Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science
Network Architectures and Services Group

Diverse routing in SRLG networks

Rob Juffermans

September 2009

A thesis submitted in partial fulfillment of
the requirements for the degree of

Master of Science

Chairman	: prof.dr.ir. P. Van Mieghem
Supervisor	: dr.ir. F.A. Kuipers
Daily supervisors	: ir. A.A. Beshir
	: ir. R. van der Pol
	: dr.ir. F. Dijkstra
Date of defense:	25 september 2009
Thesis no:	PVM 2009-059

Acknowledgment

This project was carried out from October 2008 till June 2009 at SARA in Amsterdam and the Network Architectures and Services (NAS) Group of the Faculty of Electrical Engineering, Mathematics and Computer Science of the Delft University of Technology. I would like to thank Ronald van der Pol and Freek Dijkstra from SARA for their time and insightful comments. I also would like to thank prof.dr.ir. Piet Van Mieghem, dr.ir. Fernando Kuipers and ir. Anteneh Beshir from the TU Delft for their help and ideas which made this project successful. Finally, I would like to thank my girlfriend Marjolein for being a great support during the project.

I dedicate this thesis to Bart Kortekaas who guided me in the right direction.

Abstract

As our dependency on communication increases so is the demand for protecting these communication lines. To provide failure-safe connections in optical networks, lightpaths can be protected. Protected lightpaths consist of a primary path and a backup path which are disjoint and ensure connection continuity in case of a single link failure. Optical networks consist of at least two layers, the optical layer and the physical layer. And although the primary and backup paths are disjoint in the optical layer, in the physical layer they may share the same fiber span or duct. These links are in the same Shared Risk Link Group (SRLG). If one link of a SRLG fails then all fail. Because of this, a single failure at the physical layer can causes multiple failures at the optical layer and protected paths could get disconnected if both paths have fibers in the broken fiber span. This thesis proposes an exact algorithm which finds the shortest SRLG-disjoint protected path in a network through an iterative approach.

Contents

Used symbols

Symbol	Definition
$\mathcal{G}$	Network
$\mathcal{N}$	Set of nodes of network $\mathcal{G}$
$\mathcal{L}$	Set of links of network $\mathcal{G}$
$\mathcal{C}$	Cost function
$\mathcal{P}'_a$	initial primary path (before the mixing in Bhandari's algorithm)
$\mathcal{P}'_b$	initial backup path (before the mixing in Bhandari's algorithm)
$\mathcal{P}_a$	final primary path (after the mixing in Bhandari's algorithm)
$\mathcal{P}_b$	final backup path (after the mixing in Bhandari's algorithm)
$(\mathcal{P}_a, \mathcal{P}_b)$	Protected path with primary path $\mathcal{P}_a$ and backup path $\mathcal{P}_b$
$\mathcal{R}(l)$	Set of risk groups link l is in
$\mathcal{R}(\mathcal{P}_a)$	Set of risk groups in path $\mathcal{P}_a$
$\mathcal{R}_a$	Set of Shared Risk Link Groups
$\mathcal{T}$	To do set of $(\mathcal{P}_a, \mathcal{P}_b)$ (thus set of (set of SRLG, set of SRLG))
$\mathcal{D}$	Done set of $(\mathcal{P}_a, \mathcal{P}_b)$

List of Figures

Chapter 1

Introduction

1.1 Background

1.1.1 SURFnet

SURFnet is a subsidiary of the SURF organization, in which Dutch universities, universities for applied sciences and research centers collaborate nationally and internationally on innovative ICT facilities. The national SURFnet network connects local networks from institutes together. Besides that, SURFnet is connected to other networks nationally and internationally. About every 6 years SURFnet is upgraded so more bandwidth can be supplied. Currently version 6 of the SURFnet network is operational.

SURFnet is completely fiber-based, so from end- to endpoint the signal is transmitted using light. The route through the network is therefore called "lightpath". Customers of SURFnet can request lightpaths between two locations in the network. When a lightpath is set, this will typically sustain for longer periods for example one year. Optionally the lightpath can be set as a protected path and in this case two disjoint lightpaths will be setup to guarantee connectivity in case of failure of one of the lightpaths.

1.1.2 SARA

SARA Computing and Networking Services is an institute located in Amsterdam and Almere which has been given the task to maintain the SURFnet network. SARA is one of the locations of the Amsterdam Internet Exchange (AMS-IX), which is one of the largest Internet exchanges in the world.

1.1.3 Protection

To provide failure-safe connections in an optical network, lightpaths can be protected. Protected lightpaths consist of a primary path and a backup path which are disjoint and ensure connection continuity in case of a single link failure. There are two kinds of protection: dedicated protection and shared protection. In shared protection backup paths can share backup capacity or backup bandwidth (BBW) on links. This works if only one primary path fails, but fails if more than one primary path fail of which the backup paths are shared.

In dedicated protection it is not allowed for backup paths to share bandwidth. In this thesis protection means dedicated protection unless explicitly stated otherwise. Optical networks consist of at least two layers, the optical layer and the physical layer. And although the primary and backup paths are disjoint in the optical layer, in the physical layer they may share the same fiber span or duct. Such a fiber span can get damaged during digging for example. Although this is a single failure at the physical layer, it causes multiple failures at the optical layer and protected paths could get disconnected if both paths have fibers in the broken fiber span.

1.2 Problem definition

In WDM optical networks the *diverse routing problem* is to find a protected path between a source and a destination at the optical layer such that no single failure at the physical layer can cause both paths to fail. Fiber links that all fail together in case of physical damage, like fibers in the same fiber span, are assigned to the same Shared Risk Link Group (SRLG). A SRLG can have any number of links and a link can be in more than one SRLG. Further we assume that links in the same SRLG can be anywhere in the network which is the most general approach.

Consider an undirected network $\mathcal{G}$ with nodes $\mathcal{N}$ and links $\mathcal{L}$ where the SRLGs of a link $l \in \mathcal{L}$ are given by the set $\mathcal{R}(l)$. Let $\mathcal{P}$ be a single path in the network. We define $\mathcal{R}(\mathcal{P})$ as the set of SRLGs of all links in path $\mathcal{P}$.

$$\mathcal{R}(\mathcal{P}) = \bigcup_{l \in \mathcal{P}} \mathcal{R}(l)$$

A link can be in zero or more SRLGs and a SRLG can contain any number of links. Let n_s be a source node and n_d a destination node with $n_s, n_d \in \mathcal{N}$. The objective is to find two paths $(\mathcal{P}_a, \mathcal{P}_b)$ with minimal cost between nodes n_s and n_d with $\mathcal{R}(\mathcal{P}_a) \cap \mathcal{R}(\mathcal{P}_b) = \emptyset$. Hence, these paths are SRLG-disjoint. In case of a single link failure or single SRLG failure, the connection between n_s and n_d survives.

1.3 Complexity

The SRLG diverse routing problem has been proved to be NP-complete in [1]. There are several aspects of the diverse routing problem that make it difficult. In large networks with many SRLGs it can even be hard to find a disjoint path pair without considering the minimization of the cost.

As explained in section 1.4.2, with Active Path First algorithms (APF) the choice for the primary path causes limitations for the choice of the backup path. The backup path cannot contain any links that share SRLGs with links of the primary path to obtain a SRLG disjoint solution. If a primary path is chosen in such way that there is no backup path possible that is SRLG-disjoint with the primary path then this is called a trap. There are two kinds of traps for APF algorithms: real traps and avoidable traps. A real trap occurs in networks where no SRLG-disjoint solution is possible. Each choice of the primary path causes a trap for the backup path. In cases where there is no backup path possible which

is disjoint with the primary path, but there does exists a solution $(\mathcal{P}_a, \mathcal{P}_b)$ that is disjoint then this is called a avoidable trap.

Finding a disjoint path pair is one aspect, but minimizing the cost of that pair is another difficult aspect. A disjoint path pair with least cost is preferred above a solution which uses a lot of resources and thus has high cost. The combination of these two aspects causes great complexity. Proposed algorithms in the field of SRLG diverse routing (such as in section 1.4) try to find a balance between on one side the performance of the algorithm and on the other side the exactness of the algorithm. With the approach of heuristic algorithms performance improvement can be gained at the risk of not finding a solution while such solution does exists.

While in small networks an exact solution can be calculated in reasonable time, this cannot be done in large networks. This is because of the NP-completeness of the problem. The solution space grows exponentially with the network size. For large networks with a few SRLGs many developed algorithms can return the optimal solution, but if the network contains many SRLGs the probability that an algorithm encounters avoidable traps grows and finding an optimal path pair becomes far more difficult.

For special cases there exist algorithms with polynomial running time. An example is an SRLG network in which all the links in a SRLG share a common node. This and other special cases are discussed at the end of section 1.4.

1.4 Related work

The SRLG-tree algorithm in this thesis is an SRLG diverse routing algorithm with dedicated protection. Dedicated protection or static routing means that a backup path is reserved for one primary path only. The backup path is not shared by other backup paths. If the backup paths are allowed to share bandwidth with other backup paths then this is called shared protection or dynamic routing. In the field of SRLG diverse routing a large amount of work has been done. First we will look at several Integer Linear Programming (ILP) approaches.

1.4.1 ILP

A basic ILP formulation was proposed in [1], which we will refer to as static ILP hereafter. Its objective function is to minimize the total costs of the SRLG disjoint path pair. It can obtain results in seconds for a medium size network, but is suitable for dedicated protection only.

For ILP approaches which allow shared protection, i.e. different backup paths can use the same path, [13] proposes "Best Sharing ILP for Dynamic Routing". To take into consideration shared protection when determining the path pair, besides the constraints used in the static ILP formulation, some additional constraints related to the sharing and/or link capacity are required. It can achieve optimal (minimum) route allocation, but its running time is too long to be practical for a large network.

Since best sharing ILP is time-consuming and static ILP is not suitable for shared protection, a compromise scheme called two-stage ILP was developed [13]. It uses a path determination method similar to that used by static ILP

(i.e., it does not consider path sharing at this time). However, once the primary path and the backup path are chosen, minimal capacity will be allocated on each link along the backup path, whether complete or partial information is available.

Although many improvements on the existing ILP models have been made like in [14] ILP models are not feasible for large networks due to the enormous complexity of the models for networks with many nodes and links.

Shared backup path protection has been studied extensively in [2, 3, 4, 5, 6, 7, 8, 9]. Since this is an NP-complete problem, heuristic approaches are employed to obtain near-optimal solutions in polynomial time [4, 7, 8, 9].

1.4.2 Active Path First

Generally speaking, in shared path protection, optimizing a primary path is much more important than optimizing the corresponding backup path because the primary path will carry traffic almost all the time, and the bandwidth allocated to the backup path is used only after the primary path fails and may be shared by other backup paths in the future. Many approaches for shared protection assume that the cost of a link for the backup path is only a fraction of the cost of using a link for the primary path. The path pair is said to be asymmetrically weighted [15]. Therefore the algorithms mainly focus on optimizing the primary or active path. Since Active Path First (APF) based heuristics can naturally optimize a primary path and are intuitively simple, many such heuristics have been proposed, as described below.

First we will discuss "Simple APF". For a given source and destination node pair, a shortest path is found first as the primary path. Then the algorithm excludes all the links from the network that share at least one SRLG with any link along the primary path, and try to find another shortest path as the corresponding backup path. However, the algorithm can easily fall into avoidable traps, especially in large networks. With an avoidable trap is meant that a solution is possible, but the algorithm does not find one.

[28] suggests an improved scheme in which to each link weight an addition weight is added for each SRLG the link is in. With this approach links that are in many SRLGs become less attractive. With less SRLGs in both paths the chance that they are SRLG-disjoint becomes higher. The advantage of this approach is that it is very easy to implement, but it is not difficult to create a simple network in which the algorithm does not result in two SRLG-disjoint paths while there is a solution.

The "Bypass method" [22] was proposed for fiber span failure protection, which is a special case of SRLG failure protection. Its basic idea is to construct a single layer subnetwork over the original optical network and try to find two link disjoint paths on the constructed subnetwork.

In [11] a heuristic is proposed. With Yen's algorithm [12] the first k shortest paths are calculated. Then their Iterative Modified Suurballes Heuristic (IMSH) algorithm iterates over these k paths. For each iteration the path is considered the first path in Suurballe's algorithm. Then all SRLGs that also have links in the first path are removed from the network and Suurballe continues as normal. If two paths are found then IMSH checks if the paths are indeed disjoint. The iteration with the solution with the least cost is returned as the solution.

In [25] with a heuristic a solution is searched with the least number of SRLGs shared in the primary path and the backup path while keeping the blocking probability of new requested paths in the network as low as possible. A solution that is completely SRLG-disjoint becomes a special case.

To address the trap problem [26] describes an approach to deal with this problem. After a primary path is found all links in SRLGs that also has links in the primary path get a high link cost. In this way these links become less attractive to the shortest path search algorithm for the backup path. If a backup path is found without having links in it which were set at a high cost then a disjoint path pair is found. But if the backup path does contain links with high cost then with "MostRiskyActiveLink" a link from the primary path is removed from the network and the shortest path algorithm is ran again to get a new primary path. This loops until a disjoint solution is found or no solution is found. Several suggestions are made to determine the most risky link in the primary path.

1.4.3 Other approaches

Besides the ILP- and APF-based heuristic algorithms described above, an approach called network transformation was proposed in [23]. It first transforms the upper-layer network into a network without SRLGs by adding some virtual nodes and zero cost links, and then applies the algorithm of finding node disjoint paths to the transformed network. The found node disjoint paths will also be SRLG disjoint after the virtual nodes and links are removed. However, the transformation is applicable to only a few topologies, and is intractable for an arbitrary topology.

In addition, a stochastic approach was also proposed in [24] to determine SRLG disjoint paths for shared SRLG protection. It is similar to simple APF with the major difference in how the cost of each link is assigned before the backup path is determined using a shortest path algorithm. More specifically, while simple APF assigns, say, 1 unit (e.g., a channel) worth of bandwidth as the cost of each link that can possibly be used by the backup path, the stochastic algorithm assigns $p < 1$ units, where p is the probability that backup bandwidth sharing cannot occur on the link. Its main advantage is that it is simple to implement, but the price paid is that its bandwidth efficiency is not as good as its deterministic counterparts such as simple APF. It also did not address the trap problem, and in fact can fall into as many avoidable traps as simple APF.

In [16] and [17] the scheme for Protection with Multiple Segments (PROMISE) is described. The basic idea of PROMISE is to divide the primary path into several possible overlapping active segments or ASs, and than protect each AS with a detour called backup segment instead of protecting the primary path as a whole as in path protection schemes. Note that, several other segmented protection approaches have been proposed in [18, 19, 20]. In [18], an APF based heuristic algorithm was proposed to determine segments, which cannot efficiently deal with a real trap, and in addition, does not consider backup bandwidth sharing until the paths are found. The scheme in [19] requires the node immediately upstream from the link/node failure to restore traffic along an alternate outgoing link, which limits its flexibility (and bandwidth efficiency), especially in SRLG networks. Another example of segment-based approach is

called Short Leap Shared Protection (SLSP) proposed in [20], where an AP is divided into several equal-length and overlapped segments. However, inflexible segmentation will affect its ability to get the best performance and escaping traps. In [21] PROMISE is extended. The proposed algorithm uses a dynamic programming technology and achieves a higher bandwidth efficiency and lower request blocking probability than the original PROMISE algorithm.

