Universidade de Aveiro
2014

Departamento de Eletrónica,
Telecomunicações e Informática

Miguel Filipe Ramalho Rodrigues

Otimização de Redes IP com Mecanismos de Reencaminhamento Rápido

Optimization of IP Networks with Fast Reroute Mechanisms

Universidade de Aveiro
2014

Departamento de Eletrónica,
Telecomunicações e Informática

Miguel Filipe Ramalho Rodrigues

Otimização de Redes IP com Mecanismos de Reencaminhamento Rápido

Optimization of IP Networks with Fast Reroute Mechanisms

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Amaro Fernandes de Sousa, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri

presidente

Prof. Doutor José Rodrigues Ferreira da Rocha
Professor Catedrático do Departamento de Eletrónica, Telecomunicações e Informática, Universidade de Aveiro

vogais

Prof. Doutor Rui Jorge Morais Tomaz Valadas
Professor Catedrático do Instituto Superior Técnico, Universidade de Lisboa (arguente externo)

Prof. Doutor Amaro Fernandes de Sousa
Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática, Universidade de Aveiro (orientador)

palavras-chave redes IP, mecanismos de reencaminhamento rápido, LFA, optimização

resumo

Esta dissertação estuda estratégias para a atribuição de custos IGP às interfaces dos *routers* de uma rede IP de forma a potenciar o uso de *Loop-Free Alternates* (LFA), um mecanismo de reencaminhamento rápido que tem sido recentemente implementado em *routers* comerciais. Este mecanismo permite que os *routers* reencaminhem tráfego por rotas alternativas assim que uma falha de rede é detetada, evitando uma maior perda de pacotes durante o período de recuperação da rede. O problema é que este mecanismo geralmente não oferece cobertura para todas as falhas possíveis. Além disso, as rotas de restauro podem causar congestão na rede e até mesmo ciclos de encaminhamento.

Foi então desenvolvida uma aplicação que, dada uma topologia de rede e respetiva matriz de tráfego, permite determinar custos que melhorem o desempenho da rede quando emprega este mecanismo. As estratégias implementadas procuram minimizar situações em que o uso das rotas de restauro provoca ciclos de encaminhamento ou sobrecarga das ligações, preservando desta forma a qualidade da maior parte do serviço.

Os resultados obtidos mostram que é possível minimizar os efeitos de uma falha através de uma escolha inteligente dos custos. Também é possível concluir que, na grande maioria dos casos, aumentar de forma cega a cobertura da rede através de *Loop-Free Alternates* não é a melhor estratégia. Dependendo dos recursos disponíveis, torna-se muitas vezes necessário sacrificar essa cobertura para obter melhores níveis globais de desempenho.

keywords IP networks, fast rerouting mechanisms, LFA, optimization

abstract

This dissertation studies strategies for assigning costs to the interfaces of routers inside an IP network to potentiate the use of Loop-Free Alternates (LFA). LFA is a fast reroute mechanism that has been recently deployed in commercial routers. This mechanism allows routers to forward traffic through alternative paths right after the detection of a network failure, avoiding a higher loss of packets during the network's recovery process. The problem is that this mechanism does not usually provide coverage to all possible failures. Moreover, repair paths may lead to congestion and even forwarding loops.

An application was developed that, given a network topology and its supporting traffic matrix, allows to find IGP costs that improve the network performance when employing this mechanism. The implemented strategies try to minimize situations where the use of repair paths leads to micro-loops or link overloads, thus preserving the quality of the service.

The computational results show that it is possible to minimize the effects of a failure through an intelligent choice of costs. It is also possible to conclude that, for the majority of cases, increasing the LFA coverage of a network is not the best strategy. Depending on the available resources, it becomes often necessary to sacrifice this coverage to obtain better performance levels.

Index

Figures

Tables

1 Introduction

1.1 Motivation

The Internet has experienced an incredible growth, especially over the last decade, not only because the overall increase of broadband speed allowed new applications to emerge but also because new technologies and devices such as smartphones and tablets made it easier to access the Internet. The number of users and provided services continues to grow every year. Social web and mobile technologies are pointed as two major factors for this growth, being also responsible for changing the way people use the Internet (Internet World Stats).

Today's Internet usage is much harder to predict and control, and user expectations are higher, so it has been a challenge for Internet Service Providers (ISP) to deliver a quality service. They've been particularly concerned with the increasing demand for low-latency applications (Neagle, 2012). Online gaming, live-streaming media, IP telephony and video communications are some examples of applications whose generated traffic has increased over the last years. This kind of traffic is very sensitive to delay and packet loss. On the other hand, streaming-video may consume a lot of bandwidth, depending on the requested video quality. Capacity is a limited resource, so operators already implement some traffic engineering techniques in their networks to reduce latency and avoid congestion. But when a failure occurs, common Internet Protocol (IP) networks take some time to recover from it, which may cause severe degradation to these services as illustrated in Figure 1.1. Thus, there has been an increasing need for fast failure recovery mechanisms.

Figure 1.1 - Artifacts caused by packet loss[1]

One popular solution is to use Loop-Free Alternates (LFA). This technology is an IP Fast ReRoute (IPFRR) mechanism that uses pre-calculated backup next-hops so that traffic may be rerouted and avoid a local failure right after its detection (Atlas, et al., 2008). In this way, there is no need to wait for the affected routers to rebuild their forwarding tables before resuming service. However, being one of the simplest mechanisms, it has some drawbacks. In general, LFA does not guarantee protection against all possible failure scenarios and there is a risk of creating traffic loops whenever

[1] http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-next-generation-network-ngn-video-optimized-transport/white_paper_c11-637031.html

an inappropriate backup is used. Repair traffic may also cause some links to become congested, degrading traffic which was not directly affected by the failure. Despite these limitations, LFA is the only IPFRR specification currently supported by high-end IP routers. Although it ensures at least the same availability and performance of a network without IPFRR, providers may still be unsure about the actual benefits of deploying LFA.

Many proposals have been presented to the scientific community focused on improving the LFA coverage of a network. The problem is that almost all of them make restrictive assumptions and ignore traffic engineering. Although it may be logical that a well-protected network is more likely to provide a better service, when considering the current traffic demands, the work on this dissertation will show that the overall performance of a network can be improved by actually limiting the LFA coverage.

1.2 Context

1.2.1 IP Network Failures

When rating an ISP, the two main factors that are usually considered by clients are the access bandwidth and its price. Nevertheless, another important aspect should be the service availability. All promises made by ISPs are meaningless if their services don't work. Generally, providers are supposed to guarantee an uptime around 99.9% which means that, in a year, they only expect their service to be down no more than a total of 8 hours.

This downtime is usually associated with the failure of one or more elements in the network. It is estimated that 20% of the failures happen during scheduled maintenance, while the other 80% are unplanned (Markopoulou, et al., 2008). A router can reboot for some reason or even shutdown and a link can be accidently disconnected or even cut. Approximately 70% of these unexpected failures affect a single link at a time, while 30% are shared by multiple links. A simultaneous failure of links is usually the result of a router failure. A network is never fail-free, but a good redundancy plan together with fast reroute mechanisms can help ISPs proactively deal with these unexpected failures in a way that their customers will not even notice them.

1.2.2 Convergence Time

Large IP-based networks commonly use the Open Shortest Path First (OSPF) routing protocol. OSPF is an Interior Gateway Protocol (IGP) that distributes routing information between routers within a single routing domain. Being a link-state protocol, each router builds a topology map of the network based on the information received by all other routers: the link-state database. Then, it runs Dijkstra's Shortest Path First algorithm (SPF) to determine the shortest path from itself to all existing IP networks inside the routing domain and builds its own routing table from which forwarding decisions are made. This is a dynamic protocol in the sense that it responds to topology changes. When a router detects a failure, it sends to all other routers update packets to announce the changed topology. These packets are flooded throughout the network and routers redo the SPF

calculation to update their routing tables (Moy, 1998). The time required for a network to completely recover from a failure is then determined by:

1. The time taken to detect the failure – this may be of the order of a few milliseconds when detected at the physical layer or up to tens of seconds when detected using a Hello protocol.
2. The reaction time of the local router to the failure – after some hold-down delay, the router generates and floods new routing updates. It also re-computes its forward information base (FIB).
3. The time taken to transmit the information about the failure to other routers in the network – usually between 10ms and 100ms per hop.
4. The time taken to re-calculate the forwarding tables – the total execution time of the SPF algorithm depends on the router's CPU and the network size, but is typically a few milliseconds.
5. The time taken to load the revised tables into the forwarding hardware – this can take several hundreds of milliseconds.

The service disruption will last until the routers adjacent to the failure have completed steps 1 and 2, and until all routers whose paths were affected have completed the remaining steps. The packet loss during steps 1 and 2 is unavoidable: a router will keep trying to send packets across the failure until it is detected. As for any packet loss that happens during the remaining steps, it may be caused by micro-loops that are formed due to inconsistencies between forwarding tables. Routers build their routing tables locally and the time taken to do so may vary a lot from one router to another. The loading time into the forwarding hardware (step 5) is actually the main responsible for this variation since it is very dependent on the implementation and number of affected prefixes (Shand, et al., 2010).

We are talking about a convergence time in the order of 100's of milliseconds or even seconds, during which packets are lost and loops may occur leading to artificial and much more undesirable congestion. Today's most sensitive traffic does not tolerate losses that last more than 50ms. Many efforts have been made in the recent years aiming to improve the OSPF's convergence speed (Goyal, et al., 2011), but the distributed nature of a network will always impose an intrinsic limit on the minimum convergence time that can be achieved. A better approach to solve this problem is to reroute traffic while the network converges in the background.

1.2.3 Fast Reroute Solutions

One common solution adopted by some operators is to use a Multi-Protocol Label Switching (MPLS) infrastructure and to enable MPLS Traffic Engineering (MPLS-TE), which has fast reroute (FRR) capabilities. FRR allows traffic affected by a failure to be rerouted through pre-calculated alternative paths, reducing the disruption time to tens of milliseconds. The downside of MPLS-TE is that it is complex and adds significant overhead to the network. Many operators were using MPLS-TE mainly for the FRR functionality (Doyle, 2007). In the recent past, the Internet Engineering Task

Force (IETF) proposed a FRR solution that does not require MPLS-TE, being able to work over a pure IP infrastructure employing conventional IP routing and forwarding. This solution is called IP Fast Reroute and allows operators to improve availability and keep their networks simple.

Several IPFRR mechanisms have been developed by the IETF. They present different approaches and techniques to increase the network protection and to prevent micro-loops in the event of a single failure, whether link, node, or shared risk link group (SRLG). There are mechanisms for Fast Failure Detection, mechanisms for Repair Paths and mechanisms for Micro-Loop Prevention.

Repair Paths, in particular, make it possible to recover from a failure immediately after its detection, eliminating the most-time consuming part of the IGP recovery process: flooding new routing information throughout the network. This is achieved by using pre-calculated backup routes that avoid the failed element. Network connectivity is then maintained while routers reinstall their forwarding tables in the background. Repair Paths are divided in three categories:

1. Equal cost multi-paths (ECMP). These paths exist when there are multiple primary next-hops for a single destination. Any of these next-hops not traversing the failure can trivially be used as repair paths.
2. Loop-free alternate paths. These paths exist when a direct neighbor of the router adjacent to the failure has a path to the destination that does not traverse the failure.
3. Multi-hop repair paths. These paths can be used when there are no loop-free alternate paths available. In this case, the goal is to find a router, which is more than one hop away from the one adjacent to the failure, from which traffic can be forwarded without traversing that failure.

ECMP and loop-free alternate paths offer the simplest repair paths and would normally be used when available. It is anticipated that around 80% of failures can be repaired using these two methods alone. Multi-hop repair paths, on the other hand, are much more complex, both in the computations required to determine their existence, and in the mechanisms required to invoke them (Shand, et al., 2010).

1.2.4 Loop-Free Alternates

A loop-free alternate is a pre-computed backup next-hop intended to quickly replace a primary next-hop when it fails, thus providing an alternative path to the destination. LFAs offer a certain amount of protection: they can protect against a single link failure, a single node failure, a failure of one or more links within a shared risk link group, or a combination of these. They are named loop-free because they guarantee that, when an expected failure occurs, forwarding traffic through them will not result in a routing loop.

Consider the following terminology, when describing the LFA mechanism:

- S – traffic source, the computing router;
- D – traffic destination router;
- E – a primary next-hop of S to D;

- N – a neighbor of S that is not a primary next-hop to D;
- $dist(X, Y)$ – the shortest path distance from router X to router Y.

For convenience, both E and N refer to routers that are neighbors of S, but keep in mind that a neighbor or next-hop of a router S is composed by the adjacent router and the link used by S to reach it. When we say that a primary next-hop has failed, it means that either the outgoing link has failed or the neighbor's router itself has failed.

To better explain the functionality of loop-free alternates, let us consider the basic topology example shown in Figure 1.2.

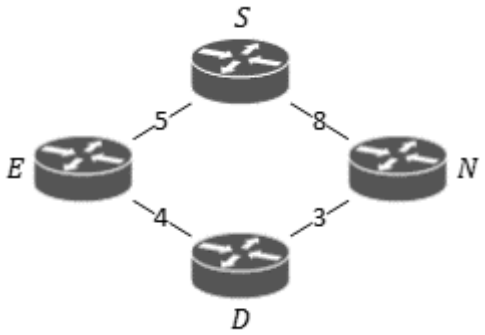

Figure 1.2 - A basic topology example

The numbers in the links correspond to the costs assigned to the interfaces they are connected with. Consider that there is a traffic flow from router S to router D. With this configuration, when S calculates its shortest path to D, it will determine the link connected to E as its primary next-hop. Without IPFRR, this is the only computed next-hop. With IPFRR, S also determines a backup next-hop to use in case the primary next-hop fails, which, in this case, is the link connected to N.

Let us now assume that the link connecting S to E fails. The IGP recovery process begins by S (and E) detecting the failure and flooding routing updates. Without IPFRR, the service disruption will last until S sets link to N as the new primary next-hop to D. With IPFRR, S removes the failed primary next-hop E from its table and installs the corresponding pre-computed backup next-hop N right after acknowledging the failure. In this way, the service is resumed almost immediately using the alternative path (no need to wait for the new primary next-hop to be set). After the network finally recovers from the failure, S can switch from the backup to the new primary next-hop (which, for this example, will be the same neighbor).

Let us now consider that the cost of link between N and D is increased from 3 to 17 (or any other value above). Notice that the cost of the path from N to D via S is also 17 (8+5+4). With this new configuration, N is no longer a loop-free alternate for router S, because half the traffic that reaches N destined to D will be forwarded back to S. The existence of a suitable loop-free alternate depends on the network topology, on the assigned link costs and on the nature of the failure.

For a neighbor N to be a loop-free alternate for a given destination D, it must meet the following condition:

$$dist(N,D) < dist(N,S) + dist(S,D) \quad (1)$$

This inequality is named Loop-Free Criterion.

