the shortest path to d without $\{e, d\}$. Similarly, the keylinks for destination d and directed edge (f, s) are $\{f, e\}$ and $\{e, d\}$. Node s forwards the packet to a on the shortest path without these two links. Finally, a has no keylink for (s, a) and destination d, so the packet is forwarded on the shortest path (depicted in Figure 2.3).

Similarly, FIFR computes the keynodes for each destination and incoming interface. When node e fails (Figure 2.5), node f reroutes the packet, and sends it back to node s. Since s computed the keynodes for (f, s) and destination d, which consist of node e, it forwards the packet to a, which is the next hop along the shortest path without e. Node a computes no keynodes, so it forwards the packets on the shortest path to d.

Although this method is very effective, it has a serious disadvantage. Theorem 2.2.1 demonstrates this disadvantage. In the sequel I call a routing *optimal*, if packets are always forwarded along shortest paths in an intact network. I call a routing *loop-free*, if there is no loop even if any subset of links and/or nodes is failed. A routing is *link failure insensitive* or *node failure insensitive*, if traffic can pass between all node pairs even if one link or one node is down respectively. A rerouting is *local*, if only the neighbours of the failed resource have information about the failure.

Theorem 2.2.1. *Suppose that packets are forwarded based on only the destination address and the interface they arrived through. In this case, there are some networks*

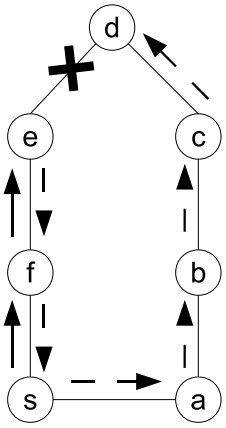


Figure 2.3. Forwarding if the link between node d and node e is down

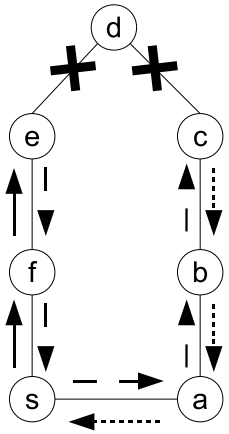


Figure 2.4. Loop if the link between node d node e and node c node d are both down

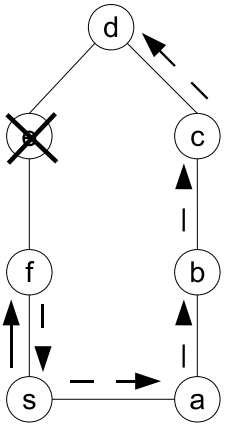


Figure 2.5. Forwarding if node e is down

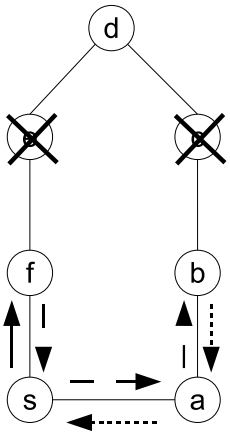


Figure 2.6. Loop if both node c and e are down

where no optimal, loop-free, link failure insensitive, interface-based local IP protection technique exists. Moreover, there are networks where no optimal, loop-free, node failure insensitive, interface-based local IP protection exists.

Proof. Consider the network depicted in Figure 2.1 and let all the lengths of the links be 1. Indirectly suppose that an optimal, loop-free, link failure insensitive, interface-based local IP protection technique is applied in this network. Hence, if there is no failure the shortest path is used (since the routing is optimal) and node s will send its packets to node d on path $s-f-e-d$ (Figure 2.2). If link $\{e,d\}$ becomes unavailable packets will be able to reach node d because the routing is link failure insensitive. Since we have local rerouting in the network, node e has no other choice than sending packets back to node f which is its only reachable neighbour, so packets will follow path $e-f-s-a-b-c-d$ (Figure 2.3). One may observe that node f and s does not send packets back to node e because interface-based forwarding is used. If link $\{c,d\}$ is also unavailable, a loop will be formed. Node c gets a packet from node b, which has c as a default next hop, so c has no information about the previous failure. It tries to bypass link $\{c,d\}$ – node c "thinks" that this is the first unavailable link – and in this way a loop is formed (Figure 2.4), which contradicts the assumption that this routing is loop-free.

Similarly, if the rerouting is node failure insensitive, the failure of node e inducts rerouting at node f, and packets reach d as depicted in Figure 2.5. If node c fails too, node b got the packet from a, which has b as a default next hop towards d. In this way, b has no information about the first failure, so it reroutes the packet to node a and a loop is formed as depicted in Figure 2.6. □

Because FIR is interface-based, optimal, link failure insensitive local rerouting mechanism, Theorem 2.2.1 proves, that there are some networks where using FIR can cause loops. Section 2.5 shows that these networks are not rare; FIR can cause loop in numerous networks. Moreover, since FIFR is interface-based, optimal, node failure insensitive local rerouting mechanism, FIFR can create loops too. Observe that FIFR is still node failure insensitive even if it is improved to cover both link and node failures at the same time [WN07].

Furthermore, although in the proof of Theorem 2.2.1 multiple failures were supposed, it is possible to create loops when the rerouting is capable to bypass link failures in 2-edge-connected networks, and a single node fails. Consider the network depicted in Figure 2.7 first without g, and suppose that node s tries to send packets to d. If all the length of links are uniformly one, the shortest path is $s-f-c-d$. Now, assume that f cannot forward the packet to c. Usually, f cannot decide, whether c or

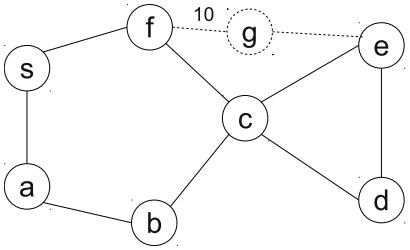


Figure 2.7. Example network for FIFR loops

the link $\{f, c\}$ is down. If an interface-based technique tries to bypass the failed link, it may create loop, if node c is the failed resource. On the other hand, it is possible to suppose that c has failed and avoid loops, but then the failure cannot be bypassed if only the link was down. In this way, techniques supposing link failures, like U-turn alternates, FIR and the improved version of FIFR presented in [WN07], can create loops even if only a single node is down.

Moreover, observe that adding g (the length of $\{f, g\}$ 10 others are 1) would not help either for U-turn alternates, or for FIR. Since the shortest detour is not through g, they would form the same loop, even if a path to the destination does exist.

2.3 Loop-Free Failure Insensitive Routing

It was shown in the previous section that both FIR and FIFR can create loops. With respect to my research objectives (Section 1.4), in this section I propose the Loop-Free Failure Insensitive Routing (LFIR) [C2, C3, P1], an IPFRR mechanism using interface-based forwarding, which always avoids loops. LFIR can always bypass a single failed link like FIR does. Naturally, according to Theorem 2.2.1, there must be a trade-off: for the price of always avoiding loops, LFIR is not an optimal routing. However, as it turns out in Section 2.5, the default paths are only slightly longer than the shortest ones. First, I study 2-edge-connected networks, then we lift this assumption.

2.3.1 2-edge-connected networks

The basic idea of LFIR is to find paths from each node to each destination in such a way that when a node gets a packet from a specific incoming interface, it can always

decide if either the default path was used or the packet is on a detour due to a failed link. If the detour has also failed, the packet must be dropped, and other nodes must be immediately informed about the need of falling back to restoration. In order to realize this scheme, we must recall a version of a theorem of Edmonds [Edm73].

Definition 2.3.1. A *branching* rooted at vertex r is a directed spanning tree with edges directed in such a way, that each vertex $x \neq r$ has one edge going out.

Remark: Note that branchings are usually defined in the reverse direction in the literature. Moreover, observe that edges in these branchings are directed so that the root can be reached along a directed path from any other vertex.

Proposition 2.3.1 (Edmonds)**.** *Let G be a digraph, which is n-edge-connected. It is possible to find n edge-disjoint branchings in this graph rooted at any $r \in V(G)$.*

One may observe that a branching is something like routing; if a packet can follow the directed edges of a branching, it eventually reaches the destination without loops. The only difficulty is that typically links can be used in both directions, so networks can be modeled by an undirected graph.

It is possible to solve this problem and find decent branchings in the undirected graph. Let G be the undirected 2-edge-connected graph of a network. Let G' be a directed graph, such that $V(G) = V(G')$ and if $\{i, j\} \in E(G)$, then both $(i, j) \in E(G')$ and $(j, i) \in E(G')$. Trivially, G' is also 2-edge-connected.

Now, the version of LFIR for 2-edge-connected networks is the following. Convert the undirected graph G to a digraph G', find two edge disjoint branchings – a red and a blue one – in G' rooted at each vertex. When a packet arrives on a branching, forward it on the same one, if it is possible – there is exactly one edge going out from each node belonging to that branching. If it is not possible (e.g., due to a link failure) and the packet used the red branching, try to forward it on the blue one. If it used the blue one, drop the packet and start restoration immediately. By default (at the source node), the packet is on the red branching. The branching is selected by the destination address and the incoming interface together; a pair of branchings belongs to each IP address (the branchings rooted at the destination) and the incoming direction selects the branching for forwarding, since they are edge-disjoint.

Back to our example, the two branchings of the previous network are depicted in Figure 2.8. Naturally, in a real network two branchings would be computed to each destination, not only to d, but a packet heading to d will use only these branchings. If there is no failure packets sent by s to d reach the destination on path $s - f - e - d$. If $\{e, d\}$ fails, e changes the branching, and sends packets back to f. Since (e, f) is

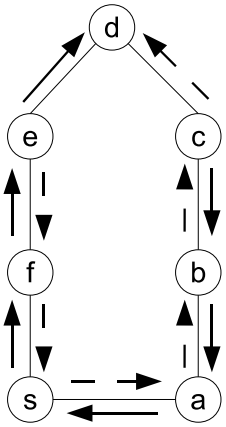


Figure 2.8. The two branchings of LFIR rooted at d (red – solid arrow, blue – dashed arrow

in the blue branching of d, f forwards the packets on this branching to s. Following the same branching d is reached, if there is no more failure. Observe that the path of packets is now $s-f-e-f-s-a-b-c-d$, since packets are locally rerouted. Moreover, if $\{c,d\}$ fails too, c detects that packets arrived on the blue branching cannot be forwarded on the same one, so drops them and starts informing other nodes about the immediate need of restoration.

However, observe the drawback of this technique: if the source is e.g., node a, the default path is still the red branching, which means that packets would be forwarded on path $a-s-f-e-d$, which is not the shortest one.

The next theorem shows that packets always reach the destination if at most one link is down and loops can never be created.

Theorem 2.3.2. *The version of LFIR used in 2-edge-connected networks is correct (it never creates forwarding loop) and complete (packets arrive if at most one link is down).*

Proof. It is easy to see that packets can travel on each link at most twice – once using the red branching and once using the blue branching –, so there cannot be a forwarding loop. It is also easy to see that packets arrive along the red branching, if all the links are available.

Now suppose that exactly one link, $\{i,j\} \in E(G)$ is failed. Naturally $(i,j) \in E(G')$ and $(j,i) \in E(G')$ and these two edges cannot belong to the same branching, because there is no cycle in branchings. Suppose that a packet cannot reach the

destination. First, it is forwarded along the red branching. But its forwarding failed, so link $\{i, j\}$ was tried to use, which means that either (i, j) or (j, i) is in the red branching. Without loss of generality, we can suppose that this edge is (i, j). So the packet left node i using the blue branching. Failing the forwarding again means that link $\{i, j\}$ was tried to use again, so (j, i) is an edge of the blue branching. But the packet could reach node j from node i, meaning that there is a path from i to j in the blue branching and with (j, i) there is a cycle, which contradicts the assumption that there is no cycle in a branching. □

2.3.2 Non-2-edge-connected networks

Next, I deal with non-2-edge-connected networks. If the network is not 2-edge-connected, two edge-disjoint branchings cannot be found, but bypassing the failures when the network remains connected is still possible.

An undirected graph can be partitioned into some disjunct inextensible 2-edge-connected components. Naturally, it is possible that some components contain only one vertex. If removing a link causes the network to fall into two parts, it means that this link – a cut-edge – is between two 2-edge-connected components. It is also true that if vertex s and d are not in the same 2-edge-connected component, there is only one edge-disjoint path between them.

Using these ideas one may observe a possibility to improve LFIR. Duplicate the cut-edges virtually in the graph of the network. This new graph is 2-edge-connected, so after the transformation into a directed graph there will be at least two edge-disjoint branchings. Packets can follow these branchings as before. If a packet following a branching crosses a cut-edge, then the node after the cut-edge cannot decide which branching was used, so use the first one (the red one) for the next forwarding.

Lemma 2.3.3. *In the graph with duplicated cut-edges, all the cut-edges are used by both branchings.*

Proof. Suppose that edge e is a cut-edge which is not used by one of the branchings. We know that there is a path from each of the vertices to the destination in that branching which does not contain the cut-edge, so there is also a path between each vertex and the destination in the original undirected graph and none of them contains edge e. This means that without edge e the graph is connected, which contradicts the assumption that e is a cut-edge. □

Lemma 2.3.4. *In the graph with duplicated cut-edges, all the cut-edges are used in*

the same direction (i.e. if $\{i, j\}$ is a cut-edge, then both branchings contain (i, j) or both contain (j, i)).

Proof. Because of Lemma 2.3.3 all the cut-edges are used by both branchings. Suppose that there is cut-edge $\{i, j\}$ which is used in different directions. This means that duplicating $\{i, j\}$ is not necessarily, so without two $\{i, j\}$ the undirected graph is 2-edge-connected (there are two edge-disjoint branchings in the directed one). This contradicts the assumption that $\{i, j\}$ is a cut-edge. □

Theorem 2.3.5. *The improved version of LFIR is correct and complete (packet will arrive, if at most one non-cut-edge is down).*

Proof. If there is no failure, packets reach node d along the red branching. Now, suppose that there is a single link failure, link $\{i, j\}$ is down, and node i is the one, which locally rerouted the packets. If i and d are in the same 2-edge-connected component, packets will reach d along the blue branching thanks to Theorem 2.3.2.

Now, suppose that this is not the case, the failure is not in a component containing d. Let $\{x, y\}$ be the first cut-edge along a path from i to d (this is the same for all the paths including the one along the red and the one along the blue tree). Thanks to Theorem 2.3.2, $\{x, y\}$ will be reached, and then the packet will be forwarded along the red tree. Thus, packet cannot return to i, to the failure, along the red tree, since the path along the red tree from i to d contains both x and y, so they are ancestors of i in the red tree, and there is no cycle in a tree, so the algorithm is complete.

Moreover, according to Theorem 2.3.2, no FRR loop can be formed inside a 2-edge-connected component, so if there was a loop, it would contain some cut-edges, let one of them be $\{x, y\}$ again, and suppose that (x, y) is contained by both of the branchings (Lemma 2.3.4). Since x is the child of y in both of the trees, and since there is no edge between ancestors and a successors of y in any of the trees, packets cannot return to any successor of x or to x. In this way, (x, y) can be used only once, which contradicts the assumption that it is contained by a loop. Thus, the algorithm is correct.

□

2.4 Implementation questions

In the previous section, I proposed an algorithm for constructing a loop-free failure insensitive routing. In this section, we discuss some implementation questions, which are still open.

2.4.1 Finding branchings

For LFIR, the most important is an effective algorithm for finding branchings. Note that, unlike the ones in the literature, our branchings are directed *towards* the destination, not *away* from it. However, this does not cause any problem since any algorithm can be tweaked to reverse the direction of the branchings found.

In the next chapter, the generalization of edge-disjoint branchings is discussed. As it will turn out, the branchings we need for edge-redundant forwarding are also named as edge-redundant trees. In [IR84] linear time algorithm was proposed, so a pair of edge-redundant trees can be found in $O(|E(G)|)$ time. However, for LFIR another algorithm was proposed, which provides good optimization heuristics.

The computation of edge-disjoint branchings for LFIR is based on the algorithm of Lovász presented in [Lov76]. This algorithm starts from the destination; this is a directed tree. Then the algorithm adds directed edges (and vertices) to this directed tree until it becomes a branching. Each time a new link with the source outside and the target inside the directed tree can be added, if the remaining graph, the graph without the edges of the directed tree including the freshly added edge, is sill connected. It is proven in [Lov76] that in this way a branching can always be found.

Checking the connectivity of the graph can be made by the Breadth First Search (BFS) algorithm: if it is possible to reach the target of the newly added edge from its source in the remaining graph (which naturally does not contain the edge itself), then the remaining graph is still connected. Because at most $O(|E(G)|)$ edges should be checked in this way finding two edge-disjoint branchings takes $O(|E(G)|^2)$ time.

It is shown in Section 2.5 that if we use Lovász's method the order of the edges added to the red branching is very important in order to keep the average length of the red branching (the default forwarding) low. As heuristics I always chose the directed edge from the set of edges that can be added to the directed tree, which provides the shortest path to the destination. Using binary heap, finding the next candidate edge takes $O(\log |E(G)|)$ time. Inserting an edge needs $O(\log |E(G)|)$ time too. Because each edge can enter and leave the heap once, heap operations need $O(|E(G)| \log |E(G)|)$ time. When these heuristics are used, BFS traversal is still needed, so the complete time of finding two branchings is $O(|E(G)|^2 + |E(G)| \log |E(G)|) = O(|E(G)|^2)$.

Naturally, it is necessary to find the branchings for all the destinations so the complete computation needs $O(|V(G)||E(G)|^2)$.

2.4.2 Finding cut-edges

If it is not sure that the network is at least 2-edge-connected, it is needed to find the cut-edges. The simplest way of finding out whether an edge is a cut-edge or not is to simply remove the edge and check if the remaining graph is still connected. Checking the connectivity can be done by a BFS traversal with $O(|E(G)|)$ complexity, so finding all the cut-edges takes $O(|E(G)|^2)$ time. Since the complexity of finding a pair of edge-disjoint branchings for all the destinations takes $O(|V(G)||E(G)|^2)$, this simple technique can also be used, and the complexity is still dominated by the problem of finding the branchings.

2.4.3 Using LFIR in distributed environment

Using LFIR in distributed environment, such as routers in a network, raises a new problem; routers must find the same two branchings. A unique priority given to all the edges can solve this problem. If there are more edges with the same distance from the root during the edge selection, choose always the one with the highest priority. In this way building the red branching and then the blue one is fully defined, so if the routers has the same information about the network the same routing will be calculated.

2.5 Evaluation

We have discussed a new IPFRR technique, LFIR, which can always avoid loops for the price of longer paths. In this section, I show by extensive simulations that forming loops is a realistic problem and my solution, which always prevents loops, use only slightly longer paths.

For the simulations I used the topology of both real and random networks. I used the topology of the NSF network [CB92] and the backbone network of Germany and Italy [GO05]. For all the networks, I assigned random link lengths; the distribution of lengths was independent, discrete and uniform between 1 and 50.

Artificial network topologies were generated using Boston university Representative Internet Topology gEnerator (BRITE) [MLMB05]. I applied Waxman algorithm with random node placement and parameters $\alpha = 0.15$ and $\beta = 0.2$. The number of nodes varied between 20 and 50 and the number of neighbours was 2 and 3. Naturally, according to Waxman algorithm, this means that a node has at least 2 or 3 neighbours.

2.5.1 Probability of forming an FRR loop

Previously, I have shown that current IPFRR techniques using interface-based forwarding may cause loops. Now, I show by extensive simulations that this is not only a theoretical possibility but rather a realistic problem. Since LFIR is capable to handle link failures, it is considered as a version of FIR. Therefore, I computed the probability of loops, when FIR is used in the network. I supposed that FIR drops a packet, if it cannot be forwarded on the detour (packet is on a detour till it is forwarded through unusual interfaces).

For the evaluation, I use random experiments. The presence of a loop with FIR can be modeled by a Bernoulli random variable X (there is loop or not). Suppose that $P(X = 1) = p$ (the chance of loop), so the expected value and the variance are $EX = p$ and $\sigma^2 X = p(1-p)$. Make n Bernoulli random experiments and let their results be $X_1, X_2, ..., X_n$. Let $Y = \frac{\sum_{i=1}^{n} X_i}{n}$. Therefore, $EY = EX = p$ and $\sigma^2 Y = \frac{\sigma^2 X}{n}$.