1.4.4 Special cases

For special cases of SRLG problems there exist algorithms with polynomial complexity. If we take the condition that a SRLG contains only one link then the problem is reduced to the link-disjoint diverse routing problem. Another commonly known problem is when all links from a node form a SRLG. This problem is equal to the node-disjoint diverse routing problem. Suurballe [31, 32] has proposed algorithms for these problems which run in polynomial time.

In [29] and [30] SRLGs are considered with no more than two links. A pair of paths is constructed in polynomial time which can survive a dual-link failure.

Another special case is considered in [27] where links in the same SRLG share a common endpoint. The algorithm in this work runs in polynomial time. After the proposed Diverse_Routing finds a first path with a shortest path algorithm like Bhandari's algorithm for example, extra virtual nodes are added to the network for link that have both endpoints in the first path. A second path is searched with Find_Second_Path. Find_Second_Path uses three different kinds of routing information per node. Which kind of routing information is used depends on which of the four cases is encountered while exploring new nodes. Define v as the last node in a path that is being explored and u as the node before the last node. The four cases are: both u and v are in the first path, both u and v are not in the first path, u is in the first path and v is not, and v is in the first path and u is not. If Find_Second_Path finds a solution then the virtual nodes are removed and both paths are checked on common segments. Like in Bhandari's algorithm segments are exchanged and common segments are deleted. The results are SRLG-disjoint paths.

1.5 Contribution of this thesis

Below are the aspects of the SRLG-tree algorithm proposed in this thesis

- Applicable to general SRLG network
- Exact
- Polynomial running time for networks with little SRLGs

In this work we consider the most general case where links in a SRLG can be any set of links of the network. The proposed SRLG-tree algorithm is an iterative approach with the focus on the optimality of the path pairs. SRLG-tree is different with most approaches in that it is not a heuristic or ILP model, but an exact algorithm that finds two SRLG-disjoint paths with minimal cost between a source node and a destination node in a network. SRLG-tree addresses the trap problem by its iterative approach. SRLG-tree has polynomial running time for networks with SRLG sparse networks. If the average number of SRLGs that

a link is in grows then the running time converges to an exponential running time.

1.6 Organization

The SRLG-tree algorithm is described in chapter 3. SRLG-tree uses the SRLG-exclusion algorithm and the SRLG-exclusion uses the ShortestPathSwitch algorithm. In chapter 2 first the ShortestPathSwitch is discussed and after that SRLG-exclusion is described. In chapter 4 the algorithms are adapted to be used in subnetwork topologies. Chapter 5 describes the developed application in which SRLG-tree is implemented. Finally chapter 6 makes suggestions for further investigations.

Chapter 2

The SRLG-exclusion algorithm

2.1 SRLG-tree introduction

SRLG-tree is based on the exclusion of SRLGs from paths. If a protected path is not disjoint for a SRLG, for example SRLG A, then if we exclude this SRLG A from either the first path or the second path then we know that the new protected path will be disjoint for SRLG A. SRLG-tree uses a tree in which each node is a pair of sets of SRLGs. For example node (AB, C) means that a protected path is searched where the first path does not have links that are in SRLG A or B, and where the second path does not have links in SRLG C. The top node $(-, -)$ means both paths are allowed to have any link and no SRLGs are excluded. SRLG-tree is a Breath-First-Search algorithm. A property of the tree is that the cost of solutions of the nodes grow while traveling down the tree. So the solution of the top node has the least total cost, but this solution does not have to be SRLG-disjoint. If a SRLG-disjoint solution is found, no child node can have a solution with lower cost, because of this property. The idea behind SRLG-tree is the lazy exclusion of SRLGs. What we mean with this is that when a found path is not disjoint for a SRLG only then we exclude this SRLG from one of the paths. The parsing tree is not known in advance, only the starting node $(-, -)$. From there, the tree evolves with branches. The branching stops in a node where a SRLG-disjoint solution is found or no solution is found.

First we explain that with a path excluded from SRLG A is meant that this path does not have links that are in SRLG A. If a found protected path of a tree node is not SRLG-disjoint, the algorithm only picks one SRLG that is in both paths. If there are multiple SRLGs that are in both paths then a SRLG can be chosen randomly. Then two new branches are made with the original exclusions and the exclusion of the picked SRLG for both paths. So if for example the original node was (A, B) and the found path is disjoint for SRLG C then two new branches are made (AC, B) and (A, BC). This makes SRLG tree branch-and-bound method. Note that the solution of the top node $(-, -)$ can be found with Bhandari's algorithm [33], because no SRLG exclusions are necessary. For finding paths excluded from certain SRLGs this thesis proposes an algorithm

called SRLG-exclusion.

2.2 SRLG-exclusion introduction

Consider an undirected network $\mathcal{G}$ with nodes $\mathcal{N}$ and links $\mathcal{L}$ where the SRLGs of a link $l \in \mathcal{L}$ are given by the set $\mathcal{R}(l)$. Let $\mathcal{P}$ be a single path in the network. We define $\mathcal{R}(\mathcal{P})$ as the set of SRLGs of all links in path $\mathcal{P}$. So

$$\mathcal{R}(P) = \bigcup_{l \in \mathcal{P}} \mathcal{R}(l)$$

It is easy to find a single path between nodes in $\mathcal{G}$ which is excluded from certain SRLGs. Just remove all the links that are in unwanted SRLGs and run a shortest path algorithm. Things get more complicated if we have two sets of SRLGs $\mathcal{R}_a$ and $\mathcal{R}_b$ and if the objective is to calculated two disjoint paths $\mathcal{P}_a$ and $\mathcal{P}_b$ where $\mathcal{P}_a$ is excluded from $\mathcal{R}_a$ and $\mathcal{P}_b$ is excluded from $\mathcal{R}_b$. If we use the same technique as for the single path then this is a form of two-times Dijkstra which not always gives a solution even if it exists. Therefore using an algorithm like Bhandari's algorithm is better. In Bhandari's algorithm after finding the first path, which we will call the initial first path, this path is made directed towards the source. If the second path, which we will define as the initial second path, uses such directed links then these links used by both paths. Therefore these links can be eliminated and this gives two new paths, which we will call the final first path and the final second path. But this causes that parts of the initial first path can be in the final second path after the elimination. And vice versa, that parts of the initial second path can be in the final first path. So this also causes that although the initial first path that was found without links that are in $\mathcal{R}_a$ the final first path does have links in $\mathcal{R}_a$ and analog for the second path. This is where the problem lies in finding two paths that are excluded from $\mathcal{R}_a$ and $\mathcal{R}_b$. We will write the algorithm for the calculation of these two paths $(\mathcal{P}_a, \mathcal{P}_b)$ as SRLG-exclusion$(\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b)$ or short as $(\mathcal{R}_a, \mathcal{R}_b)$ if it is clear what $\mathcal{G}$, n_s and n_d are.

2.3 Algorithm

We now focus on SRLG-exclusion which is used by the SRLG-tree algorithm. SRLG-exclusion is called with the parameters SRLG-exclusion$(\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b)$ where $\mathcal{G}$ is a network, n_s is a source, n_d is a destination in $\mathcal{G}$, $\mathcal{R}_a$ and $\mathcal{R}_b$ are two sets of SRLGs. The algorithm returns two paths from which the first path does not have links that are in a SRLG that is in $\mathcal{R}_a$ and the second path does not have links that are in a SRLG that is in $\mathcal{R}_b$. Both paths are also node disjoint. We will now explain how SRLG-exclusion works. First we remove all links from $\mathcal{G}$ that are in a SRLG in $\mathcal{R}_a$ and call this new network $\mathcal{G}'_a$. Then a initial primary path $\mathcal{P}'_a$ is obtained with a shortest path algorithm. $\mathcal{P}'_a$ is evidently excluded from SRLGs in $\mathcal{R}_a$. As in the Bhandari algorithm we make all links in the original $\mathcal{G}$ where the removed links are placed back, that are in $\mathcal{P}'_a$ directed from n_d to n_s. We call this new network $\mathcal{G}'$. We define

$$\mathcal{G}_a := \{(\mathcal{N}, \mathcal{L}) \in \mathcal{G}' \mid l \in \mathcal{P}'_a \text{ or } \mathcal{R}(l) \cap \mathcal{R}_a = \emptyset \ \forall l \in \mathcal{L}\}$$

$$\mathcal{G}_b := \{(\mathcal{N}, \mathcal{L}) \in \mathcal{G}' \mid l \in \mathcal{P}'_a \text{ or } \mathcal{R}(l) \cap \mathcal{R}_b = \emptyset \; \forall l \in \mathcal{L}\}$$

So $\mathcal{G}_b$ is the network $\mathcal{G}'$ where all links that are in $\mathcal{R}_b$ are removed except if they are directed because they are in $\mathcal{P}'_a$.

While searching the shortest path from n_s to n_d in $\mathcal{G}$ we keep track of the queue of paths that is being explored. We define one or more contiguous links in a path as a "part".

The algorithm starts its search in n_s in $\mathcal{G}_b$. If a path "in progress" has one part in common with $\mathcal{P}'_a$ then the algorithm uses $\mathcal{G}_a$ to explore new links. We also use the term crossing for having a common part. If a path crosses $\mathcal{P}'_a$ two times so it has two parts common with $\mathcal{P}'_a$ then the algorithm uses $\mathcal{G}_b$ again. So with an odd number of crossings the algorithm works on $\mathcal{G}_a$ otherwise on $\mathcal{G}_b$. So it is a shortest path algorithm while during the exploration of links the network on which it operates can change. An example is given below.

2.4 Example

Assume we have a network $\mathcal{G}$ shown in figure 2.1 with a source and destination. $\mathcal{G}$ contains two SRLGs, 1 and 2. Let $\mathcal{R}_a = \{1\}$ and $\mathcal{R}_b = \{2\}$. Now for the initial primary path $\mathcal{P}'_a$ we can run $ShortestPath(\mathcal{G}'_a, n_s, n_d)$ where $\mathcal{G}'_a$ is $\mathcal{G}$ without links in $\mathcal{R}_a$. This gives the path in figure 2.3 and does not contain links that are in 1. We now place back all the links that we removed that are in SRLG 1. Further we make all links of the found path directed from n_d to n_s. Now remove all undirected $l \in \mathcal{L}$ in $\mathcal{G}$ with $\mathcal{R}(l) \cap \mathcal{R}_b \neq \emptyset$. We call this new network $\mathcal{G}_b$ (figure 2.4). We are now interested in calculating a backup path without links in $\mathcal{R}_b$. We use a shortest path algorithm which works with a queue of paths. While running the algorithm, if a path goes through a directed link like in figure 2.5 then we transform our network $\mathcal{G}_b$ to $\mathcal{G}_a$ (see figure 2.6). If the path travels a unidirectional link again (figure 2.7), we transform our network back to $\mathcal{G}_b$ (figure 2.8), etc. After finding the backup path $\mathcal{P}'_b$ (figure 2.9) we put both $\mathcal{P}'_a$ (solid) and $\mathcal{P}'_b$ (dashed) in the original network $\mathcal{G}$ (figure 2.10). Then like the Bhandari algorithm we remove links in $\mathcal{G}$ which are used in both directions (figure 2.10). The result is a primary path $\mathcal{P}_a$ excluded from SRLGs in $\mathcal{R}_a$ and a backup path $\mathcal{P}_b$ as displayed in figure 2.11.

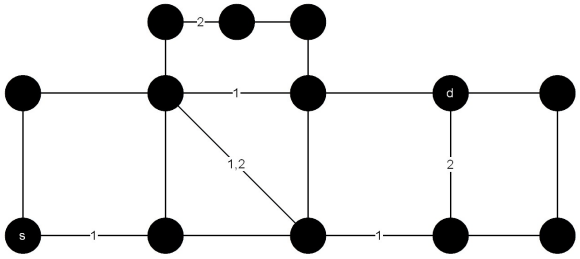


Figure 2.1: Network $\mathcal{G}$

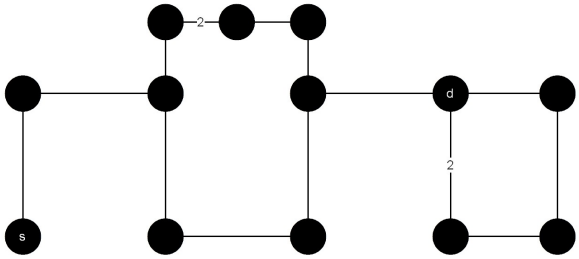


Figure 2.2: Network $\mathcal{G}'_a$

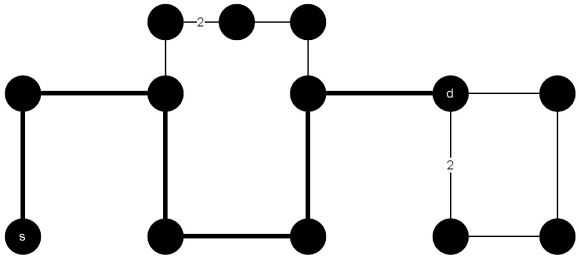


Figure 2.3: $\mathcal{P}'_a$=ShortestPath$(\mathcal{G}'_a, n_s, n_d)$

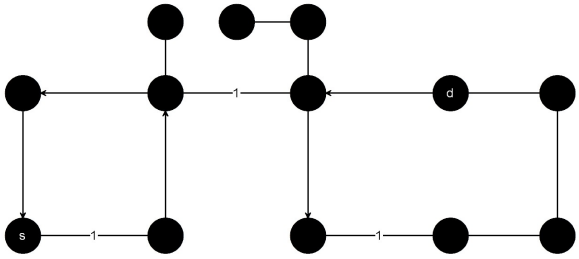


Figure 2.4: Network $\mathcal{G}_b$ with primary path and links in $\mathcal{R}_b$ removed from $\mathcal{G}$