Next, some common failure scenarios that may limit the use of LFAs are presented. For each scenario, I will describe all types of alternates, and show how they are computed and selected. For the sake of simplicity, SRLGs will not be addressed, since they were not considered in this dissertation.

A LFA can be link-protecting if it protects against a link failure, and/or node-protecting if it protects against a node failure. A router is able to detect when a certain neighbor becomes unreachable, but it does not directly know if it was due to a failed link or node. It always takes the pessimist assumption that the node has failed but may still use a link-protecting alternate to repair the traffic if a node-protecting alternate does not exist. If a LFA that only provides link protection is used to cover a node failure, there is a possibility for the repair traffic to experience micro-looping as illustrated in the example of Figure 1.3.

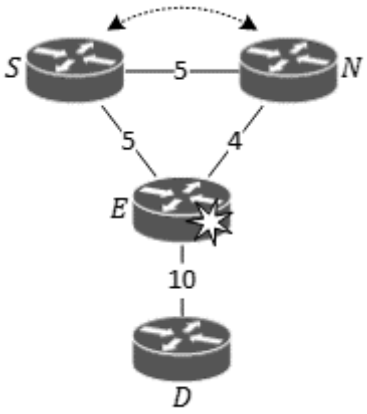

Figure 1.3 - Topology with link-protecting alternates causing a micro-loop

If link between S and E fails, S will use its backup N, and traffic will reach the destination D without any problems. But if router E fails, both S and N will detect a failure and switch to their alternates. S will redirect traffic to N while N will redirect it back to S causing a micro-loop.

Micro-loops can be an issue whenever repair traffic uses two or more consecutive backups that do not protect against the failure that has occurred. It is still a temporary problem, since they will disappear once routers finish installing the new primary next-hops. But during this period, connections containing micro-loops become congested, causing delay and packet loss to the other traffic flows that were not affected by the failure. Note that loop traffic will never reach its destination and, therefore, when router E fails in the previous example, it would be better if both S and N did not use their link-protecting alternates.

Micro-loops can be avoided by using downstream paths. A neighbor N is a downstream path if it meets the Downstream Path Criterion:

$$dist(N, D) < dist(S, D) \quad (2)$$

Note that a downstream path is more restrictive than a loop-free alternate (*i.e.*, a downstream path is a loop-free alternate but a loop-free alternate might not be a downstream path). A downstream path requires the router's neighbor to be closer to the destination than itself. In the previous example, N is a downstream alternate of S, but S is just a loop-free alternate of N. If only downstream paths were used to repair traffic, S would be able to use N after the failure of router E, but N could not use S and traffic would be discarded. The micro-loop would not occur at the cost of limiting the use of alternates.

A LFA only protects against a node failure if the repair path does not traverse the failed node. This happens when the following condition is verified, with N being the neighbor providing a loop-free alternate for primary next-hop E:

$$dist(N, D) < dist(N, E) + dist(E, D) \quad (3)$$

This is the Criteria for a Node-Protecting Loop-Free Alternate.

In order for a LFA to protect traffic from a link failure, it is necessary that the repair path does not traverse the failed link. This condition is always true when using point-to-point links: different neighbors have different links, so all LFA are link-protecting. The only case in which a LFA may not be link-protecting is when the router uses a broadcast link (given by a switching network) to reach the primary next-hop as illustrated in the case presented in Figure 1.4, because it is possible for an alternative path to traverse that broadcast link.

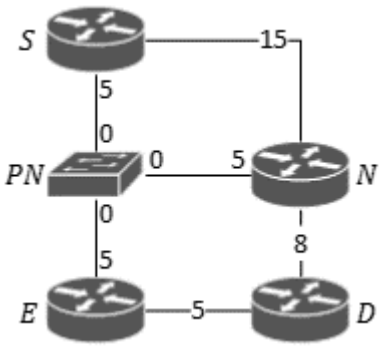

Figure 1.4 - Topology with a broadcast link

A broadcast link is modelled by the routing protocols as a pseudo-node (PN) whose interfaces have cost 0. In the topology above, N fulfills condition (1) and (3), thus offering a node-protecting LFA. To be link-protecting, the first requirement is that N does not use PN on its route towards D. This is because a failure of an interface attached to a broadcast link can mean loss of connectivity of the whole segment. N must then be loop-free with respect to the pseudo-node:

$$dist(N, D) < dist(N, PN) + dist(PN, D) \quad (4)$$

This condition is called Loop-Free Link-Protecting Criterion for Broadcast Links. In the previous example, N verifies this condition, since it uses the next-hop that is directly connected to D. The second requirement is that S does not use PN to reach N. There are two neighbors of S regarding N that we must consider: one using the broadcast link and another using the point-to-point link. The latter meets both requirements, thus becoming a link-and-node-protecting LFA. The first neighbor is only a node-protecting LFA, since it fails to meet the second requirement.

Until now, we have been considering a single primary next-hop to the destination. With Equal-Cost Multi-Path, it is possible for a router to have multiple primary next-hops to reach a certain destination. Traffic is then equally distributed and forwarded through all those next-hops. Each one of these primary next-hops has its own loop-free alternate computed separately, and this alternate may well be one of the other primary next-hops. When an alternate is also a primary next-hop, it is guaranteed that repairing traffic through it does not cause micro-loops, regardless the failure.

In summary, the three types of LFAs are:

1. Primary Next-hop – when the alternate is also a primary next-hop. The implementations prefer using this type of LFAs because the repair traffic will not be routed through unfamiliar paths. All primary next-hops are downstream paths.
2. Downstream Path – a neighbor that verifies condition (2). Unlike a simple LFA, a downstream path is guaranteed not to cause micro-loops, but the coverage for this kind of alternates may be very poor. All downstream paths are LFAs.
3. LFA – a neighbor that verifies condition (1). Always depending on the topology, the coverage of LFAs can be quite good, but it is possible to occur micro-looping because of unprotected failures.

For each primary next-hop, a router attempts to select at least one loop-free alternate from the other neighbors based on the alternate type and provided protection. Note that this selection process does not affect the previous calculation of primary next-hops via a standard SPF algorithm. According to LFA specification (Atlas, et al., 2008), a computing router S should use the following rules:

1. S should select a node-protecting LFA, if available. If not, it may select a link-protecting LFA.
2. S should select a link-and-node-protecting LFA over a node-protecting LFA.
3. If S has multiple primary next-hops, S should select either one of the other primary next-hops or a node-protecting LFA if available. If no node-protecting LFA is available and no other primary next-hop can provide link-protection, then S should select a link-protecting LFA.
4. Implementations should support a mode where other primary next-hops providing at least link or node protection are preferred over any non-primary alternates. This mode allows the administrator to preserve traffic patterns based on regular ECMP behavior.

The aim of these rules is to maximize the coverage of failure scenarios, *i.e.*, to maximize the percentage of the total number of primary next-hops with assigned backup next-hops for link failures and node failures.

In practice, the link failure coverage in common networks is usually in the order of 70-90%, while the node failure coverage is around 50-70%, which are considered very good results taking into account the simplicity of this particular IPFRR specification (Gjoka, et al., 2007). To find the alternates, routers only need to know their neighbors and calculate some distances. Remember that an OSPF router has a complete view of the network and, therefore, is able to run the SPF algorithm from the perspective of any other router. LFA can then be implemented with minor software upgrades and does not require any additional information from the other routers. This easy deployment, and the fairly reasonable failure coverage provided at the cost of almost insignificant CPU overhead, has convinced router manufacturers to support LFA in some of their latest models (Cisco Systems, 2012). Since then, LFA has gained special interest in the scientific community.

1.3 Previous Research

Most of the research works regarding loop-free alternates are concerned in maximizing the coverage of networks. Exploiting the fact that failure protection is dependent on the network topology and link costs, they make use of algorithms and heuristics to prove it is possible for ISPs to achieve higher availability with relatively low cost.

One way of improving the LFA coverage is by adding links to a topology. The LFA Graph Extension Problem consists in finding the smallest number of new links to add, so that LFA may provide full protection without modifying the original shortest paths (Rétvári, et al., 2011). The problem of this approach is that it becomes expensive and even unpractical for most operators to achieve 100% coverage: several new links may be required, especially to improve node protection. Still, it is possible for most networks to get close to full coverage by adding no more than 2-4 new links.

A less expensive and more practical approach is to adjust link costs (Rétvári, et al., 2011). This option may be adequate when ISPs have no constraints regarding primary paths, but their networks should have a considerable high link density (*i.e.*, a high number of links per router). Sparse networks, having less neighbors and less changeable costs, will not benefit much from this approach. In fact, it was proved that it is impossible to achieve 100% coverage in most common real networks using the LFA mechanism alone. Changing link costs is already an old technique used by operators to perform traffic engineering (Fortz, et al., 2002). An ISP may have its IGP costs carefully chosen to make better use of resources and improve performance. Re-adjusting costs only to obtain fast resiliency might not be very appealing if it means less forwarding efficiency during more than 99.9% of the time.

Changing the topology and optimizing costs both have their pros and cons. Even if combining these two strategies, it is safe to say that achieving 100% LFA protection is a very improbable scenario for real networks, due to the compromises required. Operators employing LFA in their

networks will need to deal with this imposed downtime in some other way. If not, it may be possible for these unrepairable failure scenarios to degrade service even more than if LFA was not enabled: repair traffic can cause traffic loops and/or link overloads. So far, the minimization of the side effects of using LFA when a network is not fully covered is an issue that has not been still addressed by the research community.

1.4 Objectives

This work aims to study the determination of the IGP costs of a given network in order to optimize the use of LFA while minimizing the undesirable effects of micro-loops and link overloads. This optimization considers not only the network's resources (*i.e.*, the link bandwidth capacity) but also the estimated traffic demands, so that solutions are filtered to meet some operational requirements. The goal is to find solutions that protect the traffic as much as possible without degrading the overall performance of the network.

The main difference of this work to previous research works is that failure coverage is considered based on a given traffic demand matrix. The focus is to protect service, instead of protecting against all possible failures. Also, the failure scenarios are analyzed to make sure that repair traffic does not cause micro-loops or overloads. Another important difference is that the optimization does not make restrictive assumptions like the use of symmetrical costs and the nonuse of ECMP and broadcast LANs.

The aim is to develop a heuristic, to obtain the optimized IGP costs for a given topology, and an application to simulate this problem and apply the heuristic for a given topology and a given traffic matrix. The computational results aim to show the effect of considering the traffic matrix in both coverage and performance and will highlight that the protection of a network may need to be sacrificed to really benefit from employing LFA.

1.5 Thesis Structure

The next chapter presents a high level view of our problem. It defines the topology model and the performance parameters used to compare solutions. Then, it presents the developed heuristic and several different possible strategies of changing costs to find better solutions.

The third chapter describes the implementation details regarding the application, developed in Java, using an object oriented language and with the help of some UML diagrams and other illustrations.

The fourth chapter shows the results of several optimization runs performed using the developed application. It presents an evaluation of the implemented heuristic algorithm using different configurations and discusses the optimization of Abilene network performance considering different test cases.

Finally, the fifth chapter summarizes the conducted work and points out the most important conclusions drawn from this work.

2 Problem Definition

This chapter describes the abstractions used to model the problem and the assumptions made. It also defines a solution to the problem and how to compare different solutions. Finally, it discusses the heuristic algorithm used to find good solutions and the considered strategies to try to eliminate both micro-loops and overloads, and to protect service as well.

2.1 Model

A network topology is modeled with a connected, weighted, directed graph. A graph $G = (V, A)$ is defined by a set of vertices, V, and a set of arcs, A. An arc establishes a certain type of relationship between two vertices. Figure 2.1 presents an example of a network topology and the corresponding graph representation:

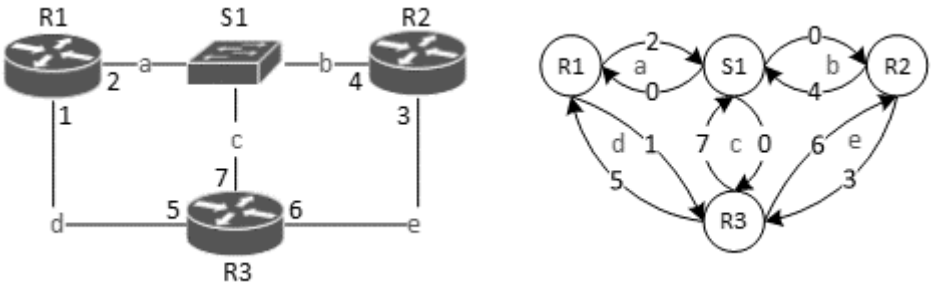

Figure 2.1 - A network topology and its graph representation

The routers and switches are the vertices of the graph, while each link is represented by two arcs, one for each link direction. An arc has a source node, a target node and a weight that matches the cost assigned to the source's output interface connected to the corresponding link (the upstream). An outgoing arc of a switch has a fixed weight of 0 with respect to the IGP routing protocol.

For traffic engineering, we also need to know the capacity of each link and the matrix of traffic demands. A demand is characterized by a source router, a destination router and the bandwidth demand value.

It is assumed that costs are asymmetrical, demands are unidirectional and link capacities are bidirectional. For simplicity, any broadcast domain (*i.e.*, any set of switches directly connected between them) is represented by a single switch, following the pseudo-node model considered by the IP routing protocols.

A solution to our problem is modeled by a list of applicable IGP costs to the outgoing arcs of routers (and only routers, because a switch interface cost is static). Considering the example of

Figure 2.1 and assuming the outgoing arcs of routers *R1*, *R2* and *R3* ordered by the corresponding link letters, the solution shown in the figure is defined as {2, 1, 4, 3, 7, 5, 6}.

2.2 Processing a Solution

As defined before, a solution is described by a set of IGP costs, one cost per router interface. The processing of a solution consists on computing all routing tables along with the backup next-hops, and sending each traffic demand through the right paths for all scenarios of interest. Three types of scenario are considered:

- the default scenario without failures;
- a single link failure scenario;
- a single node failure scenario.

Let us consider the example shown in Figure 2.2 with the service of a 100 Mb/s demand from router S to router D. It considers a network topology with 4 routers and 5 links with the IGP costs assigned to each router interface.

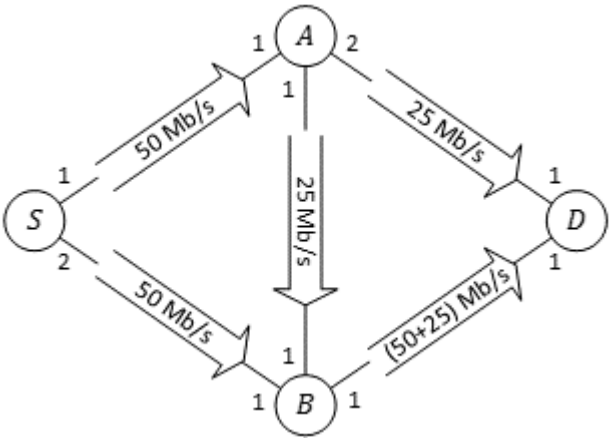

Figure 2.2 - An example of a solution's default scenario

Figure 2.2 describes the default scenario for the presented solution. Router S has two primary next-hops to router D, so traffic is split into two flows of 50 Mb/s (the ECMP rule). The same happens in router A: 25 Mb/s of the 50 Mb/s arriving from S are forwarded through primary next-hop D and the other 25Mb/s are forwarded through primary next-hop B. Router B has only one primary next-hop to the destination, serving a total of 75 Mb/s: the 50 Mb/s received from router S, plus the 25 Mb/s arriving from router A.