Now, use Chebyshev inequality, namely $P(Y - EY \geq \alpha) = P(Y - p \geq \alpha) \leq \frac{\sigma^2 Y}{\alpha^2} = \frac{\sigma^2 X}{n\alpha^2} = \frac{p(1-p)}{n\alpha^2}$. Since $0 \leq p \leq 1$, $p(1-p) \leq 0.25$. Let $\alpha = 0.01$ and $n = 250\ 000$. In this way, $P(Y - p \geq 0.01) \leq \frac{0.25}{250\ 000 \cdot 0.0001} = 0.01$. Thus, the value of Y after 250 000 random experiments is a good approximation for the chance of loops; the chance that the precision of this approximation is worse than 0.01 (so the difference is more than 1%), is less than 1%. With other words, after 250 000 experiments a symmetrical confidence interval at size 0.02 at level 99% can be calculated. Hence, I made 250 000 random experiments in each case.

First, I studied the chance that a forwarding loop can be created. Therefore, I generated 250 000 random link lengths and networks as it was described previously, and I checked, whether it is possible to find a source node s, a destination node d and two links or a single node in such a way that if the selected links or the single node fails, packets heading from s to d get into a loop. The result of these simulations is surprising: it was always possible to create loops with FIR in all the studied real and random topologies irrespectively of the concrete configuration. Naturally, this means only that network topologies, in which FIR is prone to form FRR loops, are quite common, not that FIR can create loop in all the networks.

Second, I studied the probability of loop between a randomly selected pair of nodes. Therefore, I randomly selected a source node s, a destination node d and a pair of links or a single node, and tested if packets heading from s to d gets to a forwarding loop, if the selected links or the single node fails.

The result of these simulations are presented in Table 2.1 and Table 2.2. Curiously,

Top.	Prob. of loops w/ removed edges	Prob. of loops w/ removed node
NSF	0.39 %	5.4 %
Germany	0.82 %	18.68 %
Italy	0.76 %	18.38 %

Table 2.1. Probability of FRR loops when two edges or one node is removed in networks with real topology

Node number	Neighbour	Prob. of loops w/ removed edges	Prob. of loops w/ removed node
20	2	0.32 %	11.64 %
20	3	0.02 %	1.35 %
30	2	0.48 %	22.28 %
30	3	0.05 %	4.95 %
40	2	0.61 %	30.37 %
40	3	0.11 %	9.16 %
50	2	0.7 %	36.4 %
50	3	0.14 %	13.35 %

Table 2.2. Probability of FRR loops when two edges or one node is removed from networks generated by BRITE

the probability of FIR forming loops is not negligible; when a node fails it can reach even 36%. Based on these results we must establish that creating loops is an important problem of FIR.

2.5.2 Lengths of paths

Previously, the probability of forming loop was studied. Naturally, using LFIR these loops can always be avoided. On the other hand, LFIR may use longer paths for default forwarding. In this section, the question of the length of those paths is discussed.

For the simulations computing the lengths of default paths, the same real and random topologies were applied as former. In each network, I computed the length of default paths of LFIR between each pair of nodes. The mean of these path lengths is the average path length. In Table 2.3 and Table 2.4 the average path lengths are presented (100% is the shortest path) after 250 000 random experiments[3]. These path

[3]Naturally, this is not a Bernoulli random experiment, but the law of large numbers is still true.

Top.	LFIR w/ heur.	LFIR w/o heur.
NSF	106.27 %	137.37 %
Germany	116.36 %	146.15 %
Italy	112.07 %	150.38 %

Table 2.3. Average path lengths using LFIR with and without heuristics related to using shortest paths in networks with real topology

Node number	Neighbour	LFIR w/ heur.	LFIR w/o heur.
20	2	105.53 %	128.59 %
20	3	101.68 %	127.61 %
30	2	105.26 %	129.27 %
30	3	101.63 %	129.68 %
40	2	105.04 %	129.65 %
40	3	101.56 %	131.04 %
50	2	104.86 %	129.86 %
50	3	101.5 %	132 %

Table 2.4. Average path lengths using LFIR with and without heuristics related to using shortest paths in networks with random topology

lengths are computed both with and without the heuristics presented in Section 2.4.1.

Observe that the lengths of default paths are only slightly longer, at most by 17%, than the shortest ones. This means that applying LFIR in networks does not improve the network load significantly. Moreover, observe that the heuristics proposed are effective, since they were able to decrease the default path lengths by about 30% of the shortest path.

Based on the observations above, LFIR is a useful IPFRR technique in networks with interface-based forwarding. Although it slightly increases the lengths of default paths, it helps to avoid loops and always provides protection of single links.

However, there is an observation of this chapter, more important than an IPFRR technique, namely that the scheme of special directed spanning trees can be well applied in the field of IPFRR. Therefore, the way of finding such trees becomes one of the most essential questions. I study this problem in the next chapter.

Chapter 3

Finding Vertex-Redundant Trees

3.1 Introduction

In the previous chapter, IPFRR mechanisms using interface-based forwarding were discussed. There, a pair of edge-disjoint directed spanning trees were introduced for providing failure protection. As it turns out in this chapter, these edge-disjoint branchings can be used in a much wider area.

First, consider Definition 3.1.1. Observe that we have already seen edge-redundant trees, albeit they were named edge-disjoint branchings. Since, a pair of edge-redundant trees cannot contain the same directed edge, they are edge-disjoint. Moreover, a pair of edge-disjoint branchings must be a pair of edge-redundant trees, since after removing an undirected edge from the original graph, the destination is reachable from each vertex on at least one of the trees (Theorem 2.3.2). Thus, the definition of edge-redundant trees gives only a new view to the same concept.

Definition 3.1.1. Let an undirected graph be G with vertex $r \in V(G)$. A pair of *edge-/vertex-redundant trees* of graph G rooted at r is a pair of branchings (see Definition 2.3.1) rooted at r, such that the two paths along the two branchings from any given vertex $s \neq r$ to r are edge-/vertex-disjoint respectively.

Remark: A pair of vertex-redundant trees are depicted in Figure 3.1.

On the other hand, this new definition of edge-disjoint branchings is important for generalize the previous concept. Although vertex-disjoint branchings are a nonsense, vertex-redundant trees can be defined. Observe that vertex-redundant trees are directed spanning trees, which give the possibility of bypassing even node failures.

Among others, vertex-redundant trees are well applicable in IP networks. As it was discussed in Section 1.3.3, they were applied for the IPFRR proposal called

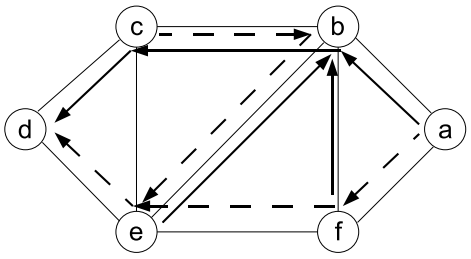


Figure 3.1. A pair of vertex-redundant trees rooted at vertex d.

IPRT [CHA07]. Furthermore, Lightweight Not-via [C5, C6, J4, P2], introduced in Chapter 5, uses the generalized concept of redundant trees, known as maximally redundant trees [C7, J4].

Redundant trees [MBFG99, XCT02c, XCT02b, XCT02a, XCT03, ZXTT05, ZXTT08, C5, C6, C7], also known as independent trees [IR84, Han98, ABS96, ZI89, Huc94, MTSIN98, CLY03, CLY06] and colored trees [Ram04, TRK06, BR06, RHK06a, RKK07, JRY09, KRKH09], are well studied objects in graph theory. They were first applied by Itai and Rodeh in [IR84]. They proposed algorithms for finding both edge- and vertex-redundant trees with computational complexity linear in the number of edges. Later, Médard *et. al.* [MBFG99] introduced these graphs as redundant trees to the field of protection in optical networks. There, the previous algorithm was generalized, albeit in this way linearity is not always retained. The concept of computation was further generalized and improved with Quality of Service (QoS) and Quality of Protection (QoP) possibilities by Xue *et. al.* [XCT03]. Later, linear time heuristics were proposed for providing QoS an QoP [ZXTT05, ZXTT08]. The first distributed algorithm was presented by Ramasubramanian *et. al.* in [RHK06a, JRY09].

In this chapter, I deal with vertex-redundant trees. Accordingly, the simple term of "redundant trees" means vertex-redundant trees in the sequel. Moreover, it is trivial that edge-redundant trees can only be found in 2-edge-connected graphs and vertex-redundant trees can only be found in 2-vertex-connected graphs. As it was proved by Itai and Rodeh, these criteria are not only necessary, but also sufficient [IR84]. Therefore, in this chapter I always assume that graphs are 2-vertex-connected. This artificial assumption is lifted in the next chapter, where maximally redundant trees are studied, which can be found in arbitrary connected graphs.

In the remaining part of this chapter, first, the problems necessarily raising, when one tries to implement a linear time algorithm proposed in [ZXTT05, ZXTT08], are discussed. This algorithm is the first linear time algorithm, which provides QoS by decreasing the total cost of the trees. Moreover, this algorithm is cited several times

in the literature (e.g., [RHK06b, TRK06, JRY09, CHA07]). Second, I propose a new technique, which overcomes these difficulties, and which can provide the same QoS level. Finally, I show a new algorithm, which finds a pair of redundant trees rooted at each vertex in linear time. Although there are numerous linear time techniques capable to find a single pair of redundant trees rooted at a given vertex, finding a pair rooted at each vertex needed quadratic time till now.

3.2 Problems with implementing Zhang's linear time algorithm

Zhang *et. al.* have studied the possibility of endowing redundant trees with Quality of Service (QoS) capabilities in [ZXTT05, ZXTT08]. They have presented an algorithm, called *ReducedCostV*, which is capable to find a pair of vertex-redundant trees in linear time with low total cost.[1] Unfortunately, as it will turn out, in some special cases this algorithm is not linear. In this chapter, I briefly introduce ReducedCostV, present a simple example, which makes the processing clear and finally, I point out the difficulties which ruin linearity. Further details can be found in [ZXTT05, ZXTT08].

The algorithm, given by Zhang *et al.*, is based on a special Depth First Search (DFS) traversal, which computes both the DFS and lowpoint numbers. The DFS number of a given vertex v is the number of vertices visited by the DFS traversal before v. The lowpoint number is a bit more complex: for a vertex different from the starting point of the DFS, find the child with the lowest lowpoint number and the neighbour with the lowest DFS number; the lowest one from these two values is the lowpoint number. The starting point of the DFS traversal has no lowpoint number. These values can be computed by a special DFS traversal, presented in Algorithm 1 (Line 3 is not needed for ReducedCostV, but it is essential for all my later algorithms).

When DFS tree is computed, the algorithm walks down on it selecting always the "eldest" child, the child with the lowest lowpoint number. At this point, one may observe some differences between the algorithms, since [ZXTT05] uses stack and [ZXTT08] uses FIFO. Fortunately, it does not really influence the behavior of the algorithm.

Since the lowpoint of a child must be lower than the DFS number of its parent (this claim will be proved in Lemma 3.3.1), walking down on the tree by always selecting the eldest child means selecting vertices with lowpoint number strictly lower

[1] Cost function is discussed later.

Algorithm 1 Revised DFS for graph G and root vertex r

1: Start a DFS traversal on the graph from root r. Set DFS number D_v at each vertex v, so that D_v denotes the number of vertices visited before v.
2: Recursively compute the lowpoint number for each vertex v as $\min(L, D)$, where L is the smallest lowpoint number of v's children and D is smallest DFS number among v's neighbours (since a leaf has no child, the recursion always comes to an end).
3: For each vertex v, associate a directed edge (v, x), where x is the vertex from v received its lowpoint number. If it is possible, choose an arbitrary child as x.

than the DFS number of the starting vertex x as long as possible. Sooner or later the algorithm encounters a "jump" in the lowpoint number, when it becomes higher than or equal to the DFS of x. At this point, along the path we found some vertices were visited, which are now called as an *ear*. The sequence of vertices specified by the freshly found ear is added to the first tree in one direction and to the second tree in the reverse direction.

Unfortunately, it does matter that which direction of the ear is added to which tree. For deciding the correct direction a partial order of the vertices, called voltage, is used. When a new ear is found, the two endpoints of it are compared, and the direction towards the lower one is always added to the first (say red) tree and the other direction to the second (say blue) tree. Finally, voltages are assigned to the vertices in the freshly found ear, so that these voltages are between the voltages of the two endpoints.[2]

The voltage of the starting point (the root of the redundant trees) is special. It can be either the smallest or the highest in the network. Moreover, when the first ear is found, its both endpoints are the root, so it will be needed to compare with itself. In this case, either one of the directions is acceptable for the red tree and the opposite is acceptable for the blue one.

Finally, before presenting the example, we discuss the cost function, since this is a heuristic algorithm, which tries to decrease the cost of the redundant trees. The cost of the two trees in this case is the number of edges used by *either* of the two trees (so an edge used by both or one of the trees counts 1); thus, the cost of the trees in Figure 3.1 is 8. This is useful for networks, where links are needed to be

[2]Supposing that voltages are taken from a finite, totally ordered set, which, as it will turn out, is needed for linear complexity, raises the need of a bit stricter rules for the new voltages, in order to make sure that there is no two vertices with the same voltage. In order to keep the reasoning simple, I do not deal with these details, since they are not needed in the sequel; reader can find them in [ZXTT08].

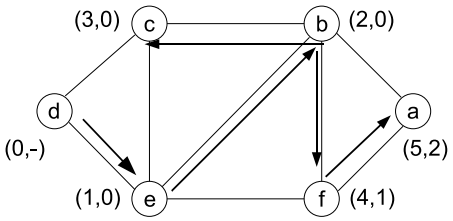


Figure 3.2. A possible DFS, the DFS and the lowpoint numbers.

reserved for protection (e.g., optical networks).[3] Using this cost function, adding an ear with n new vertices to the trees costs $n+1$ (there are $n-1$ edges between the new vertices and two to connect the endpoints), so covering the graph with k ears costs $|V|-1+k$ (the root is immediately covered). Therefore, the algorithm needs large ears in order to minimize their number, so it walks down on the DFS tree as long as possible [ZXTT08].

Before turning to discuss the problems this algorithm suffers from, I present a simple example. Consider again the network depicted in Figure 3.1. Suppose that we are computing a pair of redundant trees rooted in vertex d. First a DFS is needed, let this be the one depicted in Figure 3.2. The DFS and the lowpoint number is written next to each vertex.

The computation of redundant trees is started from vertex d, we set it *ready*, and then walk down along the DFS tree from it. If a vertex has more than one child, such as in the case of vertex b, choose the eldest child. In this way we find ear e, b, c, and we stop at c, where a jump in the lowpoint number is, since it has no child with lowpoint number equals to the DFS number of d.[4] There we 'tie the knot' to the vertex whose DFS number equals the lowpoint number of c, which is vertex d.

The newly found ear is special, since its both endpoints are d, so the direction is not important. Now, set the voltages in such a way that $v(d) \prec v(e) \prec v(b) \prec v(c)$ (where $v(x)$ is the voltage of vertex x), so edges (c,b), (b,e) and (e,d) are added to the red tree, edges (e,b), (b,c) and (c,d) are added to the blue one (as it is presented in Figure 3.1), and vertices c, b, e are set *ready*. Next we take a *ready* vertex, let it be vertex b. We could take either vertex e or vertex c as well, but then we would continue with another *ready* vertex, since they have no child, which is not *ready* yet. From b we walk to vertex f, which has a child, albeit the lowpoint number of a is

[3]The cost function under discussion is undoubtedly hardly useful for IP networks. However, here I give a corrected version of ReducedCostV, so we need to give good results with respect to these costs. I deal with the problem of optimizing trees for IP networks in Chapter 4.4.

[4]The algorithm needs a child with lowpoint number strictly lower than DFS number of the starting point of the ear (here d), or one with lowpoint number 0.

equal to the DFS number of b, so the next ear is vertex f alone and voltages $v(b)$ and $v(e)$ are needed to be compared. Since $v(e) \prec v(b)$, (f, e) is added to the red tree, (f, b) is added to the blue tree, f is now *ready*, and $v(f)$ is set in such a way that $v(e) \prec v(f) \prec v(b)$. Similarly, the last ear is vertex a alone, $v(f) \prec v(b)$, so (a, f) is added to the red tree, and (a, b) to the other one.

The most important problem of this algorithm stems from the need of storing the voltages, the order of vertices. One may observe that a data structure would be needed, where both inserting a new item and comparing two vertices takes $O(1)$ time. Unfortunately, simple arrays are immediately ruled out, since insert may need moving some elements. Moreover, linked lists or tree structures (e.g., binary trees) do not come to rescue either, since comparing cannot be done in $O(1)$ time using these structures.

According to private mailing, the authors of [ZXTT05, ZXTT08] assigned platform-native numbers as voltages. Although first it seems a practical solution, as computers encode numbers in a finite number of bits (e.g., an IEEE double precision float uses 64 bits, so it can represent less than 2^{64} different values), the algorithm might easily run out of assignable distinct voltage values, rendering the result incorrect.

Theorem 3.2.1. *Suppose that voltages take their values from a finite, totally ordered set S. Then, there is a graph of at most $3\log_2(|S|) + 7$ vertices and a possible DFS traversal, on which an implementation of the algorithm gives incorrect answer.*

Proof. Suppose that the elements of S are identified by a unique natural number, which we shall use to assign voltages, and by $v_i < v_j$ we shall mean that the voltage of vertex v_i is less than that of v_j. Consider the graph depicted in Fig. 3.3 and let r be the root vertex. The arrows show a possible DFS traversal of the graph: $r, b_1, c_1, a_1, a_2, c_2, b_2, \ldots, b_k, c_k, a_k$. Note that the last three items can vary if k is even, but the proof remains essentially the same.

In the ith step, the algorithm finds the vertex sequence a_i, c_i, b_i between vertices b_{i-1} and c_{i-1}. Without loss of generality, suppose that $b_{i-1} < c_{i-1}$ (otherwise, the proof proceeds similarly, only the relations are the other way around). So, to the tree along which voltages increase we need to add the path $b_i \to c_i \to a_i$, so we set $b_i < c_i < a_i < c_{i-1}$. Similarly, along the decreasing tree we write $a_i > c_i > b_i > b_{i-1}$, so we have $b_{i-1} < b_i < c_i < a_i < c_{i-1}$. We choose the voltage of c_i pessimistically to $\lfloor \frac{a_i+b_i}{2} \rfloor$ (otherwise, if $c_i - b_i > a_i - c_i$ held, we could reconstruct the graph by connecting a_{i+1} to a_i instead of c_i, and b_{i+1} to c_i instead of b_i, which would yield a suboptimal subdivision of S). In consequence, we have that $a_i - b_i < \frac{a_{i-1}-b_{i-1}}{2}$ and, for a general k, we have $a_k - b_k < \frac{|S|}{2^{k-1}}$. Finally, we observe that the algorithm fails

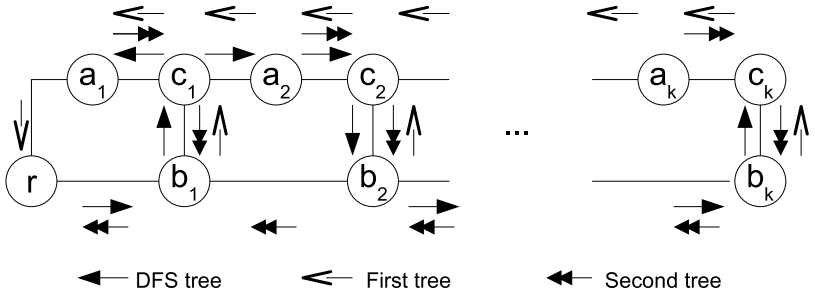


Figure 3.3. Illustration for Theorem 3.2.1.

if we run out of distinct voltage values in S, that is, if $a_k = b_k$, which occurs when $\frac{|S|}{2^{k-1}} < 1$. Therefore, for arbitrary finite S, we can show a graph of $k = \log_2(|S|) + 2$ ears (or $3(\log_2(|S|) + 2) + 1 = 3\log_2(|S|) + 7$ vertices), for which the algorithm fails to find correct redundant trees. □

The smaller the set S of voltages, the sooner this pathologic behavior emerges. For 32 bit integers, we only need 103 vertices for the algorithm to fail, and for 64 bit integers we need 199 vertices. Floating point arithmetic does not come to rescue here either: double precision floats of 64 bits run short again at 199 vertices at the most. A solution would be to use arbitrary precision arithmetics, however, such arithmetics does not provide $O(1)$ insertion and/or comparison, rendering the implementation worse than linear.