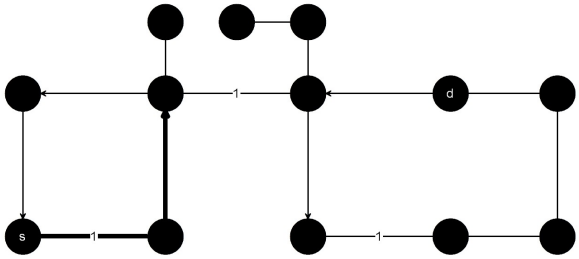


Figure 2.5: Crossing directed link

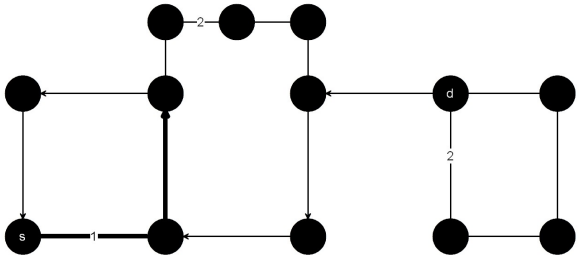


Figure 2.6: Add links in $\mathcal{R}_b$, remove links in $\mathcal{R}_a$

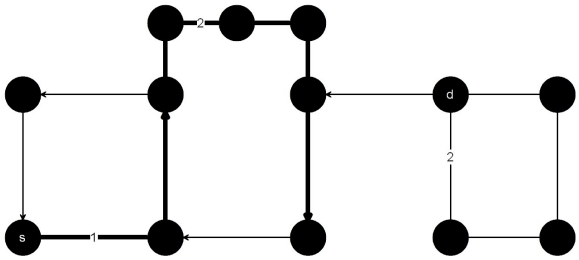


Figure 2.7: Crossing directed link

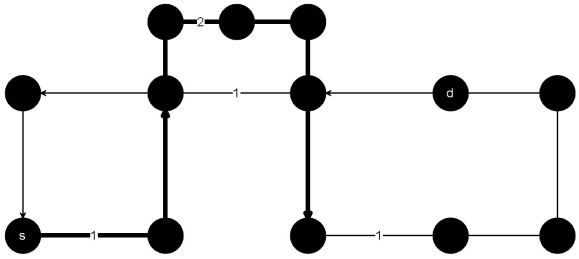


Figure 2.8: Add links in $\mathcal{R}_a$, remove links in $\mathcal{R}_b$

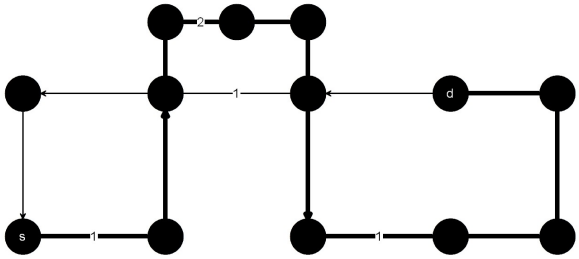


Figure 2.9: Backup path found

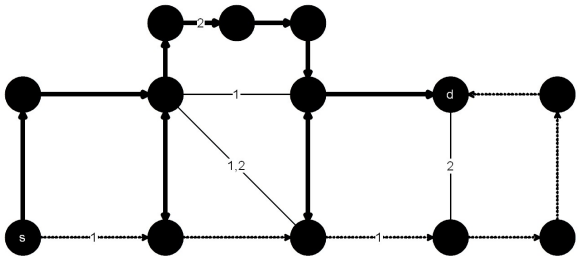


Figure 2.10: Both paths in $\mathcal{G}$, remove links that are traversed in both directions

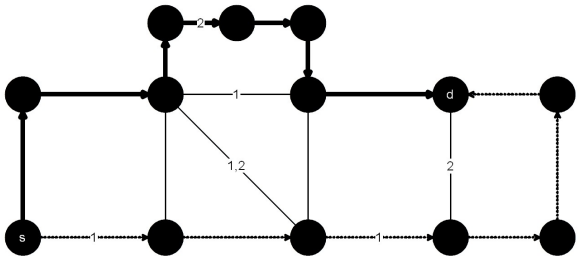


Figure 2.11: Result of SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b$)

2.5 Problem of mixing paths

In the example we saw that final solution had a primary path and backup path which were excluded from the intended SRLGs. But with the used method this can easily go wrong. While we used the switching between two networks for the backup path, we did not do this for the first path. This can cause that after mixing the two initial paths, links with unwanted SRLGs can end up in the backup path. We give an example where this happens. In this example we again search a primary path without $\mathcal{R}_a = \{1\}$ and the backup path without $\mathcal{R}_b = \{2\}$.

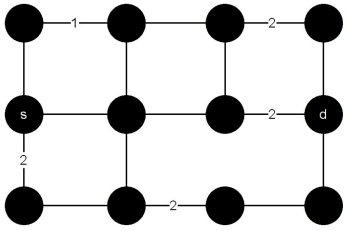


Figure 2.12: Original network $\mathcal{G}$

In figure 2.18 the bold solid path is the primary path and the dashed path is the backup path. As we see the backup path contains SRLG 1 and SRLG 2 although we searched a backup path without SRLG 2. This is caused by the mixing of the initial primary and backup path.

There is a way to prevent this. After finding the initial primary path $\mathcal{P}'_a$

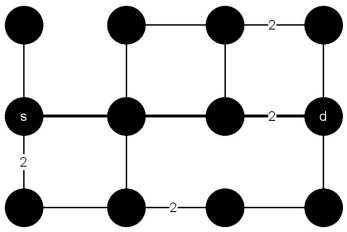


Figure 2.13: Initial primary path $\mathcal{P}'_a$ in $\mathcal{G}'_a$

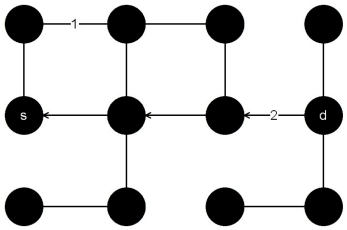


Figure 2.14: Network $\mathcal{G}_b$

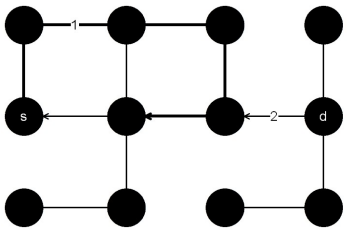


Figure 2.15: Crossing initial primary path therefore switch network.

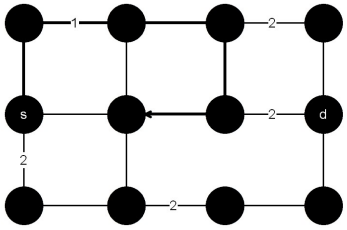


Figure 2.16: Network $\mathcal{G}_a$

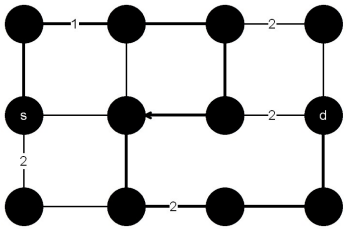


Figure 2.17: Initial backup path $\mathcal{P}'_b$

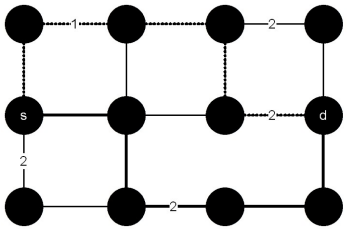


Figure 2.18: Final primary and backup path after mixing

remove from that path all links that are in a SRLG in $\mathcal{R}_b$. This leaves one or more parts. Also remove the last part, even if there is only one part. This leaves zero or more parts. During the exploring of the second initial path $\mathcal{P}'_b$ we demand that the number of times that $\mathcal{P}'_b$ crosses a part is even. The final paths $\mathcal{P}_a$ and $\mathcal{P}_b$ will be correct after mixing.

Assume again a network with SRLGs 1 and 2. We want to calculate $(1, 2)$ which means that the primary path does not have links in SRLG 1 and the backup path does not have links in SRLG 2. In figure 2.19 we only show $\mathcal{P}'_a$. If we remove the two links from $\mathcal{P}'_a$ that are in SRLG 2 we get three parts from which we ignore the last part. For finding $\mathcal{P}'_b$ we demand that the number of times that $\mathcal{P}'_b$ crosses each of the two parts is even. Figure 2.20 shows a possible $\mathcal{P}'_b$ that is valid. The first part is crossed two times and the second is crossed zero times. So each part is crossed an even number of times. The last part which is crossed once is ignored. After mixing the initial paths the final paths in figure 2.21 are a valid solution for $(1, 2)$.

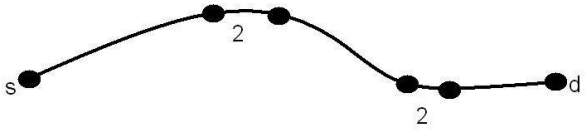


Figure 2.19: Initial primary path $\mathcal{P}'_a$

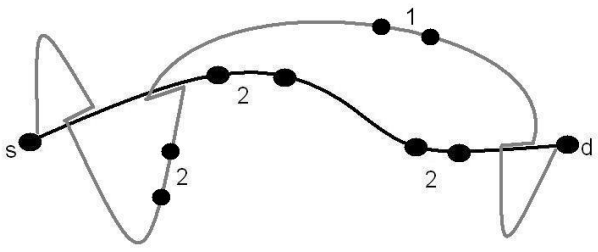


Figure 2.20: Valid path for $\mathcal{P}'_b$

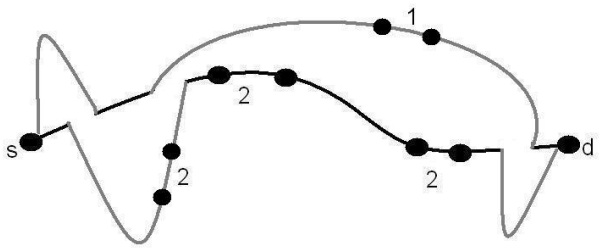


Figure 2.21: Result after mixing

2.6 Time complexity

Let $\mathcal{N}$ be the nodes in a network and $\mathcal{L}$ the links. The calculation of $\mathcal{P}'_a$ is done with a shortest path algorithm. We can use an algorithm based on Dijkstra for this which has time complexity $\mathcal{O}(|\mathcal{L}| + |\mathcal{N}|log|\mathcal{N}|)$. The time complexity for calculating $\mathcal{P}'_b$ takes some attention. In normal shortest path calculations for each node in the network the minimal cost for reaching that is note is kept up. In SRLG-exclusion a node can be visited in each of the two networks $\mathcal{G}_a$ and $\mathcal{G}_b$. This means that for both networks the minimal cost for reaching a node has to be tracked. All links and nodes are visited twice in the worst case so the time complexity for finding $\mathcal{P}'_b$ is $\mathcal{O}(2|\mathcal{L}| + 2|\mathcal{N}|log(2|\mathcal{N}|))$ and so the total time complexity for SRLG-exclusion is $\mathcal{O}(3|\mathcal{L}| + 3|\mathcal{N}|log(|\mathcal{N}|) + 2|\mathcal{N}|log(2)) = \mathcal{O}(|\mathcal{L}| + |\mathcal{N}|log(|\mathcal{N}|) + |\mathcal{N}|)$. We assume that the switching between the two networks does not add to the running time.

2.7 Exactness

Theorem 2.7.1. *Let $\mathcal{G}$ be a network with nodes $\mathcal{N}$ and links $\mathcal{L}$. n_s is a source, n_d is a destination in $\mathcal{G}$, $\mathcal{R}_a$ and $\mathcal{R}_b$ are two sets of SRLGs. SRLG-exclusion$(\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b)$ or short as $(\mathcal{R}_a, \mathcal{R}_b)$ finds a solution if one exists.*

Proof. Let $(\mathcal{P}^*_a, \mathcal{P}^*_b)$ be a solution so $\mathcal{P}^*_a \cap \mathcal{R}_a = \emptyset$ and $\mathcal{P}^*_b \cap \mathcal{R}_b = \emptyset$. For calculating $(\mathcal{R}_a, \mathcal{R}_b)$ we first calculate the initial primary path $\mathcal{P}'_a$. Such path is always possible, because calculation of $\mathcal{P}'_a$ is exact (i.e. it is just Dijkstra's algorithm). We then switch to $\mathcal{G}_b$ as defined above and calculate $\mathcal{P}'_b$. If a direct path from source to destination is possible over undirected links then we found a solution $(\mathcal{P}'_a, \mathcal{P}'_b)$. If no direct path $\mathcal{P}'_b$ in $\mathcal{G}_b$ is possible then $\mathcal{P}'_a \cap \mathcal{P}^*_b \neq \emptyset$ holds otherwise $\mathcal{P}'_b = \mathcal{P}^*_b$ is a path in $\mathcal{G}_b$ which is a contradiction. For constructing $\mathcal{P}'_b$ we first follow $\mathcal{P}^*_b$ until $\mathcal{P}'_a$. Then we follow the directed links until we encounter $\mathcal{P}^*_a$. If $\mathcal{P}^*_a \cap \mathcal{P}'_a = \emptyset$ then this is all the way back to the source. SRLG-exclusion then switches to $\mathcal{G}_a$. If a direct path to the destination is possible then we are done otherwise we follow $\mathcal{P}^*_a$ in the direction of the destination until we reach $\mathcal{P}'_a$ again. Then follow the directed links until we encounter $\mathcal{P}^*_b$ again. By continuing this process we finally reach the destination, because we follow $\mathcal{P}^*_a$ and $\mathcal{P}^*_b$ towards the destination. □

Because the main algorithm SRLG-tree has to calculate the shortest SRLG-disjoint protected path in a network, we want SRLG-exclusion to return the

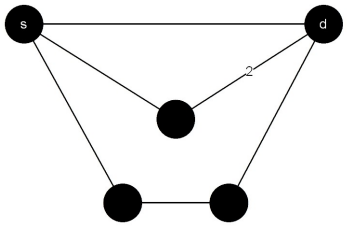


Figure 2.22: Original network $\mathcal{G}$

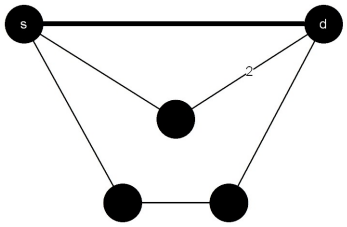


Figure 2.23: Path $\mathcal{P}'_a$

shortest protected path. But SRLG-exclusion does not guarantee to give the solution with minimal cost. We will show this in an example where we again calculate $(1, 2)$ in a network $\mathcal{G}$ with SRLGs 1 and 2 shown in figure 2.22. First $\mathcal{P}'_a$ is calculated and shown in figure 2.23. In figure 2.24 network $\mathcal{G}_b$ is shown and in figure 2.25 is the calculated $\mathcal{P}'_b$. In figure 2.26 the primary path is drawn as a thick, solid lines and the backup path is dashed. This solution is obviously not the optimal solution. The optimal solution is shown in figure 2.27. Although SRLG-exclusion does not guarantee to return a solution with minimal cost, we can prove the next theorem.