Let us now analyze the link failure scenarios. The topology has 5 links, so there are 5 possible scenarios, but there is only one scenario in which service is not protected: if the link between B and D fails, there is no backup next-hop for primary next-hop D, since the other two neighbors of B, S and A, are not loop-free. This means that 75 Mb/s will not be served. All other routers have two primary next-hops in the way to D, so when one next-hop fails, they will send traffic through the other.

As for the node failure scenarios, we only consider the scenario where router A fails and the scenario where router B fails. When router A fails, the 100 Mb/s will reach the destination by going through router B (note that the backup next-hop B is node-protecting since it fulfils conditions (1) and (3)). If router B fails, the demand is still served, despite the backup next-hop A of S not being node-protecting. If link between A and D did not exist, router A would have no backup for next-hop B and traffic would be discarded. If we have considered that routers S and D could fail, service protection for those scenarios would no longer depend on the chosen costs, since it would be impossible to serve the demand.

A node can be classified as transit or non-transit. By definition, switches are always transit and routers with a single link are always non-transit. Other non-transit nodes are routers which are the source and/or the destination of at least one traffic demand. All other routers are assumed to be transit. We assume that non-transit routers are secured and do not fail, otherwise, some bandwidth demands would stop to be served anyway. Therefore, we assume that node failure scenarios only involve a failure of a transit node.

After completely processing a solution, its data includes all routing tables and the information from each possible scenario (*e.g.*, the load on each directed link).

2.3 Evaluating a Solution

In order to find optimum link costs, we must be able to compare solutions given some objective function. Solutions are evaluated using the following parameters:

1. LFA protection level
2. Micro-loop ratio
3. Overload ratio
4. Maximum overload
5. Average overload
6. Maximum load
7. Average load
8. Served bandwidth
9. Service ratio

Next subsections describe each of these parameters.

The LFA protection level is computed from analyzing the routing tables, while the other parameters are based on the considered failure scenarios: the micro-loop ratio of the solution is calculated based on the micro-loop ratio of each failure scenario, and so on.

Each scenario has its parameters calculated separately. The performance parameters of the default scenario are used only to validate a solution (for instance, an operator may wish to impose maximum load limits on their links). Failure scenarios are divided into two groups, one for the link failures and another for node failures so that we may define the weight of each failure type in the calculation of the solution parameters. This is important because, otherwise, the parameters of a

solution could be wrongly influenced by link failures, since a network typically has more links than nodes. Also, operators might consider different probabilities for their links and nodes to fail.

Let us assume p_L as the adopted weight of a link failure and p_N as the adopted weight of a node failure. Considering $\bar{s}_L$ and $\bar{s}_N$ the average value of parameter s regarding link failure scenarios and node failure scenarios, respectively, the parameter s for the solution is the weighted average between those two values:

$$\bar{s} = \frac{\bar{s}_L p_L + \bar{s}_N p_N}{p_L + p_N} \quad (5)$$

2.3.1 LFA Protection Level

The LFA protection level of a solution is a weighted average between the link and node protection levels. Both these levels are calculated by checking each primary next-hop and its assigned alternate.

The LFA link protection level η_{LP} of a network G is generally calculated using with the following expression:

$$\eta_{LP}(G) = \frac{\#(s,d)\ pairs\ with\ link\text{-}protecting\ LFA}{\#\ all\ (s,d)\ pairs} \quad (6)$$

This expression uses the assumption that each source-destination pair has only one primary next-hop (thus only one possible backup). In order to consider ECMP, we must adjust the numerator and define each pair protection as the proportion of primary next-hops with a link-protecting LFA.

The calculation of the node protection level, $\eta_{NP}(G)$, is done in a similar way, with the exception that, when the destination router d is the next-hop of source router s, a link-protecting LFA is counted as also being node-protecting. The reason for this assumption is that the node protection level would be miscalculated because when the primary next-hop's router is the same as the destination, any existing alternate is never node-protecting.

2.3.2 Micro-loop Ratio

The micro-loop ratio of a failure scenario is simply given by a flag, so it is 0% when the scenario is free of micro-loops or 100% when at least one micro-loop has occurred. Micro-loops might happen during node failure scenarios when using link-protecting alternates. Their presence is detected while routing each traffic demand from its source to the destination. Detecting only one micro-loop is sufficient because others may be redundant and it is a situation we must always try to avoid, regardless the affected link(s) and flow(s). When considering a group of failure scenarios, the micro-loop ratio is then the percentage of scenarios which contain at least one micro-loop.

2.3.3 Overload Ratio

A link is overloaded in one direction when the total amount of required bandwidth value to be routed through it exceeds its capacity. Considering a directed link to be one of the two possible directions of a link, the overload ratio of a failure scenario is given by the proportion of directed

links which are overloaded. For instance, considering a topology with 6 links, an overload ratio of 25% means that, for the failure scenario in question, 3 directed links are overloaded ($0.25 \times 6 \times 2 = 3$). We consider that a scenario is overloaded when it is free of micro-loops and its overload ratio is not zero. It should be free of micro-loops because micro-loops are usually more harmful than overloads, and overloads can also be a consequence of micro-loops. When considering a group of failure scenarios, the overload ratio is the percentage of overloaded scenarios.

2.3.4 Maximum Overload

The maximum overload of a scenario is the maximum amount of bandwidth in excess from all directed links which are overloaded. The maximum overload for a group of scenarios is the average of the maximum overloads from the overloaded scenarios of that group.

2.3.5 Average Overload

The average overload of a scenario is the average amount of bandwidth in excess on the directed links which are overloaded. The average overload for a group of scenarios is the average of the average overloads from the overloaded scenarios of that group.

2.3.6 Maximum Load

The load (or relative occupancy) of a directed link is the percentage of transported bandwidth in relation to the link capacity. For instance, a link with a capacity of 1 Gb/s carrying 750 Mb/s has a load of 75%. The maximum load of a scenario is the maximum load of all directed links. Considering a group of scenarios, the maximum load is the average of the maximum loads from the scenarios which are free of both overloads and micro-loops. If scenarios contain micro-loops or overloads, they are excluded from the calculation since this value would be adulterated otherwise.

2.3.7 Average load

The average load of a scenario is the average of the loads of all directed links. Considering a group of scenarios, the average load becomes the average of the average loads from the scenarios which are free of both overloads and micro-loops.

2.3.8 Served Bandwidth

The served bandwidth of a scenario is the percentage of requested bandwidth which has reached its destination, considering all the demands. The served bandwidth for a group of scenarios is the average of the served bandwidths from the scenarios which are free of both overloads and micro-loops.

2.3.9 Service Ratio

The service ratio of a scenario is the percentage of demands that have been fully served. A demand is fully served when 100% of its requested bandwidth has reached the destination. The service ratio for a group of scenarios is the average of the service ratios from the scenarios which are free of both overloads and micro-loops.

2.4 Optimization Strategies

This section discusses the strategies used to compute IGP costs that increase LFA general efficiency, rather than just failure coverage. Finding optimum link costs for load-balancing purposes is already known to be an NP-hard problem. The complexity of our optimization problem should be equivalent, so we must adopt some heuristic strategy in order to find good solutions.

The objective function considered for this problem uses a priority system with the following parameters:

- Micro-loop ratio
- Overload ratio
- Maximum overload
- Served bandwidth

The micro-loop ratio has higher priority than the overload ratio, the overload ratio has higher priority than the maximum overload, and the maximum overload has higher priority than the served bandwidth.

Let us represent this objective function by F. A value of F is a quadruple (A, B, C, D), where A is the micro-loop ratio, B is the overload ratio, C is the maximum overload and D is the served bandwidth. A solution x is ideal if $F(x) = (0, 0, 0, 1)$. Note that for most networks, due to topology limitations, ideal solutions are impossible to reach since it would imply a LFA coverage of 100%. A solution x is better than a solution y (*i.e.*, $F(x) > F(y)$) when:

$$\begin{aligned} F(x) > F(y) \Leftrightarrow & \big(A(x) < A(y)\big) \vee \big[\big(A(x) = A(y)\big) \wedge \big(B(x) < B(y)\big)\big] \\ & \vee \big[\big(A(x) = A(y)\big) \wedge \big(B(x) = B(y)\big) \wedge \big(C(x) < C(y)\big)\big] \\ & \vee \big[\big(A(x) = A(y)\big) \wedge \big(B(x) = B(y)\big) \wedge \big(C(x) = C(y)\big) \\ & \wedge \big(D(x) > D(y)\big)\big] \end{aligned} \qquad (7)$$

Let us consider a topology with 10 failure scenarios and four solutions S_1, S_2, S_3 and S_4 with the corresponding objective values: (0.1, 0.2, 300, 0.5); (0.0, 0.3, 500, 0.4); (0.0, 0.3, 250, 0.3) and (0.0, 0.0, 0, 0.25). Ignoring different failure types and assuming that all scenarios have the same weight in the solution performance parameters, S_1 has exactly one failure scenario containing micro-loop(s), S_2 and S_3 both have three overloaded scenarios, and solution S_4 fails to protect all demands. S_4 is the best of all solutions, while S_1 is the worst. Despite protecting less bandwidth, assigning the costs of S_4 ensures that repair traffic does not cause any micro-loop or congestion. We use the maximum overload criteria for the possibility of not being able to eliminate all overloads. S_3 is slightly better than S_2 because it is able to reduce the average maximum overload of the overloaded scenarios, which represents less packet loss.

Note that the objective function F ignores maximum and average load statistics. Having concerns involving load-balance during convergence time does not make much sense. It is not critical if links become heavy loaded due to repair traffic (as long as they are not overloaded), because this situation is still temporary. However, it may be important to optimize the use of

resources during the rest of the time, when no failures affect the network. Some load-balance constraints should then be applied regarding the default scenario. These constraints might even favor a more efficient use of LFA (it could free capacity to be used later on by repair traffic), but they can also have a negative effect in the optimization process. Less flexible constraints may narrow the set of possible solutions, reducing the chance of finding good solutions. Therefore, the adopted strategy is to simply use an acceptable threshold for the default scenario maximum load. Solutions above this threshold are considered invalid.

The most naive approach to solve this optimization problem is to explore all possible combinations of costs in a full search manner. This ensures that we find the optimal solutions, but it might take too long to do so. For instance, consider a simple topology with just 6 point-to-point links connecting 4 routers, 1 demand and 8 failure scenarios (besides the default scenario). Using cost values between 1 and 10 inclusive, there is a total of $10^{6\times2} = 1000000000000$ possible cost combinations. Imagine that half of those combinations (5×10^{11}) are rejected because of the load-balance constraint. If we consider that a solution takes 1 microsecond to be processed (*i.e.*, to build all routing tables) and 0.1 microseconds to process each one of the nine possible scenarios (the default scenario plus the eight failure scenarios), the total execution time of this algorithm would be approximately $5 \times 10^{11} \times (1\mu s + 0.1\mu s) + 5 \times 10^{11} \times (1\mu s + 9 \times 0.1\mu s)$, which is about 17 days an 9 hours. Real topologies are typically larger, and OSPF costs can be configured with any integer value between 1 and 65535. Therefore, a more elaborate method is required to find good solutions in a reasonable amount of time.

A better approach would be to start with one solution and gradually fix the detected issues by changing some IGP cost values. First, we try to eliminate existing micro-loops; then, we try to eliminate overloads; and, finally, we focus on increasing service protection. Each cost change is expected to generate a better solution. The solutions generated by the change of cost values depend a lot from the starting solution and may still be far from ideal. So, the process should be repeated several times with different initial solutions. The heuristic algorithm is described as follows:

Algorithm 1: HEURISTIC(Topology, min_cost, max_cost, max_load, time_to_leave)

```
1  F_best ← -∞
2  S_best ← { }
3  while StopOptimization() = FALSE do
4      x ← CreateRandomValidSolution(min_cost, max_cost, max_load)
5      (F', S') ← CreateBetterSolutions(x, min_cost, max_cost, max_load, time_to_leave)
6      if F' > F_best then
7          F_best ← F'
8          S_best ← S'
9      else if F' = F_best then
10         S_best ← S_best ∪ S'
```

```
11      end if
12  end while
```

F is the objective function, S represents a group of solutions and x is a single solution. All solutions in S are equivalent in terms of F. The algorithm logic is very simple: for each iteration, a newly random solution x is created; x is then used as the starting point to find better solutions; the returned solutions are finally compared with the current best. The optimization cycle can be stopped by simply establishing an iteration limit.

The method CreateRandomValidSolution(min_cost, max_cost, max_load) creates a solution with random costs between min_cost and max_cost. The default scenario maximum load must not exceed max_load. This generation of random costs is a simple straightforward process which is repeated if the costs fail the maximum load requirement. The real optimization is done in the next step.

The method CreateBetterSolutions(x, min_cost, max_cost, max_load, time_to_leave) tries to create better solutions from the initial solution x. It involves identifying the issues and trying to fix them by applying minimum changes in the costs of a solution. One change equals one new child solution when the change is within the cost limits. Child solutions may or may not become parents themselves, forming a tree-like search structure. A leaf is not further optimized if one of three following situations occur:

- The solution is invalid or worse than its parent;
- The solution is better, but has no fixable issues;
- The solution is equivalent to its parent but the time_to_leave was reached – each leaf has a counter to avoid entering in some sort of vicious cycle when descendants are constantly unable to improve their parents.

The method terminates when no more leaves are created. The best solutions are returned, along with the objective value. For example, imagine this method has originated the tree of solutions shown in Figure 2.3.

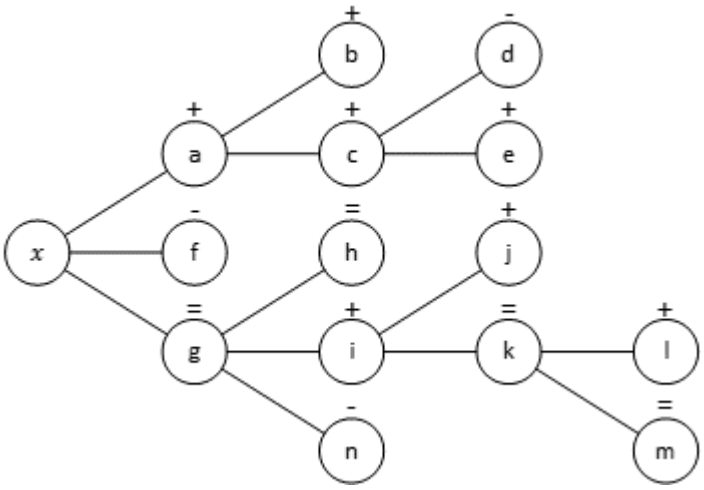

Figure 2.3 - A simple optimization flow

The comparison result between each child solution and its parent is given by symbols + (better), - (worse) and = (equivalent). The optimization used a time_to_leave of 2 solutions. The solutions represented by leaves *h* and *m* have expired because they have reached the time_to_leave value. Solution *k* has reset the time_to_leave counter. Solutions *d*, *f* and *n* were ignored because they were worse than their father solutions. Solutions *b*, *e*, *j* and *l* are optimized leaves with no issues left to fix, but they may be different between each other. The method should then return only the best of these four solutions *b*, *e*, *j* and *l*.