3.3 Finding redundant trees in strictly linear time

In this section, I present a revised algorithm for finding a pair of redundant trees in linear time [C7]. This algorithm is divided into 3 distinct phases: in the first phase, it performs a DFS traversal of the graph, then computes an intermediate graph representation based on which in the final phase it obtains the redundant trees. Since special care is taken to ensure that each phase terminates in linear time, the resultant algorithm will still be linear. Moreover, this algorithm does not use voltages, so the traps discussed in the previous section are avoided. Furthermore, as it is shown in Section 3.3.4, the cost of redundant trees created by this algorithm is only slightly larger than the ones found by the original but not linear algorithm.

3.3.1 Phase I – DFS traversal

In the first phase, my algorithm computes the DFS and lowpoint numbers with exactly the same algorithm as previously (Algorithm 1). In the sequel, the root vertex (starting vertex of DFS) will be denoted by r, and the DFS and lowpoint number of some vertex v will be denoted by D_v and L_v respectively (naturally, $D_r = 0$).

Next, I present a simple technical lemma for characterizing DFS numbers, which is necessary in the sequel.

Lemma 3.3.1. *Let v be a vertex of an undirected 2-vertex-connected graph. Do a DFS traversal and start it at $r \neq v$. Let the DFS parent of v be p. If $D_v \geq 2$, then $L_v < D_p$. If $D_v = 1$, then $L_v = 0$.*

If $D_v \geq 2$, walking down as long as possible along the DFS tree from v by always selecting a child c, such that $L_c < D_p$, leads to a successor with such a neighbour y in G, that y is a DFS ancestor of p. Otherwise, if $D_v = 1$, walking down as long as possible along the DFS tree from v by always selecting a child c, such that $L_c = 0$, leads to a successor, which is a neighbour of r.

Remark*: Note that if $D_v \geq 2$, it is possible that v has no child c with $L_c < D_p$. Then we "walk down" zero hops along the DFS tree and y is a neighbour of v.*

Proof. If $D_v = 1$, v is a child of r, so r is a neighbour of v, and $L_v = 0$. If $D_v \geq 2$, let the DFS subtree rooted at v be T (so v and its successors are in T). Since r and the vertex with DFS number 1 are not in T, $|V(G) \setminus V(T)| \geq 2$. Thus, since G is 2-vertex-connected, there must be at least two $\{m, x\}$ edges in G, such that $m \in V(T)$, $x \in V(G) \setminus V(T)$ and the endpoints of these edges in $V(G) \setminus V(T)$ are different. One of these edges have p as an endpoint in $V(G) \setminus V(T)$, but there must be at least one, which has another vertex y outside T. Consider this $\{m, y\}$ edge.

Thanks to the properties of the DFS traversal, since y is a neighbour, it must be either a successor or an ancestor. Since $y \notin V(T)$, y must be an ancestor of both m and v. Moreover, since $y \neq p$, y must be an ancestor of p too. Thus, $L_v \leq D_y < D_p$.

Suppose that $D_v \geq 2$. Walk down along the DFS tree, and always select the child c, such that $L_c < D_p$ as long as possible. Sooner or later we get to a vertex n, which has no decent child. Since all the children of n has higher lowpoint number than L_n, n got its lowpoint number from a neighbour outside T. Let this neighbour be z. Since $D_z = L_n < D_p$, z is an ancestor of n, v and p.

If $D_v = 1$, following the vertices with lowpoint 0 along the DFS tree as long as possible sooner or later leads to a vertex q, which has no decent child. Since $L_q = 0$, q must be a neighbour of r. Since the graph is 2-vertex connected, there must be a cycle containing both r and v, thus $q \neq v$. □

Note that Algorithm 1 is implementable with a slight modification of the standard DFS traversal algorithm, and thus its complexity is $O(|V|+|E|) = O(|E|)$ (the graph is 2-vertex-connected, so $|V| \leq |E|$).

3.3.2 Phase II – Finding an ADAG

As it was mentioned previously, my algorithm is divided into three phases. Below, we discuss the second, intermediate phase, when a special directed spanning graph is computed, which is called spanning ADAG (Almost Directed Acyclic Graph) since it "almost" fulfils the DAG (Directed Acyclic Graph) property. This intermediate step is important for two reasons: first, it facilitates a cleaner, modular implementation; second, as it will be discussed in the next chapter, this intermediate graph will be well applicable for finding multiple redundant trees, a pair rooted at each vertex, in the same time.

Next, I give the definition of ADAG, which can be well applied in 2-vertex-connected graphs. Since in the next chapter the concept of redundant trees will be generalized to arbitrary connected graphs, this definition will be improved, and Generalized ADAG will be introduced.

Definition 3.3.1. Let a strongly connected digraph D be an *ADAG* with vertex r as a root, if for each $v \in V(D)$ there is directed cycle containing both v and r, and all the cycles in D contain r.

Remark: Note that without r, D is a DAG.

A spanning ADAG is depicted in Figure 3.4. Here, I also define exactly the concept of ear.

Definition 3.3.2. Let an *ear* be a sequence of vertices we push to the stack at the same time (Line 11 or Line 21 of Algorithm 2).

Finding a spanning ADAG is done in a similar way as redundant trees were found by the algorithm of Zhang *et. al.*; the edges of the ADAG will be directed always towards the higher voltage. In order to avoid the traps of voltages, the main idea is to ensure that we always proceed from lower voltage vertices towards higher voltage vertices, so we always know in which direction to attach new paths (recall that voltages were used to help deciding on the order of vertices as they are added to the trees). To maintain this invariant, we need to ensure that we only leave a vertex when we have found not only all those ears *emanating* from it (as the original algorithm of Zhang *et al.* does), but also those ones that *terminate* in it, by sometimes traversing the

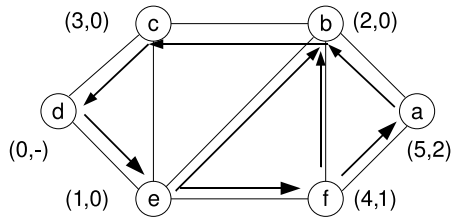


Figure 3.4. A sample graph and the computed ADAG, for root vertex d. The computation is based on the DFS traversal depicted in Figure 3.2

DFS tree upwards instead of moving always downwards. This idea is implemented in Algorithm 2. Note that Algorithm 2 does not compute the redundant trees right away, it instead builds an intermediate graph representation D.

Furthermore, observe that this algorithm tries to minimize the cost of the trees found (the cost is the same as it was previously for Zhang's algorithm). As it will be discussed in Section 3.3.3, the cost of the redundant trees cannot be higher than the number of edges of this ADAG. Therefore, this algorithm also tries to find long ears. When we are walking down on the DFS tree, we do the same as the original algorithm does. On the other hand, when we are walking up, we only choose a neighbour, if it got the lowpoint from the *current* node. More detailed description is presented in Section 3.3.3.

Before we turn to discuss the specifics of Algorithm 2, I first provide a short example of the algorithm's procession. Considering the same network and the same DFS traversal (Figure 3.2) as before; the graph D calculated by Algorithm 2 is given in Figure 3.4. We start from vertex d and the first ear we find is $e \to b \to c$, so edges (d, e), (e, b), (b, c) and (c, d) are added to D. Now, stack S contains "ebc", so the next vertex we pop from the top is vertex e. Vertex e has only a *ready* child, so we do not enter the branch at Line 6. However, we observe that there is a neighbouring vertex still not marked *ready*, vertex f, so we take the branch at Line 16 and we move upwards along the DFS tree until we arrive to a *ready* vertex, vertex b. Therefore, the next ear is made up by vertex f alone, and edges (e, f) and (f, b) are added to D. Now, the stack contains "fbc". We pop f, whose only child is vertex a, so next we find the ear consisting of the sole vertex a, and (f, a) and (a, b) are added to D. At this point all vertices are marked *ready*, so the the algorithm terminates (after popping the remaining entries "abc" from the stack).

Next, I show that Algorithm 2 always terminates. For this, we only need to show that the two main branches of the algorithm (Line 6 and Line 16) terminate.

Lemma 3.3.2. *The branches at Line 6 and 16 always terminate.*

Algorithm 2 Finding a spanning ADAG for graph G and root vertex r

Compute a DFS tree using Algorithm 1. Initialize the ADAG D with the vertices of G and an empty edge set. Create an empty stack S. Set the *ready* bit at each vertex to $false$.
push r into S and set *ready* bit at r to *true*
while S is not empty
    $current \leftarrow$ pop S
    for each child n of *current*
        if n is not *ready* **then**
            while n is not *ready*
                Let e be the child of n with lowpoint number 0 or less than the DFS number of *current*. If there is no such child, let e be the node, where n got its lowpoint from.
                $n = e$
            end while
            Let the found vertices be $x_0 \rightarrow x_1 \rightarrow ... \rightarrow x_k$, where x_k is *ready*, and x_0 is the neighbour of *current*. Set the *ready* bit at $x_0, x_1, ..., x_{k-1}$ to *true* and push them into S in reverse order, so eventually the top of the stack will be $x_0, x_1, ..., x_{k-1}$
            Add edges in the path $current \rightarrow x_0 \rightarrow x_1 \rightarrow ... \rightarrow x_k$ to D.
        end if
    end for
    for each neighbour n of *current* which is not a child
        if n is not *ready* and n got lowpoint number from *current* **then**
            while n is not *ready*
                let e be the parent of n in the DFS tree
                $n = e$
            end while
            Let the found vertices be $x_0 \rightarrow x_1 \rightarrow ... \rightarrow x_k$, where x_k is *ready*, and x_0 is the neighbour of *current*. Set the *ready* bit at $x_0, x_1, ..., x_{k-1}$ to *true* and push them into S in reverse order, so eventually the top of the stack will be $x_0, x_1, ..., x_{k-1}$.
            Add edges in the path $current \rightarrow x_0 \rightarrow x_1 \rightarrow ... \rightarrow x_k$ to D.
        end if
    end for
end while

Proof. First, I show by mathematical induction that all DFS ancestors of an arbitrary *ready* vertex are always marked *ready*. Initially, this is true, since only r is *ready*. Then, after finding an ear either at line 6 or at Line 16, the claim remains true, since all the ancestors of a vertex in the ear became *ready* too.

At the end of the branch at Line 6, we always arrive to r or to an ancestor of the starting vertex, since there is such a path thanks to Lemma 3.3.1. From r we return to r, and from any other vertex we eventually reach an ancestor (which is *ready* at this point as I have shown above), so the branch at Line 6 indeed terminates. On the other hand, in the branch at Line 16 we always move upwards in the DFS tree, heading towards r. Since r is *ready*, a *ready* vertex is always reached finally, so the branch at Line 16 also terminates. □

Next, I show that the output graph is an ADAG.

Lemma 3.3.3. *The output graph of Algorithm 2 is a spanning ADAG of G.*

Proof. First, I show that r is in all the cycles in D. Remove r from D and let this new graph be D'. Observe that in both cases when we add edges to D', the endpoints of the edges in the ear appear exactly in the same order both in the edge and in the stack. Consider an ear the algorithm finds either at Line 11 or Line 21. This ear starts at *current* and terminates at another vertex, say, x. Since $r \notin V(D')$, claims about *current*, where $current = r$ or claims about x, where $x = r$ are not important (and not always true). Otherwise, the following claims hold for *current* and x:

- $current \neq x$ (at branch 6, this is true due to Lemma 3.3.1, and at branch 16 because all the children have been made *ready* by branch 6)
- *current* has already left the stack and
- x is still on the stack (since it has a neighbour, the last vertex of the ear, which is either a child or which got its lowpoint number from x).

Now, let $V = v_1, v_2, ..., v_n$ be the sequence of vertices as they leave stack S. Observe that if there is a (v_i, v_j) edge in D', then v_i and v_j was either in the same ear or (v_i, v_j) was an end of the ear (one of the vertices was *current* or x). According to the argumentation above, when we add edge (v_i, v_j) to D' one of the following two cases hold

- v_i has already left the stack when we push v_j or
- v_i appears above v_j in the stack.

Thus, v_i will leave the stack before v_j, which means $i < j$. Therefore, we have that for each (v_i, v_j) in D', $i < j$ holds, so V is a topological order and hence D' is a DAG.

Next, I use mathematical induction in order to prove that D is strongly connected, and for each $v \in V(D)$, there is a directed cycle, which contains both r and v. Initially, when D contains only r, the claim is true. Suppose that after adding some ears it is still true.

Now, we add a new ear from *current* to x. There must be a path from r to *current* and one from x to r, thanks to strong connectivity. Moreover, there can be no directed path from x to *current*, which does not contain r, otherwise with the newly found ear there would be a cycle not containing r. Consequently, there is a cycle in D, which starts from r, goes through *current*, then all the vertices of the new ear, and finally returns to r through x. Naturally, the graph remains strongly connected thanks to this cycle.

Now, we have seen that D is an ADAG. In order to prove that this is a spanning ADAG of G, it is needed to observe that all the vertices of G becomes *ready*. Since a *ready* vertex leaves the stack sooner or later, a neighbour of a *ready* vertex must be also *ready* when Algorithm 2 terminates. Since there was *ready* vertex (at least r is *ready*), and since the graph was connected, all the vertices are needed to be *ready*, when the algorithm terminates. □

Lemma 3.3.4. *The computational complexity of Algorithm 2 is* $O(|E(G)|)$.

Proof. Each vertex is pushed into S and popped from S once, so the most important part of the algorithm is at Line 6 and at Line 16, where the ears are found. Line 16 is simpler, since the parent can be stored at each vertex during the DFS traversal, so finding it takes $O(1)$ time. In this way, finding ears at Line 16 takes at most $O(|V(G)|)$ time altogether.

Finding an ear at Line 6 is more difficult, since it is needed to find the eldest child, at each vertex. The simplest way for finding it is to check each of the children. Fortunately, this is needed only once (e.g., we do not need the second eldest child), so in the worst case each edge of the DFS tree will be used once. Thus, since the DFS tree has $|V(G)| - 1$ edges, the complexity is $O(|V(G)|)$ again.

Since a DFS traversal takes $O(|E(G)|)$ time in a connected graph, the overall complexity is $O(|E(G)|)$. □

3.3.3 Phase III – Constructing redundant trees

In the final phase, I construct a pair of redundant trees from the spanning ADAG produced by Algorithm 2. First, a simple definition is needed.

Definition 3.3.3. Let D be a spanning ADAG of graph G. Split the root vertex r in D into two vertices, r^+ and r^-, in such a way that edges only enter to r^+ and only leave r^-. Let this new graph be D'. Define a relation $(\prec)$ on $V(D')$ as follows: $u \prec v$ if and only if there is a directed path from u to v in D' $(u, v \in V(D'))$.

Generalize this relation; for given vertex x and y let $x \preceq y$ be true, if $x \prec y$ or $x \equiv y$.

It is easy to see that $(V(D'), (\preceq))$ makes up a bounded partially ordered set (poset); since D' is a DAG, it is reflexive, transitive and antisymmetric. Additionally, since there is a cycle in D for any vertex x, which contains r and x, there is a path from r^- to r^+ through x. Thus, r^- is less than any other vertex, so the minimum element is exactly r^-. Similarly, r^+ is the maximum element.

Theorem 3.3.5. *Perform a Breadth First Search (BFS) traversal from vertex r on the spanning ADAG D, yielding a directed tree R. Perform a second BFS traversal from r, but now taking the edges of D in reverse direction, yielding another tree B. Now, R and B are a pair of redundant trees rooted at r (when the edges are directed towards r).*

Proof. Create a new graph D' from D, by splitting vertex r into two vertices, r^+ and r^-, in such a way that edges only enter r^+ and only leave r^-. According to Definition 3.3.3, using this DAG, a partial order can be defined.

Observe that moving from a given vertex v towards r in R equals traversing the vertices in decreasing direction. Conversely, moving towards r along the other tree, B, means moving in increasing direction. This ensures that what we obtain by taking the paths $v \rightsquigarrow r$ in R and B, respectively, are two vertex-disjoint paths, which concludes the proof of the theorem. □

Note that this final phase again can be performed in a linear number of steps, since both BFS traversals are linear in the number of edges in D. Since each of the three phases turned out to be linear, the overall complexity of the algorithm is linear too.

Note that the cost of the two trees cannot be higher than the number of edges of the spanning ADAG. In order to minimize the number of edges of the spanning ADAG, special care was taken: when we walk down on the DFS tree, we do the same

Topology	Number of nodes	Original cost	Cost w/ revised algorithm	Increase
Germany	17	19.82	19.82	0%
NSF	26	31.65	31.8	0.47%
Cost266	37	43.49	44.13	1.47%
Germany50	50	57.06	57.78	1.26%

Table 3.1. Average cost of redundant trees on real topologies

as the original algorithm does, we walk down as long as possible without giving up the chance to get to an ancestor, therefore the ears found are quite long. On the other hand, at some point *current* vertex may have some not *ready* neighbours, which are covered by a single ear. Suppose that *current* has two not *ready* neighbours, a and b, such that a is an ancestor of b and $L_a = L_b$. Then, any ear, walking up on the DFS tree covering b, covers a as well. Therefore, a does not choose *current* as a source of the lowpoint number in Algorithm 1 at Line 3, and Algorithm 2 does not get to a from *current* at Line 16. The efficiency of these heuristics is studied in Section 3.3.4.

Getting back to our example, it is easy to construct the redundant trees from the ADAG depicted in Figure 3.4 using Theorem 3.3.5. These redundant trees coincide with the ones given in Figure 3.1; the tree marked by solid lines in Figure 3.1 coincides with B, while the tree marked by dashed lines is exactly R.

3.3.4 Evaluation of total cost

It was discussed previously that the algorithm presented by Zhang *et. al.* in [ZXTT05, ZXTT08] may lose linear time complexity on some graphs, because of the need of keeping up voltages. Therefore, a revised algorithm was presented, which is strictly linear. This revised algortihm was designed so that it tries to minimize the cost of the redundant trees in the same way as the original algorithm does. Unfortunately, in some cases the revised algortithm needs more ears to cover all the vertices of the graph, in this way finds trees with slightly higher cost.

In this section, the problem of this increase in cost is discussed. Therefore, I implemented both algorithms and made intensive simulations using real and random graphs. For real graphs I have chosen the topology of NSF network [SND] the German (Germany) and the European (Cost266) backbone network [GO05] and an extended 50 node version of the German backbone (Germany50) [SND]. For generating random topologies, I used BRITE again with the same configuration as previously in Section 2.5: Waxman algorithm with random node placement and parameters $\alpha = 0.15$

Node number	Neighbour	Original cost	Cost w/ revised algorithm	Increment
20	2	25.74	26.31	2.21%
20	3	23.94	24.38	1.84%
30	2	38.76	39.73	2.5%
30	3	36.26	37.07	2.23%
40	2	51.76	53.04	2.47%
40	3	48.57	49.69	2.31%
50	2	64.68	66.3	2.5%
50	3	60.76	62.16	2.3%

Table 3.2. Average cost of redundant trees on topologies generated by BRITE

and $\beta = 0.2$. The number of vertices varied from 20 to 50, and each had 2 or 3 neighbours. In each case 250 000 topologies were generated. Since the selection of the root vertex can silghtly influence the results, I computed the redundant trees for each vertex as a root for every topology and I computed the mean of these costs. The results are presented in Table 3.1 and Table 3.2.

As it can be observed, the efforts taken to decrease the cost of the redundant trees found by the revised algorithm were successful. Since the cost of redundant trees computed in linear time are only about 2% greater than the ones computed by the original algorithm, my new technique also provides about the same level of QoS.

3.3.5 Notes on st-numbering

In this section, we digress from the main line of the discussion for short, in order to introduce st-numbering, an interesting concept that has important relations to our topic of main interest. As it will turn out, st-numbering is very similar to to the concept of ADAG, therefore it is important to point out the differences, and reason, why the concept of ADAG was developed.