Theorem 2.7.2. *Let $\mathcal{G}$ be a network with nodes $\mathcal{N}$ and links $\mathcal{L}$. n_s is a source, n_d is a destination in $\mathcal{G}$, $\mathcal{R}_a$ and $\mathcal{R}_b$ are two sets of SRLGs. $(\mathcal{R}_a, \mathcal{R}_b)$ or $(\mathcal{R}_b, \mathcal{R}_a)$ returns an optimal solution if exists with a path excluded from links in SRLGs in $\mathcal{R}_a$ and a path excluded from links in SRLGs in $\mathcal{R}_b$.*

Proof. The difference between the two SRLG-exclusion calculations is with which path is started, the path without $\mathcal{R}_a$ or the path without $\mathcal{R}_b$. The first path from $(\mathcal{R}_a, \mathcal{R}_b)$ we call $\mathcal{P}'_{a\mathcal{R}_a}$ and the initial primary path from $(\mathcal{R}_b, \mathcal{R}_a)$ we call $\mathcal{P}'_{a\mathcal{R}_b}$. Let $(\mathcal{P}^*_a, \mathcal{P}^*_b)$ be the optimal solution in $\mathcal{G}$. If $\mathcal{P}'_{a\mathcal{R}_a} \cap \mathcal{P}^*_a = \emptyset$ and $\mathcal{P}'_{a\mathcal{R}_b} \cap \mathcal{P}^*_b = \emptyset$ then if $\mathcal{P}'_{a\mathcal{R}_a} \cap \mathcal{P}'_{a\mathcal{R}_b} = \emptyset$ then because $\mathcal{C}(\mathcal{P}'_{a\mathcal{R}_a}) \leq \mathcal{C}(\mathcal{P}^*_a)$ and $\mathcal{C}(\mathcal{P}'_{a\mathcal{R}_b}) \leq \mathcal{C}(\mathcal{P}^*_b)$ it holds that $\mathcal{C}(\mathcal{P}'_{a\mathcal{R}_a}, \mathcal{P}'_{a\mathcal{R}_b}) \leq \mathcal{C}(\mathcal{P}^*_a, \mathcal{P}^*_b)$ and $(\mathcal{P}'_{a\mathcal{R}_a}, \mathcal{P}'_{a\mathcal{R}_b})$ is a solution. Else if $\mathcal{P}'_{a\mathcal{R}_a} \cap \mathcal{P}'_{a\mathcal{R}_b} \neq \emptyset$ then $(\mathcal{P}'_{a\mathcal{R}_a}, \mathcal{P}^*_b)$ is a optimal solution and $(\mathcal{P}^*_a, \mathcal{P}'_{a\mathcal{R}_b})$ is a optimal solution.

So now we look at the case that $\mathcal{P}'_{a\mathcal{R}_a} \cap \mathcal{P}^*_a \neq \emptyset$ or $\mathcal{P}'_{a\mathcal{R}_b} \cap \mathcal{P}^*_b \neq \emptyset$. Assume $\mathcal{P}'_{a\mathcal{R}_a} \cap \mathcal{P}^*_a \neq \emptyset$. The prove for $\mathcal{P}'_{a\mathcal{R}_b} \cap \mathcal{P}^*_b \neq \emptyset$ is analog. If $\mathcal{P}'_{a\mathcal{R}_a} \cap \mathcal{P}^*_b = \emptyset$ then $(\mathcal{P}'_{a\mathcal{R}_a}, \mathcal{P}^*_b)$ is an optimal solution. Assume $\mathcal{P}'_{a\mathcal{R}_a} \cap \mathcal{P}^*_b \neq \emptyset$. But then we can use the same construction as in the previous proof to construct $\mathcal{P}'_b$. The optimal route for the initial backup path is from the source over $\mathcal{P}^*_b$ until $\mathcal{P}'_{a\mathcal{R}_a}$. Then follow the directed links until $\mathcal{P}^*_a$ is encountered. Follow $\mathcal{P}^*_a$ towards the destination until the destination node or until $\mathcal{P}'_{a\mathcal{R}_a}$ is reached. If we are

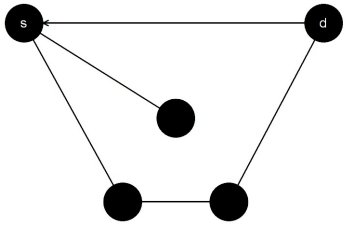


Figure 2.24: Network $\mathcal{G}_b$

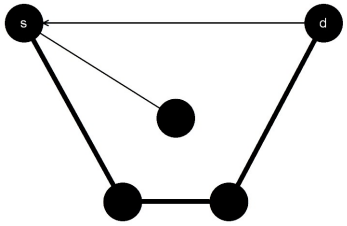


Figure 2.25: Path $\mathcal{P}'_b$

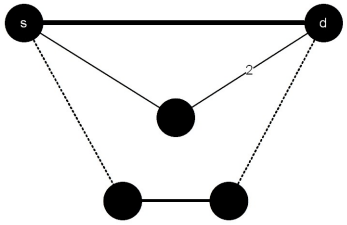


Figure 2.26: Final paths, but not optimal.

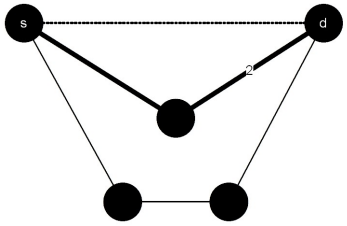


Figure 2.27: Optimal solution

in the destination we are done otherwise follow the directed links until $\mathcal{P}_b^*$ is encountered. Then follow $\mathcal{P}_b^*$ towards the destination. This process is repeated until the destination node is reached. After mixing the initial paths the final paths are $\mathcal{C}(\mathcal{P}_a, \mathcal{P}_b) \leq \mathcal{C}(\mathcal{P}_a^*, \mathcal{P}_b^*)$. □

Chapter 3

The SRLG-tree algorithm

We can now focus on the main algorithm of this work, the SRLG-tree algorithm. Given a network $\mathcal{G}$ with nodes $\mathcal{N}$, links $\mathcal{L}$ and a source node n_s and a destination node n_d with $n_s, n_d \in \mathcal{N}$. Links in $\mathcal{G}$ can be in one or more SRLGs and a SRLG can contain any number of links. The goal of the SRLG-tree algorithm is to find the two shortest paths between n_s and n_d which are node, link and SRLG disjoint. To calculate the cost $\mathcal{C}$ of a path we can use any cost function.

3.1 Algorithm

Below follows the global layout of the algorithm. Afterwards it is explained step by step. **SRLG-tree(G,n_s,n_d)**

1: Initialize $\mathcal{T} \leftarrow \emptyset$ (todo set of tree nodes)
2: Initialize $\mathcal{D} \leftarrow \emptyset$ (done set of tree nodes)
3: Initialize $\mathcal{P}_{min} \leftarrow \emptyset$ (current two shortest paths)
4: Initialize $\mathcal{C}_{min} \leftarrow \infty$ (cost of current two shortest paths)
5: $(\mathcal{P}_a, \mathcal{P}_b) \leftarrow Bhandari(\mathcal{G}, n_s, n_d)$ (normal Bhandari)
6: **if** $(\mathcal{P}_a, \mathcal{P}_b) = \emptyset$ **then**
7: exit (no solution possible, stop)
8: **else**
9: **if** $\mathcal{R}(\mathcal{P}_a) \cap \mathcal{R}(\mathcal{P}_b) = \emptyset$ **then**
10: $\mathcal{P}_{min} \leftarrow (\mathcal{P}_a, \mathcal{P}_b)$ (save found solution)
11: $\mathcal{C}_{min} \leftarrow \mathcal{C}(\mathcal{P}_a, \mathcal{P}_b)$ (save cost)
12: exit (solution is SRLG disjoint, stop)
13: **else**
14: **for all** $r \in \mathcal{R}(\mathcal{P}_a) \cap \mathcal{R}(\mathcal{P}_b)$ (is single path without SRLG possible?) **do**
15: $\mathcal{L}^* = \{l \in \mathcal{L} | r \notin \mathcal{R}(l)\}$
16: $\mathcal{G}^* = \mathcal{G}(\mathcal{N}, \mathcal{L}^*)$
17: **if** $ShortestPath(\mathcal{G}^*, n_s, n_d) = \emptyset$ **then**
18: exit (no SRLG-disjoint paths possible)
19: **end if**
20: **end for**
21: $r \in \mathcal{R}(\mathcal{P}_a) \cap \mathcal{R}(\mathcal{P}_b)$
22: $\mathcal{T} \leftarrow \mathcal{T} \cup \{(\{r\}, \emptyset), (\emptyset, \{r\})\}$

```
    end if
end if
while $\mathcal{T} \neq \emptyset$ (loop if todo is not empty) do
    $(\mathcal{R}_a, \mathcal{R}_b) \leftarrow \{(\mathcal{R}_1, \mathcal{R}_2) \in \mathcal{T}, \min\{|\mathcal{R}_1| + |\mathcal{R}_2| \mid (\mathcal{R}_1, \mathcal{R}_2) \in \mathcal{T}\}\}$
    $(\mathcal{P}_a, \mathcal{P}_b) \leftarrow$SRLG-exclusion$(\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b)$
    if $(\mathcal{P}_a, \mathcal{P}_b) = \emptyset$ then
        $\mathcal{D} \leftarrow \mathcal{D} \cup (\mathcal{R}_a, \mathcal{R}_b)$ (add to done set)
    else
        if $\mathcal{R}(\mathcal{P}_a) \cap \mathcal{R}(\mathcal{P}_b) = \emptyset$ (no overlapping SRLGs?) then
            if $\mathcal{C}(\mathcal{P}_a, \mathcal{P}_b) < \mathcal{C}_{min}$ then
                $\mathcal{P}_{min} \leftarrow (\mathcal{P}_a, \mathcal{P}_b)$
                $\mathcal{C}_{min} \leftarrow \mathcal{C}(\mathcal{P}_a, \mathcal{P}_b)$ (if currently shortest sol. then save)
            end if
            $\mathcal{D} \leftarrow \mathcal{D} \cup (\mathcal{R}_a, \mathcal{R}_b)$ (add to done set)
        else
            if $\mathcal{C}(\mathcal{P}_a, \mathcal{P}_b) < \mathcal{C}_{min}$ then
                $r \in \mathcal{R}(\mathcal{P}_a) \cap \mathcal{R}(\mathcal{P}_b)$
                $\mathcal{T} \leftarrow \mathcal{T} \cup \{(\mathcal{R}_a \cup \{r\}, \mathcal{R}_b)\} \cup \{(\mathcal{R}_a, \mathcal{R}_b \cup \{r\})\}$
            else
                $\mathcal{D} \leftarrow \mathcal{D} \cup (\mathcal{R}_a, \mathcal{R}_b)$ (not possible to become opt. sol.)
            end if
        end if
    end if
    $\mathcal{T} \leftarrow \mathcal{T} \backslash (\mathcal{R}_a, \mathcal{R}_b)$ (evaluated $(\mathcal{R}_a, \mathcal{R}_b)$, remove from todo set)
    $\mathcal{T} \leftarrow \{(\mathcal{T}_1, \mathcal{T}_2) \in \mathcal{T} \mid \forall (\mathcal{D}_1, \mathcal{D}_2) \in \mathcal{D}, \mathcal{D}_1 \cup \mathcal{D}_2 \nsubseteq \mathcal{T}_1 \cup \mathcal{T}_2\}$
end while
return $\mathcal{P}_{min}$
```

3.2 Explanation

If a solution exists, the algorithm returns two node/link/SRLG-disjoint paths with minimal cost. The algorithm starts with the Bhandari algorithm to find two initial paths. If we find paths that are SRLG-disjoint then we are done, otherwise we check for each SRLG that is in both paths if a single path from n_s to n_d without links in the SRLG is possible. We can do this by removing links that are in the SRLG and run a shortest path algorithm. If there is a SRLG for which no single path can be found then there cannot be two paths that are SRLG-disjoint and the algorithm stops.

We pick a SRLG that is in both paths, say A, and add $\{(A, \emptyset)\}$ and $\{(\emptyset, A)\}$ to $\mathcal{T}$. $\mathcal{T}$ is the set we still have to evaluate. We pick an element in $\mathcal{T}$ with the least number of SRLGs, say $\{(A, \emptyset)\}$. If SRLG-exclusion$(\mathcal{G}, n_s, n_d, \{A\}, \{\})$ returns two paths then we know that the first path will not have links that are in SRLG A. So the two paths will be disjoint for SRLG A. If the two paths are disjoint for all SRLGs then we found a possible solution, but if the two paths have another overlapping SRLG B then we remove the current element $\{(A, \emptyset)\}$ from $\mathcal{T}$ that is being evaluated and add $\{(AB, \emptyset)\}$ and $\{(A, B)\}$ to $\mathcal{T}$. Our set $\mathcal{T}$ now contains $\{(\emptyset, A)\}$, $\{(AB, \emptyset)\}$ and $\{(A, B)\}$. We now evaluate the next element of $\mathcal{T}$ with the least SRLGs, $\{(\emptyset, A)\}$. This loop continues until the set $\mathcal{T}$ is empty. Every time a solution is found, the cost is compared with the

currently optimal solution. If the cost is lower then the new solution is saved.

If we calculate SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b$) where $\mathcal{R}_a$ and $\mathcal{R}_b$ are sets of SRLGs, there are three possible outcomes. SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b$) has a solution $(\mathcal{P}_a, \mathcal{P}_b)$ where the SRLGs of both paths are disjoint so $\mathcal{R}(\mathcal{P}_a) \cap \mathcal{R}(\mathcal{P}_b) = \emptyset$. The second possibility, SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b$) finds two paths with one or more SRLGs that are in both paths, $\mathcal{R}(\mathcal{P}_a) \cap \mathcal{R}(\mathcal{P}_b) \neq \emptyset$. And the last possibility is that SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b$) does not find a solution.

Theorem 3.2.1. *Let $\mathcal{R}_a, \mathcal{R}_b, \mathcal{R}_a^+, \mathcal{R}_b^+$ be sets of SRLGs with $\mathcal{R}_a \subseteq \mathcal{R}_a^+$ and $\mathcal{R}_b \subseteq \mathcal{R}_b^+$. If SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b$) has a SRLG disjoint solution $(\mathcal{P}_a, \mathcal{P}_b)$ then SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a^+, \mathcal{R}_b^+$) cannot have a solution $(\mathcal{P}_a^+, \mathcal{P}_b^+)$ with $\mathcal{C}(\mathcal{P}_a^+, \mathcal{P}_b^+) < \mathcal{C}(\mathcal{P}_a, \mathcal{P}_b)$.*

Proof. SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b$) has a SRLG disjoint solution $(\mathcal{P}_a, \mathcal{P}_b)$ with minimal cost where the primary path is excluded from SRLGs in $\mathcal{R}_a$ and the backup path is excluded from SRLGs in $\mathcal{R}_b$. Let us assume that SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a^+, \mathcal{R}_b^+$) returns a solution $(\mathcal{P}_a^+, \mathcal{P}_b^+)$ with $\mathcal{C}(\mathcal{P}_a^+, \mathcal{P}_b^+) < \mathcal{C}(\mathcal{P}_a, \mathcal{P}_b)$. Because the primary path $\mathcal{P}_a^+$ is excluded from SRLGs in $\mathcal{R}_a^+$, it is also excluded from SRLGs in $\mathcal{R}_a$ because $\mathcal{R}_a \subseteq \mathcal{R}_a^+$. Analog is $\mathcal{P}_b^+$ excluded from SRLGs in $\mathcal{R}_b \subseteq \mathcal{R}_b^+$. So paths $(\mathcal{P}_a^+, \mathcal{P}_b^+)$ are also a valid solution for SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b$), but with less cost than $(\mathcal{P}_a, \mathcal{P}_b)$. But SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b$) gives a solution with minimal cost, so this is a contradiction and $\mathcal{C}(\mathcal{P}_a^+, \mathcal{P}_b^+) \geq \mathcal{C}(\mathcal{P}_a, \mathcal{P}_b)$. □

The result is that if we evaluate SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b$) and find a SRLG disjoint solution, we do not have to search any further for solutions of SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a^+, \mathcal{R}_b^+$). If SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b$) does not find a solution it is evident that SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a^+, \mathcal{R}_b^+$) also will not find a solution.