The time_to_leave factor can be very important, not only to control the depth of the search tree, but also to adapt the optimization algorithm to the size of the topology. The effect of one minimum cost change has more impact in smaller topologies. A very large topology may need several cost changes to reach solutions with different objective parameters, which leads us with three alternatives: increase the magnitude of each change, use a larger time_to_leave value, or a combination of these two. The latter can be performed by changing costs exponentially at first, and linearly after, once the child does not exist, is invalid or worse than its parent.

Figure 2.4 summarizes the entire optimization process.

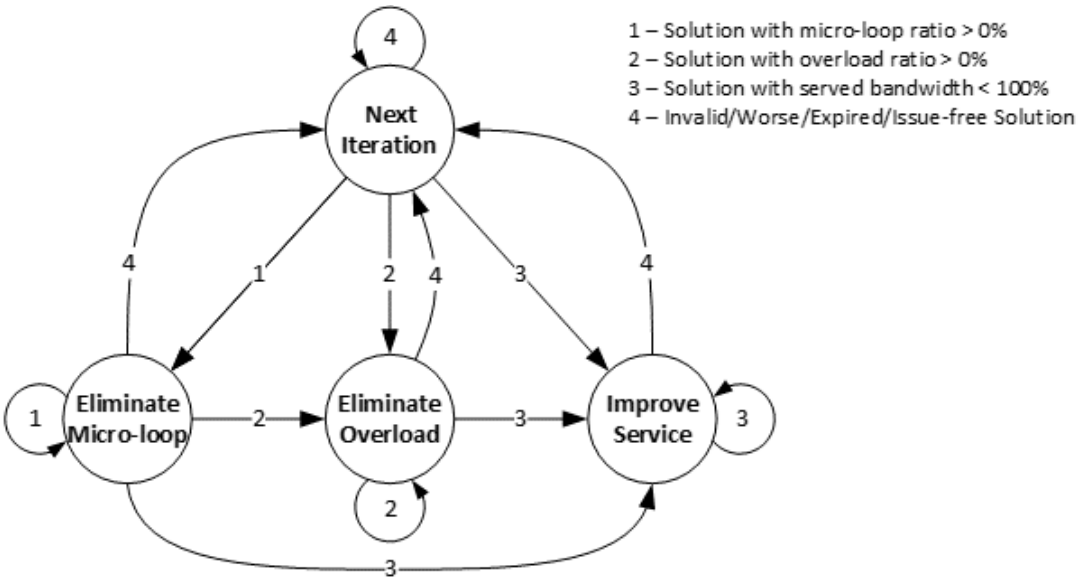

Figure 2.4 - Optimization process

The next subsections discuss the possible techniques to try fixing the three main issues: micro-loops, overloads and unrepaired traffic.

2.4.1 Eliminating Micro-loops

A micro-loop caused by repair flows is the result of using at least two link-protecting LFAs to forward traffic affected by a node failure. To eliminate a micro-loop, we should disable at least one of the responsible LFAs. A backup next-hop N can be disabled by using costs that force the failure of the Loop-Free Criterion (1), which means that the following condition must be verified:

$$dist(N, D) \geq dist(N, S) + dist(S, D) \quad (8)$$

We can simply increase the distance between N and D, decrease the distance between N and S and/or decrease the distance between S and D. There are several different ways of changing the shortest path distance between two routers. Decreasing a distance can be achieved by simply lowering some primary next-hop's outgoing link cost in the path to the destination. To make sure we increase a shortest distance, we should add the same value to the cost of each interface of the source router. However, there are some problems with these two approaches. It is important that we try minimum cost changes at a time in order to explore the closest surrounding solutions. We will consider four rules when performing cost changes to generate a child solution:

1. The change must involve only one router.
2. The change should affect the minimum number of interfaces possible.
3. The change should be the minimum possible.
4. The change should try to preserve the primary paths.

Decreasing the shortest path distance by lowering some primary next-hop's outgoing link cost would not be in accordance with rule 4 (because of the ECMP), while changing a distance by

modifying the costs of all the interfaces of the source router may not be in accordance with rule 2. Note that using this last method to meet condition (8) would be counter-productive: changing distance between N and D would also change distance between N and S in the same exact way, and vice-versa, thus maintaining the loop-free status of neighbor N. In order to consider all the above rules, we have decided to increase/decrease the distance between two routers in a minimum of Δ units by incrementing/decrementing the costs of the source outgoing links, corresponding to the primary next-hops to the destination, in exactly Δ units. Note that this technique may still not do the trick, because primary paths can still change and there is also the possibility that some changes may not be allowed due to cost limits. For these reasons, it might be a good idea to individually explore all three possibilities for disabling a LFA:

- Increase the distance between N and D.
- Decrease the distance between N and S.
- Decrease the distance between S and D.

Which of the LFAs causing a micro-loop should we try to disable? To answer this question, consider the case illustrated in Figure 2.5.

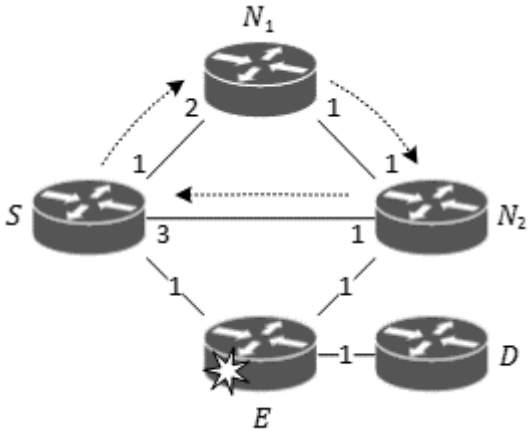

Figure 2.5 - A micro-loop affecting three links

Router S has two neighbors offering a link-protecting LFA for primary next-hop E, N_1 and N_2. Using the distance as tiebreaker, S will select N_1 to be the backup of E ($S \rightarrow N_1 \rightarrow N_2 \rightarrow E \rightarrow D$ = 4 and $S \rightarrow N_2 \rightarrow E \rightarrow D$ = 5). S is the link-protecting LFA for primary next-hop E of router N_2 in the way to D. Consider we have a demand with source in S and destination in D and that router E fails. A micro-loop will then be formed between three links. To remove this micro-loop, we should eliminate backup N_1 in S and/or backup S in N_2. Disabling the first repair seems the best option. Assuming primary paths are kept, it would release all the unnecessary repair traffic from links $S \rightarrow N_1$ and $N_1 \rightarrow N_2$. Consider that we change the cost of link $N_1 \rightarrow N_2$ to 2. N_1 is no longer a LFA of S, but a micro-loop will still occur because now N_2 becomes the LFA of S. If we remove the other backup instead, by changing cost of $S \rightarrow E$ to 4, the primary path will change but the micro-loop will be fixed with a single cost change.

It is always a valid option to test solutions that try to disable each of the backup next-hops causing the micro-loop, but disabling the first responsible LFA will generally produce better results, since it helps to avoid overloads as well.

Next, we present the algorithm to try disabling a backup next-hop n of a router s to a destination router d. This algorithm uses function $changeDistance(a, b, u)$, which schedules the execution of a child solution where the costs of the interfaces of all primary next-hops of router a to router b are adjusted in u units with respect to the current solution.

Algorithm 2: DISABLE_BACKUP(n, s, d)

$u \leftarrow dist(n, s) + dist(s, d) - dist(n, d)$ // n is a LFA, thus u is positive
if $n \neq d$ **then**
    $changeDistance(n, d, +u)$
end if
$changeDistance(n, s, -u)$
$changeDistance(s, d, -u)$

2.4.2 Eliminating Overloads

As discussed before, a load-balanced network is generally less prone to experience overloading caused by repair traffic. So, increasing the costs of heavy loaded links and decreasing those of lightly loaded links may seem a good strategy at first. The problem is that it lacks precision and can create some optimization pitfalls, like not being able to solve an overload or going back to having micro-loops. Increasing the cost of an overloaded link can actually help deflect other traffic and preserve the repairs, but it may also cause other links to become overloaded and a vicious cycle can be easily started.

Remember that a network loses resources after a failure. Depending on the failure extent and the average load, it may become impossible for the network to serve all the traffic. Considering a valid solution, overloads only happen because of repair traffic, so the only way to really ensure they disappear is to eliminate those repairs. By identifying the backup next-hop that is originally responsible for a certain repair flow inside an overloaded link, we can remove that LFA the same way we do for the micro-loop case.

Once again, which of the LFAs contributing for the overload should we eliminate? Note that an overloaded link may be transporting different repair flows with different bandwidths originated by LFAs that may not be the same. In this case, it might be better to try removing all the different backups, one at a time, since it is very hard to predict which LFA removal would generate a better solution. Therefore, Algorithm 2 should be executed to try disabling each one of the backup next-hops that are originally responsible for the different repair flows inside an overloaded link.

2.4.3 Fixing Service

In the previous subsections, we have been disabling backup next-hops to eliminate both micro-loops and overloads. Having less repair paths, the served bandwidth is expected to drop. Fixing service will take the opposite road of previous fixes. To protect traffic affected by a failure we should provide an adequate LFA. For each failure scenario, we should identify the failed primary next-hops of interest and change costs to meet at least condition (1). It may be possible to find child solutions that are able to improve global service without causing micro-loops or overloads. A promising way of fixing service while avoiding micro-loops is to consider the occurred failure when enabling the LFAs: for a primary next-hop that failed to protect traffic from a link failure, we should perform cost changes that try enabling a simple LFA of each of the other neighbors; for a primary next-hop that failed to protect traffic from a node failure, we should be more conservative and perform cost changes that try enabling a downstream path LFA of each of the other neighbors. The following algorithms are then used to schedule child solutions that try enabling a backup to a primary next-hop e of a router s to destination router d. Algorithm 3 performs cost changes to enable loop-free alternates, while Algorithm 4 tries to enable downstream paths.

Algorithm 3: ENABLE_LFA(e, s, d)

```
for c ∈ s.neighbors do
        n ← c.router
        if n = e.router and c.outgoingLink = e.outgoingLink then
                // neighbor is the primary next-hop, skip it
                continue
        end if
        u ← dist(n, d) − dist(n, s) − dist(s, d) + 1    // e has no LFAs, thus u is positive
        if n ≠ d then
                changeDistance(n, d, −u)
        end if
        changeDistance(n, s, +u)
        changeDistance(s, d, +u)
end for
```

Algorithm 4: ENABLE_DOWNSTREAM (e, s, d)

```
1  for c ∈ s.neighbors do
2          n ← c.router
3          if n = e.router and c.outgoingLink = e.outgoingLink then
4                  // neighbor is the primary next-hop, skip it
```

```
5                continue
6          end if
7          u ← dist(n, d) − dist(s, d) + 1        // e has no LFAs, thus u is positive
8          if n ≠ d then
9                changeDistance(n, d, −u)
10         end if
11         changeDistance(s, d, +u)
12   end for
```

When trying to enable a downstream path from a neighbor N of a router S, decreasing distance between N and the destination router D will never be allowed using the previously described implementation for method changeDistance. This happens because N uses S to reach D (N is not loop-free). Without changing primary paths, the distance between N and D will always be higher than the distance between S and D. The change would then result in a negative cost for an outgoing link from N to S, being rejected. Therefore, lines 8-10 of Algorithm 4 can be removed.

3 Implementation Details

This chapter describes the most relevant decisions regarding the implementation of the application. The application was developed using Java Development Kit 1.7 in Netbeans IDE. Java language was used mainly because of its high portability and multithreading facilities. Appendix A shows the main windows of the developed application.

During the first implementation, the goal was to minimize execution time. Depending on the complexity of the problem, even the best heuristics may need considerable time to find good solutions and, therefore, it is important to use efficient algorithmic implementations. Usually, the execution time of an algorithm can be improved by increasing the amount of space required to run it, either using temporary variables or more complex data structures. Some preliminary tests have shown that the first version of the application had some serious memory issues, causing the application to crash most of the times. Increasing Java's maximum heap size did not solve the problem and a major review of the implementation was required in order to use memory more efficiently, regardless of any additional execution overhead. Some of the techniques used to avoid memory exhaustion are also described along this chapter.

3.1 Overview

The following diagram presents the interactions between some of the main entities of the application:

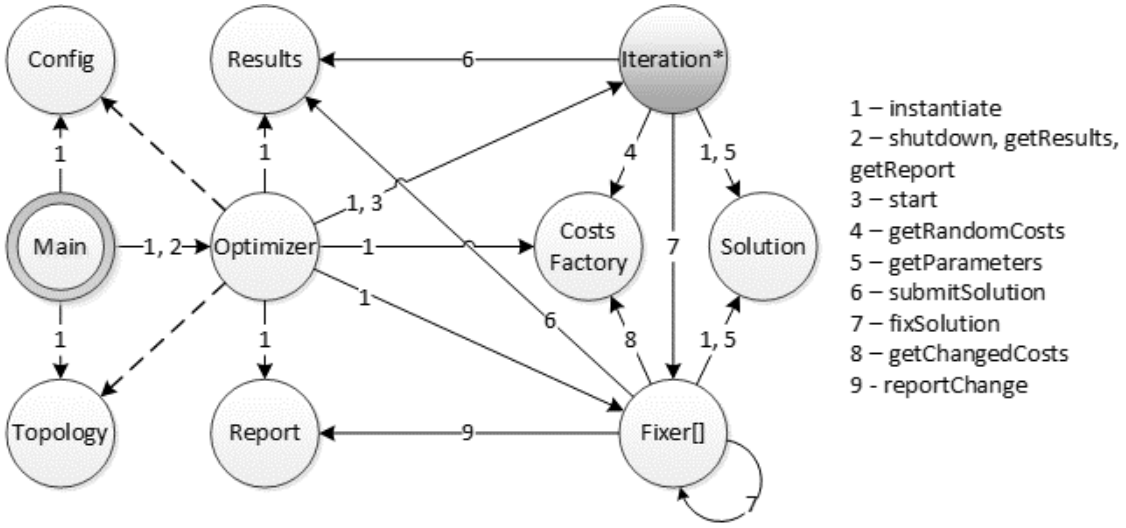

Figure 3.1 - Simplified interaction diagram of the application

The *Main* class is responsible for setting up the optimization environment. It loads the *Configuration*, creates a *Topology* and initializes an *Optimizer*. The *Optimizer* entity reads the configuration and initializes a group of fixers and three supporting entities:

- The *CostsFactory* entity, responsible for creating *Costs*;
- The *Results* entity, which applies the objective function to submitted solutions, containing a record of the best costs and best objective function value found during optimization;
- The *Report* entity, which is used to register the effect of each cost change performed by each *Fixer*, so that we may analyze the efficiency of fixers.