One may observe that any spanning ADAG computed by Algorithm 2 has only one edge entering the root (Lemma 3.3.6). Thanks to this fact, removing this edge cuts all the cycles in the ADAG and creates a DAG.

Lemma 3.3.6. *In a spanning ADAG, found by Algorithm 2 in a 2-vertex-connected graph G, only one edge enters to the root.*

Remark: *Note that not all the possible spanning ADAGs have this property, but the ones, which were found by Algorithm 2.*

Proof. Let x be the vertex with DFS number 1 (this is the vertex visited by the DFS traversal after vertex r). Since G is 2-vertex-connected, it is possible to reach all the vertices of G from x without using r, so r have only one DFS child. Thanks to this fact, only the first ear can have r as both endpoints. Since r is left only, when all its neighbours are ready, all the remaining ears of r have r as only the starting endpoint (*current*). □

Thus, a spanning ADAG with a sole edge entering to the root (like one found by Algorithm 2) is closely related to st-numbering (Definition 3.3.4 [ET76]). This connection is proven below.

Definition 3.3.4. Given any edge $\{s, t\}$ in a 2-vertex-connected graph G. Let an st-numbering be a bijective function $n : V(G) \to \{x \in \mathbb{Z}^+ : 1 \geq x \geq |V(G)|\}$, such that:

1. $n(s) = 1$,
2. $n(t) = |V(G)|$
3. each other vertex $x \neq s \wedge x \neq t$ have two neighbours y and z, such that $n(y) \geq n(x) \geq n(z)$.

It can be observed that an st-numbering is something similar to the concept of ADAG; actually it can be easily converted into an ADAG. Let $n : V(G) \to \{x \in \mathbb{Z}^+ : 1 \geq x \geq |V(G)|\}$ be an st-numbering of undirected 2-vertex connected graph G. Create a new directed graph D in such a way, that $V(D) = V(G)$ and $E(D) = \{(x, y) : \{x, y\} \in E(G) \wedge n(x) < n(y) \wedge (x, y) \neq (s, t)\} \cup \{(t, s)\}$. Trivially, there is no cycle in D without edge (t, s) thanks to the properties of st-numbering. Moreover, since each vertex $x \in V(G) \setminus \{s, t\}$ has at least two neighbours, one with less and one with greater st-number, there is a directed path from s to each of the vertices and there is a directed path from each of the vertices to t. Thus, with edge (t, s) there is such a cycle, which contains s, t and any arbitrary vertex, so D is an ADAG with s as a root and only a sole edge enters to s in D.[5]

However, observe that even if it is easy to convert an st-numbering into an ADAG, not all the ADAGs containing a single edge entering to the root can be produced in this way. Since an st-numbering gives a full order, an ADAG generated from such a numbering *must* contain all the edges of G in either direction.

[5]It is easy to prove that an ADAG with a sole edge entering to the root can also be converted into some st-numberings. However, since this converting is not needed in the followings, I do not deal with it.

Unfortunately, some algorithms, like my revised one for computing trees fulfilling QoS criteria, would be restricted by such an ADAG. As it was mentioned in Section 3.3.3 and as it was proved in Section 3.3.4, the revised algorithm for finding a pair of redundant trees can provide heuristics for reducing the cost of such trees, since the ADAG is made up by huge ears and therefore few edges. Naturally, this means that there are some edges of the original graph, which are not contained by the ADAG in any direction, which is now useful. The information about which edge should *not* be used in the trees is essential part of these heuristics. Similar observations was made by Xue *et. al.* in [XCT03].

3.4 Computing multiple redundant trees

In the previous section the way of computing a single pair of redundant trees rooted at a given vertex was discussed. This solves the routing problem with respect to a single root node. However, in real networks there are multiple destinations. Thus, usually a pair of redundant trees is needed for each vertex as a root, which would take $O(|V(G)||E(G)|)$ time with the algorithm presented in Section 3.3. In this section, I present a distributed algorithm, which can compute the same information in $O(|E(G)|)$ time. Naturally, not only the computational complexity, but the quality of these trees is important; this will be studied and improved in Section 4.4 and Section 4.5.

One may observe that computing a pair of redundant trees must need at least $\Omega(|V(G)|^2)$ time, since a single spanning tree has $|V(G)| - 1$ edges, so even writing the computed trees into memory would take at least $2|V(G)|(|V(G)| - 1)$ steps. At first, this fact seems to make impossible to create an algorithm for computing the trees in $O(|E(G)|)$ time. Fortunately, in common networks no node needs the whole redundant trees, but only the next hops, or more precisely, the edges going out from the vertex representing the given node. In this way, in a network, using the algorithm I present in this section, no node would know completely any of the redundant trees, albeit packages could be forwarded along the trees.

One may observe that there is another distributed algorithm, presented in [JRY09][6], computing the same information. However, the most important designing goal of the technique in [JRY09] was to minimize the number of messages, thus it uses only local information, and communication between the nodes is needed for the computation

[6]Note that there the trees are referred as colored trees.

itself. In contrast, since my solution is designed for intra-domain routing in IP networks, I suppose that the whole topology is explored (link state routing is used, like OSPF) and no cooperation is needed for computation. With these assumptions the algorithm presented in [JRY09] would need $O(|V(G)||E(G)|)$ time as well to compute all the redundant trees rooted at each vertex (my algorithm needs only $O(|E(G)|)$).

In the sequel, I will assume, that the input is exactly the same for all the nodes in the network. If it is not true (e.g., the order of vertices/edges is not the same) some precomputation may be needed for making exactly the same redundant trees.

I use the partial order of Definition 3.3.3 in the followings.

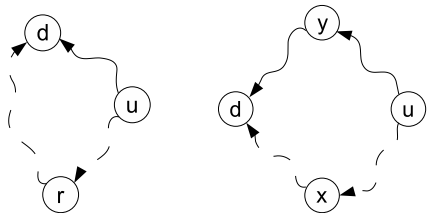
Definition 3.4.1. Let the spanning ADAG of graph G be D and let D' be the graph created by splitting the root of D into two (r^+, r^-). For some vertex u, let V_u^+ be the set of the vertices larger than u: $V_u^+ = \{v \in V(D') : u \prec v\}$. Similarly, let V_u^- be the set of vertices smaller than u: $V_u^- = \{v \in V(D') : v \prec u\}$.

Additionally, let $f_u^+(d)$ denote the first directed edge along some directed path from u to $d \in V_u^+$. Reverse the edges of D' and similarly let $f_u^-(d)$ be the fist edge along some directed path from u to $d \in V_u^-$. Moreover, let $f_u^+(r) = f_u^+(r^+)$, $f_u^-(r) = f_u^-(r^-)$. For the root vertex r, define $f_r^+(d) = f_{r^-}^+(d)$ and $f_r^-(d) = f_{r^+}^-(d)$ for all $d \in V(D') \setminus \{r^+, r^-\}$.

V^+ and $f^+(\cdot)$ can be computed by a BFS traversal of D. Similarly, V^- and $f^-(\cdot)$ come from a reverse BFS. This way, $f^+(\cdot)$ and $f^-(\cdot)$ encode the next-hop along the minimum-hop path, which makes paths shorter. Note that in general $V_u^+ \cap V_u^- = \emptyset$, but $V_u^+ \cup V_u^- \neq V(D') \setminus \{u\}$, because some vertices might not be ordered with respect to u.

Theorem 3.4.1. *Let the spanning ADAG be such that there is only one edge entering to the root (e.g., Algorithm 2 finds such an ADAG thanks to Lemma 3.3.6). Let vertices u and d be given in such a way that $u \neq d$, and choose the edge of the primary $(h_u^P(d))$ and the secondary $(h_u^S(d))$ tree rooted at d going out from u as follows:*

1. *If $d \in V_u^+$: $h_u^P(d) = f_u^+(d)$ and $h_u^S(d) = f_u^-(r^-)$*
2. *If $d \in V_u^-$: $h_u^P(d) = f_u^+(r^+)$ and $h_u^S(d) = f_u^-(d)$*
3. *Else: $h_u^P(d) = f_u^-(r^-)$ and $h_u^S(d) = f_u^+(r^+)$*
4. *Special rules apply at the root vertex (if $u = r$):*
   $h_r^P(d) = f_r^+(d)$ and $h_r^S(d) = f_r^-(d)$



(a) ordered case (b) unordered case

Figure 3.5. Illustration for Theorem 3.4.1.

Then, interleaving the primary ($h^P(d)$) and the secondary ($h^S(d)$) makes up a pair of redundant trees rooted at d.

Proof. To prove the theorem, it is enough to show that following the primary and the secondary next-hops comprises two loop-free, vertex-disjoint paths. The rules encode the intuitive idea: following the next-hops $h^P(d)$ we move in increasing direction in the poset, along $h^S(d)$ in decreasing direction, and if u and d are not mutually ordered, we move downwards in the poset until we can move upwards (or *vice versa*).

First, I show that for two vertices $v, w : v \prec w$, what we obtain by following the primary next-hops $h^P(w)$ is a loop-free $v \rightsquigarrow w$ path. Let $h_v^P(w) = (v, x)$. Using this edge, we either get to w, if $x = w$, or get to a vertex x, $v \prec x$. Moreover, $x \prec w$, since (v, x) is the first edge along a path from v to w, so there is a path from x to w too. Therefore, if $x \neq w$, we can repeat the same reasoning till we eventually arrive to w. Along the similar lines, following $h^S(w)$ yields a loop-free $v \rightsquigarrow w$ path for $v, w : v \succ w$

If $d = r$, the claim is trivial. Suppose $d \neq r$ and there is an order between u and d, say $u \prec d$. Now, following $h^P(d)$ yields an $u \rightsquigarrow d$ path p_p (the path marked by solid arrow in Figure 3.5a), and following $h^S(d)$ yields first a $u \rightsquigarrow r^-$ path p_s^1 and then an $r^+ \rightsquigarrow d$ path p_s^2 (dashed arrow in Figure 3.5a). Based on the observation above, these subpaths are indeed paths and they are loop-free. The concatenation of p_s^1 and p_s^2 gives the secondary path p_s. Finally, p_p and p_s are vertex-disjoint: vertices along p_p belong to the interval $[u, d]$, p_s^1 to $[r^-, u]$ and p_s^2 to $[d, r^+]$, and these intervals are disjunct except the endpoints.

If there is no order between u and d, the situation is slightly more difficult: following $h^P(d)$ first yields an $u \rightsquigarrow x$ path p_p^1 and then a $x \rightsquigarrow d$ path p_p^2, where x is the first vertex for which $u \succ x$ and $x \prec d$ holds (see the dashed arrows in Figure 3.5b). Similarly, $h^S(d)$ yields first a $u \rightsquigarrow y$ path p_s^1 and then an $y \rightsquigarrow d$ path p_s^2 for the first $y : u \prec y$ and $y \succ d$ (solid arrows in Figure 3.5b). Again, concatenation

of the corresponding subpaths yields two vertex-disjoint paths: first, p_p^1 and p_s^1 are vertex-disjoint because $p_p^1 \in V_u^-$, $p_s^1 \in V_u^+$ and $V_u^- \cap V_u^+ = \emptyset$; second, p_p^1 and p_s^2 are also vertex-disjoint because the vertices of p_p^1 are not ordered with respect to d but those of p_s^2 are; third, p_p and p_s cannot both traverse r, because $r^+ \succ y$ (since only one edge enters to r, we have a vertex m, for which $m \succ v : v \in V \setminus \{r^+, m\}$, so the secondary path turns back in m at the very latest). Similar reasoning applies to see that the rest of the subpaths are mutually vertex-disjoint too. □

Since the rules of Theorem 3.4.1 gives the possibility to compute the edges of a pair of redundant trees going out from a given vertex u in $O(1)$ time[7], computing all the edges takes $O(|V(G)|)$ time. Since computing the ADAG and doing two BFS traversals for finding V_u^+, V_u^-, $f_u^+(d)$ and $f_u^-(d)$ take $O(|E(G)|)$ time, computing all the trees takes $O(|E(G)|)$ time in any connected graph, when there is a processor assigned to each of the vertices, which is exactly the case in typical IP networks.

In this chapter redundant trees were discussed. I have presented algorithms for finding a pair of complete redundant trees and only the outgoing edges of a pair of redundant trees rooted at each node. In the next chapter I generalize both of the algorithms for finding *maximally* redundant trees in arbitrary connected graphs.

[7]Observe that deciding, whether a particular vertex is an element of V_u^+ or V_u^- respectively, can be realized in $O(1)$ time by assigning a bit to each of the vertices.

Chapter 4

Improving Redundant Trees

4.1 Introduction

In the previous chapter, algorithms for finding a pair of vertex-redundant trees were discussed. Although redundant trees are useful for precomputed detours, these trees have two significant drawbacks.

First, redundant trees always provide edge- or vertex-disjoint paths, so they can be found only in 2-edge-connected or 2-vertex-connected graphs. Although we search for such trees in communication networks, which are usually designed with redundancy, several such networks do not fulfil 2-vertex-connected requirement even when they are intact (e.g., see Abeline, AT&T in [SND] or Italian backbone in [GO05]). Moreover, even networks with proper redundancy may easily lose this capability after a failure.

Naturally, any arbitrary connected graph can be partitioned into 2-edge-connected or 2-vertex-connected subgraphs, and theoretically finding edge-redundant or vertex-redundant trees in these subgraphs is possible, albeit providing always the maximum redundancy (providing maximally edge- and vertex-disjoint paths simultaneously) in this way can easily become complex. In this chapter, I generalize the concept of redundant trees, in order to find decent spanning trees in any connected graph.

Second, the paths in several real networks are optimized somehow, thus usually it is not enough to simply find maximally redundant trees, but they need to fulfil some extra requirements. As an example, one may consider algorithm ReducedCostV proposed by Zhang *et. al.* in [ZXTT05, ZXTT08]. As it was discussed in the previous chapter, this algorithm uses heuristics in order to reduce the cost of redundant trees.

ReducedCostV was proposed for networks, where explicitly reserving resources (links) for protection is needed. Since in IP networks link lengths are selected in

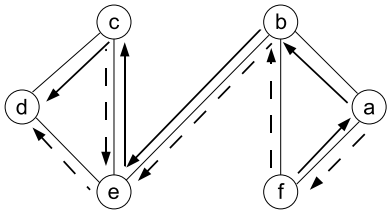


Figure 4.1. A pair of maximally redundant trees rooted at vertex d.

such a way that the shortest paths are the ones, which should be chosen, in these networks it is more desirable to select trees, which contain short paths. Therefore, in this chapter I propose heuristics, which do not ruin the linear complexity of the distributed algorithm, and which make paths significantly shorter in average.

4.2 Finding Maximally Redundant Trees

Previously, the concepts of edge- and vertex-redundant trees were discussed. Since paths in these threes are always edge-disjoint or vertex-disjoint, for finding such trees 2-edge-connectivity or 2-vertex-connectivity is needed. Now, I introduce the concept of *maximally redundant trees* [C7, J4], which lifts this artificial assumption networks may not be able to always fulfil.

Definition 4.2.1. Let an undirected graph be G with vertex $r \in V(G)$. A pair of *maximally redundant trees* of graph G rooted at r is a pair of branchings (see Definition 2.3.1) rooted at r, such that the two paths along the two branchings from any given vertex $s \neq r$ to r have the minimum number of vertices in common.

Remark: Note that this definition says that the paths along maximally redundant trees have only the unavoidable cut-vertices and cut-edges (a cut-edge has two cut-vertices as endpoints) in common. In this way the paths are always as edge-disjoint and as vertex-disjoint as it is possible.

A pair of maximally redundant trees is depicted in Figure 4.1.

As it will turn out, finding maximally redundant trees in a connected graph is always possible. In this section we discuss the way of finding these trees rooted at a single, arbitrary given vertex. There is also a distributed algorithm, similar to the one presented in Section 3.4, for finding all the maximally redundant trees rooted at not only one but each vertex simultaneously, which is discussed in Section 4.3.

4.2.1 Generalized ADAG

The algorithm for finding maximally redundant trees is based on the same idea as the one for finding simple redundant trees: an intermediate graph representation called spanning Generalized ADAG is needed. Previously, only a simple definition of ADAG was presented, which can be found only in 2-vertex-connected graphs. Now, I generalize this concept.

Definition 4.2.2. Let D be a strongly connected digraph and choose an arbitrary vertex r as *global root*. Consider a path from vertex $x \in V(D) \setminus \{r\}$ to r. Let r_x be the first weak cut-vertex after x along this path. If there is no such vertex (so x is a neighbour of r, or x and r are in the same weakly 2-vertex-connected component), let $r_x = r$. Let r_x be the *local root* belonging to x and global root r. The global root has no local root.

For simplicity, I also refer on global root as *root*. Moreover, in a graph with a given root, r_x denotes the local root of x in the sequel.

Definition 4.2.3. Let D be a strongly connected digraph with arbitrary chosen root r. Let C be the set of the maximum (here means inextensible) weakly 2-vertex-connected components of D. For each vertex $x \in V(D) \setminus \{r\}$, add x and r_x with the edges between them to C as a component, if there is no $A \in C$, such that $x, r_x \in V(A)$. Set C is the set of *clusters* of D and root r.

Let $r_A \in V(D)$ be the local root of cluster $A \in C$, if $r_A = r_x$ for all $x \in V(A) \setminus \{r_A\}$. (Note that for all paths from A to r, r_A is the last vertex in A.)

Definition 4.2.4. Let D be a strongly connected digraph with vertex r as root. Let the set of clusters in D be C. D is a *Generalized ADAG (GADAG)* with r as a root, if each cluster $A \in C$ is an ADAG with r_A as a root.

Remark: An equivalent definition is that D is a GADAG, if for all $x \in V(D)$ there is a directed cycle in D containing both x and r_x, and $A \in C$ is a DAG without r_A.

Although one may find these definitions a bit complicated at the first time, it is not so difficult to understand. Simply consider a GADAG as several ADAGs "glued" together at the weak cut-vertices (the local roots) which are the roots of these components[1]. Naturally, any ADAG D is a GADAG. Since an ADAG is weakly 2-vertex-connected, C has only one element, D itself, which is definitely an ADAG.

Next, I present a simple example. In Figure 4.2, a GADAG rooted at d is presented. This graph is made up by two weakly 2-vertex-connected components, a, b, f

[1]Note that r is a local root, albeit it is not necessarily a weak cut-vertex.

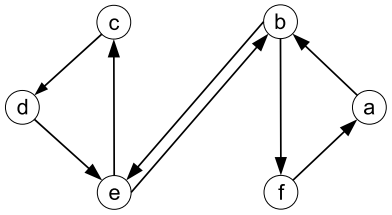
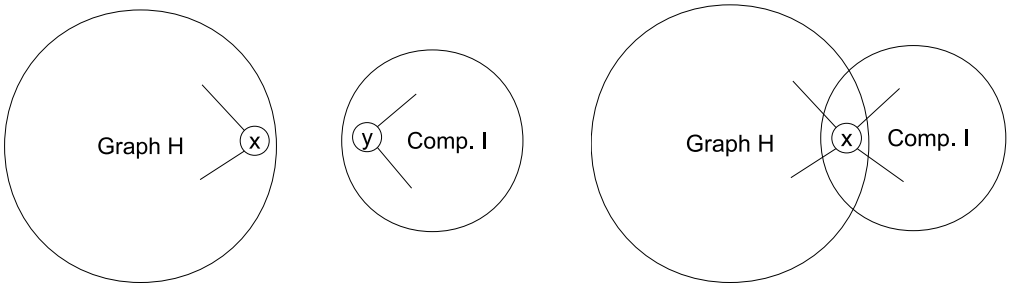


Figure 4.2. A Generalized ADAG rooted at vertex d.



(a) The graph and the new component.

(b) The component is "glued" to the graph.

Figure 4.3. Example for Lemma 4.2.1.

(let it be cluster X) and c, d, e (let it be cluster Y). Since there is no weakly 2-vertex-connected component, which contains b and its local root e, so C also contains b and e with the two edges between them as a cluster (let it be cluster Z). It is easy to see, that $r_c = r_e = d$, $r_a = r_f = b$, $r_b = e$, $r_X = d$, $r_Y = b$ and $r_Z = e$. Trivially, for each vertex there is a directed cycle containing the vertex and its local root. Moreover, without the local root, any of the three elements of C is a DAG, so the graph depicted in Figure 4.2 is a GADAG.