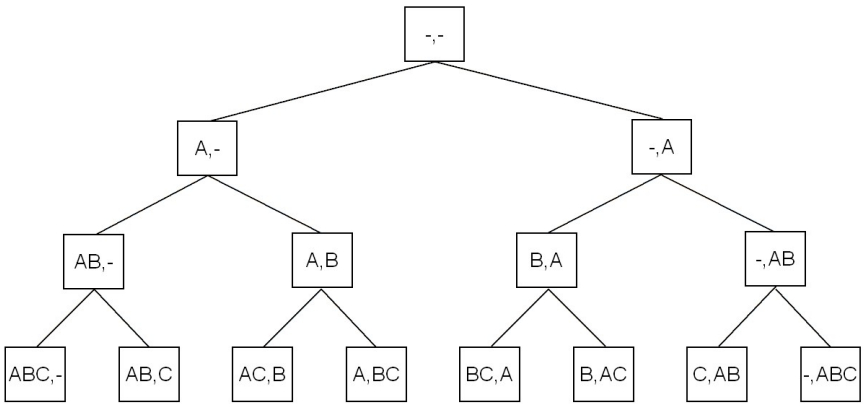


Figure 3.1: Tree-view

Assume a network has three SRLGs: A, B and C and we write SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b$) as $(\mathcal{R}_a, \mathcal{R}_b)$. Then all the possibilities are shown in figure 3.1. The algorithm starts at the top with calculating (-,-) which stands for

SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b$) with $\mathcal{R}_a = \mathcal{R}_b = \emptyset$. If two paths are found which are not SRLG-disjoint one SRLG is picked, A in the example, and from the top node we make two branches (A,-) and (-,A), and add them to the set $\mathcal{T}$ of nodes which still have to be parsed. The top node we have parsed is removed from $\mathcal{T}$. We then pick from this set the next node with the least number of SRLGs which is either (A,-) or (-,A). We pick (A,-) and run SRLG-exclusion. If we find two disjoint paths then we found a solution and from theorem 3.2.1 we know that further parsing the children of node (A,-) will not result in a solution with less cost. If (A,-) does not find a solution then children of (A,-) also do not have solutions. If (A,-) results again in two paths which is not SRLG-disjoint then we know that both paths are disjoint for A. In the example the paths share B and the node branches into (AB,-) and (A,B). Next (-,A) is picked and the this process continuous until $\mathcal{T}$ is empty.

3.3 Analyses

Let $\mathcal{G}$ be a network and $\mathcal{R}_\mathcal{G}$ the set of all SRLGs in the network. Now let $(\mathcal{P}_a^*, \mathcal{P}_b^*)$ be two SRLG-disjoint paths with minimal cost in $\mathcal{G}$. We now define

$$\mathcal{R}_a^* = \{r \in \mathcal{R}_\mathcal{G} \mid r \notin \mathcal{R}(\mathcal{P}_a^*)\}$$

$$\mathcal{R}_b^* = \{r \in \mathcal{R}_\mathcal{G} \mid r \notin \mathcal{R}(\mathcal{P}_b^*)\}$$

Lemma 3.3.1. *If $(\mathcal{P}_a^*, \mathcal{P}_b^*)$ is SRLG disjoint then $\mathcal{R}_a^* \cup \mathcal{R}_b^* = \mathcal{R}_\mathcal{G}$.*

Proof. Assume $\mathcal{R}_a^* \cup \mathcal{R}_b^* \neq \mathcal{R}_\mathcal{G}$. Then $\exists r \in \mathcal{R}_\mathcal{G}$ with $r \notin \mathcal{R}_a^*$ and $r \notin \mathcal{R}_b^*$. But then $r \in \mathcal{R}(\mathcal{P}_a^*)$ and $r \in \mathcal{R}(\mathcal{P}_b^*)$. This is a contradiction, because the two paths are SRLG disjoint. So $\mathcal{R}_a^* \cup \mathcal{R}_b^* = \mathcal{R}_\mathcal{G}$. □

Lemma 3.3.2. *Let $(\mathcal{P}_a, \mathcal{P}_b)$ be a SRLG disjoint path in $\mathcal{G}$ and let $(\mathcal{P}_a^*, \mathcal{P}_b^*)$, $\mathcal{R}_a^*$, $\mathcal{R}_b^*$ be defined as above. The algorithm SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a^*, \mathcal{R}_b^*$) or SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_b^*, \mathcal{R}_a^*$) returns a SRLG disjoint solution $(\mathcal{P}_a^*, \mathcal{P}_b^*)$ with $\mathcal{C}(\mathcal{P}_a^*, \mathcal{P}_b^*) \leq \mathcal{C}(\mathcal{P}_a, \mathcal{P}_b)$*

Proof. SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a^*, \mathcal{R}_b^*$) or SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_b^*, \mathcal{R}_a^*$) returns the optimal solution $(\mathcal{P}_a^*, \mathcal{P}_b^*)$. Then by definition if $(\mathcal{P}_a, \mathcal{P}_b)$ is an optimal solution then $\mathcal{C}(\mathcal{P}_a, \mathcal{P}_b) = \mathcal{C}(\mathcal{P}_a^*, \mathcal{P}_b^*)$ else $\mathcal{C}(\mathcal{P}_a^*, \mathcal{P}_b^*) < \mathcal{C}(\mathcal{P}_a, \mathcal{P}_b)$. □

Theorem 3.3.3. *SRLG-tree finds two SRLG-disjoint paths with minimal cost in a network $\mathcal{G}$, if such paths exists.*

Proof. Assume an optimal SRLG disjoint solution $(\mathcal{P}_a^*, \mathcal{P}_b^*)$ exists. Let $\mathcal{R}_a^*$, $\mathcal{R}_b^*$ be defined as above. The quest for the optimal disjoint path in the algorithm begins with SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b$) where $\mathcal{R}_a = \mathcal{R}_b = \emptyset$. As we will see, for every iteration holds $\mathcal{R}_a \subseteq \mathcal{R}_a^*$ and $\mathcal{R}_b \subseteq \mathcal{R}_b^*$. We already described that there are three possible results. The first is that we find a disjoint solution $(\mathcal{P}_a, \mathcal{P}_b)$. Then because of theorem 3.2.1 $\mathcal{C}(\mathcal{P}_a, \mathcal{P}_b) \leq \mathcal{C}(\mathcal{P}_a^*, \mathcal{P}_b^*)$. Because $(\mathcal{P}_a^*, \mathcal{P}_b^*)$ is an optimal disjoint solution, it must be that $\mathcal{C}(\mathcal{P}_a, \mathcal{P}_b) = \mathcal{C}(\mathcal{P}_a^*, \mathcal{P}_b^*)$ and the iterations stop. The second option is that SRLG-exclusion($\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b$) does not find a solution. But this is not possible because $(\mathcal{P}_a^*, \mathcal{P}_b^*)$ is a valid solution. The third possible outcome is a path $(\mathcal{P}_a, \mathcal{P}_b)$ which is not SRLG disjoint. The algorithm takes a $r \in \mathcal{R}(\mathcal{P}_a) \cap \mathcal{R}(\mathcal{P}_b)$ and

creates two new searches SRLG-exclusion$(\mathcal{G}, n_s, n_d, \mathcal{R}_a \cup \{r\}, \mathcal{R}_b)$ and SRLG-exclusion$(\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b \cup \{r\})$. If $r \in \mathcal{R}_a^*$ then in this proof we only consider SRLG-exclusion$(\mathcal{G}, n_s, n_d, \mathcal{R}_a \cup \{r\}, \mathcal{R}_b)$ otherwise we will look at SRLG-exclusion$(\mathcal{G}, n_s, n_d, \mathcal{R}_a, \mathcal{R}_b \cup \{r\})$. Actually if $r \in \mathcal{R}_a^*$ and $r \in \mathcal{R}_b^*$ then we could focus on either one. From this follows that the new $\mathcal{R}_a \subseteq \mathcal{R}_a^*$ and new $\mathcal{R}_b \subseteq \mathcal{R}_b^*$. From here we move on to the next iteration. Because of lemma 3.3.2, if $\mathcal{R}_a = \mathcal{R}_a^*$ and $\mathcal{R}_b = \mathcal{R}_b^*$ then an optimal disjoint solution is found and the iterations stop. So during the iterations $\mathcal{R}_a \subseteq \mathcal{R}_a^*$ and $\mathcal{R}_b \subseteq \mathcal{R}_b^*$. □

3.4 Time complexity

Assume a network with nodes $\mathcal{N}$, and links $\mathcal{L}$ and R SRLGs. In each iteration of the SRLG-tree algorithm the SRLG-exclusion algorithm is ran. We have shown that the time complexity of SRLG-exclusion is $\mathcal{O}(|\mathcal{L}| + |\mathcal{N}|log(|\mathcal{N}|) + |\mathcal{N}|)$. In the worst-case the SRLG-tree algorithm has to iterate through the complete tree shown in figure 3.1 which has $2^{R+1} - 1$ nodes. For SRLG-tree this gives a time complexity of $O((2^{R+1} - 1) \cdot (3|\mathcal{L}| + 3|\mathcal{N}|log(|\mathcal{N}|) + 2|\mathcal{N}|log(2)) = O(2^R \cdot (|\mathcal{L}| + |\mathcal{N}|log(|\mathcal{N}|) + |\mathcal{N}|)$.

3.5 Improvement

Finding a disjoint solution as soon as possible is important, because with this solution certain branches of the parsing tree can be disregarded. Theorem 3.2.1 states that no child nodes can have a solution with less cost than the solution of the parent node. So if we have a disjoint solution we can compare it with solutions of other nodes. If the cost of a solution of a node has higher cost then we do not need to explore that node any further.

To create a disjoint solution SRLG-tree can use the primary path it has already found in its iteration and than search a disjoint backup path. This can be done by eliminating all links that share an SRLG with the primary path and than run a shortest path algorithm. By doing this in each iteration SRLG-tree quickly obtains disjoint solutions and is able to compare the cost with solutions of tree nodes.

3.6 Worst-case scenario's

The worst-case scenario for SRLG-tree is that it will run SRLG-exclusion $2^{R+1} - 1$ times, where R is the number of riskgroups. This is for example the case in the following situation. The algorithm starts with Bhandari. Assume this results in two paths which share SRLG A. Next SRLG-exclusion will run which will eliminate A being shared by both paths, but these new paths now share B. If SRLG-exclusion searches for paths disjoint in A and B, a new shared SRLG C pops up. This continues until all R SRLGs are encountered. In this situation SRLG-tree performs poorly and has to run SRLG-exclusion $2^{R+1} - 1$ times and performs with time complexity $O(2^R \cdot (|\mathcal{L}| + |\mathcal{N}|log(|\mathcal{N}|) + |\mathcal{N}|)$.

In figure 3.2 an example of such network is shown. This network is strongly simplified and has only a source node s and a destination node d, and has four SRLGs A, B and C. The link cost/length are within the brackets beside the

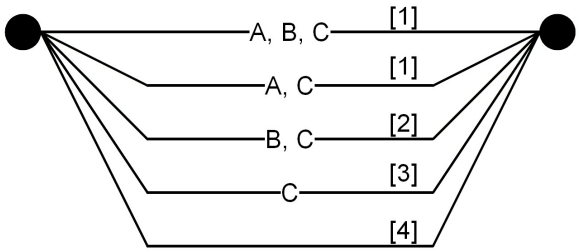


Figure 3.2: Worst-case scenario example network

links. Starting with Bhandari, this returns two paths which share A and C. SRLG-tree picks one of these SRLGs, we take A. SRLG-exclusion returns a path which is disjoint for A, but now we have paths that share B. The tree that is fully parsed by SRLG-tree is shown in figure 3.1. All nodes at the lowest level result in a SRLG-disjoint solution, but only $ABC, -$ is the optimal solution. We have to note that if SRLG-tree picked C when Bhandari returned two paths that shared A and C then SRLG-exclusion would have directly resulted in the optimal solution.

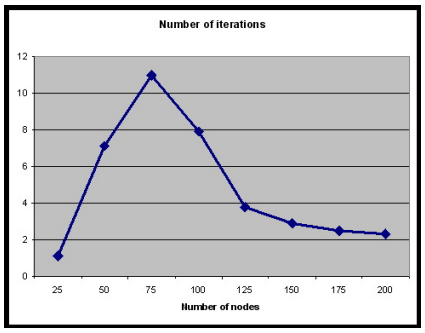
3.7 Performance of SRLG-tree

For measuring the performance of the SRLG-tree algorithm we create two kind of networks, random and scale-free networks. The default network has 100 nodes, 15 SRLGs and SRLG probability, the probability of a link being in a SRLG, of 0,1. In case of the random network the link probability of the default network is 0,05. We will vary each variable while keeping the others static.

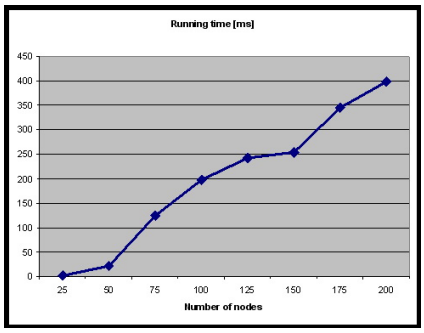
When the random network is created each possible link between any node pair has probability p to be in the network. For creation of the scale-free network a power law distribution function is used. The link probability of the nodes are uniformly distributed along the x axis of then the corresponding link probability is given by the power law function. This probability varies between 0 and 1. Then for each possible link between any node pair (m, n) with link probabilities p_m and p_n, the probability for the link to be in the network is $max(p_m, p_n)$.In both type of networks we also vary the probability p_r that a link is in a SRLG. For each link and for each SRLG it has probability p_r that the link is in that SRLG.

For each of the different networks with N nodes and R SRLGs we created 10 random sample networks with N nodes and R SRLGs and in each sample network calculated 10 random protected paths. We measured the running time of the SRLG-tree algorithm and the number of nodes in the iteration tree of the SRLG-tree algorithm. The averages of these 100 outcomes are calculated and shown in the figures below.