Several iterations are then initialized and executed. Each *Iteration* begins by requesting a random set of costs from the *CostsFactory* to obtain a valid *Solution*. Then, the failure scenarios are processed to see if the solution contains any fixable issue (*i.e.*, micro-loops, overloads or unprotected traffic). If so, the *Fixer* that is responsible for that issue will try to fix the solution by generating new costs.

3.2 Configuration

User can setup the configuration before running the optimization. Table 3.1 shows a description of all configurable parameters of the application.

Table 3.1 - Configuration properties

Property	Description
Minimum Cost	The minimum IGP cost.
Maximum Cost	The maximum IGP cost.
Default Cost	The IGP cost to use for the default solution.
Default Capacity	The default link capacity. A value above zero overrides all predefined link capacities.
Demand Percentage	Applies a percentage to all demand values. It is used to increase (>100) or decrease (<100) the amount of traffic.
Demand Delta	Adds a randomly chosen percentage within range [-delta, +delta] to each demand value. It is used to change traffic patterns. The Demand Delta is applied after applying the Demand Percentage.
Transit Incidence	Forces a minimum percentage of transit nodes. This configuration is used to change the number of node failure scenarios.
Weight Link	The weight of link failure scenarios in the solution parameters. When the weight is 0, link failures are not considered.
Weight Node	The weight of node failure scenarios in the solution parameters. When the weight is 0, node failures are not considered.
Load Tolerance	Sets the maximum load exceeding tolerance for default scenarios (in percentage).
Time To Leave	The maximum number of child solutions unable to improve the parent solution.
Maximum Solutions	Maximum number of solutions to test. When set, it is used as stopping criteria for the optimization.

Maximum Iterations	Maximum number of iterations to run. When set, it is used as stopping criteria for the optimization.
Maximum Changes	Maximum number of individual cost changes to perform. When set, it is used as stopping criteria for the optimization.
Execution Time	Maximum execution time. When set, it is used as stopping criteria for the optimization.
Memorize Costs	When set, the *CostsFactory* will memorize generated *Costs*, so it does not repeat sets of costs that have already been considered.
Criteria	Determines the objective function to use in the optimization.
Micro-loop Fixer	Determines the micro-loop fixer to use in the optimization.
Overload Fixer	Determines the overload fixer to use in the optimization.
Service Fixer	Determines the service fixer to use in the optimization.

3.3 Topology Representation

A network topology is modeled with a graph where each node is a vertex of the graph and each link is divided in two arcs with opposite directions. There are two standard ways of implementing graphs: using an adjacency matrix or a collection of adjacency lists. The choice between these two representations should consider the graph properties and the required functionalities. Let us consider a graph with n vertices and m arcs. An adjacency matrix is a two-dimensional $n \times n$ array where the entry in row i and column j is either 1, if the arc from vertex i to vertex j exists, or 0, otherwise. It requires n^2 of space, being preferable to use when the graph is dense (m close to n^2). Though providing fast lookups to check if two vertices are directly connected, it is slow when the algorithm requires iterating over arcs (it must check all vertices). Also, it is complex to use when multiple arcs are required between two vertices (when there are multiple links between the same routers), since it would need a list of arcs in the corresponding matrix entry. The adjacency list representation associates to each vertex a list containing all arcs originated on that vertex. The space usage is then proportional to the number of arcs, being adequate for sparse graphs. Real networks are usually sparse, so we might save some memory using adjacency lists. Adjacency lists may also speed up the execution of Dijkstra's algorithm due to fast iteration between arcs. Adding to the fact that it is trivial to support link redundancy, the adjacency list representation is by far the best algorithmic choice for this problem. Figure 3.2 shows an adjacency list representation for the topology from Figure 2.1. Note that to make it possible to have multiple links, each adjacency must refer to a link.

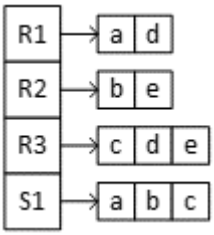

Figure 3.2 - An example of an adjacency list representation

Figure 3.3 shows a class diagram describing the topology implementation used in the application.

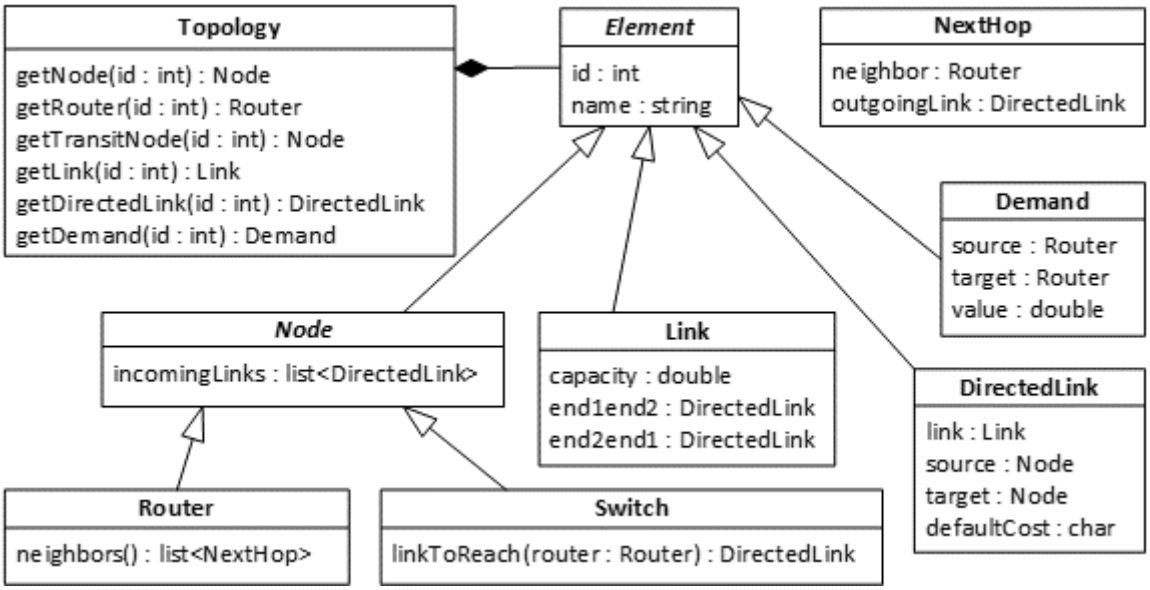

Figure 3.3 - Simplified class diagram of a Topology

The *Topology* class provides access to all elements within the network and to traffic demands as well. Since costs are asymmetrical, we create for each *Link* two directed links with opposite directions. Each *DirectedLink* has a source node, a target node and a defaultCost. It also contains a reference to the *Link* to know its capacity, enforcing the bidirectional assumption of links capacity. Note that an adjacency list is a member of a *Node* and contains the directed links that arrive to that node. Though it is common to use outgoing adjacencies, our Dijkstra's implementation uses the incoming arcs for reasons that will be discussed later in this chapter. There are two subtypes of *Node*: a router and a common layer 2 switch. When a *Topology* is built, each *Router* searches for its neighbors and saves them in a list. This list is used for the backup next-hops calculation. A *Switch* needs to have some kind of switching table, which maps each router connected to it with the corresponding outgoing interface. This table is very simple to calculate because switches represent pseudo-nodes. It is used both for neighbor discovery and to forward demands. Unlike routing tables, the switch tables are the same for all solutions, so they only need to be calculated once during the *Topology* initialization.

Each *Optimizer* is client of a single *Topology* instance. This is a central class, so all accesses should be optimized. Simple arrays are used to store the topology elements. Nodes, links, directed links and demands are all stored in four separate arrays. Each index of the array matches the corresponding element id, so that accesses are done the fastest way possible. An element id for a certain element type must then be a unique integer value inside interval $[0, N[$, where N is the total number of existing elements of that type. While the assignment of ids for links and demands can be arbitrary, for nodes and directed links it should follow specific rules. For example, for a topology with 8 routers, 2 switches and 2 demands, one from router A to router B and another from router C to router D, Figure 3.4 shows how the array of Nodes will look like for this topology (remember from section 2.2 that any router which is the source/destination of at least one demand is a non-transit router).

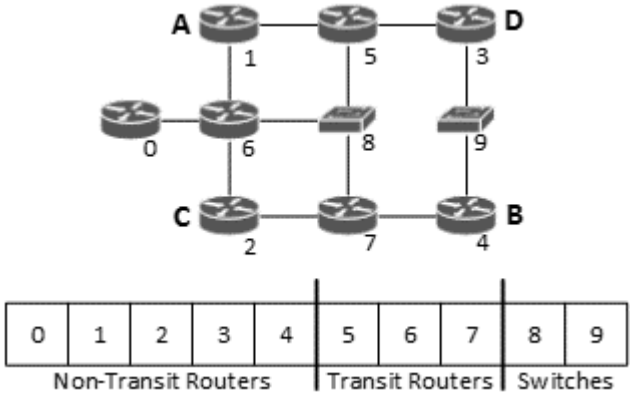

Figure 3.4 - A topology and the corresponding node array

Non-transit routers are inserted at the beginning of the array, transit routers at the middle and switches at the end. This configuration determines the element id assigned for each node. This configuration was adopted in order to provide efficient access with minimal memory usage according to the following implementation requirements:

- the Dijkstra's algorithm needs to access all nodes and needs to be able to identify each one with a non-negative integer value (the element id);
- to build the routing tables, all routers must be accessible;
- the transit nodes must be identifiable to process the corresponding failure scenarios.

Consequently, the Topology class provides three methods to access different ranges of the array:

- *getNode* – accepts values between 0 and N-1, where N is the total number of nodes;
- *getRouter* – accepts values between 0 and R-1, where R is the total number of routers;
- *getTransitNode* – accepts values between 0 and T, where T is the total number of transit nodes. The element id is then given by adding T to the value.

As for the array of directed links, the outgoing links of switches are saved at the end of the array. This configuration allows using the element id to access directed links whose cost is changeable.

3.4 Costs Factory

A solution associates a topology to one set of costs. These costs 'override' the default costs assigned to directed links (except when the cost is zero) and can be easily implemented using a simple array. Consider that an array c represents the costs of a solution, the cost assigned to the interface modeled by the directed link with id i is given by $c[i]$.

All costs providing the best solutions are kept in the Results object and should remain in memory during the optimization process, so that they can be saved once the optimization ends. This suggests an efficient implementation of the *Costs* object in terms of required space in memory. Another option would be to keep only the first *Costs* found (or the last), but two solutions may have the same objective function value and be considerably different. Early implementations used the 'int' data type for a cost value, simply because it is common practice. However, given the subsequent memory issues, this decision was revised. A Java primitive integer uses 4 bytes by default, while an OSPF interface cost is a positive value up to 65535, which only requires 2 bytes, the same space required for a 'char' type. Therefore, the data type representing each cost was changed from 'int' to 'char' allowing to save almost half the memory for each array of costs. Another memory optimization was to reduce the length of these arrays by not considering the costs of switches interfaces. This is why the outgoing links of switches are saved at the end of the *DirectedLink* array. These tweaks introduce some minor execution overhead:

- when Dijkstra's algorithm requests the cost of a directed link, it must check the default cost first, to see if that interface is attached to a router or a switch;
- when a *Fixer* requests a single cost change, the operation will involve one explicit downcast from 'int' to 'char' before saving the new cost, which must be checked at runtime.

There is another possible way to reduce the memory used by an instance of the Results object. It consists in ignoring equivalent costs (*e.g.*, [1,1,1,1], [2,2,2,2], [3,3,3,3],…), but this would require even more overhead to check the equivalence, which would degrade the optimization speed, specially for topologies with many links.

To create new *Costs*, we use a *CostsFactory* object which provides three methods:

- *getDefaultCosts* – returns a new *Costs* instance whose array is filled with the configured default cost.
- *getRandomCosts* – returns a new *Costs* instance whose array is filled with random values between the configured minimum cost and the configured maximum cost.
- *getChangedCosts* – returns a new *Costs* instance whose array is a modified copy of the array of a given *Costs* object.

Costs are randomly generated at the start of each iteration and changes are performed based on the resulting solution to generate new costs. These changes consist on adding, multiplying, subtracting or dividing a single cost value (or a small subset when altering shortest distances) by a specific amount. Any generated set of cost values must always follow the lower and upper cost boundaries and, for that reason, some changes may not be allowed. When this happens, the *getChangedCosts* method returns NULL and the requested solution is not created.

The *Configuration* provides an option to memorize the *Costs* that are generated, so that the *CostsFactory* does not return cost combinations that have already been considered. User can then choose between two different *CostsFactory* implementations. The base implementation has no memory of previously generated *Costs*, so it may repeat cost combinations. The second implementation, the *BoundedCostsFactory*, is a specialized *CostsFactory* containing a memory of the generated costs, so that it only returns sets of costs that have not been used before. This memory is implemented using a hash table mechanism to allow efficient lookups. To ensure that the application is safe, we should control the size of this hash table. A massive number of sets of costs may be requested during optimization and saving all of them could easily exhaust the machine's available memory. Whenever a new *Costs* object is generated, the factory checks if current memory usage exceeds 60% of maximum memory and, if so, it clears its memory (*i.e.*, it removes all *Costs* saved in the hash table). By having optimized the space required by a *Costs* object, we are able to save more costs, thus improving the efficiency of the *BoundedCostsFactory*.

3.5 Routing Table Construction

An OSPF router uses the information from the link-state database to build its routing table. Using Dijkstra's algorithm, it constructs a tree of shortest paths to all nodes in the network with itself as the root, being able to calculate the distance and primary next-hops for each destination router. This algorithm extends the classic Single-Source Shortest Path variation to allow Equal Cost Multi-Paths. The following box describes an efficient version of this algorithm. It uses a graph G and a source vertex s as inputs.

Algorithm 5: SOURCE_DIJKSTRA(G, s)

for $v \in G$.vertices() **do**
    $\text{dist}[v] \leftarrow +\infty$
    $\text{pred}[v] \leftarrow \{\}$
    $\text{visited}[v] \leftarrow \text{FALSE}$
end for
$\text{dist}[s] \leftarrow 0$
Q.insert(s, dist[s])
while not Q.isEmpty()
    $u \leftarrow Q$.deleteMin()

```
        visited[u] ← TRUE
        for a ∈ G.outgoingArcs(u) do
            v ← G.target(a)
            if visited[v] = FALSE then
                d ← dist[u] + G.weight(a)
                if d < dist[v] then
                    dist[v] ← d
                    pred[v] ← {a}
                    if Q.contains(v) then
                        Q.decreaseKey(v, d)
                    else
                        Q.insert(v, d)
                    end if
                else if d = dist[v] then
                    pred[v] ← pred[v] ∪ {a}
                end if
            end if
        end for
    end while
```

In Algorithm 5 (and the next Algorithm 6), G is a graph with the following methods:

- vertices() – returns all vertices in G;
- outgoingArcs(v) – returns all outgoing arcs of vertex v;
- incomingArcs (v) – returns all incoming arcs of vertex v;
- source(a) – returns the source of arc a;
- target(a) – returns the target of arc a.