Algorithm 3 computes a spanning GADAG in an arbitrary undirected connected graph. Note that this is a version of Algorithm 2. The most important modification is at Line 9, which is needed for entering a new 2-vertex-connected component; other modifications are only important for finding multiple redundant trees, thus they will be discussed later, here they do not influence the result. Trivially, if Algorithm 3 is applied to a 2-vertex-connected graph, this modification at Line 9 means only that ears found by walking down on the DFS tree are a bit shorter, but Lemma 3.3.2 and Lemma 3.3.3 still holds true.

Lemma 4.2.1. *Let G be an arbitrary connected graph with vertex r. With graph G and vertex r as an input, Algorithm 3 always terminates, finds a spanning GADAG*

Algorithm 3 Finding a spanning GADAG for graph G and root vertex r. Note that Line 2, Line 14 and Line 29 are needed only for finding multiple trees.

1: Compute a DFS tree using Algorithm 1. Initialize the GADAG D with the vertices of G and an empty edge set. Create an empty stack S. Set the *ready* bit at each vertex to $false$.
2: Set *localRoot* at each vertex to $NULL$
3: push r into S and set *ready* bit at r to *true*
4: **while** S is not empty
5: $current \leftarrow$ pop S
6: **for** each child n of *current*
7: **if** n is not *ready* **then**
8: **while** n is not *ready*
9: let e be the vertex from where n got its lowpoint number
10: $n = e$
11: **end while**
12: Let the found vertices be $x_0 \rightarrow x_1 \rightarrow ... \rightarrow x_k$, where x_k is *ready*, and x_0 is the neighbour of *current*. Set the *ready* bit at $x_0, x_1, ..., x_{k-1}$ to *true* and push them into S in reverse order, so eventually the top of the stack will be $x_0, x_1, ..., x_{k-1}$
13: Add edges in the path $current \rightarrow x_0 \rightarrow x_1 \rightarrow ... \rightarrow x_k$ to D.
14: **if** $current = x_k$ **then**
15: Set *localRoot* to *current* at $x_0, x_1, ..., x_{k-1}$
16: **else**
17: Set *localRoot* to *current.localRoot* at $x_0, x_1, ..., x_{k-1}$
18: **end if**
19: **end if**
20: **end for**
21: **for** each neighbour n of *current* which is not a child
22: **if** n is not *ready* **then**
23: **while** n is not *ready* and n got lowpoint number from *current*
24: let e be the parent of n in the DFS tree
25: $n = e$
26: **end while**
27: Let the found vertices be $x_0 \rightarrow x_1 \rightarrow ... \rightarrow x_k$, where x_k is *ready* and x_0 is the neighbour of *current*. Set the *ready* bit at $x_0, x_1, ..., x_{k-1}$ to *true* and push them into S in reverse order, so eventually the top of the stack will be $x_0, x_1, ..., x_{k-1}$.
28: Add edges in the path $current \rightarrow x_0 \rightarrow x_1 \rightarrow ... \rightarrow x_k$ to D .
29: Set *localRoot* to x_k.*localRoot* at $x_0, x_1, ..., x_{k-1}$.
30: **end if**
31: **end for**
32: **end while**

rooted at vertex r, and there is exactly one edge entering to the local root in each cluster. Moreover, the cut-vertices of G are exactly the weak cut-vertices of D.

Proof. In order to keep the proof simple, in this proof I say that an undirected graph with two vertices connected by a single edge is 2-vertex-connected. For such a graph the first claim is trivially true. Moreover, thanks to Lemma 3.3.2 and Lemma 3.3.3 this claim is true for any 2-vertex-connected graph.

Next, I will use mathematical induction. We start from a 2-vertex-connected graph, and add 2-vertex-connected components to it as follows: let the graph be graph H the component be I; choose vertex $x \in V(H)$ and vertex $y \in V(I)$. Let $V(H') = V(H) \cup V(I) \setminus \{y\}$ and $E(H') = E(H) \cup E(I) \setminus \{\{y, v\} : \{y, v\} \in E(I)\} \cup \{\{x, v\} : \{y, v\} \in E(I)\}$ (depicted in Figure 4.3, uniting x and y). Note that any arbitrary undirected connected graph can be constructed by "gluing" components in this way.[2]

Suppose that after some components were added, the claim still holds true. Now, add one more component, let it be component A. Component A has a vertex in common with the previous graph (vertex x). The spanning GADAG of the graph without A will be computed by Algorithm 3 as in the previous step of the induction. In contrast, now, when x is popped from stack S, the algorithm enters to component A and traverses A, as it was a 2-vertex-connected graph with x as a root, and adds a new cluster to the GADAG found. In this way, the new resulting graph must be a GADAG as well. Moreover, since Lemma 3.3.6 holds true for A, there is only one edge in A entering to r_A.

Now, suppose that there is a weak cut-vertex x of D, such that x is not a cut-vertex of G. Since x is not a cut-vertex of G, it must be inside a 2-vertex-connected component A of G, so all the neighbours of x are in A. Therefore, x is a weak cut-vertex of the directed subgraph found in A as well. However, Algorithm 3 finds an ADAG inside A, and there is no weak cut-vertex in an ADAG, which contradicts the assumption that x is a weak cut-vertex. Naturally, since D is a spanning graph, all the cut-vertices of G must be weak cut vertices of D. □

Naturally, the computational complexity of Algorithm 3 is linear, since the proof of Lemma 3.3.4 remains true for arbitrary connected graphs.

[2]One may think that two 2-vertex-connected components may be connected with a sole edge without a common vertex, and it may seem that this case cannot be realized by this scheme. However, observe that now two vertices connected by a sole edge is considered as a 2-vertex-connected graph, thus this case can be realized in two steps.

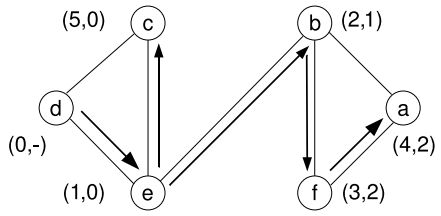


Figure 4.4. A possible DFS, the DFS and the lowpoint numbers.

Consider the network and the DFS traversal depicted in Figure 4.4. This DFS is computed by Algorithm 1. Now, use Algorithm 3 for finding a spanning GADAG. First, stack S contains d, so the algorithm starts from root vertex d, and then gets to e. It walks down along the DFS tree, always choosing the eldest child, which is vertex c, and ear e, c is found. The algorithm adds e and c to stack S (now it contains "ec"), sets them *ready* and adds (d, e), (e, c) and (c, d) to GADAG D. The next vertex popped out from S is e. Vertex e has a not *ready* child, vertex b. Since b got its lowpoint number from e, the next ear found is vertex b alone, it is set to *ready*, pushed on the top of S (which now contains "bc"), and the algorithm adds (e, b) and (b, e) to D. Next, node b is popped out from S, and the algorithm finds ear f, a, adds (b, f), (f, a) and (a, b) to D. Although the stack contains "fac", the nodes are all *ready*, so the algorithm terminates without adding any more edge to D. The computed GADAG coincides with the one depicted in Figure 4.2.

4.2.2 Constructing the maximally redundant trees

In the previous subsection not the maximally redundant trees, but only an intermediate graph representation called GADAG was computed. Now, I present the technique to construct the maximally redundant trees themselves. Naturally, the technique is very similar to the one presented in the previous chapter.

Theorem 4.2.2. *Perform a Breadth First Search (BFS) traversal from vertex r on the spanning GADAG D, yielding a directed tree R. Perform a second BFS traversal from r, but now taking the edges of D in reverse direction, yielding another tree B. Now, R and B are a pair of maximally redundant trees, rooted at r (when the edges are directed towards r).*

Proof. Starting from an arbitrary given vertex x, and walking along tree R yields a directed path from x to r taking the edges of D in reverse direction. Similarly, using

tree B yields a directed path taking the edges in normal direction. If x and r are in the same weakly 2-vertex-connected component, the claim is trivial (Theorem 3.3.5).

Let the set of clusters be C, and let $A \in C$. Let the first vertex of the path from x to r in R and in cluster A be a and the last be b. Vertex a must be either a weak cut-vertex or x. Similarly, vertex b must be either a weak cut-vertex or r. Thus, both the paths in R and B must contain both a and b. Since a and b are in the same cluster of GADAG D, it is trivial on the same line as the proof of Theorem 3.3.5 that the paths between a and b are vertex-disjoint. In this way only x, r and the weak cut-vertices between them are contained by both paths, so R and B are maximally redundant trees. □

Now consider the GADAG depicted in Figure 4.2. The computed trees would coincide with the ones depicted in Figure 4.1; tree R is the one marked with dashed lines, and tree B is the one marked with solid lines.

4.3 Computing multiple maximally redundant trees

Previously, the way of computing a single pair of maximally redundant trees was discussed. In this section, I show a distributed algorithm, similar to the one presented in Section 3.4. At this point one may think that the algorithm for computing multiple redundant trees would be able to compute maximally redundant trees as well, if the computed intermediate graph was a GADAG. Unfortunately, this is not so simple.

The problem stems from the fact that the previous algorithm "knew", which vertex was the local root. Since there is only one local root if the graph is 2-vertex-connected, the global root itself, it is possible to split this vertex into two. Fortunately, for maximally redundant trees finding the local roots can be done by adding some extra lines to Algorithm 2; these extra operations are at Line 2, Line 14 and Line 29 in Algorithm 3.

One may observe that if and only if Algorithm 3 finds an ear with the same vertex x as both endpoints at Line 7, the algorithm got in a new component $A \in C$. The local root of this component is x.[3] Both endpoints cannot be the same at Line 22, since such ear could be found at Line 7 sooner. In this way finding the local roots is easy, albeit it is still needed to know, which local root belongs to which vertex.

For vertex y in an ear with the same vertex x as both endpoints, this problem is trivial, $r_y = x$ (first possibility at Line 14). Other vertices in this component would

[3]Note that this is a common technique for finding cut-vertices. Here, we do not need to execute it separately, but we can find these vertices while the GADAG is being constructed.

have either two endpoints, which both have the same local root, or vertex x as the first endpoint (in this case $current = x$) and another one, vertex z. At Line 14, z is an ancestor of x thanks to Lemma 3.3.1, so $r_y = r_x$ (second possibility at Line 14). On the other hand, at Line 29, we walked upwards along the DFS tree, until we get to a *ready* vertex, which is definitely a successor of x, so $r_y = r_z$.

Now, it is needed to guarantee that the paths are maximally vertex-disjoint. This can be done similarly to the solution applied in Section 3.4: an order of the vertices is needed. Unfortunately, creating the order is not so simple; when the graph is 2-vertex-connected the only vertex ruining the DAG property is the root, and splitting this single vertex can provide the possibility of a partial order. In contrast, in arbitrary connected graphs all the local roots ruin the DAG property, and simply splitting them cannot solve the problem.

The main idea for computing the outgoing edges of maximally redundant trees for a given node x is to compute the trees first only for the 2-vertex-connected components containing x, and finding the paths to remaining vertices only after it. Since a 2-vertex-connected component contains only vertices with the same local root and the local root itself, it is easy to separate the necessary parts of the GADAG.

Procedure 4 SetEdge(vertex x)

1: **if** $h_u^P(x) = NULL \wedge h_u^S(x) = NULL$ **then** # Both, or neither is $NULL$
2: call SetEdge(r_x)
3: $h_u^P(x) = h_u^P(r_x)$
4: $h_u^S(x) = h_u^S(r_x)$
5: **end if**

Algorithm 5 computes the edges of the maximally redundant trees going out from a given vertex u. It is made up by two phases. In the first phase, it computes the edges for trees rooted at vertices in clusters of D containing u. Then, in the second phase, first the edges for the trees rooted at the global root are set, then the edges belonging to all the remaining trees.

As one may observe, the first phase is almost the same as the algorithm presented in Section 3.4. For vertices, which have vertex u as a local root, $h_u^P(x)$ and $h_u^S(x)$ are the first edges along the paths computed by the BFS traversals taking the edges in normal and reverse direction, which is exactly the same as Rule 4 in Theorem 3.4.1. Moreover, for vertices which have the same local root, Phase 1 computes the same edges as Rule 1, Rule 2 and Rule 3.

The second phase computes the vertices which are not in any of the clusters containing u. For the global root r, the edges going out are trivially correct. For

Algorithm 5 Computing the primary and secondary edges for all root d ($h_u^P(d)$, $h_u^S(d)$) going out from a given vertex u.

For all d set $h_u^P(d) = NULL$ and $h_u^S(d) = NULL$. Use Algorithm 3 for computing a spanning GADAG D with a given r as root (r is the same for any given u). Create digraph D' by splitting local root r_u into two vertices, so that edges only enter to vertex r_u^+ and only leave r_u^-. For each vertex x set $x.V^+ = false$ and $x.V^- = false$. If $u = r$ (r has no local root), do not split any of the vertices.

Phase 1: vertices in the same cluster

Do a BFS traversal on D' from u taking the edges in normal direction. Do not visit vertex x, if $x \neq r_u^+ \wedge x.localRoot \neq u \wedge x.localRoot \neq u.localRoot$. At visited vertex x set $x.V^+ = true$, and set $h_u^P(x)$ to the first edge along the path to x computed by the BFS.

Do a BFS traversal on D' from u taking the edges in reverse direction. Do not visit vertex x, if $x \neq r_u^- \wedge x.localRoot \neq u \wedge x.localRoot \neq u.localRoot$. At visited vertex x set $x.V^- = true$, and set $h_u^S(x)$ to the first edge along the path to x computed by the BFS.

if $u \neq r$ **then**
    set $h_u^P(r_u) = h_u^P(r_u^+)$
    set $h_u^S(r_u) = h_u^S(r_u^-)$
end if
for all vertex $x \neq u$, $x.localRoot = u.localRoot$
    if $x.V^+ = true$ **then**
        set $h_u^S(x) = h_u^S(r_u)$
    else if $x.V^- = true$ **then**
        set $h_u^P(x) = h_u^P(r_u)$
    else
        set $h_u^P(x) = h_u^S(r_u)$
        set $h_u^S(x) = h_u^P(r_u)$
    end if
end for

Phase 2: other components

if $u \neq r$ **then**
    set $h_u^P(r) = h_u^P(r_u)$
    set $h_u^S(r) = h_u^S(r_u)$
end if
for all vertex $x \neq r \wedge x \neq u$
    call SetEdge(x) # Procedure 4
end for

other vertices there is a recursion similar to a recursive lookup. Since both $h_u^P(r)$ and $h_u^S(r)$ are computed, this recursion always computes an edge for the given vertex.

Theorem 4.3.1. *Let an undirected connected graph G and vertex d be given. For all $u \in V(G)$, interleaving the edges $h_u^P(d)$ and $h_u^S(d)$ computed by Algorithm 5 makes up a pair of maximally redundant trees rooted at d.*

Proof. Let X be the computed GADAG, and let its global root be r. Let the set of clusters of X be C.

First, I prove that Algorithm 5 always terminates and computes two edges, $h_u^P(d)$ and $h_u^S(d)$, for any given vertex d. It is trivial that Phase 1 always terminates. Suppose that there is vertex d, such that there is $A \in C$, $d, u \in A$ and either $h_u^P(d)$ or $h_u^S(d)$ is still $NULL$ after Phase 1. If $u = r_d$, both traversals reach d, so $h_u^P(d)$ and $h_u^S(d)$ are set. Otherwise, if $d.localRoot = u.localRoot$, both $d.V^+$ and $d.V^-$ cannot be *true*, since in this case both BFS traversals would reach d, which is impossible, since all the cycles in A contains $r_u = r_A$. If only one of $d.V^+$ and $d.V^-$ is *true*, then one of the edges is computed by the BFS traversals, and the other one is set at Line 13 or Line 15. Since if none of them is *true*, the edges are set at Line 17, the only possibility is $d = r_u$. However, $r_u^+.V^+ = true$ and $r_u^-.V^- = true$, so both $h_u^P(r_u)$ and $h_u^S(r_u)$ are set at Line 8.

Phase 2 terminates, if Procedure 4 terminates. Since the recursion gets always to the local root, sooner or later r is reached. If $u = r$, $h_u^P(r)$ and $h_u^S(r)$ are not computed, however, all the cut-vertices in component(s) containing r are computed, so the recursion stops before reaching r. Otherwise, $h_u^P(r)$ and $h_u^S(r)$ are already computed, so Phase 2 terminates and we get edges as $h_u^P(d)$ and $h_u^S(d)$.

Next, I show that interleaving the edges makes up two paths from any vertex u to any vertex d, which are as vertex-disjoint as possible. Naturally, since for any given vertex x $h_x^P(d)$ and $h_x^S(d)$ are edges going out from x, interleaving these edges either makes up paths from u to d (reaching d is possible) or walks, which contain cycles.

If there is $A \in C$, so that $u, d \in A$, the claim is trivial, thanks to Theorem 3.4.1. Suppose that u and d are not in the same cluster. Create digraph T, and let $V(T) = C \cup \{r\}$. For all $A \in C$, $r \in V(A)$ add edge (A, r) to T (observe that such an r can only be the global root of X, since other vertices of X are not in T). Moreover, for all $A, B \in C$, $r_A = r_{r_B}$ add edge (B, A) to T. Since a local root is always on the path to r, r is reachable on a directed path from any vertex in T, so T is weakly connected. Moreover, T is a directed tree, since each cluster has only one local root, so there is only one edge going out from each $x \in V(T) \setminus \{r\}$ and no edge leaves r, so $|E(T)| = |V(T)| - 1$.

Let the cluster closest to r in T containing u be U, and similarly let the closest cluster containing d be D (here closest means that the path from U or D to r has minimum number of vertices). If U is on the path from D to r in T, Procedure 4 finds cut-vertex x in the cluster closest to D containing u, and sets $h_u^P(d) = h_u^P(x)$ and $h_u^S(d) = h_u^S(x)$. Since any path from u to d contains x, the walks leave each cluster at the right vertex, so both walks are paths and reach d. If U is not on the path from D to r, then Procedure 4 sets $h_u^P(d) = h_u^P(r)$ and $h_u^S(d) = h_u^S(r)$. In this way, the walks go up towards r in T until it reaches the first vertex P, which is either on the path from D to r or which contains d, so there is no cycle again, and the walks are paths. Since the paths inside a cluster are vertex-disjoint, the two paths are maximally vertex-disjoint. □

Next, I show that Algorithm 5 is linear in the number of edges.

Theorem 4.3.2. *The computational complexity of Algorithm 5 is $O(|E(G)|)$ for any connected graph.*

Proof. Computing GADAG D and doing the DFS traversals need $O(|E(G)|)$ time. The main question is the complexity of Procedure 4. Each time Procedure 4 is called recursively (from the procedure itself), a vertex x is needed with $h_u^P(x) = NULL$ and $h_u^S(x) = NULL$, so it can be called recursively at most $O(|V(G)|)$ times altogether. Since it is called from Algorithm 5 $|V(G)|$ times, the overall complexity of the algorithm is $O(|V(G)| + |E(G)|) = O(|E(G)|)$ (the graph is connected). □

4.4 Optimizing maximally redundant trees

In the previous section the way of computing a pair of maximally redundant trees to each vertex as a root was discussed. There, the length of paths was not taken into account, however it is very important for real applications. In this section, I present linear time heuristics to cut these lengths down. Since finding a single pair of redundant trees with short paths was already studied in [XCT02b, XCT03], here I focus on the problem of decreasing the lengths of paths found by the previous distributed algorithm.[4] Thus, although the technique presented in the sequel optimize the spanning GADAG, and such "better" GADAG would mean shorter paths for the

[4]I try to decrease the length of paths along *both* trees, so now we do not have a tree used for default forwarding. Having a tree for default forwarding is needed for techniques like LFIR (Chapter 2), but does not fit to techniques like Lightweight Not-via, an IPFRR technique being introduced in Chapter 5. Here, I optimize for the later ones.

centralized algorithm as well, I focus on the distributed algorithm here and in the next section.