For low number of nodes and/or low link probability the network can be disconnected and SRLG-tree quickly detects this. For example in figure 3.3a the network will be disconnected in the beginning when the number of nodes is low. In both types of networks similar graphs show for varying the SRLG probability. First both number of iterations and running time grow exponentially, but then

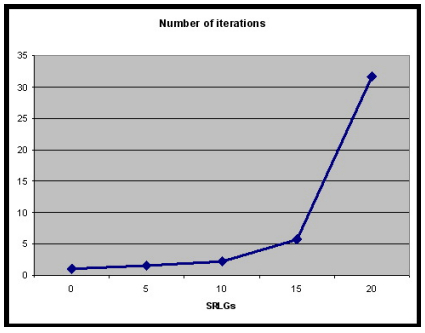


(a) Iterations

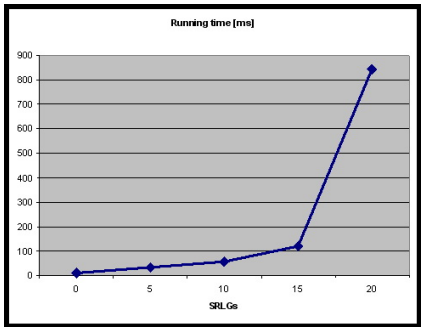


(b) Running time

Figure 3.3: Random network: Variable number of nodes



(a) Iterations

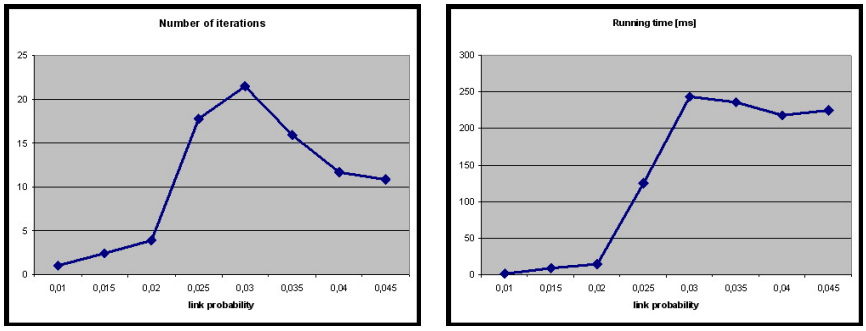


(b) Running time

Figure 3.4: Random network: Variable number of SRLGs

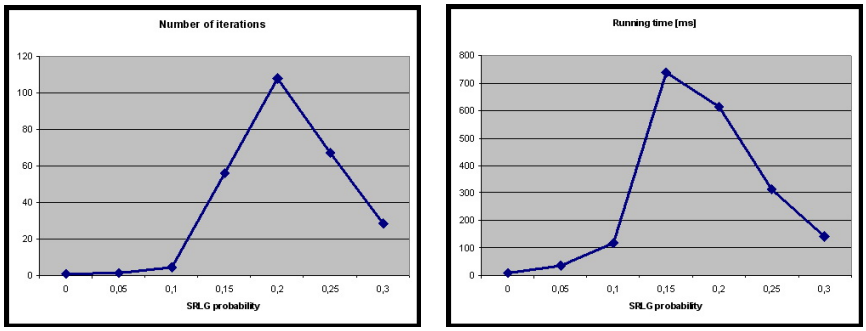
the growth reverses. An explanation for this can be that if links are in more SRLGs then the possible number of disjoint pairs in the network decreases. The same we see in the figure of the random network where we vary the number of nodes. If the network becomes larger the change that a disjoint path between two random nodes is possible becomes smaller. In case such path is not possible then SRLG-tree detects this in one iteration.

The maximum number of iterations is $2^{R+1} - 1$ where R is the number of SRLGs and this gives the maximum iterations in the parsing tree in the worst-case scenario. In average SRLG-tree parses only a fraction of this. In the cases where $SRLGs = 0$ SRLG-tree only needs 1 iteration and performs equal to Bhandari's algorithm.



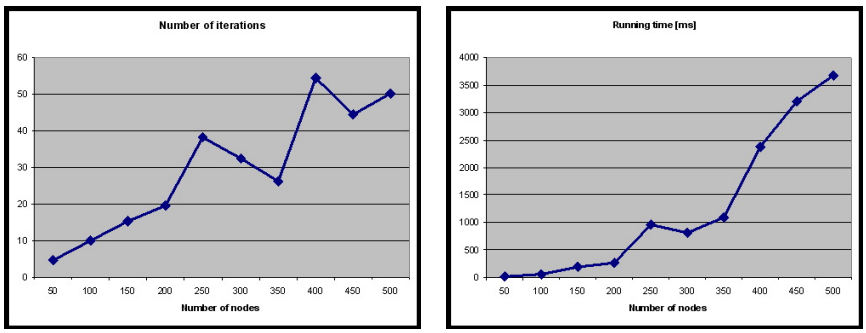
(a) Iterations (b) Running time

Figure 3.5: Random network: Variable link probability



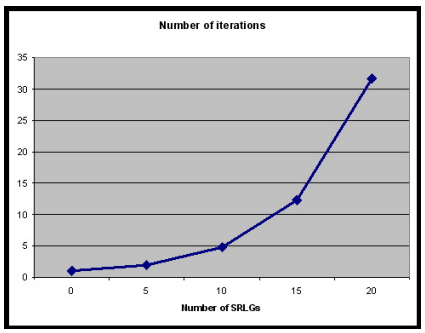
(a) Iterations (b) Running time

Figure 3.6: Random network: Variable SRLG probability

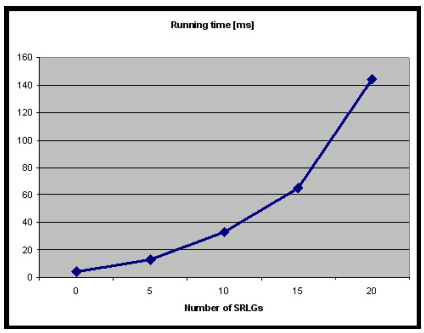


(a) Iterations (b) Running time

Figure 3.7: Scale-free network: Variable number of nodes

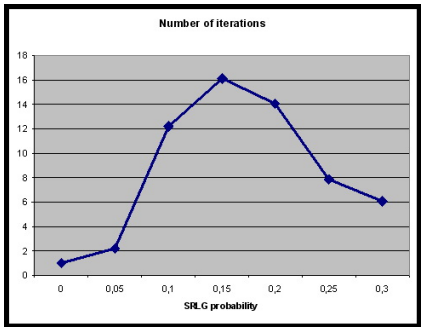


(a) Iterations

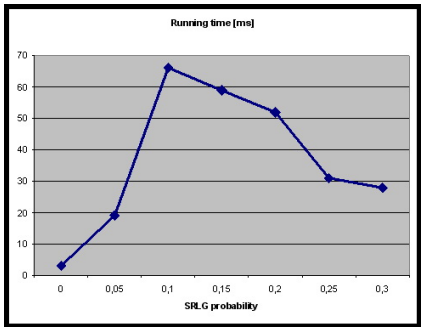


(b) Running time

Figure 3.8: Scale-free network: Variable number of SRLGs



(a) Iterations



(b) Running time

Figure 3.9: Scale-free network: Variable SRLG probability

Chapter 4

SRLG-tree in subnetwork topology networks

4.1 Introduction

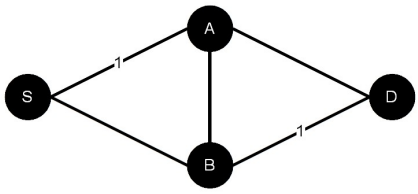


Figure 4.1: Protected path with shortcut

The SURFnet network consists of separate rings. These rings are connected to each other in two center states, Amsterdam1 and Amsterdam2. These center states or hubs are directly connected with each other. A protected path between a source and destination node on separate rings always go via the two hubs, the primary path through one hub and the backup path through the other. Sometimes there is no such SRLG-disjoint protected path possible.

Assume that we are allowed to make a shortcut between a path pair between a source and destination node. The simplest example of a protected path with a shortcut is shown in figure 4.1. The protected path consists of a pair of paths, the primary path over nodes $S \rightarrow A \rightarrow D$ and a backup path over nodes $S \rightarrow B \rightarrow D$. The shortcut is the path $A \rightarrow B$ which connects the primary path and the backup path. In a protected path with a shortcut there are four ways to go from the source node to the destination. Besides the primary path and backup path, by using the shortcut there are also the paths $S \rightarrow A \rightarrow B \rightarrow D$ and $S \rightarrow B \rightarrow A \rightarrow D$ possible. These two shortcut paths are not link/node disjoint with the primary path or backup path. A shortcut path uses the first part of the primary path until the starting node of the shortcut, then goes over the shortcut and finally uses the second part of the backup path to reach the

destination node. And the other shortcut path goes vice versa. If a primary path and a backup path share a SRLG then with a shortcut a connection could survive if this SRLG fails. In figure 4.1 the primary path and the backup path share SRLG 1. If this group fails then because of path $S \rightarrow B \rightarrow A \rightarrow D$ connection between source and destination survives. But if for example the shared SRLG 1 had links (S, A) and (S, B) then with even with the shortcut the connection does not survive. To use the same terminology we will call a protected path with a shortcut, which can survive a single SRLG failure, a SRLG-disjoint path.

Let $(\mathcal{P}_a, \mathcal{P}_b)$ be a protected path between a source node n_s and a destination node n_d, then a shortcut $\mathcal{P}_s$ for $(\mathcal{P}_a, \mathcal{P}_b)$ is a path between a node n_a in $\mathcal{P}_a$ to a node n_b in $\mathcal{P}_b$ where n_a and/or n_b are not the source or destination node. We also demand that n_a and n_b are the only nodes in $\mathcal{P}_s$ that are in $\mathcal{P}_a$ or $\mathcal{P}_b$. Below we give a formal definition.

Definition 4.1.1. *Given a protected path $(\mathcal{P}_a, \mathcal{P}_b)$ between source n_s and destination n_d in a network $\mathcal{G}$ with nodes $\mathcal{N}$ and links $\mathcal{L}$. Path $\mathcal{P}_s = \bigcup_{i=0..n}(u_i, v_i)$ where $u_i, v_i \in \mathcal{N},\ i = 0..n$ is a shortcut for $(\mathcal{P}_a, \mathcal{P}_b)$ if*

$$u_0 \in \mathcal{P}_a \backslash \{n_s, n_d\}$$

$$v_n \in \mathcal{P}_b \backslash \{n_s, n_d\}$$

$$\bigcup_{i=1..n-1} (u_i, v_i) \cap (\mathcal{P}_a \cup \mathcal{P}_b) = \emptyset$$

We call $(\mathcal{P}_a, \mathcal{P}_b, \mathcal{P}_s)$ a "protected path with shortcut".

Assume a network which is constructed of multiple subnetworks. These subnetworks are not directly connected to each other, but are connected through hubs. An example of such topology with only one hub is shown in figure 4.2. Nodes in the same subnetwork can connect directly through the subnetwork they are in, but nodes in different subnetworks have to connect through a hub.

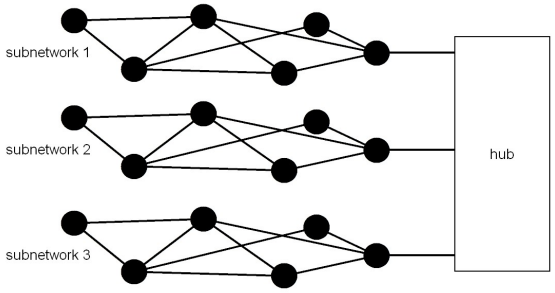


Figure 4.2: ring topology

The SURFnet network consists of rings which are connected in two center states, Amsterdam1 and Amsterdam2, which are directly connected with each other. Therefore we assume a network with two hubs where the two hubs are directly connected. We will regard this direct connection between the hubs as the shortcut described above. If there is a protected path with shortcut between

two nodes on different subnetworks then a flat view of the path in the network can look like depicted in figure 4.3 and shows the four possible paths from source n_s to destination n_d through hubs/nodes h_1 and h_2.

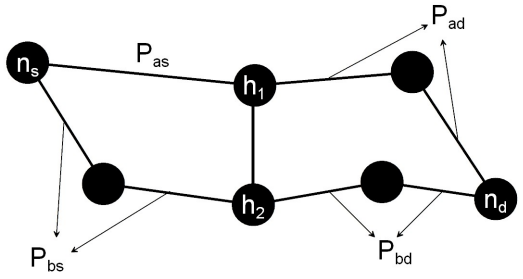


Figure 4.3: Protected path with shortcut between two hubs

4.2 Problem definition

Assume an undirected network $\mathcal{G}$ with SRLGs, constructed of subnetworks which are connected through two hubs. Network $\mathcal{G}$ has nodes $\mathcal{N}$ of which two hub nodes h_1, h_2 and links $\mathcal{L}$. The link between the two hubs is not in any SRLG. Let n_s be a source node and n_d a destination node with $n_s, n_d \in \mathcal{N}$. The objective is to find a SRLG-disjoint protected path with shortcut $(\mathcal{P}_a, \mathcal{P}_b, \mathcal{P}_s)$ between nodes n_s and n_d with minimal cost and with the shortcut between h_1 and h_2.

4.3 Related work

For this problem we need an algorithm that is able to calculate a SRLG-disjoint path between a source node and two destination nodes. The destination nodes are the hubs. We can then calculate a disjoint path from source to the two hubs and from the destination to the two hubs. If we combine these two protected paths and add a shortcut between the two hubs then the result is also SRLG-disjoint. Every algorithm in section 1.4 which is able to calculate a disjoint path between a source and two destination nodes can be used for this problem.

4.4 Algorithm

We are interested in describing an algorithm to find a protected path with shortcut that is SRLG-disjoint with least total cost. Let $\mathcal{G}$ be a network with nodes $\mathcal{N}$ and links $\mathcal{L}$. Besides a source node n_s and a destination node n_d, we have two hub nodes h_1 and h_2 which are the end nodes of the shortcut path. We define the path from n_s to h_1 as $\mathcal{P}_{as}$, the path from n_s to h_2 as $\mathcal{P}_{bs}$, the path from h_1 to n_d as $\mathcal{P}_{ad}$ and the path from h_2 to n_d as $\mathcal{P}_{bd}$ as shown in figure 4.3.

For the protected path with shortcut to be SRLG-disjoint we need the condition that $\mathcal{R}(\mathcal{P}_{as}, \mathcal{P}_{bs}) = \emptyset$ and $\mathcal{R}(\mathcal{P}_{ad}, \mathcal{P}_{bd}) = \emptyset$ holds. So the SRLG-diverse routing problem is split up in two subparts. The first subpart is a diverse routing problem from a source n_s to two different destinations h_1 and h_2. The other subpart is the SRLG-disjoint route from h_1 and h_2 to n_d. If both sub problems have SRLG-disjoint solutions then we can join the solutions together to obtain a protected path with shortcut between n_s and n_d which is SRLG-disjoint.

With minor changes the algorithms in this work can also be applied to paths with one or two sources and/or one or two destinations. The algorithm SRLG-exclusion has parameters n_s for the source and n_d for the destination node. SRLG-exclusion returns two paths with the same source and destination. To enable the algorithm to return two paths with different sources and destinations it first has to get two extra parameters. Instead of the parameters n_s and n_d the algorithm gets n_{s1}, n_{s2}, n_{d1} and n_{d2}. If we want two paths with the same source node, but different destination node then parameters n_{s1} and n_{s2} are set the same and n_{d1} and n_{d2} different.

The main algorithm SRLG-tree needs also extra parameters. Also here we have instead of parameters n_s and n_d the new parameters n_{s1}, n_{s2}, n_{d1} and n_{d2}. These new parameters are given as input to the new SRLG-exclusion in SRLG-tree.

With this new SRLG-tree algorithm we are now able to solve the two sub problems of the SRLG diverse routing problem with shortcut. To obtain a SRLG-disjoint protected path between n_s and n_d with the given shortcut between node h_1 and h_2. The new SRLG-tree algorithm has to be used twice.

$$(\mathcal{P}_{as}, \mathcal{P}_{bs}) = SRLG - tree(\mathcal{G}, n_s, n_s, h_1, h_2)$$

$$(\mathcal{P}_{ad}, \mathcal{P}_{bd}) = SRLG - tree(\mathcal{G}, h_1, h_2, n_d, n_d)$$

If both return a solution then we can create a protected path with shortcut $(\mathcal{P}_a, \mathcal{P}_b, \mathcal{P}_s)$ that is SRLG-disjoint with $\mathcal{P}_a = \mathcal{P}_{as} \cup \mathcal{P}_{ad}$ and $\mathcal{P}_b = \mathcal{P}_{bs} \cup \mathcal{P}_{bd}$. $(\mathcal{P}_a, \mathcal{P}_b, \mathcal{P}_s)$ is then a protected path with shortcut with least total cost.