In Algorithm 5 (and the next Algorithm 6), Q is a priority queue used to store the vertices whose distances from source are yet to be defined. It supports the following operations:

- isEmpty() – returns true if the queue is empty, false otherwise;
- insert(v, k) – inserts value v in the queue with priority k;
- deleteMin() – removes and returns the value with minimum priority from the queue;
- decreaseKey(v, k) - decreases the current priority of value v in the queue to k;
- contains(v) – returns true if the queue contains value v, false otherwise.

When the algorithm terminates, $\mathrm{dist}[i]$ contains the total cost of the shortest path(s) from s to vertex i and $\mathrm{pred}[i]$ is a list of incoming arcs of vertex i in the shortest path(s) from s to i. Note that each predecessor is a list of arcs, while for most of Dijkstra's algorithms found on literature it represents one single vertex. The reason for using lists is to enable ECMP, while the use of arcs instead of vertices is to consider possible link redundancy.

After running this algorithm, extracting the primary next-hops will need further processing of predecessors. Using the Single-Target variation of Dijkstra's will make this task much easier because this algorithm uses successors instead of predecessors. The following box contains the pseudo-code to calculate the shortest paths from all vertices to a single target vertex t within a graph G.

Algorithm 6: TARGET_DIJKSTRA(G, t)

```
for v ∈ G.vertices() do
        dist[v] ← +∞
        succ[v] ← { }
        visited[v] ← FALSE
end for
dist[t] ← 0
Q.insert(t, dist[t])
while not Q.isEmpty()
        v ← Q.deleteMin()
        visited[v] ← TRUE
        for a ∈ G.incomingArcs(v) do
                u ← G.source(a)
                if visited[u] = FALSE then
                        d ← dist[v] + G.weight(a)
                        if d < dist[u] then
                                dist[u] ← d
                                succ[u] ← {a}
                                if Q.contains(u) then
                                        Q.decreaseKey(u, d)
                                else
                                        Q.insert(u, d)
                                end if
                        else if d = dist[u] then
                                succ[u] ← succ[u] ∪ {a}
                        end if
                end if
        end for
end while
```

The difference from previous algorithm is that the processing is done backwards from the destination to all possible sources, so it must use incoming arcs. When the algorithm terminates, dist[i] now contains the total cost of the shortest path(s) from vertex i to target t, while succ[i] is the list of outgoing arcs of vertex i in the shortest path(s) to t.

Computing the primary next-hops is now almost trivial: if the vertex of a successor represents a router, then we found a primary next-hop; if the vertex represents a switch, each successor of that switch represents a new primary next-hop.

The only downside of using this variation instead of Single-Source's is that it encourages to build destination based routing tables instead of common source-based ones. Each routing table is then associated to a destination router and each line will refer to a source router. But this is just a minor representation issue and has no impact whatsoever in the optimization problem. In fact, it may even speed up the execution of a solution: to route each demand we only need the destination's routing table, so less accesses are required.

Dijkstra's algorithm is executed a lot of times during a single optimization and, so, the efficiency of this algorithm is crucial in order to find good solutions faster. The running time of Dijkstra's algorithm depends on its priority queue implementation. There is a vast area of research around priority queues and several different implementations. An efficient priority queue is typically backed up by a heap data structure. The most popular heap realization is the Binary Heap. Although a Fibonacci Heap offers better theoretical performance, it is a very complex data structure and proven to be more efficient only for denser graphs. A Binary Heap can be easily implemented using an array. Java already provides a simple binary heap with class java.util.PriorityQueue. Nevertheless, this heap does not support the decreaseKey() operation; to perform this operation, a client must remove the value and insert it again with the new priority. For this reason, a more efficient priority queue was implemented. The most important aspects of this implementation are:

- The heap structure is a primitive array of integers using 1-based indexing. Using 1-based instead of 0-based indexing makes the formulas for parent node, left-child and right-child simpler, so their calculation should be faster. This heap will contain the values ordered by their corresponding key. If the heap is not empty, the value at index 1 of the array always has the minimum key.
- A supporting array is used to store the indexes of values in the heap. In this way, a value can be found in constant time which will improve the performance of the decreaseKey() operation.

This priority queue has the limitations of being bounded, not allowing duplicates and accepting only integer values within interval $[0, N[$, where N is the heap capacity. The queue will then be used to store the element id of routers and switches.

Figure 3.5 shows the class diagram for the module responsible for the construction of all routing tables.

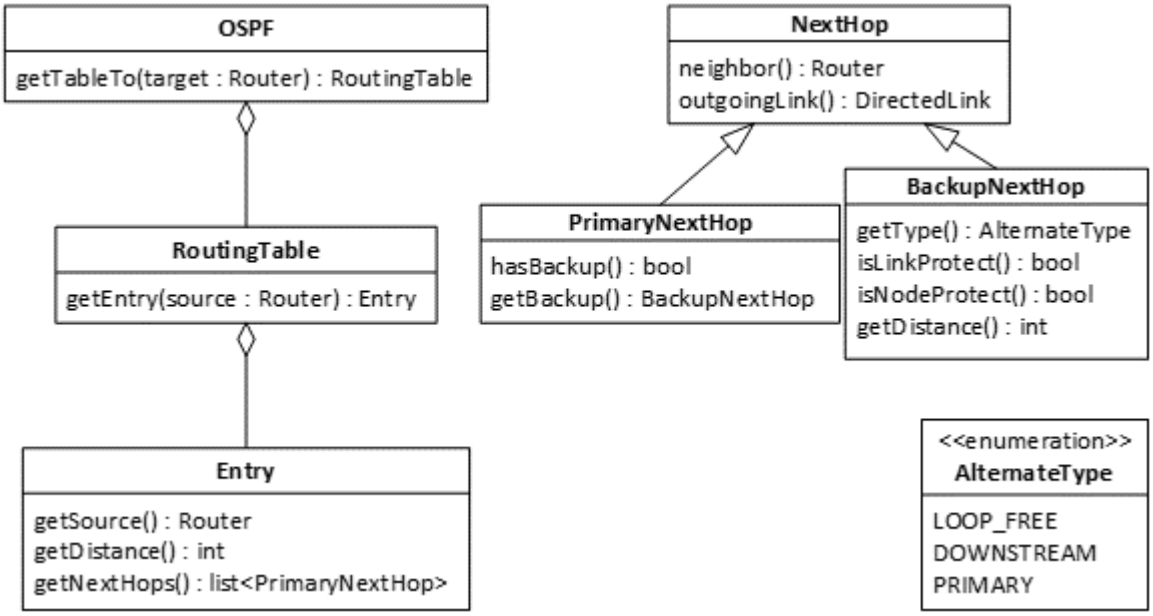

Figure 3.5 - Class diagram for the OSPF module

Each *Solution* initializes one *OSPF* class providing all the information needed for calculating the LFA coverage and to properly route traffic demands for each scenario. This initialization comprehends the following steps:

1. Execute Target Dijkstra's algorithm for all nodes
2. For all destination routers:
    a. For all source routers different from the destination:
        i. Find all primary next-hops from current source router to the destination
        ii. Compute the backup next-hop for each primary next-hop found in previous step

3.6 Loop-Free Alternates

Consider a solution x, a source router s, a destination router d and the list of primary next-hops p. The algorithm used to compute the backup next-hop for a primary next-hop $e \in p$ is as follows:

Algorithm 7: ComputeLFA(x, s, d, e, p)

```
1  for each NextHop n ∈ s.neighbors() do
2      if n = e then continue
3      if dist(n,d) ≥ dist(n,s) + dist(s,d) then continue        // not loop-free
4      temp ← BackupNextHop.init(n)
5      if n ∈ p then
6          temp.type ← PRIMARY
7      else if dist(n,d) < dist(s,d) then
```

```
8                   temp.type ← DOWNSTREAM
9           else
10                  temp.type ← LOOP_FREE
11          end if
12          if dist(n, d) < dist(n, e) + dist(e, d) then
13                  temp.nodeProtect ← TRUE
14          else
15                  temp.nodeProtect ← FALSE
16          end if
17          l ← e.outgoingLink()
18          if n.outgoingLink() = l then
19                  temp.linkProtect ← FALSE
20          else if l.isBroadcast() and
21                      dist(n, d) < dist(n, l.target) + dist(l.target, d)
22                  temp.linkProtect ← FALSE
23          end if
24          temp.distance ← x.getCost(n.outgoingLink()) + dist(n, d)
25          if Compare(temp, e.backup) > 0 then
26                  e.backup ← temp
27          end if
28  end for
```

This algorithm follows the selection procedure described in the LFAs specification. The main difference is that it does not maintain a list of equivalent backup next-hops. Since only one backup next-hop is used at a time, I consider that a backup next-hop is always better or worse than another. The function Compare($b1$, $b2$), defined in Algorithm 8, returns a positive value if backup next-hop $b1$ is better than backup next-hop $b2$:

Algorithm 8: Compare($b1$, $b2$)

```
1   if b2 = NULL return 1
2   if b1.type = PRIMARY and (b1.linkProtect or b1.nodeProtect) and b2.type ≠ PRIMARY
3           return 1
4   if b2.type = PRIMARY and (b2.linkProtect or b2.nodeProtect) and b1.type ≠ PRIMARY
5           return -1
6   if b1.nodeProtect and not b2.nodeProtect return 1
7   if b2.nodeProtect and not b1.nodeProtect return -1
8   if not b1.linkProtect and b2.linkProtect return -1
9   if not b2.linkProtect and b1.linkProtect return 1
10  if b1.type > b2.type return 1
11  if b1.type < b2.type return -1
```

```
12  if b1.distance < b2.distance return 1
13  if b1.distance > b2.distance return -1
14  return b2.neighbor.id – b1.neighbor.id
```

The LFA specification suggests the use of tiebreakers like IP addresses. So, I used the neighbor id as the final tiebreaker. This id is determined when routers find their neighbors during the initialization of a *Topology*. There is some micro-optimization in Algorithm 8: the order of terms, in the 'if' conditions, is chosen so that the second term is not evaluated most of the times.

3.7 Scenarios

After computing all routing tables, and for each scenario, the algorithm processes the amount of demand routed through each directed link from the demand's source router to the demand's destination router. The processing of each scenario must keep some sort of record of the flows that traverse each directed link. This record is not only used to calculate the scenario parameters, but also to identify any possible issues and provide all the needed information so that the assigned *Fixer* is able to apply accurate cost changes. Figure 3.6 shows the class diagram for the scenarios module.

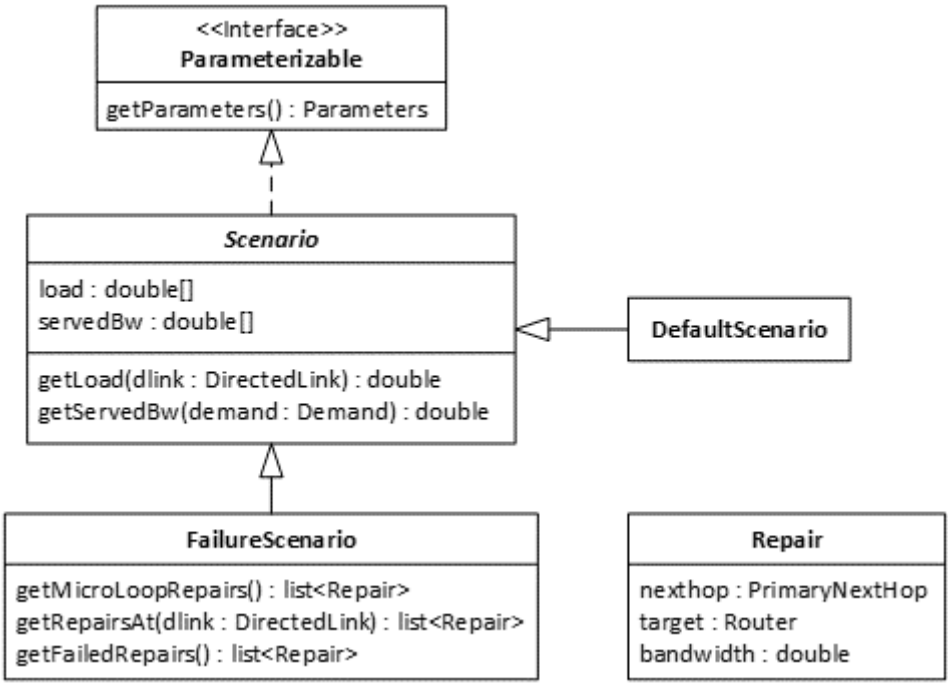

Figure 3.6 - Class diagram for the scenarios module

In order to save the load at each directed link, an array of doubles is used whose indexes correspond to the directed link id. To save the total served bandwidth of each demand, an array of doubles is used whose indexes correspond to the demand id. A *FailureScenario* needs three extra structures:

- a list of the first repair (one for each failure scenario) responsible for causing a micro-loop – this list is used to try to eliminate any existing micro-loop;
- an array of lists to save the repairs that traverse each directed link – this list is used to try to eliminate any existing overload;
- A list of repairs that were not performed due to the absence of a backup next-hop – this list is used to try to improve service.

A *Repair* object is created whenever a primary next-hop is down while routing a traffic demand. It contains information about the failed primary next-hop and the demand target so that we have all the needed tools to try disabling the respective backup next-hop (if it caused a micro-loop or overload) or enabling a LFA through other neighbors if the primary next-hop has no backup.

The processing of a default scenario is trivial: for each demand, we get the corresponding destination-based routing table and start traversing the directed links towards the destination, adding bandwidth to the load array in the process. The served bandwidth array is merely a formality, since the demands are always served.

The processing of a failure scenario is much more complex, because we must deal with failed primary next-hops, save the repair data and be aware of possible micro-loops. To help routing demands, an auxiliary structure named *Request* was created. A demand request is attended by each router in the way towards the destination. A *Request* has access to all the requests that were previously processed, so that micro-loops can be detected. It must also have a queue to save all *Repair* objects created along the way.

3.8 Iteration

Figure 3.7 presents the flowchart of one iteration of the optimization process.