As it was already mentioned, my heuristics do not increase the computational complexity of the previously discussed distributed algorithm. Minimizing the path lengths in linear complexity can only mean minimizing the number of vertices of the paths (so the edge lengths are uniform). Therefore, here and in the next section, shorter or shortest path means the path with less or minimal number of vertices. However, all the following techniques can be applied for arbitrary non-negative edge lengths, if the BFS traversals in Algorithm 5 at Line 5 and Line 6 are exchanged to two runs of Dijkstra's algorithm. Naturally, in this case the overall complexity of the algorithm is increased, it is equal to the complexity of Dijkstra's shortest path algorithm ($O(|V(G)| \log |V(G)| + |E(G)|)$).

The most important aspect influencing the number of vertices along the paths of maximally redundant trees is the spanning GADAG; using a "better" GADAG, BFS traversals can find better paths. Observe that when a GADAG is found, there can be some edges in the original graph, which are not used in either direction. Adding these edges in a direction, which keeps up the GADAG property may reduce the length of paths.

Moreover, observe that for any vertex v, optimizing the whole spanning GADAG is not necessary; it is enough to add some edges to the clusters of the GADAG, which contain v. Since paths towards vertices in different clusters are paths towards a decent cut-vertex, optimizing the paths in the local clusters optimizes all the paths.

Unfortunately, note that simply keeping up the GADAG property is not enough, since Algorithm 5 needs special spanning GADAG, which has clusters fulfilling Lemma 3.3.6. Therefore, adding edge in the direction entering to the local root must also be avoided.

Considering these observations, it is possible to construct some simple linear time heuristics for some vertex v:

- Compute GADAG D of graph G with set of clusters C.
- For all $A \in C$, where $v \in V(A)$, remove the single edge entering into r_A. Make a topological order where r_A is the minimum element. Edges of G, used in D in neither direction, can be added in a direction such that the source is the lower, the target is the higher vertex with respect to the topological order.

Trivially, in this way the GADAG property is kept up, and no new edge entering a local root is added. Moreover, since topological ordering is linear, these heuristics do not increase the complexity of the algorithm.

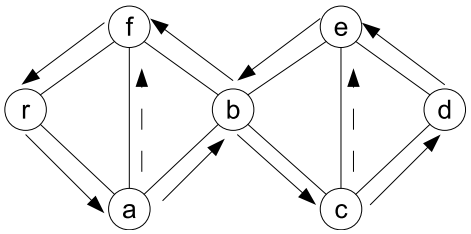


Figure 4.5. The original GADAG of a simple graph (solid arrows), and the improved GADAG (dashed and solid arrows).

Note that these heuristics can not only decrease, but also increase the number of vertices of paths in the maximally redundant trees. The optimization definitely decreases the length of paths, if two vertices are ordered. Unfortunately, if they are not ordered, the sorter paths to the local roots may make the paths between the source and the destination longer. Moreover, some unordered vertices may become ordered in the improved GADAG, in this way the path lengths can increase too. In the next section using extensive simulations, I prove that although some paths may become longer, in average there is a significant decrease.

Next, I present a simple example. Consider the network in Figure 4.5. A possible spanning GADAG is depicted in this figure by solid arrows. As one may observe, there are some edges, edge $\{a, f\}$ and $\{c, e\}$, which are not used in the GADAG in either direction, so improving is possible.

This GADAG is made up by two clusters, the first one contains r, a, b, f and the second one contains b, c, d, e. After removing the edges entering to r and b, the topological orders are $r \prec a \prec b \prec f$ and $b \prec c \prec d \prec e$. In the first component $a \prec f$ and in the second component $c \prec e$, so (a, f) and (c, e) can be added to the GADAG. Naturally, as it was discussed above, only b computes both these edges, since only b is in both clusters.

4.5 Evaluation of heuristics

Previously, some heuristics for decreasing the number of vertices of paths in maximally redundant trees were discussed. In this section, I study the efficiency of these heuristics with extensive simulations. Moreover, observe that for 2-connected graphs my results without heuristics are the same, as we would get with the original algorithm described in Section 3.4.

For the simulations, I used both the graphs of real word and artificial networks,

Network	Node number	Suurballe	Xue	Prim. path w/o heur.	Sec. path w/o heur.	Prim. path w/ heur.	Sec. path w/ heur.
Abilene	12	–	–	210%	212%	168%	171%
Germany	17	135%	136%	231%	230%	191%	190%
AT&T	22	–	–	221%	224%	166%	167%
NSF	26	121%	124%	224%	222%	178%	174%
Italy	33	–	–	248%	247%	175%	174%
Cost266	37	129%	154%	250%	253%	190%	194%
Germany50	50	118%	160%	304%	309%	212%	214%

Table 4.1. Average number of vertices along paths of maximally redundant trees in real word networks (100% is the path with minimum number of vertices).

Node number	Neighbours	Suurballe	Xue	Prim. path w/o heur.	Sec. path w/o heur.	Prim. path w/ heur.	Sec. path w/ heur.
20	2	120%	147%	217%	224%	173%	174%
20	3	116%	155%	298%	313%	180%	181%
30	2	120%	147%	235%	243%	182%	182%
30	3	115%	152%	332%	352%	190%	190%
40	2	119%	148%	250%	259%	190%	189%
40	3	114%	151%	361%	385%	198%	197%
50	2	118%	148%	263%	273%	197%	195%
50	3	113%	150%	388%	415%	205%	203%

Table 4.2. Average number of vertices along paths of maximally redundant trees in artificial networks (100% is the path with minimum number of vertices).

as previously. As real networks, I selected the Abilene, NSF, AT&T and 50 node German backbone network from [SND], and the Italian, German and European Cost266 backbone network from [GO05]. For each of these networks, I computed the maximally redundant trees with respect to each vertex as root, and I averaged the length of the resultant paths.

Random networks were generated in the same way, as in Chapter 2 and Chapter 3: the topologies were generated by BRITE [MLMB05], using Waxman algorithm, with random node placement and parameters $\alpha = 0.15$ and $\beta = 0.2$. The number of nodes varied between 20 and 50 and the number of neighbours was 2 and 3. In each case, I made 250 000 random experiments to get the expected value of the length of paths.

Since several real networks are 2-vertex-connected when no failure exists, for these topologies I computed two optimal vertex disjoint paths using Suurballe's algorithm.[5] Moreover, I also implemented the heuristics proposed by Xue *et. al.* in [XCT02b, XCT03] for minimizing the path lengths of redundant trees. The mean of the lengths of path pairs computed by these two algorithms and the lengths of paths computed by Algorithm 5 with and without heuristics are presented in Table 4.1 and Table 4.2.

One may observe that paths get significantly shorter when the heuristics proposed

[5]Recall that these paths do not make up trees.

in Section 4.4 are applied. Unfortunately, these paths are significantly longer than the optimal ones are. Thus, we can identify an interesting trade-off here: using my maximally redundant tree algorithm instead of Suurballe's algorithm or Xue's heuristics is clearly advantageous in performance-sensitive applications, because its complexity is much smaller (linear, $O(|E|)$) than that of Suurballe's algorithm (for all the vertex pairs $O(|V(G)|^3 \log |V(G)|)$) or that of Xue's heuristics (a tree rooted at each vertex is $O(|V(G)|^3(|E(G)| + |V(G)| \log |V(G)|))$). On the other hand, my technique gives suboptimal protection paths, whose length may be significantly larger than the optimal path length. My simulations reveal that the increase is at most two-fold, which not necessarily poses difficulties if these paths are only used for protection in out-of-order situations, which, supposedly, only last a couple of seconds, and the default paths can still be optimal shortest paths. But perhaps most importantly, my algorithm is much better suited to certain applications, namely those based on the hop-by-hop forwarding paradigm like IP, because in these applications we only need the next-hops along the recovery trees instead of the entire protection paths as returned by Suurballe's or Xue's algorithm. In the next chapter, I present such an application.

Chapter 5

Lightweight Not-Via

5.1 Introduction

Previously, we have discussed redundant and maximally redundant trees, in order to use them for IP Fast ReRoute. Now, in this chapter, we return to real networks, and a possible application of these graphs is presented. We focus on the IPFRR technique Not-via, and reconsider it using the possibilities of maximally redundant trees, in order to overcome the drawbacks.

As it was discussed in Chapter 1, one of the most promising IPFRR techniques is the one called Not-via [BSP10]. First, this technique is able to cover 100% of single node or link failure cases, second, it does not need significantly new hardware or protocol, third, Not-via has probably the strongest backing both in IETF and industry among the techniques, which have 100% coverage. These facts make Not-via a very likely future IPFRR standard.

As it will turn out in this chapter, despite Not-via is one of the best solutions currently, it still has important drawbacks. In order to overcome these weaknesses, I propose a new version called Lightweight Not-via, where the bases of rerouting remained intact, but the detours are completely changed by applying maximally redundant trees.

In the first part of this chapter, I introduce the original Not-via algorithm, and we discuss its drawbacks experienced in real operation of a full fledged Not-via testbed deployed at BME-TMIT [ST08, ERC+] by students I supervised. Then, I turn to introduce a modified version of Not-via, called Lightweight Not-via [C5, C6, J4, P2], which overcomes these problems. Finally, I present simulation and test results measured in a Not-via and Lightweight Not-via testbed, which confirm the efficiency of

this new technique.

5.2 IPFRR using not-via addresses

As it was discussed previously, the most important aspect of IPFRR techniques is the way of local rerouting. Not-via uses explicit marking; it puts packets into an IP-in-IP tunnel with a special destination address, sends it around the failure and puts it back to its ordinary path as soon as it safely got past the failed component. This safe decapsulation point is the so called next-next hop (NNH), the second closest node along the shortest path tree. The NNH is certainly closer to the destination than the encapsulating router, so it cannot loop back, and it is, hopefully, beyond the failed component (the next hop). Moreover, observe that in this way both node and link failures are handled, distinguishing the two types is not needed for the neighbour.

Additionally, a packet, while on detour, must be given special treatment to ensure that it bypasses the failed component. Therefore, the destination address of the tunnel describes not only the endpoint of the tunnel, but the failed resource as well. This address comes from an address space safely isolated from ordinary addresses, so that it has a distinct entry in the routing table facilitating to apply special routing decisions.

Perhaps a simple example is in order. Consider the network depicted in Figure 5.1, and suppose that a packet entering the network at node a is forwarded to the egress node c. Furthermore, assume that the shortest path (marked by bold arrows) goes through node b, but a suddenly loses contact with b. This is either because the link $\{a, b\}$ went down, or because node b failed. Repairing to the NNH, node c in this case, protects against both events, so a encapsulates the packet in a new IP header and passes it to node d. Note that if no special care were taken, then d would pass back the packet along its shortest path to a, thus forming a forwarding loop. Therefore, the encapsulated packet is destined to a special IP address, "c not via b" (denoted shortly by c_b), bearing the semantics "route this packet to node c without traversing node b in any ways". Node d accomplishes this by computing the route corresponding to c_b with b removed from the network. So d routes the packet through LAN l and node e (or f) to node c, where it is decapsulated and sent further along its normal path as if no failure had happened.

Unfortunately, the situation is not always so simple. Suppose for instance that it is now b which loses contact to c. Now, b has no NNH to pass the packet to, since c is the destination as far as b is concerned. This is the so called *last-hop problem*,

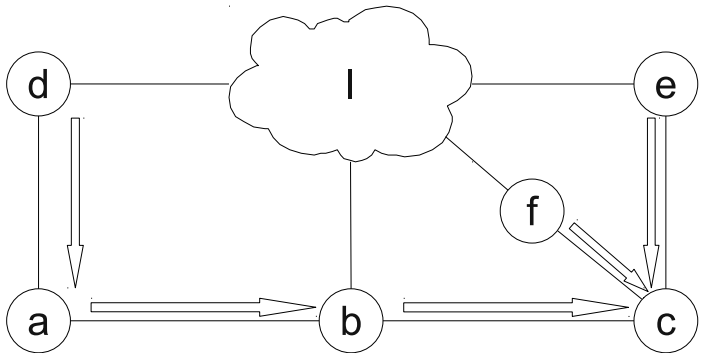


Figure 5.1. Sample network with IP routers a, b, c, d and e and LAN l. Bold arrows mark the shortest path to c.

and Not-via resolves it by simply assuming that it is not node c that went down but only the link $\{b, c\}$. Thus, it tunnels the packet to the same not-via address c_b as previously, which ensures that the packet avoids using link $\{b, c\}$. A similar problem arises when a node detects the loss of a next hop that provides the only connectivity to a certain destination. We cannot blatantly remove this next hop from the topology, since this would disconnect all possible backup paths. This situation is called the *bridge problem*, and it is handled similarly as the previous one. Finally, there is the *problem of LANs*. Suppose that node d, willing to send a packet to e through l, detects a failure in LAN l. In the simplest case, d pessimistically assumes that the entire LAN went down with all the routers attached to it. This way, we need only one not-via address to cover the failure of the LAN and we do not need costly point-to-point liveliness detection between each neighbour. In our case, however, this simple LAN repair mechanism would cause d losing all connectivity to e, even if d supposes that e is still alive (e is in the LAN too). If, on the other hand, failure detection has the granularity to distinguish particular neighbours in the LAN, then d can decide whether the LAN or a router went down; losing multiple connections is likely to be the result of a LAN failure. Therefore, if LAN l is down, d can tunnel the packet into e_l (denoting that LAN l is unavailable). However, if the shortest path from c to e is $c \to b \to a \to d \to l \to e$ and the second shortest without d is $c \to f \to l \to e$ (e.g. the direct link has very low capacity), node a would need to use another address e_d when it lose connection to d. Hence, generally we need distinct not-via addresses with respect to all possible combinations of neighbours in the LAN, plus one more protection address for each node, which describes that the LAN is down, so the number of IP addresses needed by Not-via scales quadratic with the number of nodes in LANs, as it is described in [BSP10].

Despite these issues, Not-via is still a practical and rather straight-to-the-point solution. It handles all single link and node failures, it is robust against multiple failures[1] and it has strong and stable industrial backing. In contrast to most IPFRR proposals, it does not require significant changes to legacy IP equipment. What is more, it is implementable in the fast-path. It is for these reasons that we chose Not-via to base our IPFRR testbed onto. The testbed, deployed at BME-TMIT, consists of a handful of PC routers running GNU/Linux, complete with kernel-based fast failure detection using Bidirectional Forwarding Detection [KW08], full support for communicating with the IGP to query topology information, globally synchronized transient-to-persistent failure switch-over and a distributed measurement system [TL07, ST08]. After dealing with all the intricacies of implementing the standard and experimenting with it in operation, it is possible to identify some of its pressing limitations.

Burdening address management: The first question an implementor inevitably faces is how to assign and distribute not-via addresses. As of this writing, there is no official protocol support for advertising not-via addresses into the routing domain. The situation is worsened by the fact that a not-via address has a compound meaning, as it encodes both a destination node and a component to be bypassed, and there is currently no way to communicate this rich semantics between routers. As a workaround, network operators resort to statically assigning local not-via addresses and concocting ad-hoc policies like "the IP address *.*.X.Y/32 means router X not via component Y". Such policies, however, are inflexible and subject to human configuration errors, and they break down rapidly as the network increases. Just the sheer number of not-via addresses can pose problems: the simple network of Figure 5.1 would require a total of 26 not-via addresses, which all appear in the routing tables and are all subject to individual routing calculations.

Considerable computational overhead: In an ordinary IP network, the next hops towards all destinations are obtained by a single shortest path tree (SPT) calculation. With Not-via, a router must execute as many SPT instances as there are components that can fail, with the failed component removed from the topology. Using some simple heuristics one can go down to some few dozen additional SPT calculations [LFY07], which is still significant. Note that substantial additional costs is needed to be paid due to having to deal with an increased number of entries in the routing tables, establish, maintain and tear down tunnels, etc.

Complexity and special cases: As mentioned above, Not-via brings in subtle intricacies into routing and in many cases it overrides well-known IP routing mechanisms.

[1] It is easy to detect second failure along a detour, since packets have special destination address.

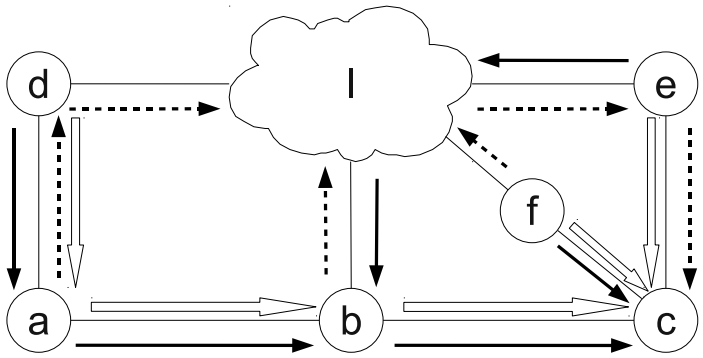


Figure 5.2. Sample network. Bold arrows mark the shortest path to C, dashed arrows mark the primary and solid arrows mark the secondary maximally redundant tree rooted at C.

The corner cases, discussed above, make implementations convoluted and operation of the protocol hardly tractable by operators.

In Section 5.4, I support the above claims with measurement results obtained on an operational IP testbed. In addition, note that similar observations were reported in the literature [LFY07]. In the next section, I propose deliberate modifications to Not-via in order to remove, or at least mitigate, these compelling issues.

5.3 An improved lightweight Not-via

Perviously the original Not-via technique was discussed. In this section, I modify the way of computing detours of Not-via by applying the concept of maximally redundant trees.

I define Not-via over maximally redundant trees in the following way (Figure 5.2). Suppose a has a packet to send to node c. As long as its default next hop, b, is alive, a simply passes the packet to b. Moreover, for resilience purposes a has computed a pair of maximally redundant trees (a primary and a secondary) rooted at c. If, however, b goes down, a must find a backup path, or at least a next hop that can push the packet further, towards c. So it encapsulates the packet and sends it along the primary tree to d. Assuming that d computed the exact same maximally redundant tree to c (which is not hard to ensure), d will pass the packet through LAN l (LANs are considered as vertices; further details in Section 5.3.2) and node e sends it to c, where it is decapsulated and sent further. If, instead, it is now node e that has to get a packet to c and it finds that connectivity to c went away, both its shortest path and its primary backup path are affected by the failure. In this case, the packet is encapsulated to

the secondary backup path and sent through l to b. Note that the secondary backup path cannot be impacted by the failure in this case, as it is node disjoint from the primary path. Finally, a packet forwarded along the primary path gets rerouted to the secondary path, when it encounters a failure (this might be the very same failure that pushed the packet to the detour in the first place) but not *vice versa*.

5.3.1 Redefining the semantics of not-via addresses

In Lightweight Not-via, a node v has only three addresses: a default routable IP address, denoted by $\mathcal{D}_v$, an IP address $\mathcal{P}_v$ that belongs to the primary tree and an IP address $\mathcal{S}_v$ that belongs to the secondary tree. The address space is, correspondingly, split into three disjunct zones: a default routable address zone $\mathcal{D}$, a primary backup address zone $\mathcal{P}$ and a secondary backup zone $\mathcal{S}$. There are distinct entries in the routing table for all three addresses for each node, and there is a common understanding between routers as to which address belongs to which zone. A router, therefore, always unambiguously knows along which path it received a packet.

In order for a router to participate in the forwarding process, it needs to compute the next hop corresponding to any of the potential destination addresses it can find in a packet. In our case, the next hop corresponding to the default routable address of some node v, $\mathcal{D}_v$, is obtained from the shortest path tree, and can be computed for all nodes in one pass spawning a single instance of Dijkstra's algorithm. The next hops for the primary and the secondary backup addresses, $\mathcal{P}_v$ and $\mathcal{S}_v$, are obtained from computing a pair of maximally redundant trees to v. As it was discussed in Chapter 4, it is possible to compute the edges going out from a given node for all the maximally redundant trees in $O(|E(G)|)$ time. Since these edges, going out from a given node, are the next hops, it is easy to compute all the needed information, and the computation will still be dominated by the Dijkstra's algorithm computing the default hops ($O(|V(G)| \log |V(G)| + |E(G)|)$). In contrast, computing the detours for the original Not-via takes $O(|V(G)|(|V(G) \log |V(G)| + |E(G)|))$ even for point-to-point networks. These ideas are presented in Algorithm 6, where $\mathrm{nh}(\mathcal{X})$ and $\mathrm{nnh}(\mathcal{X})$ represents the next hops and the next-next hops respectively towards node with address $\mathcal{X}$.