If a solution to the SRLG diverse routing problem can optionally have a shortcut then problem has to be split up in the normal SRLG diverse routing problem (without shortcut) and the SRLG diverse routing problem with shortcut. If both produce a result the one with the least cost is chosen.

4.5 Time complexity

Let a $\mathcal{G}$ be a network with nodes $\mathcal{N}$ and links $\mathcal{L}$. The number of subnetworks of $\mathcal{G}$ is S. Assume a source node in a subnetwork and a destination node in another subnetwork. We want to know the time complexity for a protected path between these nodes over the two hub nodes. In each subnetwork are $|\mathcal{N}|/S$ nodes and $|\mathcal{L}|/S$ links on average. SRLG-tree is ran for each subnetwork. So the time complexity is $2 \cdot O((2^{R+1} - 1) \cdot (3|\mathcal{L}|/S + 3|\mathcal{N}|log(|\mathcal{N}|/S)/S + 2|\mathcal{N}|log(2)/S))$ with $S \geq 2$. The maximum value is obtained for $S = 2$ which gives $O((2^{R+1} - 1) \cdot (3|\mathcal{L}| + 3|\mathcal{N}|log(|\mathcal{N}|/2) + 2|\mathcal{N}|log(2))) = O(2^R \cdot (|\mathcal{L}| + |\mathcal{N}|log(|\mathcal{N}|) + |\mathcal{N}|))$.

Chapter 5

Implementation

For the practical work of my thesis, I worked at SARA in Amsterdam. I developed a web-based Java/J2EE application "PathPlanner", which implements the SRLG-tree algorithm, for planning lightpaths in SURFnet. The functionalities of PathPlanner are:

- Calculate the optimal single or protected path between two nodes.
- Display information about active lightpaths.
- Display ranking of all active lightpaths sorted on efficiency.
- Display hot spots.

While planning a path, the user can prefer and/or exclude certain routes, nodes. This gives the user some control over the planning of the path. Displaying information of a lightpath gives the user detailed information for example which route the path takes and which devices the path uses in lower layers of the network. For the ranking for each lightpath the following costs are calculated: cost of current active path, cost of rerouted path and cost of rerouted path in an empty network. For the cost of the rerouted path the active path is first removed from the network and than rerouted with SRLG-tree. The percentage of improvement is also shown and the list is sorted on this percentage from high to low. This gives the user an overview of lightpaths that are not provisioned optimally. The last functionality of PathPlanner is the list of hot spots. Here first all active lightpaths are routed in an empty network and than all calculated paths are added to an empty network. A list of all nodes with the needed capacity is shown. This gives the user an idea which nodes could be potential bottlenecks. If the for example the capacity of a node is exceeded then there could be lightpaths that are not routed optimal. By increasing the capacity of this node, resources could be reduced.

Chapter 6

Further investigations

- Can the approach of SRLG-tree can be adapted to be used for shared protection routing?
- If a near-optimal solution is sufficient, how much can be gained on the running time?
- How "near" is this solution to the optimal solution?
- Ragarding the routing problem with shortcuts from chapter 4, is there an efficient exact algorithm for non-fixed shortcut paths?
- Are there good heuristics for this non-fixed shortcut problem?

Chapter 7

Appendix

7.1 Snapshots Pathplanner on SURFnet

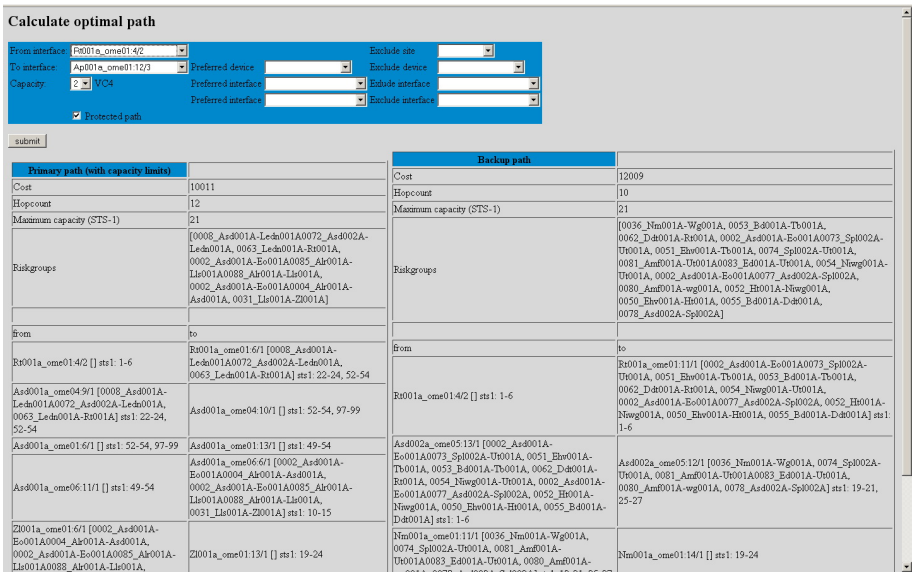


Figure 7.1: Protected path between Rotterdam and Apeldoorn

7.2 Extract source code PathPlanner

```
private List<List<Interface>> SRLGtree(List<Interface> from,
        List<Interface> to, int sts1capacity, boolean noifcweight) {

    if (from == null || to == null || from.size() == 0
            || from.size() != to.size()) {
        return null;
    }

    boolean protectedpath = from.size() > 1;
```

Interface ranking

Interface	Used timeslots reroute	Current used timeslots	Capacity
Ut001a_ome01:6/1	231	132	192
Asd001a_ome04:12/1	231	132	192
Asd001a_ome03:2/1	216	192	192
Asd001a_ome10:5/1	216	192	192
Asd001a_ome02:14/1	210	150	192
Asd001a_ome03:11/1	210	150	192
Rt002a_ome01:5/1	192	192	192
Rt001a_ome01:13/1	192	192	192
Rt001a_ome01:12/1	192	192	192
Rt002a_ome01:10/1	192	192	192
Rt001a_ome01:14/1	192	192	192
Rt002a_ome01:9/1	192	192	192
Asd002a_ome03:14/1	189	120	192
Asd001a_ome10:9/1	189	120	192
Asd001a_ome07:10/1	183	189	192
Asd001a_ome02:6/1	183	189	192
Gv001a_ome02:5/1	180	159	192
Gv002a_ome01:9/1	180	159	192
Asd001a_ome05:9/1	174	174	192
Ehv001a_ome01:6/1	174	174	192
Asd002a_ome05:6/1	171	174	192
Tb001a_ome01:9/1	171	174	192
Asd002a_ome03:9/1	165	129	192
Asd001a_ome10:6/1	165	126	192
Dt001b_ome01:9/1	165	129	192
Asd001a_ome04:14/1	165	126	192
Gv001a_ome02:10/1	162	183	192
Gv002a_ome01:6/1	162	183	192

Figure 7.2: “Hotspots” in the network

Lightpath ranking

Lightpath	Inefficiency	Current cost	Rerouted cost	Rerouted cost empty network
DRAC-loopback	100,000%	1	0	0
2054LP_Ht01-Ehv01_FONTYS-O06(Ht02-OPN)	98,717%	1014016	13014	13014
2141LP_Dt08-Ut01_TNO-O22(Dt08-Ut32)	90,080%	2218014	220020	219518
2069LR2_Mdb01-Vs01_HZEELAND-Zebi-O28	79,995%	15011	3003	3003
2038LR2_Ah01-Nm01_HAN-O07	74,994%	12009	3003	3003
2040LR2_Ah01-Nm01_HAN-O07	74,994%	12009	3003	3003
2067LE_Ap01-Ah01_LSOP-O14(NP_Ap01-Ap02)	74,994%	12009	3003	3003
2034LR2_Amr01-Asd02_OU-O15(Amr02-OPN)	72,192%	18013	5009	5009
2107LR2_Ap01-Lls01_LSOP-O14(OPN)	62,504%	16015	6005	6005
2094LP_Ah01-Nm01_UMCN	59,992%	15012	6006	6006
2078LR2_Ut01-Dt08_TNO-O22(Zt01-OPN)	57,153%	14015	6005	6005
2113LR2_Vl02-Asd02_BOBZ-O13(Vl03) P15238	55,527%	18013	8011	8011
2073LR1_Lw01-Asd01_OU-O15(Lw-OPN)	51,583%	15513	7511	7511
0028IR2_Vs01-Bd01_HZEELAND	49,996%	12009	6005	6005
2047LP_Asd01-Asd02_OWINSP-O20(B2B-OPN)	49,900%	8012	4014	4014
2041LR2_Lw01-Ap01_LSOP-O14(Drachten-OPN)	47,992%	25021	13013	13013
2030LE_Ut01-Asd01_UU-SARA(EYR)	46,135%	6507	3505	3505
2145LP_Ut02-Ut01_SURFnet(DLP1) P15351	44,646%	215012	119018	119018
2146LP_Ut02-Ut01_SURFnet(DLP2) P15351	44,646%	215012	119018	119018
2123LR2_Amr01-Asd02_IKA@Amr05(DigiBobOPN) P14632	44,388%	9007	5009	5009
2118LR2_Ap01-Gw01_LSOP-O14(Gv16)	43,759%	16015	9007	9007
2093LR2_Ap01-Boz01_LSOP-O14(Osd01-Ap01)	42,851%	28023	16015	16015
2130LR2_Rt01-Vs01_SBBZWN-O14(Vdg02-Gs01)	42,316%	26023	15011	15011