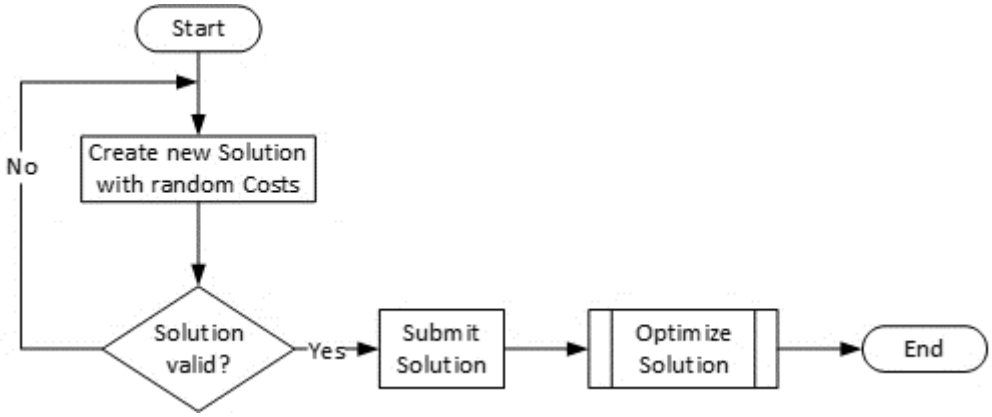

Figure 3.7 - Iteration flow

The *Iteration* comprehends lines 4-11 of the heuristic algorithm presented in Algorithm 1 with an important modification regarding the time at which the resulting solutions are evaluated. The process ‘Submit Solution’ is responsible for applying the objective function to the submitted solution. This process will then save the solution *Costs* whenever the objective function value is

better or equal than current best value saved in the *Results*. Instead of submitting the best solutions found at the end of the iteration, this process is called whenever the *Fixer* is able to find a solution that is at least as good as its parent regarding the current issue that is being fixed. A solution that is worse than its parent is not submitted, since it is guaranteed to have a worse objective function value than current best in the *Results*. Note that this variation of the algorithm executes a lot more comparisons which introduces some execution overhead, but it allows saving memory, since Algorithm 1 requires a record to save all the best solutions found by fixers.

Figure 3.8 shows the sub-process Optimize Solution in more detail.

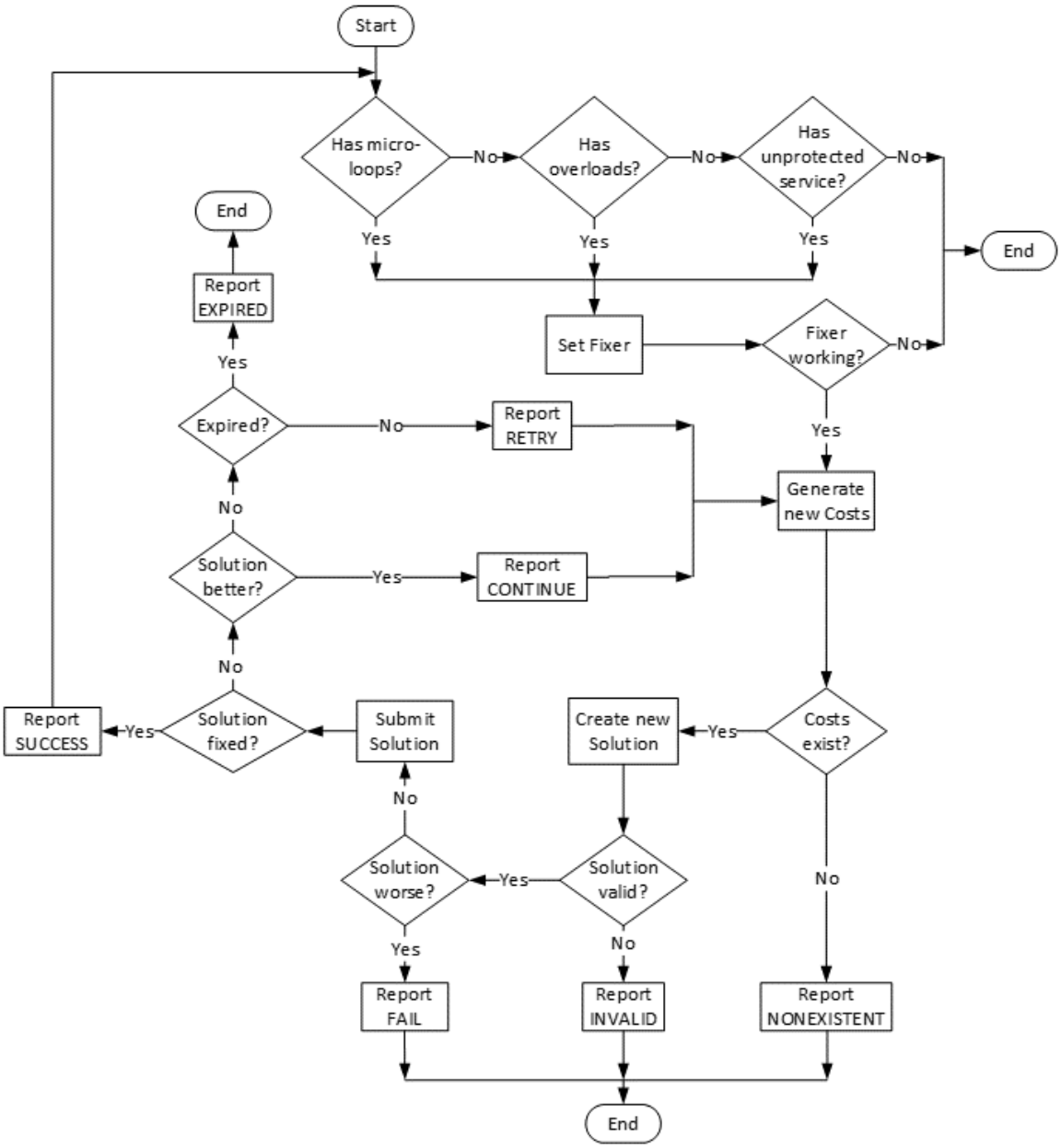

Figure 3.8 - Optimize Solution sub-process

The first step of this process is to determine if current solution has some fixable issue (*i.e.*, any micro-loops, overloads or unprotected traffic). If not, the solution is optimized and the process terminates, otherwise, it chooses an adequate *Fixer* to deal with that specific issue. The *Fixer* performs several cost changes to generate child solutions that may or may not be better than their parent. When the *Fixer* produces a solution that solves current issue, this solution is checked for any remaining issues and another *Fixer* can be assigned.

Each time a new cost change is performed by a *Fixer*, it must report the outcome of that specific change to the *Report* object. The *Report* object maps each change to several counters, one for each outcome. There are eight possible outcomes considered:

- NONEXISTENT – if the *CostsFactory* was unable to produce the requested change, due to cost limits violation;
- INVALID – if the change created a solution whose maximum load for the default scenario is invalid;
- FAIL – if the change created a solution which aggravates current issue;
- SUCCESS – if the change created a solution that solves current issue;
- CONTINUE – if the change created a solution that is closer to solve current issue;
- EXPIRED – if the change created a solution presenting the same issue and the time to leave is zero;
- RETRY – if the change created a solution presenting the same issue and the time to leave is above zero.

3.9 Multi-threading

One of the reasons to use Java is because of its multi-threading facilities. The application was designed to take advantage of multi-core machines by using multiple threads to perform the optimization. Each thread will be responsible for executing one iteration. To save resources and reduce thread creation overhead, the *Optimizer* uses a thread pool implemented with the help of Java's Executor Service. The number of threads in this pool is fixed and configurable at the start of each optimization process. Figure 3.9 illustrates the multi-threading feature using a simple interaction diagram.

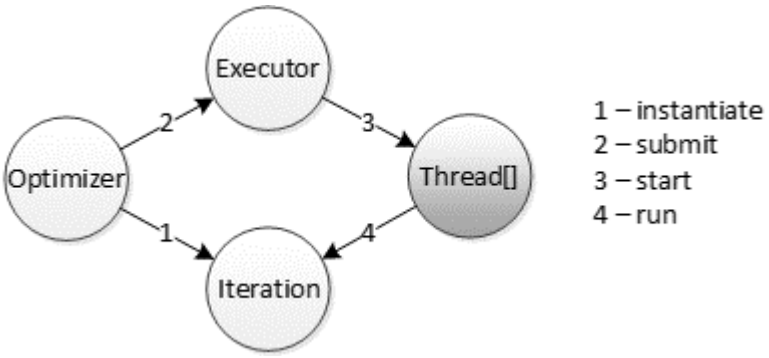

Figure 3.9 - Multi-threaded iterations

The *Optimizer*, *Executor* and *Iteration* are passive entities, while each *Thread* inside the thread pool (in dark gray) is the active entity that executes an *Iteration*. The *Executor* is responsible for managing the assignment of iterations to threads. Each time the *Optimizer* submits an *Iteration* to the *Executor*, it will wait until there is an idle thread in the pool, ready to run the submitted iteration. Several iterations are then executed concurrently and, therefore, it must be assured thread-safety for all objects shared by iterations to avoid any race-conditions and data corruption. The shared objects are: the *Topology*, the *CostsFactory*, the *Results* and the *Report*. The Topology and the base implementation of *CostsFactory* are immutable objects (*i.e.*, their state cannot be modified once they are created), which means they are inherently thread-safe. The *BoundedCostsFactory* must synchronize the access to the hash table containing the memorized *Costs*. As for the *Results* and the *Report* objects, thread-safety is achieved using simple mutual exclusion over shared resources.

4 Results and Discussion

This chapter presents several optimization results obtained with the use of the developed application. These results focus on a single topology and use different traffic matrices to generate different test scenarios and prove the efficiency of the developed heuristic algorithm. The optimization progress is also analyzed to confirm that high levels of LFA protection may lead to undesirable situations, especially when networks are more congested. The well-known Abilene network was used, so that results may be compared with previous research studies. Figure 4.1 shows the topology of Abilene network and Table 4.1 presents its topology characteristics.

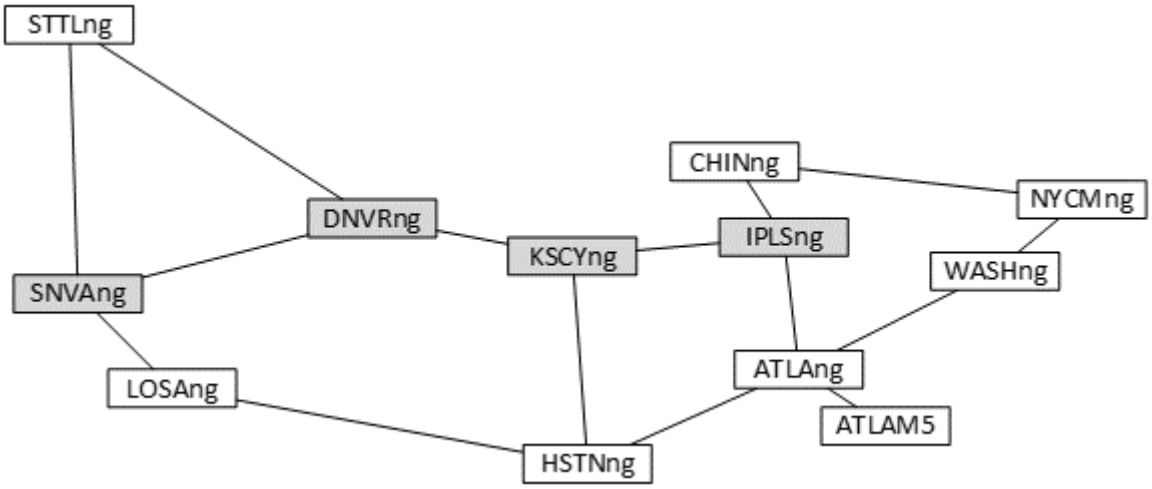

Figure 4.1 - Abilene network topology

Table 4.1 summarizes the topology characteristics of Abilene network.

Table 4.1 - Topology characteristics of Abilene network

Number of nodes	12
Number of links	15
Minimum node degree	1
Maximum node degree	4
Average node degree	2.50
Link density	22.73%

This is a relatively sparse network which suggests that it may be impossible to obtain 100% LFA coverage, regardless the chosen IGP costs. In fact, node ATLAM5 has only one neighbor and, thus, there will always be at least one primary next-hop without a backup.

We have considered that all links of the topology have a capacity of 100 Gbps. The base demand values used in test scenarios can be found in Appendix B and were taken from SNDlib library (http://sndlib.zib.de), which provides several examples of traffic demands for real networks,

intended to be used by researchers in their optimization problems. There are demand values for all source-destination pairs in a total of 132 demands. In order to enable some node failure scenarios, the four routers involving less traffic bandwidth were set to be transit routers and the traffic demands from/to these routers were discarded. As a result, throughout this chapter, we only consider 56 demands and the possible failure of routers KSCYng, SNVAng, IPLSng and DNVRng (highlighted in Figure 4.1 with the gray color).

We have defined two types of demand values, type 1 and type 2. Type 1 uses the original values from SNDlib, while type 2 adds a randomly chosen percentage within range [-50%, +50%] to each demand value of type 1. We may then conclude about the effects of using two different sets of demand values in both the optimization process and final results.

Before executing any long run to obtain optimized costs for this topology, several short runs were executed in order to determine the configuration values that best improve the overall quality of the optimization process. The following section presents the results of these preliminary tests. The application was executed in Java Virtual Machine (JVM) with 1.4GB of memory, running in Windows 7 Operating System of 64 bits and using an Intel Core i7 CPU at 2.3 GHz. The optimization of all test scenarios was conducted using 8 threads.

4.1 Preliminary tests

It may be hard to predict the behavior and outcome of a fixing strategy. Depending on the topology and the configuration used, some fixers may be more efficient than others. As discussed before, the more number of iterations we run, the better, because the final solutions of an iteration depend on the starting random solution. Therefore, we do not wish to spend too much time in a single iteration, especially if it is not finding better solutions. Fast iterations, on the other hand, might have the problem of returning worse solutions, since they explore a smaller number of cost changes. There should be some kind of tradeoff between the success rate of a fixer and the runtime it imposes to the iteration. Several tests were run to measure the efficiency of each fixer using different configuration values. All these preliminary tests consider the demand values type 1 (Appendix B).

4.1.1 Micro-loop Fixers

Only one micro-loop fixer named M1 was implemented, so we just need to find the configurations that provide better results. This fixer tries to disable the first backup next-hop responsible for a micro-loop using Algorithm 2 of chapter 2. To disable the backup next-hop, it creates a maximum of three child solutions with the following cost changes:

- M1_SD_DEC – change corresponding to a decrease of the distance between the source router and the destination router;
- M1_NS_DEC – change corresponding to a decrease of the distance between the neighbor providing the backup next-hop and the source router;

- M1_ND_INC – change corresponding to an increase of the distance between the neighbor providing the backup next-hop and the destination router.

To speed up the tests for this fixer and increase the probability for micro-loops to happen, the following configurations were used:

- Demand Percentage: 1%
- Load tolerance: 100%
- Link failures weight: 0
- Node failures weight: 1

By using very low bandwidth demands and a load tolerance of 100%, a solution will always be valid, since the maximum load for the default scenario will be lower than 100%. In this way, the fixer does not waste time with invalid solutions. Moreover, micro-loops can only occur during node failures, so we only consider node failure scenarios by setting the link failures weight to 0.

Any overload and service fixers are disabled during the optimization process of these runs. A solution is then considered to be optimized when the micro-loop ratio is zero.

Using the above setup, we are able to get more cost changes in shorter runtimes.