The forwarding process, responsible for passing a packet further towards the destination address, is given in Algorithm 7. Note that the operation `push` $\mathcal{X}$ in routing terminology means "encapsulate the packet into an IP-in-IP tunnel and set its outer destination address to $\mathcal{X}$". The operation $\mathcal{X} \leftarrow$ `pop` does the reverse: decapsulates the packet and puts the address of the innermost IP header to $\mathcal{X}$.

Algorithm 6 Calculation of routing entries for interior nodes at node u, given network $G(V, E)$

1: Run Dijkstra's algorithm on $G(V, E)$ to find the next hops $\mathrm{nh}(\mathcal{D}_v)$ for each $v \in V \setminus \{u\}$. If $\mathrm{nh}(\mathcal{D}_v) \neq v$ compute the next-next hop on this shortest path tree and let them be $\mathrm{nnh}(\mathcal{D}_v)$. If $\mathrm{nh}(\mathcal{D}_v) \equiv v$, let $\mathrm{nnh}(\mathcal{D}_v) = \mathrm{nh}(\mathcal{D}_v)$.
2: Run Algorithm 5 for computing the edges of redundant trees going out from u. Let the target of the primary and secondary edge of the trees rooted at a given node v be node t_v^P and t_v^S.
3: **for** each node $v \in V \setminus \{u\}$
4: $\quad \mathrm{nh}(\mathcal{P}_v) = t^P_{\mathrm{nnh}(\mathcal{D}_v)}$
5: $\quad \mathrm{nh}(\mathcal{S}_v) = t^S_{\mathrm{nnh}(\mathcal{D}_v)}$
6: **end for**

It is easy to see intuitively that this forwarding rule is correct. First, in the absence of failures, packets get to their destination along the shortest path as usual. In case of a single failure, a packet first gets to the NNH along either the primary or the secondary backup path, if the network remained connected. Both backups cannot be affected by the failure at the same time, as they are maximally redundant. So single node or link failures are handled correctly. Finally, packets cannot get into loops in the presence of multiple simultaneous failures or in single failure cases, which split the network into two, as a packet is unconditionally dropped and restoration is started when it meets a failure along the secondary path.

Without these modifications, a not-via address covers only a single failure scenario. After redefining Not-via in terms of maximally redundant trees, a not-via address protects many components: the primary backup address protects components along the default path and the secondary backup protects the primary backup. Consequently, in this way the number of necessary addresses is decreased to 2 per node, a constant per router (note that 2 not-via addresses per node is the absolute minimum achievable with the original Not-via, only realizable in point-to-point rings). This, obviously, alleviates the pain of assigning not-via addresses, helps shrinking routing tables and reduces the number of tunnels, in this way mitigating many of the address management issues traditional Not-via raises. Additionally, distributing not-via addresses with the IGP also became easier: a router can, for instance, advertise three addresses either as loopback, or as virtual stub interfaces or using some multi-topology IGP extension [PMR+07], and other routers can follow the policy that the lowest such IP address is the default, the second lowest one is the primary backup and the largest one is the secondary backup address. This was impossible with the original Not-via

Algorithm 7 Forwarding process at node u for a packet destined to address $\mathcal{A}$, given the set of unavailable neighbours F

if $\mathcal{A} = \mathcal{P}_u$ or $\mathcal{A} = \mathcal{S}_u$ **then**
    $\mathcal{A} \leftarrow$ pop
end if
if $\mathcal{A} = \mathcal{D}_u$ **then**
    consume the packet
end if
if $\mathcal{A} \in \mathcal{D}$ and $\text{nh}(\mathcal{A}) \in F$ **then**
    let v be the NNH to $\mathcal{A}$
    if $\text{nh}(\mathcal{P}_v) \notin F$ **then**
        push $\mathcal{P}_v$ and forward packet to $\text{nh}(\mathcal{P}_v)$
    else if $\text{nh}(\mathcal{S}_v) \notin F$ **then**
        push $\mathcal{S}_v$ and forward packet to $\text{nh}(\mathcal{S}_v)$
    end if
end if
if $\mathcal{A} \in \mathcal{P}$ and $\text{nh}(\mathcal{A}) \in F$ **then**
    $\mathcal{X} \leftarrow$ pop
    $\mathcal{S}_X \leftarrow$ the secondary backup address for $\mathcal{X}$
    if $\text{nh}(\mathcal{S}_X) \notin F$ **then**
        push $\mathcal{S}_X$ and forward packet to $\text{nh}(\mathcal{S}_X)$
    end if
end if
if $\text{nh}(\mathcal{A}) \notin F$ **then**
    forward packet to $\text{nh}(\mathcal{A})$
else
    drop the packet and start restoration
end if

due to the complex semantics born by not-via addresses.

What is more, in certain cases one can avoid using not-via addresses completely. In a traditional IP network, a router holds separate, globally routable IP addresses for each of its interfaces. Lightweight Not-via can safely use these addresses as not-via addresses. Say, a router has a routable loopback address (the one management uses to reach the router, or the one (i)BGP runs at, etc), and at least two interfaces with distinct, unique IP addresses. Now, we can designate the loopback as the default routable address, the smallest interface address as the primary backup address and the second smallest one as the secondary. Since these addresses are always disseminated by the IGP, other routers can easily learn which address belongs to which address zone. Note, however, that applications directly addressing any of the interfaces lose fast protection, because their traffic automatically travels through an unprotected backup path. This can be avoided by addressing routers through the loopback, as it is usually the case with applications talking to IP routers.

While in a conventional IP network this technique removes the need to maintain additional not-via addresses, it must be emphasized that it is not applicable to any arbitrary IP network. Namely, IP backbones running over unnumbered point-to-point links (e.g., MPLS LSPs) still need to maintain at least two additional not-via addresses per router, since interfaces usually don't have unique IP addresses assigned in such cases.

5.3.2 Removing corner cases

Not-via has some subtle details, making it more difficult to implement correctly and understand in operation. Though, redefining Not-via in terms of maximally redundant trees removes most of the corner cases.

For instance, bridge problem is immediately solved, since maximally redundant trees pass through bridges. Moreover, LAN problem is solved too. It is possible to consider a LAN as a vertex in the graph of the network, and we can compute a pair of maximally redundant trees in this new graph as well. Since paths on maximally redundant trees try to avoid common vertices, one of the trees always bypass the LAN if it is possible. One may observe that in this way the only difference between vertices representing IP nodes in the network and vertices representing LANs is that it is not needed to compute a pair of redundant trees rooted in LAN vertices.

At first one may think that Algorithm 5 may have difficulties when needs to deal with LANs, since sometimes the edges computed by this algorithm would terminate in a LAN vertex, which cannot tell the next hop in IP network. However, this problem

can be easily handled by modifying the two BFS traversals at Line 5 and Line 6 in such a way that when they get to a node x after leaving a LAN vertex neighbouring starting vertex u (in this case x must be a router), $h_u^P(x)$ or $h_u^S(x)$ should be the edge leaving the LAN vertex, not the previous one entering to it.

Unfortunately, maximally redundant trees do not remove the last hop problem. This can be treated as the original technique does: if there is no next-next hop (since the next hop is the destination), try to send the packet to the next hop on a detour.

5.3.3 The endpoints of detours

Observe that there are at least three candidates for the endpoints of detours. Although the original Not-via was limited to select the next-next hop, in order to more or less limit the number of protection IP addresses, we can tunnel to the destination (the egress router in a transport network), the next-next hop or the first node closer to the destination. All the possibilities have their pros and cons.

Tunneling to the destination helps further lowering the number of IP addresses; if the endpoints of a detours can only be egress routers, only these routers need extra IP addresses, in this way the number of IP addresses, management cost, computational cost (less maximally redundant trees are needed) and forwarding table entries can be decreased. On the other hand, these paths would mean that the endpoints of detours get quite far away, and since these detours are longer than the shortest paths (Section 4.5), detours may become unnecessarily long.

Tunneling to next-next hop (as Not-via does) keeps the detours relatively local, and the endpoint selection simple. The most important complication with this possibility raises, when the next-next hop is a LAN[2], which cannot decapsulate packets. In this case the next-next IP hop must be selected (which is the next-next-next hop on the shortest path in the graph containing both routers and LANs). Considering even this special case (what is much simpler than assigning extra addresses to LANs as Not-via does), it is possible to keep the protection overhead at a low level even in this way.

Undoubtedly, the best endpoint candidate would be the nearest node along the maximally redundant trees, which never sends the packets back, which is definitely closer to the destination. Unfortunately, since there is no strict connection between the shortest paths and the paths along the maximally redundant trees, the simplest way to find this node is to check all the trees inside the same cluster of the GADAG.

[2]Recall, that LAN problem is handled by considering LANs as vertices.

Naturally, this is quite complicated, and can easily increase even the computational complexity.

Considering the reasoning above, Lightweight Not-via is defined so that it selects the next-next hop as an endpoint. Therefore, in the next section, the performance evaluation of this version is presented.

5.4 Performance evaluation

It is sure that it is not some deep theoretical limitation or trade-off that hampers the wide-scale deployment of IPFRR the most, but rather a couple of very technical and very concrete practical issues. In order to confirm this claim, we implemented and tested both the prevailing IPFRR proposal, Not-via, and also Lightweight Not-via in an operational IP testbed. In the rest of this chapter, I report on the most important observations.

The test system is a full-fledged Not-via prototype, deployed on 9 PC routers running a stock Debian GNU/Linux distribution, the Open Shortest Path First routing protocol (OSPF) from the `Quagga` suite of routing daemons [Qua] and `kbfd`, a kernel-based implementation of the Bidirectional Forwarding Detection [KW08] protocol. In order to be able to react to failures as rapidly as possible, a Not-via daemon takes over the responsibility of failure detection and forwarding table maintenance: it keeps a BFD session with all neighbours, learns topology information from OSPF, computes the next hops corresponding to all ordinary and not-via addresses and it builds a distinct forwarding table with respect to every potentially failing neighbour (plus the default table). This way, when BFD signals the loss of contact to one of the neighbours, the router simply switch to the corresponding forwarding table without having to selectively update the entries affected by the failure one by one. Additionally, in the case of a persistent topology change, the forwarding tables are completely rebuilt. Although this implementation strategy leaves some quite obvious room for potential improvement, I still believe that my results are indicative as to how much resource a streamlined IPFRR implementation actually uses. In addition, this design makes it possible to modularize the code and hence to easily incorporate Lightweight Not-via proposal.

My first experiences were aimed at measuring the raw failure recovery speed. I found that IP Fast ReRoute is just what it promises to be: fast. Configuring BFD so that any failure is detected in at most 9 ms, but no sooner than 6 ms (BFD interval = 3 ms, BFD multiplier = 3), Not-via repairs single link and node failures in 16.65

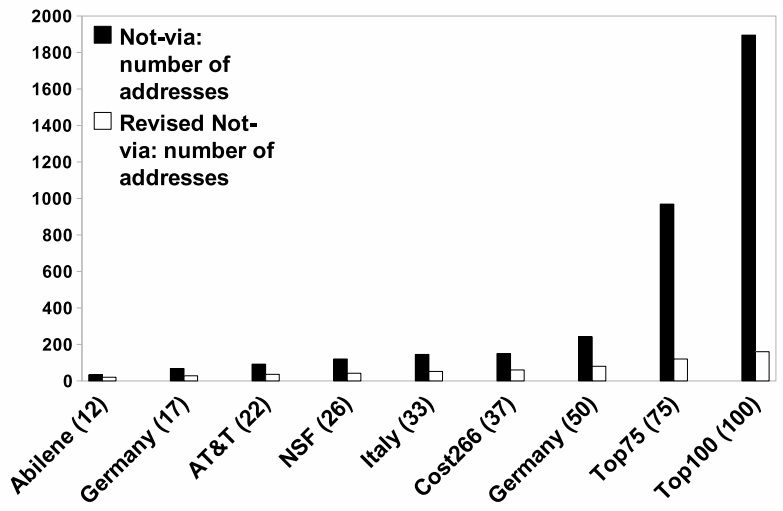


Figure 5.3. Number of additional addresses for the original and lightweight Not-via in commonplace ISP topologies (number of nodes is given in parentheses), with every fifth node substituted by a LAN.

ms on average and 18.5 ms at maximum, irrespectively of the actual topology or the nature and the location of the failure. With conventional OSPF, on the other hand, one can measure anything between 120 ms to several seconds depending on the specifics of the topology, the configuration of the routing protocol, the actual failure detection technique, etc. This fact may alone motivate IPFRR enough.

Next, I studied how many additional addresses Not-via needs. I chose some network topologies, which were previously used: the Abilene, NSF and the AT&T topologies from [SND]; the German (Germany), Italian (Italy) and the European (Cost266) backbone topologies from [GO05]; an extended 50 node version of the German backbone (Germany50, [SND]); plus two random network topologies: one of 75 nodes (Top75) and one of 100 nodes (Top100), both generated by the BRITE tool [MLMB05] using the router-level Waxman model ($m = 4$). Naturally, it was impossible to build up such huge networks like the previous ones with the limited number of PCs I had, so I modified the code, and made it possible to inject topologies into a router; in this way the router executed exactly the same computation as the topology was the studied one, only the topology exploring part of the operation was simulated. Since the type of the links is not specified in these topologies, I repeated the measurements first with every link set as a point-to-point link, and then with substituting every fifth node (20%) with a LAN connecting the neighbours of the node.

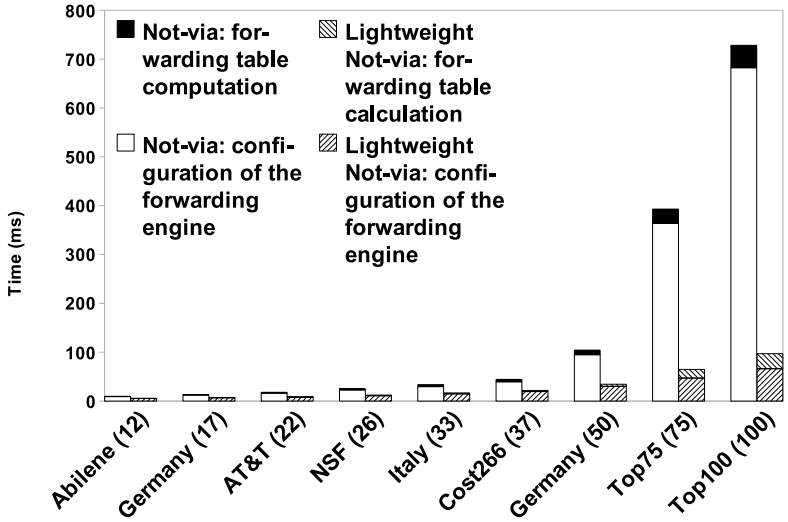


Figure 5.4. Execution time of computing the forwarding tables and configuring the forwarding engine for the original and the Lightweight Not-via, with every fifth node substituted by a LAN.

			Not-via					Lightweight Not-via				
Topology			w/o LANs		w/ LANs		alt to	w/o LANs		w/ LANs		alt to
Name	node	link	addr	route	addr	route	nnh	addr	route	addr	route	nnh
Abilene	12	15	30	196	33	260	3.7	24	187	20	180	5.3
Germany	17	26	50	318	68	371	3.6	34	272	28	238	5.8
AT&T	22	39	76	460	92	470	3.2	44	357	36	324	5
NSF	26	44	88	542	120	612	3.5	52	425	42	370	6.1
Italy	33	56	112	703	145	780	3.7	66	544	52	478	6.3
Cost266	37	57	114	736	150	1078	3.9	74	612	60	628	7
Germany50	50	88	176	1110	242	2576	3.8	100	833	80	1102	7.2
Top75	75	300	600	3330	969	9098	2.6	150	1258	120	1860	5.2
Top100	100	400	800	4457	1895	17490	2.6	200	1683	160	2490	5.8

Table 5.1. Forwarding table operations (route), the total number of not-via addresses in the network (addr) and the number of vertices along the alternative paths to the next-next hops.

My measurements were primarily aimed at identifying the management cost of Not-via. I found that considerable management complexity arises from the need to hand out and maintain vast numbers of not-via addresses. Figure 5.3 gives this number for both the original Not-via and Lightweight Not-via, as computed by the prototype system for some commonplace ISP topologies (exact values are presented in Table 5.1). Observe that using lightweight Not-via the number of additional addresses remains modest even in very large topologies with LANs.

Obviously, configuring several thousands of not-via addresses by hand is next to impossible, and it remains cumbersome and prone to human errors even using some centralized network management software. The problem is worsened by the need to retain the compound semantics of not-via addresses in a consistent manner all over the network. But it is not only central network management that is overwhelmed by the sheer volume of not-via addresses: just dealing with so many addresses can overload even the IP routers themselves. Every single not-via address handed out in the network comes at a high price: a distinct forwarding table entry must be computed and configured, an IP-in-IP tunnel needs to be set up for those addresses that can potentially be local or remote endpoints of detours, etc. Table 5.1 gives an idea of the magnitude of the address management load, in terms of the number of forwarding table operations involved. Results are given for the original Not-via and the Lightweight Not-via both with and without LANs. Note that it is only necessary to execute all these steps when the topology changes persistently, but even in this case managing so many forwarding entries can be a tedious task. To confirm this claim, I measured the time spent by a router from computing the next hops until the forwarding entries are all downloaded into the forwarding engine. The results are given in Figure 5.4.

The time needed to compute the forwarding tables for the original Not-via grows dramatically in increasingly sized networks, to the point that it takes tens of milliseconds for larger topologies. My measurements indicated a very visible improvement with the Lightweight Not-via in this regard, thanks to the technique for finding maximally redundant trees in linear time, which was introduced in the previous chapter. However, forwarding table calculation time straight-out vanishes when compared to the amount of related management work: configuring the forwarding engine with several thousand entries easily bogs down a router for half a second or even more. While this observation might be surprising, it is in line with the rest of the literature [SG01].

Finally, observe that original Not-via uses always a shortest possible path for getting to the next-next hop without using the next-hop. Unfortunately, paths along

redundant trees applied by Lightweight Not-via must be longer even if heuristics proposed in Section 4.5 are applied, which brings up a trade-off: while Lightweight Not-via significantly decreases management and computational complexity, it increases the lengths of these detours (see Table 5.1). However, as my results have shown, this increment takes only some extra hops (about 2–3), and since these paths are used only for a short time while restoration reconfigures the network, this drawback can be acceptable for most networks.

Thus, my measurement results cast Not-via in a completely different light: although the computational complexity of Not-via is substantial, yet it is the extra management burden caused by the extension of the address pool that dominates its complexity. My measurements reproduce this burden spectacularly even in small and middle-sized topologies, and we can expect it to become prohibitive in larger networks. On the other hand, it is exactly this burden where the advantages of the Lightweight Not-via really manifest themselves: the time of computing the next hops and configuring the forwarding engine decreases by an order of magnitude into the range of some few hundred milliseconds, which falls well within the time range contemporary IP routers perform ordinary shortest path routing [SG01]. I believe that these advantages can easily compensate that Lightweight Not-via use slightly longer detours.

Chapter 6

Conclusion

In this dissertation, I studied the efficiency of IP Fast ReRoute proposals. These mechanisms are currently studied heavily, because of the increasing importance of IP networks. Since these networks suffer from the lack of a native protection scheme, operators are forced to use some extra network layers (e.g., MPLS) in order to fulfil QoS requirements imposed on IP networks nowadays. Since using MPLS just for providing fast convergence is not acceptable for several operators, currently serious efforts are being taken in order to endow IP with protection capability.

In the first chapter, we reviewed the the principles of IPFRR, and current proposals. Then, we observed, that there are several requirements a modern IPFRR technique must fulfil. We also found that none of the current proposals is able to fulfil all of these requirements. Based on these observations, I was able to construct better solutions.

In Chapter 2, the possibilities of interface-based forwarding were discussed. Although it is hard to realize such a router, since the forwarding mechanism is needed to be changed, it is definitely possible. We found that the most important drawback of these techniques is their unavoidable prone to form loops, if shortest paths are used for default forwarding. Next, based on this observation, I proposed a technique, which can always avoid loops for the price of a bit longer paths.

However, the most important finding of Chapter 2 is not LFIR itself, but the observation that decent spanning trees can be extremely well applied for IPFRR. Therefore, in Chapter 3 and Chapter 4, we discussed the well studied field of redundant trees. Here, I proposed the first linear time algorithm, capable to find a pair of redundant trees rooted at *each* vertex. Moreover, I generalized the concept of redundant trees, introduced maximally redundant trees, and proposed linear time algorithms for finding maximally redundant trees both with centralized and distributed

manner.