Figure 7.3: Ranking of currently most inefficient planned lightpaths

```
System.out.println("\nCalculate path\nfrom: " + from + "\nto: " + to
        + "\nprotected: " + protectedpath);

capacity = sts1capacity;

Set<List<Set<String>>> todo = new HashSet<List<Set<String>>>();
Set<String> necessaryriskgroups = new HashSet<String>();
boolean checkednecessary = false;
List<Set<String>> ls = new ArrayList<Set<String>>();
ls.add(new HashSet<String>());
ls.add(new HashSet<String>());
todo.add(ls);
List<Set<String>> current = null;
List<List<Interface>> shortestDijkstra = new ArrayList<List<Interface>>();
shortestDijkstra.add(new ArrayList<Interface>());
shortestDijkstra.add(new ArrayList<Interface>());
int shortestDijkstraCost = 0;
exploredsets.clear();
int iterations = 0;
while (todo.size() > 0) {
    System.out.println(todo);
    iterations++;
    int min = network.riskgroups.size() + 1;
    for (List<Set<String>> lss : todo) {
        int sum = lss.get(0).size() + lss.get(1).size();
        if (sum < min) {
            min = sum;
            current = lss;
        }
    }
    // Suurballe free
```

```
System.out.println("first path without " + current
        + ", backup path free");
iterationresultfree = new ArrayList<List<Interface>>();
iterationresultnotfree = new ArrayList<List<Interface>>();
iterationresultfree.add(Dijkstra(from.get(0), to.get(0), current
        .get(0), false, true));
iterationresultnotfree.add(new ArrayList<Interface>());
for (Interface ifc : iterationresultfree.get(0)) {
    iterationresultnotfree.get(0).add(ifc);
}

if (protectedpath && iterationresultfree.get(0).size() > 0) {
    FlagLinks();
    iterationresultfree.add(Dijkstra(from.get(1), to.get(1),
            current.get(1), false, false));
    RemoveCommonLinks(iterationresultfree.get(0),
            iterationresultfree.get(1));
} else {
    iterationresultfree.add(new ArrayList<Interface>());
}
// check if two paths are free of riskgrous in current.get(0) and current.get(1)
List<List<Interface>> history = new ArrayList<List<Interface>>(); //check for cycles
while (iterationresultfree.get(1).size() > 0
        && RiskgroupsOverlap(RiskgroupsInPath(iterationresultfree
                .get(1)), current.get(1))
        && history.indexOf(iterationresultfree.get(1)) == -1) {
    history.add(iterationresultfree.get(1));
    iterationresultfree.remove(1);
    network.ResetFlags();
    negativeWeight.clear();
    shortestPath = iterationresultfree.get(0);
    shortestPathWithout = current.get(0);
    FlagLinks();
    iterationresultfree.add(Dijkstra(from.get(1), to.get(1),
            current.get(1), false, false));
    RemoveCommonLinks(iterationresultfree.get(0),
            iterationresultfree.get(1));
}
if (history.indexOf(iterationresultfree.get(1)) != -1) {
    iterationresultfree.get(1).clear();
}

// evaluate result
if (!protectedpath || iterationresultfree.get(1).size() > 0) {
    System.out.println(iterationresultfree);
    System.out.println("solution found");
    int cost = TotalCost(iterationresultfree, true, noifcweight);
    if (shortestDijkstra.get(0).size() == 0
            || cost < shortestDijkstraCost) {
        System.out
                .println("shortest (cost:" + cost + ") -> saving");
        shortestDijkstra = iterationresultfree;
        shortestDijkstraCost = cost;
    } else {
        System.out.println("not shortest");
        cost = TotalCost(iterationresultfree, false, noifcweight);
        if (shortestDijkstraCost != 0
                && cost >= shortestDijkstraCost) {
            // no possiblity to lower value below shortestDijkstraCost
            System.out
                    .println("cost can't can lower then current shortest -> add to done");
            exploredsets.add(current);
```

```
        todo.remove(current);
        RemoveExploredsets(todo);
        continue;
    }
}

Set<String> rg1 = RiskgroupsInPath(iterationresultfree.get(0));
Set<String> rg2 = RiskgroupsInPath(iterationresultfree.get(1));
rg1.removeAll(necessaryriskgroups);
rg2.removeAll(necessaryriskgroups);
if (RiskgroupsOverlap(rg1, rg2)) {
    boolean expand = true;
    // check if some riskgroups are nessacary
    if (current.get(0).size() == 0
            && current.get(1).size() == 0 && !checkednecessary) {
        expand = false;
        checkednecessary = true;
        for (String s : rg1) {
            if (rg2.contains(s)) {
                Set<String> temp = new HashSet<String>();
                temp.add(s);
                if (Dijkstra(from.get(0), to.get(0), temp,
                        false, true).size() == 0
                        && Dijkstra(from.get(1), to.get(1),
                                temp, false, true).size() == 0) {
                    List<Set<String>> lss = new ArrayList<Set<String>>();
                    lss.add(temp);
                    lss.add(new HashSet<String>());
                    exploredsets.add(lss);
                    lss = new ArrayList<Set<String>>();
                    lss.add(new HashSet<String>());
                    lss.add(temp);
                    exploredsets.add(lss);
                    necessaryriskgroups.add(s);
                    System.out.println("necessary: "
                            + necessaryriskgroups);
                }
            }
        }
        rg1.removeAll(necessaryriskgroups);
        rg2.removeAll(necessaryriskgroups);
        if (RiskgroupsOverlap(rg1, rg2)) {
            expand = true;
        }
    }
    // if overlapping -> expand
    if (expand) {
        System.out.println("overlapping");
        System.out
                .println("trying to find backup path without riskgroups of primary path...");
        Set<String> rginprimary = RiskgroupsInPath(iterationresultnotfree
                .get(0));
        iterationresultnotfree.add(Dijkstra(from.get(1), to
                .get(1), rginprimary, true, false));
        RemoveCommonLinks(iterationresultnotfree.get(0),
                iterationresultnotfree.get(1));

        // check if two paths are free of riskgrous in current.get(0) and current.get(1)
        history.clear();
        while (iterationresultnotfree.get(1).size() > 0
                && RiskgroupsOverlap(
                        RiskgroupsInPath(iterationresultnotfree
```

```
                        .get(1)), current.get(1))
        && history.indexOf(iterationresultnotfree
                .get(1)) == -1) {
    history.add(iterationresultnotfree.get(1));
    iterationresultnotfree.remove(1);
    network.ResetFlags();
    negativeWeight.clear();
    shortestPath = iterationresultnotfree.get(0);
    shortestPathWithout = current.get(0);
    FlagLinks();
    iterationresultnotfree.add(Dijkstra(from.get(1), to
            .get(1), current.get(1), false, false));
    RemoveCommonLinks(iterationresultnotfree.get(0),
            iterationresultnotfree.get(1));
}
if (history.indexOf(iterationresultnotfree.get(1)) != -1) {
    iterationresultnotfree.get(1).clear();
}

if (iterationresultnotfree.get(1).size() > 0) {
    System.out.println(iterationresultnotfree);
    System.out.println("solution found");
    cost = TotalCost(iterationresultnotfree, true,
            noifcweight);
    if (shortestDijkstra.get(0).size() == 0
            || cost < shortestDijkstraCost) {
        System.out.println("shortest (cost:" + cost
                + ") -> saving");
        shortestDijkstra = iterationresultnotfree;
        shortestDijkstraCost = cost;
    } else {
        System.out.println("not shortest");
    }
}
System.out.print("expand ");
// if overlapping solution then expand todo
for (String s : rg1) {
    if (rg2.contains(s)) {
        System.out.print(s + " ");
        // add overlapping element to first set
        List<Set<String>> copy = new ArrayList<Set<String>>();
        Set<String> copyss = new HashSet<String>();
        for (String st : current.get(0)) {
            copyss.add(st);
        }
        copyss.add(s);
        copy.add(copyss);
        copyss = new HashSet<String>();
        for (String st : current.get(1)) {
            copyss.add(st);
        }
        copy.add(copyss);
        todo.add(copy);
        // add overlapping element to second set
        copy = new ArrayList<Set<String>>();
        copyss = new HashSet<String>();
        for (String st : current.get(0)) {
            copyss.add(st);
        }
        copy.add(copyss);
        copyss = new HashSet<String>();
        for (String st : current.get(1)) {
```

```
                            copyss.add(st);
                        }
                        copyss.add(s);
                        copy.add(copyss);
                        todo.add(copy);
                        break; // add only 1 element of overlapping set
                    }
                }
                System.out.println("");
            }
        } else {
            exploredsets.add(current);
        }
    } else {
        //System.out.println("no solution");
        // no solution
        exploredsets.add(current);
    }
    todo.remove(current); // remove current from todo's
    // check of todo's are allready explored
    RemoveExploredsets(todo);
}

// sort list by to
if (shortestDijkstra.get(1).size() > 0
        && to.get(0) != shortestDijkstra.get(0).get(
                shortestDijkstra.get(0).size() - 1)) {
    shortestDijkstra.add(shortestDijkstra.get(0));
    shortestDijkstra.remove(0);
}

System.out.println("# iterations: " + iterations);
System.out.println("riskgroups primary path: "
        + RiskgroupsInPath(shortestDijkstra.get(0))
        + "\nriskgroups in backup path: "
        + RiskgroupsInPath(shortestDijkstra.get(1)));
/*
 * System.out.println("\nPRIMARY PATH:");
 * System.out.println(NetworkStateDB.getPathWithCpls(shortestDijkstra
 * .get(0))); System.out.println("\nBACKUP PATH:");
 * System.out.println(NetworkStateDB.getPathWithCpls(shortestDijkstra
 * .get(1)));
 */
return shortestDijkstra;
}

private void ExploreDevice(Interface from, Interface to, int path,
        Set<String> without, boolean costriskgroup, boolean firstpath,
        List<Interface> freeifcs) {
    Interface lastifc = explore.get(path).get(explore.get(path).size() - 1);
    int index = 0;

    // if cost higher than allready found path -> delete
    if (shortestPathCost > 0 && explorecost.get(path) >= shortestPathCost) {
        RemovePathExplore(path);
        return;
    }

    // check for cycles
    if (DeviceInPath(lastifc.getDevice(), explore.get(path).subList(0,
            explore.get(path).size() - 1))) {
        RemovePathExplore(path); // remove if cycle
```

```
    return;
}
// check if we've already been on this device
index = done.indexOf(lastifc.getDevice());
if (index != -1) {
    int cost = -1;
    if (exploreswitch.get(path)) {
        cost = nodecost.get(index).get(1);
    } else {
        cost = nodecost.get(index).get(0);
    }
    if (cost != -1 && cost <= explorecost.get(path)) {
        RemovePathExplore(path); // there is a shorter path
        return;
    } else {
        if (exploreswitch.get(path)) {
            nodecost.get(index).set(1, explorecost.get(path));
        } else {
            nodecost.get(index).set(0, explorecost.get(path));
        }
    }
} else {
    done.add(lastifc.getDevice());
    nodecost.add(new ArrayList<Integer>());
    if (exploreswitch.get(path)) {
        nodecost.get(nodecost.size() - 1).add(-1);
        nodecost.get(nodecost.size() - 1).add(explorecost.get(path));
    } else {
        nodecost.get(nodecost.size() - 1).add(explorecost.get(path));
        nodecost.get(nodecost.size() - 1).add(-1);
    }
}

// check if target is on device
ArrayList<Interface> ifcs = explore.get(path).get(
        explore.get(path).size() - 1).getDevice().getInterfaces();
if (ifcs.contains(to)) { // target found!
    AddInterface(path, to, costriskgroup, freeifcs);
    if (shortestPath.isEmpty()
            || explorecost.get(path) < shortestPathCost) { // if shortest -> save path
        shortestPath = explore.get(path);
        shortestPathCost = explorecost.get(path);
        if (firstpath) {
            shortestPathWithout = without;
        }
    }
    RemovePathExplore(path);
} else {
    ifcs.clear();
    // create new path for every interface-out
    ifcs = lastifc.getDevice().getConnectedInterfaces();
    for (int i = ifcs.size() - 1; i >= 0; i--) {
        if ((capacity > 0 && (ifcs.get(i).getFreeCapacity() < capacity || ifcs
                .get(i).getConnectedTo().getFreeCapacity() < capacity))
                || lastifc.getConnectedTo() != null
                && lastifc.getConnectedTo().getDevice() == ifcs.get(i)
                        .getConnectedTo().getDevice()
                || (!exploreswitch.get(path) && ifcs.get(i)
                        .RiskgroupOverlap(without))
                || (exploreswitch.get(path) && ifcs.get(i)
                        .RiskgroupOverlap(shortestPathWithout))) {
            ifcs.remove(i);
```

```
            }
        }
        if (ifcs.size() == 0) {
            RemovePathExplore(path); // dead end
        } else {
            int newpath;
            for (int i = 0; i < ifcs.size(); i++) {
                if (i == 0) {
                    newpath = path;
                    AddInterface(newpath, ifcs.get(i), costriskgroup,
                            freeifcs); // add to existing path
                } else {
                    // copy existing path and add new interface
                    newpath = explore.size();
                    for (int j = 0; j < explore.get(path).size() - 2; j++) {
                        AddInterface(newpath, explore.get(path).get(j),
                                costriskgroup, freeifcs);
                    }
                    AddInterface(newpath, ifcs.get(i), costriskgroup,
                            freeifcs);
                }
                AddInterface(newpath, ifcs.get(i).getConnectedTo(),
                        costriskgroup, freeifcs); // make jump
            }
        }
    }
}
```

Bibliography

[1] J.Q. Hu, "Diverse routing in optical mesh networks", *IEEE Transactions on Communications*, vol. 51, no. 3, pp. 489-494, 2003.

[2] S. Ramamurthy and B. Mukherjee, "Survivable WDM mesh networks. Part I-protection", *Proceedings of Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. INFOCOM'99*, vol. 2, New York, NY, USA, 1999, pp. 744-751

[3] S. Ramamurthy, L. Sahasrabuddhe, and B. Mukherjee, "Survivable WDM mesh network", *Journal of Lightwave Technology*, vol. 21, no. 4, 2003, pp. 870-883.

[4] W. D. Grover, "Mesh-based Survivable Networks: Options and Strategies for Optical, MPLS, SONET/SDH, and ATM networking", *Prentice Hall*, PTR, 2004.

[5] S. Ramamurthy and B. Mukherjee, "Survivable WDM mesh network.II. restoration", *IEEE International Conference on Communications*, vol. 3, Vancouver, BC Canada, 1999, pp. 2023 - 2030.

[6] C. Mauz, "Unified ILP formulation of protection in mesh networks", *Proceedings of the 7th International Conference on Telecommunication COMTEL*, vol. 2, 2003, pp. 737-741.

[7] H. Zang, C. Ou, and B. Mukherjee, "Path-protection routing and wavelength assignment in WDM mesh networks under shared-riskgroup constraints", *Asia-Pacific Optical and Wireless Communications (APOC 2001) Conference*, Beijing, China, 2001, pp. 49-60.

[8] —-, "Path-protection routing and wavelength assignment (RWA) in WDM mesh networks under duct-layer constraints", *IEEE/ACM Transactions on Networking*, vol. 11, no. 2, 2003, pp. 248-258.

[9] L.Guo, H. Yu and L. Li, "Joint routing-selection algorithm for a shared path with differentiated reliability in survivable wavelength-division-multiplexing mesh networks", *OPTICS EXPRESS*, vol. 12, no. 11, 2004.

[10] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, pp. 221, SP4, 1979.

[11] A. Todimala and B. Ramamurthy, "IMSH: An Iterative Heuristic for SRLG Diverse Routing in WDM Mesh Networks", *Communications and Networks, 2004, ICCCN 2004. Proceedings. 13th International Conference on*, pp. 199-204, 2004.

[12] J. Y. Yen, "Finding the k shortest Imp less paths in a network", *Management Science*, vol. 17, pp. 712-716, 1971.

[13] D. Xu et al, "Trap avoidance and Protection Schemes in Networks with Shared Risk Link Groups", *IEEE J. Lightwave Tech.*, vol. 21, no. 11, Nov. 2003, pp. 2683-93.

[14] A. Todimala, B. Ramamurthy, "A scalable approach for Survivable virtual topology routing under shared risk link groups in WDM networks", *Proceedings. First International Conference on Broadband Networks, 2004. BroadNets 2004*, pp. 130-139, 2004.

[15] P. Laborczi, et. al., "Solving asymmetrically weighted optimal or near-optimal disjoint path pair for the survivable optical networks", *Third International Workshop On Design Of Reliable Communication Networks (DRCN 01)*, Oct. 2001.

[16] D. Xu, Y. Xiong, and C.Qiao, "Novel algorithms for shared segment protection", *IEEE Journal on Selected Areas in Communications, accepted for publication*, 2003.

[17] D. Xu, Y. Xiong, and C. Qiao, "Protection with multi-segments in networks with shared risk link groups (SRLG)", *40th Annual Allerton Conference on Communication, Control, and Computing*, 2002.

[18] C. V. Saradhi and C. S. R. Murthy, "Dynamic establishment of segmented protection paths in single and multi-fiber WDM mesh networks", *OPTICOMM02*, pp. 211222, 2002.

[19] M. Kodialam and T. V. Lakshman, "Dynamic routing of locally restorable bandwidth guaranteed tunnels using aggregated link usage information", *INFOCOM01*, pp. 376385, 2001.

[20] P.-H. Ho and H. Mouftah, "A framework for service-guaranteed shared protection in WDM mesh networks", *IEEE Communications Magazine*, vol. 40, no. 2, pp. 97103, 2002.

[21] D. Xu, Y. Xiong, C. Qiao, "A new PROMISE algorithm in networks with shared risk link groups", *IEEE Global Telecommunications Conference, 2003. GLOBECOM '03*, vol. 5, 2003, pp. 2536-2540.

[22] G. Li, B. Doverspike, C Kalmanek, "Fiber Span failure Protection in Mesh Optical Networks", *Opt. Net. Mag.*, vol 3, no 3, May/June 2002.

[23] R. Bhandari, "Optimal Diverse Routing in Telecommunications Fiber Networks", *INFOCOM '94*, 1994.

[24] E. Bouillet el al., "Stochastic Approaches to Compute Shared Mesh Restored Lightpaths in Optical Network Architectures", *lNFOCOM '02*, pp. 801-07, 2002.

[25] Xu Shao, Luying Zho, Xiaofei, Cheng, Weiguo Zheng, Yixin Wang, "Best Effort Shared Risk Link Group (SRLG) Failure Protection in WDM Networks", *Communications, 2008. ICC '08. IEEE International Conference on*, pp. 5150-5154.

[26] D. Xu, Y. Xiong, C. Qiao, G. Li, "Failure protection in layered networks with shared risk link groups", *Network, IEEE*, vol. 18, issue 3, pp. 36-41, May-June 2004.

[27] Xubin Luo, Bin Wang, "Diverse Routing in WDM Optical Networks with Shared Risk Link Group (SRLG) Failures", *Proceedings of the 5th IEEE International Workshop on Design of Reliable Communication Networks (DRCN)*, October 16-19, 2005, Island of Ischia (Naples), Italy.

[28] Oki, E., Matsuura, N., Shiomoto, K., Yamanaka, N., "A disjoint path selection scheme with SRLG in GMPLS networks", *High Performance Switching and Routing, 2002. Merging Optical and IP Technologies. Workshop on*, pp. 88-92, 2002

[29] P. Datta and A. K. Somani, "Diverse Routing for Shared Risk Resource Groups (SRRG) Failures in WDM Optical Networks", *BROADNETS'2004, San Jose CA USA*, pp. 120-129, October 2004.

[30] J. Doucette and W. D. Grover, "Capacity Design Studies of Span-Restorable Mesh Transport Networks with Shared-risk Link Group Effects", *Proceedings of OptiComm, Boston MA USA*, pp. 25-38, August 2002.

[31] J.W. Suurballe, "Disjoint paths in a network", *Networks*, vol. 4, pp. 125-145, 1974.

[32] J.W. Suurballe and R.E. Tarjan, "A quick method for finding shortest pairs of disjoint paths", *Networks*, vol. 14, pp. 325-336, 1984.

[33] R. Bhandari, *Survivable Networks: Algorithms for Diverse Routing*. Norwell, MA, USA: Kluwer Academic Publishers, 1998.