Finally, I applied maximally redundant trees for creating Lightweight Not-via, an IPFRR technique improving Not-via. Considering my research objectives (Section 1.4), Lightweight Not-via is the only proposal capable to fulfil all the requirements. Simple solutions are immediately ruled out by the requirement of 100% single failure coverage. Not-via and MRC need too many addresses, which raises heavy management problems. Most of the interface-based mechanisms are unacceptable too, since they can create FRR loops. The technique presented in [KRKH09] is not acceptable, since it is not able to handle node failures. Although IPRT is very close to Lightweight Not-via, unfortunately, this technique is not capable to handle networks, which are not 2-vertex-connected.

LFIR is not capable to handle node failures, albeit it is theoretically possible to exchange edge-disjoint branchings to maximally redundant trees. However, observe that this revised LFIR would still have an important drawback compared to Lightweight Not-via: it would use interface based forwarding, which requires a possible, but still quite difficult change in the forwarding engine, and avoiding the changes as much as possible was one of the main requests.

6.1 Further possibilities

Currently, there is a high pressure on router vendors: on one hand, several operators need native IP protection in order to avoid using MPLS and guarantee QoS criteria in the same time, while on the other hand, as it turned out from this dissertation, the industry has not yet found a satisfying standard. Therefore, router vendors like Cisco, Juniper, Alcatel-Lucent or Ericsson (formerly RedBack) have already implemented or just implementing LFA into there products, but naturally, LFA cannot be a permanent solution with its very limited protection capability.

Hence, Ericsson has started its own research project, where I continue my work. The results of this and other projects, which undoubtedly run at other companies, will be applied in real networks in some years. It seems that the time of IPFRR has just come; it has got out from universities and research labs, and we have a great chance that IP finally gets rid off one of its fundamental weaknesses.

Index

References

[ABS96] F. Annexstein, K. Berman, and R. Swaminathan. Independent spanning trees with small stretch factors. Technical report, 1996.

[AJY00] C. Alaettinoglu, V. Jacobson, and H. Yu. Towards milli-second igp convergence. Internet Draft, available online: `http://tools.ietf.org/html/draft-alaettinoglu-isis-convergence-00`, November 2000.

[Atl06] A. Atlas. U-turn alternates for ip/ldp fast-reroute. Internet Draft, available online: `http://tools.ietf.org/html/draft-atlas-ip-local-protect-uturn-03`, February 2006.

[AZ08] A. Atlas and A. Zinin. Basic specification for IP Fast-Reroute: Loop-Free Alternates. Internet Engineering Task Force: RFC 5286, March 2008.

[BFPS05] S. Bryant, C. Filsfils, S. Previdi, and M. Shand. IP fast-reroute using tunnels. Internet Draft, available online: `http://tools.ietf.org/html/draft-bryant-ipfrr-tunnels-03`, April 2005.

[BR06] R. Balasubramanian and S. Ramasubramanian. Minimizing average path cost in colored trees for disjoint multipath routing. In *15th International Conference on Computer Communications and Networks, ICCCN 2006*, pages 185–190, October 2006.

[BSP10] S. Bryant, M. Shand, and S. Previdi. IP fast reroute using Not-via addresses. Internet Draft, available online: `http://tools.ietf.org/html/draft-ietf-rtgwg-ipfrr-notvia-addresses-06`, 2010.

[CB92] B. Chinoy and H. W. Braun. The National Science Foundation network. Tech. Rep., CAIDA, available online: `http://www.caida.org/outreach/papers/1992/nsfn/nsfnet-t1-technology.pdf`, Sep 1992.

[CHA07] T. Cicic, A. F. Hansen, and O. K. Apeland. Redundant trees for fast IP recovery. In *Broadnets*, pages 152–159, 2007.

[CHK+10] T. Cicic, A. F. Hansen, A. Kvalbein, M. Hartmann, R. Martin, M. Menth, S. Gjessing, and O. Lysne. Relaxed multiple routing configurations: IP fast reroute for single and correlated failures. IAccepted for publication, IEEE Transactions on Network and Service Management, available online: `http://www3.informatik.uni-wuerzburg.de/staff/menth/Publications/papers/Menth08-Sub-4.pdf`, September 2010.

[Cic06] T. Cicic. An upper bound on the state requirements of link-fault tolerant multi-topology routing. In Lutfi Yenel, editor, *International Conference on Communications (ICC)*, page Electronic. IEEE, 2006.

[CLY03] S. Curran, O. Lee, and X. Yu. Chain decompositions and independent trees in 4-connected graphs. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 186–191, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[CLY06] S. Curran, O. Lee, and X. Yu. Finding four independent trees. *SIAM Journal on Computing*, 35(5):1023–1058, 2006.

[Edm73] J. Edmonds. Edge-disjoint branchings. *Combinatorial Algorithms*, pages 91–96, 1973.

[ERC+] G. Enyedi, G. Rétvári, A. Császár, P. Szilágyi, and Z. Tóth. Instant fault recovery in IP networks. Presentation and demo at the High Speed Networking Workshop, Balatonkenese, Hungary, May 2008.

[ET76] S. Even and R. E. Tarjan. Computing an st-numbering. *Theoretical Computer Science*, (2), 1976.

[FB05] Pierre Francois and Olivier Bonaventure. An evaluation of ip-based fast reroute techniques. In *Proceedings of ACM CoNext 2005*, 2005.

[FFEB05] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure. Achieving sub-second igp convergence in large IP networks. *SIGCOMM Comput. Commun. Rev.*, 35(3):35–44, 2005.

[FHHK06] B. Fenner, M. Handley, H. Holbrook, and I. Kouvela. Protocol independent multicast - sparse mode (pim-sm): Protocol specification (revised). Internet Engineering Task Force: RFC 4601, August 2006.

[fS02] International Organization for Standardization. OSI IS-IS intra-domain routing protocol. ISO/IEC 10589:2002, 2002.

[Gjo07] M. Gjoka. Evaluation of IP fast reroute proposals. In *Proceedings of IEEE Comsware*, 2007.

[GO05] M. L. Garcia-Osma. TID scenarios for advanced resilience. Tech. Rep., The NOBEL Project, Work Package 2, Activity A.2.1, Advanced Resilience Study Group, Sep 2005.

[Han98] Dagmar Handke. Independent tree spanners. In *Graph-Theoretic Concepts in Computer Science*, pages 203–214, 1998.

[Huc94] A. Huck. Independent trees in graphs. *Graphs and Combinatorics*, 10(1):29–45, 1994.

[ICM+02] G. Iannaccone, C. Chuah, R. Mortier, S. Bhattacharyya, and C. Diot. Analysis of link failures in an IP backbone. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 237–242, New York, NY, USA, 2002. ACM.

[IR84] A. Itai and M. Rodeh. The multi-tree approach to reliability in distributed networks. In *SFCS '84: Proceedings of the 25th Annual Symposium onFoundations of Computer Science, 1984*, pages 137–147, Washington, DC, USA, 1984. IEEE Computer Society.

[JRY09] G. Jayavelu, S. Ramasubramanian, and O. Younis. Maintaining colored trees for disjoint multipath routing under node failures. *IEEE/ACM Transactions on Networking*, 17(1):346–359, 2009.

[KHC+06] A. Kvalbein, A. F. Hansen, T. Cicic, S. Gjessing, and O. Lysne. Fast IP network recovery using multiple routing configurations. In *Proc. of the 25th International Conference on Computer Communications (IEEE INFOCOM)*, 2006.

[KHC+08] A. Kvalbein, A. F. Hansen, T. Cicic, S. Gjessing, and O. Lysne. Fast IP network recovery using multiple routing configurations. In *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, 2008.

[KHv+09] Amund Kvalbein, Audun Fosselie Hansen, Tarik Čičic, Stein Gjessing, and Olav Lysne. Multiple routing configurations for fast IP network recovery. *IEEE/ACM Trans. Netw.*, 17(2):473–486, 2009.

[KRKH09] S. Kini, S Ramasubramanian, A. Kvalbein, and A. F. Hansen. Fast recovery from dual link failures in IP networks. INFOCOM 2009, April 2009.

[KW08] D. Katz and D. Ward. Bidirectional forwarding detection. Internet Draft, available online: `http://tools.ietf.org/html/draft-ietf-bfd-base-08`, March 2008.

[LFY07] A. Li, P. Francois, and X. Yang. On improving the efficiency and manageability of NotVia. In *Proceedings of ACM CoNEXT*, pages 1–12, 2007.

[LLW+09] R. Luebben, G. Li, D. Wang, R. Doverspike, and X. Fu. Fast rerouting for IP multicast in managed iptv networks. In *17th International Workshop on Quality of Service, 2009. IWQoS.*, pages 1–5, July 2009.

[Lov76] L. Lovász. On two minimax theorems in graph theory. *Journal of Combinatorial Theory*, pages 96–103, 1976.

[LYN+04] S. Lee, Y. Yu, S. Nelakuditi, Z.-L. Zhang, and C.-N. Chuah. Proactive vs reactive approaches to failure resilient routing. In *IEEE Infocom'04*, 2004.

[MBFG99] M. Médard, R. A. Barry, S. G. Finn, and R. G. Galler. Redundant trees for preplanned recovery in arbitary vertex-redundant or edge-redundant graphs. *IEEE/ACM Transactions on Networking*, 7(5):641–652, Oct 1999.

[MIB+04] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C. Chuah, and C. Diot. Characterization of failures in an IP backbone. In *IEEE Infocom'04*, pages 2307–2317, March 2004.

[MLMB05] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: Boston university Representative Internet Topology gEnerator. `http://www.cs.bu.edu/brite`, 2005.

[Moy98] J. Moy. OSPF version 2. Internet Engineering Task Force: RFC 2328, April 1998.

[MP06] E. Mannie and D. Papadimitriou. Recovery (protection and restoration) terminology for generalized multi-protocol label switching (gmpls). Internet Engineering Task Force: RFC 4427, March 2006.

[MTSIN98] K. Miura, D. Takahashi, S.-I.Nakano, and T. Nishizeki. A linear-time algorithm to find four independent spanning trees in four-connected planar graphs. In *WG '98: Proceedings of the 24th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 310–323, London, UK, 1998. Springer-Verlag.

[NLY+07] S. Nelakuditi, S. Lee, Y. Yu, Z.-L. Zhang, and C-N. Chuah. Fast local rerouting for handling transient link failures. *IEEE/ACM Transaction on Networking*, 15(2):359–372, 2007.

[NLYZ03] S. Nelakuditi, S. Lee, Y. Yu, and Z.-L. Zhang. Failure insensitive routing for ensuring service availability. In *Proceedings International Workshopon Quality of Service (IWQoS)*, 2003.

[PMR+07] P. Psenak, S. Mirtorabi, A. Roy, L. Nguyen, and P. Pillay-Esnault. Multi-topology (MT) routing in OSPF. Internet Engineering Task Force: RFC 4915, June 2007.

[Pos81] J. Postel. Internet protocol. Internet Engineering Task Force: RFC 791, September 1981.

[PSA05] P. Pan, G. Swallow, and A. Atlas. Fast reroute extensions to rsvp-te for lsp tunnels. Internet Engineering Task Force: RFC 4090, May 2005.

[Qua] GNU Quagga routing software. `http://www.quagga.net`.

[Ram04] S. Ramasubramanian. Supporting multiple protection strategies in optical networks. Technical report, Department of Electrical and Computer Engineering, University of Arizona, November 2004.

[RHK06a] S. Ramasubramanian, M. Harkara, and M. Krunz. Distributed linear time construction of colored trees for disjoint multipath routing. In *IFIP Networking*, pages 1026–1038, May 2006.

[RHK06b] S. Ramasubramanian, M. Harkara, and M. Krunz. Distributed linear time construction of colored trees for disjoint multipath routing. 5th International IFIP-TC6 Networking Conference, May 2006.

[RKK07] S. Ramasubramanian, H. Krishnamoorthy, and M. Krunz. Disjoint multipath routing using colored trees. *Computer Networks*, 51(8):2163–2180, 2007.

[RSM03] S. Ramamurthy, L. Sahasrabuddhe, and B. Mukherjee. Survivable wdm mesh networks. *Journal of Lightwave Technology*, 21(4):870–883, 2003.

[SB10a] M. Shand and S. Bryant. A framework for loop-free convergence. Internet Engineering Task Force: RFC 5715, January 2010.

[SB10b] M. Shand and S. Bryant. IP Fast Reroute framework. Internet Engineering Task Force: RFC 5714, January 2010.

[SG01] A. Shaikh and A. Greenberg. Experience in black-box OSPF measurement. In *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 113–125, 2001.

[SND] Survivable fixed telecommunication Network Design library (SNDlib). `http://sndlib.zib.de`.

[SRM02] L. Sahasrabuddhe, S. Ramamurthy, and B. Mukherjee. Fault management in IP-Over-WDM networks: Wdm protection versus IP restoration. *IEEE Journal on Selected Areas in Communications*, 20(1):21–33, January 2002.

[ST08] P. Szilágyi and Z. Tóth. Design, implementation and evaluation of an IP Fast ReRoute prototype. Technical report, BME, 2008. to appear at Scientific Student Conference'08, available online: `http://opti.tmit.bme.hu/~enyedi/ipfrr/`.

[TH00] D. Thaler and D. Hopps. Multipath issues in unicast and multicast next-hop selection. Internet Engineering Task Force: RFC 2991, November 2000.

[TL07] Z. Tóth and A. B. Lajtha. Fast failure recovery in IP networks. Technical report, BME, November 2007. at Scientific Student Conference'07, in Hungarian, available online: `http://opti.tmit.bme.hu/~enyedi/ipfrr/`.

[TRK06] P. Thulasiraman, S. Ramasubramanian, and M. Krunz. Disjoint multipath routing in dual homing networks using colored trees. In *IEEE Globecom*, pages 1–5, November/December 2006.

[VPD04] J. Vasseur, M. Pickavet, and P. Demeester. *Network Recovery: Protection and Restoration of Optical, SONET-SDH, IP, and MPLS*. Elsevier, 2004.

[WN07] J. Wand and S. Nelakuditi. IP fast reroute with failure inferencing. In *Proc. of ACM SIGCOMM Workshop on Internet Network Management – The Five-Nines Workshop*, 2007.

[WZN06] J. Wang, Z. Zhong, and S. Nelakuditi. Handling multiple network failures through interface specific forwarding. In *IEEE Globecom*, November 2006.

[XCT02a] G. Xue, L. Chen, and K. Thulasiraman. Cost minimization in redundant trees for protection in vertex-redundant or edge-redundant graphs. In *PCC '02: Proceedings of the Performance, Computing, and Communications Conference, 2002. on 21st IEEE International*, pages 187–194, Washington, DC, USA, 2002. IEEE Computer Society.

[XCT02b] G. Xue, L. Chen, and K. Thulasiraman. Delay reduction in redundant trees for preplanned protection against single link/node failure in 2-connected graphs. In *IEEE Globecom*, November 2002.

[XCT02c] G. Xue, L. Chen, and K. Thulasiraman. QoS issues in redundant trees for protection in vertex-redundant or edge-redundant graphs. In *IEEE International Conference on Communications (ICC)*, volume 5, pages 2766–2770, 2002.

[XCT03] G. Xue, L. Chen, and K. Thulasiraman. Quality-of-service and quality-of-protection issues in preplanned recovery schemes using redundant trees. *IEEE Journal on Selected Areas in Communications*, 21(8):1332–1345, October 2003.

[ZI89] A. Zehavi and A. Itai. Three tree-paths. *Journal of Graph Theory*, 13(2):175–188, 1989.

[ZNY+05] Z. Zhong, S. Nelakuditi, Y. Yu, S. Lee, J. Wang, and C-N Chuah. Failure inferencing based fast rerouting for handling transient link and node failures. In *IEEE Infocom'05*, 2005.

[ZXTT05] W. Zhang, G. Xue, J. Tang, and K. Thulasiraman. Linear time construction of redundant trees for recovery schemes enhancing QoP and QoS. INFOCOM 2005, March 2005.

[ZXTT08] W. Zhang, G. Xue, J. Tang, and K. Thulasiraman. Faster algorithms for construction of recovery trees enhancing QoP and QoS. *IEEE/ACM Trans on Networking*, 16(3):642–655, 2008.

Publication of new results

[J] **Journals**

[J1] András Császár, **Gábor Enyedi**, Gábor Rétvári, Marcus Hidell, Peter Sjödin, "Converging the Evolution of Router Architectures and IP Networks", *IEEE Network Magazine*, Special Issue on Advances in Network Systems Architecture, 21:4(8–14) 2007.

[J2] Péter Fodor, **Gábor Enyedi**, Gábor Rétvári, Tibor Cinkler, "Layer-Preference Policies in Multi-layer GMPLS Networks", *Photonic Network Communications* 2009.

[J3] **Gábor Enyedi**, Gábor Rétvári, "Gyors hibajavítás IP hálózatokban (Hungarian)", *Híradástechnika*, 64:3–4(20–24) 2009.

[J4] **Gábor Enyedi**, Gábor Rétvári, "Finding Multiple Maximally Redundant Trees in Linear Time", *Accepted to Periodica Polytechnica Electrical Engineering*, available online: `http://opti.tmit.bme.hu/~enyedi/ipfrr/`, 2010.

[C] **Conferences**

[C1] Péter Fodor, **Gábor Enyedi**, Tibor Cinkler, "A Fast and Efficient Traffic Engineering Method for Transport Networks", V Workshop in G/MPLS Networks (WGN5), pages 129–141, 2006.

[C2] **Gábor Enyedi**, Gábor Rétvári, Tibor Cinkler, "A Novel Loop-free IP Fast Reroute Algorithm", 13th EUNICE Open European Summer School and IFIP TC6.6 Workshop on Dependable and Adaptable Networks and Services (EUNICE), Twente, The Netherlands, 2007. BEST PAPER AWARD

[C3] **Gábor Enyedi**, Gábor Rétvári "A Loop-Free Interface-Based Fast Reroute Technique", 4th EURO-NGI Conf. on Next Generation Internet Networks (EuroNGI), Krakków, Poland, pages 39–44, 2008.

[C4] Péter Fodor, **Gábor Enyedi**, Gábor Rétvári, Tibor Cinkler, "An Efficient and Practical Layer-preference Policy for Routing in GMPLS Networks", 13th Int. Telecommunications Network Strategy and Planning Symposium (NETWORKS), Budapest, Hungary, 2008.

[C5] **Gábor Enyedi**, Péter Szilágyi, Gábor Rétvári, András Császár, "IP Fast ReRoute: Lightweight Not-Via without Additional Addresses", IEEE INFOCOM-MiniConference, Rio de Janeiro, Brazil, April 2009.

[C6] **Gábor Enyedi**, Gábor Rétvári, Péter Szilágyi, András Császár, "IP Fast ReRoute: Lightweight Not-Via", IFIP Networking, Aachen, Germany, May 2009.

[C7] **Gábor Enyedi**, Gábor Rétvári, András Császár, "On Finding Maximally Redundant Trees in Strictly Linear Time", IEEE Symposium on Computers and Communications, ISCC, Sousse, Tunisia, July 2009.

[C8] Gábor Rétvári, János Tapolcai, **Gábor Enyedi**, András Császár "IP Fast ReRoute: Loop-free Alternates Revisited", Accepted to IEEE INFOCOM, available online: `http://opti.tmit.bme.hu/~enyedi/ipfrr/`, Shanghai, China, April 2011.

[P] **Patent Applications**

[P1] András Császár, **Gábor Enyedi**, "Link failure recovery method and apparatus", Patent application WO/2009/010090, 2009.

[P2] András Császár, **Gábor Enyedi**, "A System and Method of Implementing Lightweight Not-via IP Fast ReRoutes in a Telecommunications Network", Patent application WO/2010/055408, 2010.

[P3] András Császár, **Gábor Enyedi**, "Fast Path Notification", Patent application PCT/EP2010/059391, 2010.

[P4] András Császár, **Gábor Enyedi**, "IP Fast ReRoute Relying on Fast Path Notification", Patent application PCT/EP2010/065040, 2010.