We have considered 4 different test cases to evaluate fixer M1: M1 Test 1, M1 Test 2, M1 Test 3 and M1 Test 4. M1 Test 1 uses all possible costs and a time to leave of 0 child solutions and it serves mainly as the basis for comparison to other tests. M1 Test 2 uses costs between 1 and 10, while M1 Test 3 uses costs between 1 and 100. The objective of these tests is to determine the effect of using different cost intervals in the optimization. M1 Test 4 uses the same interval of costs as M1 Test 3, but it considers a time to leave of 1 child solution. This test will tell us if there is any advantage on optimizing child solutions that were unable to improve the micro-loop ratio. All four test cases use a stopping criteria of 1000000 cost changes, which means each optimization run will end once the outcome of change number 1000000 is reported.

4.1.1.1 M1 Test 1: Costs between 1 and 65535; time to leave 0

Table 4.2 presents some statistics regarding the optimization process for this test case. These statistics tell us if the algorithm spends too much time in a single iteration. Figure 4.2 shows a bar chart presenting the amount of different reported outcomes for each individual change, while Figure 4.3 shows a pie chart with the global percentages of those results. This will be the approach used to present the results for all preliminary tests.

Remember that a fixer reports the outcome of each processed change. For this particular fixer, the SUCCESS report means that the change was able to eliminate all micro-loops, while CONTINUE means that the change resulted in a solution with a lower micro-loop ratio than the parent solution. The RETRY and EXPIRE reports mean that the change resulted in a solution with equal micro-loop ratio while the FAIL report means a solution with higher micro-loop ratio than its parent. INVALID means that the change originated an invalid solution and the NONEXISTENT report means that the

change requested nonexistent costs. We do not expect to get any INVALID results while testing this fixer with current configuration. Using a time to leave of 0, the RETRY report does not occur as well.

Table 4.2 - M1 Test 1: optimization statistics

Total execution time	0:01:47
Total number of iterations	685004
Total number of tested solutions	1139259
Iterations per second	6401,9
Solutions per iteration	1,7

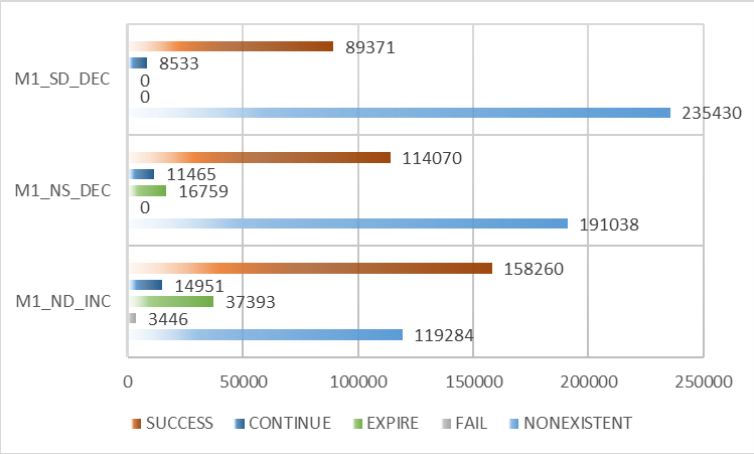

Figure 4.2 - M1 Test 1: change report

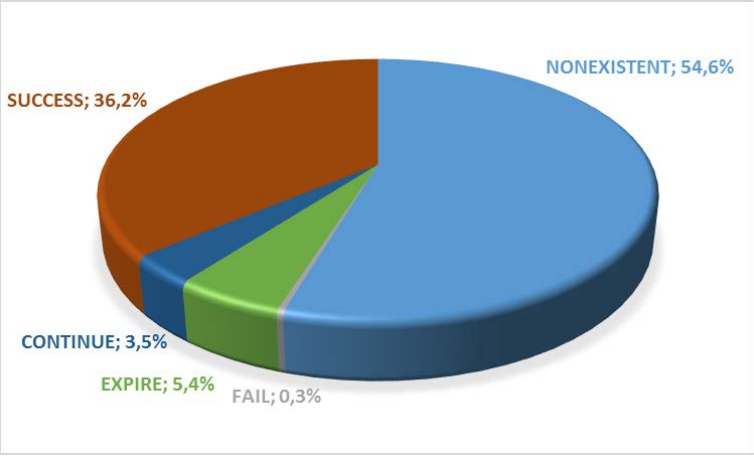

Figure 4.3 - M1 Test 1: change results

The chart from Figure 4.3 shows that only 0.3% of the applied changes resulted on a worse solution against the 39.7% (36.2% SUCCESS + 3.5% CONTINUE) of changes that were able to reduce the number of scenarios with micro-loop. This confirms that the adopted strategy is efficient to

eliminate micro-loops. Analyzing the effectiveness of each individual change in Figure 4.2, we can conclude that increasing the distance between the backup next-hop and the destination (M1_ND_INC) is more likely to disable it. The other changes were not so effective mostly because of nonexistent costs. 54.6% of the changes requested costs that were outside limits. It can be difficult to change shortest distances between two nodes by simply changing one interface cost, especially when considering large topologies and smaller cost intervals. Allowing all possible costs, despite generally augmenting shortest distances, should make it less probable for a cost change not to be allowed. Let us now see the effect of using smaller cost intervals in the number of nonexistent cost changes.

4.1.1.2 M1 Test 2: Costs between 1 and 10; time to leave 0

Table 4.3 - M1 Test 2: optimization statistics

Total execution time	0:01:50
Total number of iterations	693814
Total number of tested solutions	1083395
Iterations per second	6307,4
Solutions per iteration	1,6

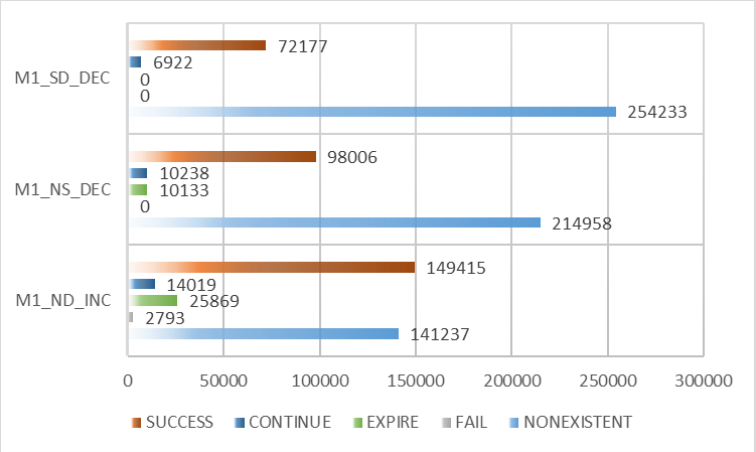

Figure 4.4 - M1 Test 2: change report

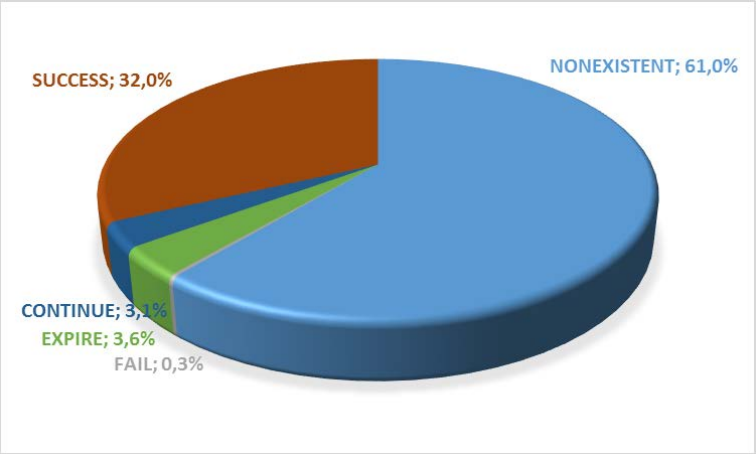

Figure 4.5 - M1 Test 2: change results

This cost interval is not adequate for this topology. By limiting the number of usable costs, we give less space for the required cost changes. Figure 4.5 shows that using a costs between 1 and 10 has significantly increased the number of nonexistent changes to 61% and decreased the overall success rate to 32%+3.1%=35.1%.

The conclusions from the analysis of Figure 4.4 are similar to the previous test.

4.1.1.3 M1 Test 3: Costs between 1 and 100; time to leave 0

Table 4.4 - M1 Test 3: optimization statistics

Total execution time	0:01:44
Total number of iterations	687597
Total number of tested solutions	1134233
Iterations per second	6611,5
Solutions per iteration	1,6

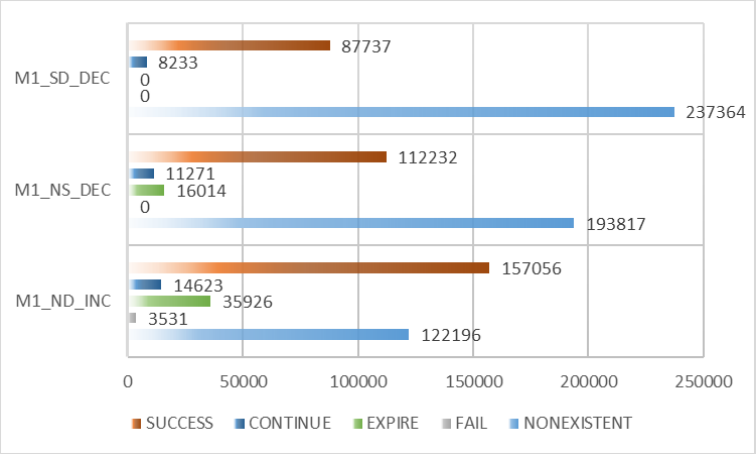

Figure 4.6 - M1 Test 3: change report

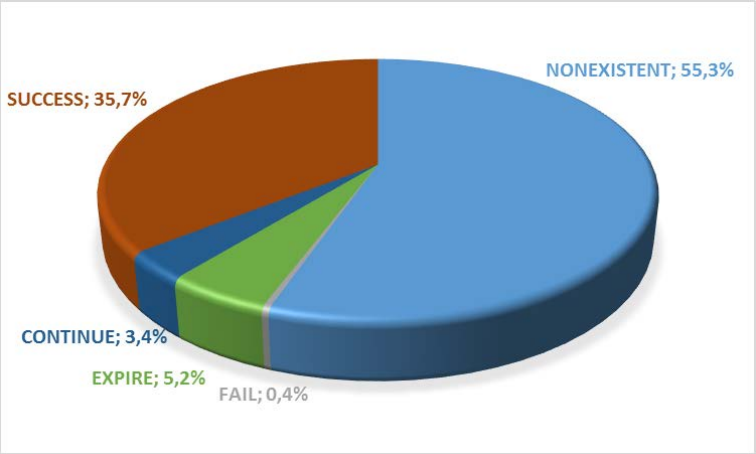

Figure 4.7 - M1 Test 3: change results

Figure 4.7 shows that using a range of costs between 1 and 100 produces far better results than Test 2 and only slightly worse results than Test 1. For this reason, this will be the configuration used in the final optimizations, since it uses more friendly values for IGP costs.

The conclusions from the analysis of Figure 4.6 are similar to previous conclusions for Test 1.

Note that 5.2% of the requested changes have expired, *i.e.*, they resulted on solutions that neither improved nor worsened the micro-loop ratio. If we use a time to leave of at least 1 child solution, the fixer might be able to eliminate the backups causing micro-loops on these solutions.

4.1.1.4 M1 Test 4: Costs between 1 and 100; time to leave 1

Table 4.5 - M1 Test 4: optimization statistics

Total execution time	0:01:37
Total number of iterations	511856
Total number of tested solutions	955355
Iterations per second	5276,9
Solutions per iteration	1,9

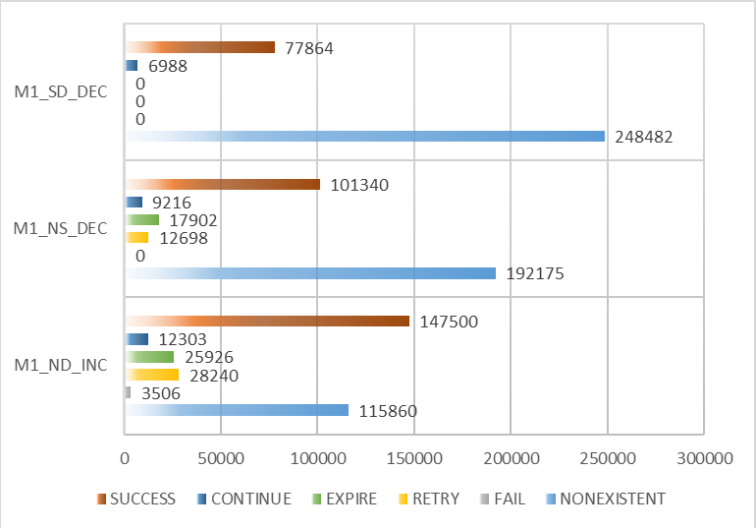

Figure 4.8 - M1 Test 4:change report

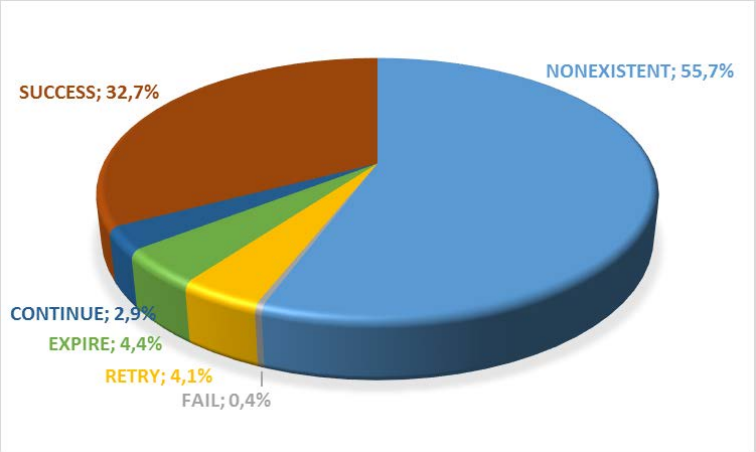

Figure 4.9 - M1 Test 4: change results

Using a time to leave of 1 degrades the overall performance of the fixer. Figure 4.9 shows that we have only slightly decreased the number of EXPIRED reports in comparison with previous test. This does not compensate reducing the number of iterations per second (by having more solutions per iteration). The time to leave factor has less importance when the fixer applies pre-calculated changes.

4.1.2 Overload Fixers

Four overload fixers were implemented (O1, O2, O3 and O4), each one using its own strategy to decrease the overload ratio. Fixer O1 applies the same methodology as fixer M1 in order to disable each backup next-hop that was responsible for a repair flow traversing each overloaded link. The other fixers use a naive approach to try to eliminate overloads. Fixer O2 performs minimum changes to solutions by incrementing the cost of an overloaded uplink by one unit. Fixer O3 performs deeper

changes by doubling the current cost of an overloaded uplink. Fixer O4 combines deeper changes with smaller ones: it keeps doubling the current cost of an overloaded uplink at first, but once the change is nonexistent or resulted in an invalid or worse child solution, it begins to increment the uplink cost by an amount corresponding to one tenth of the maximum allowed cost value. We have tested fixers O2, O3 and O4 mainly to prove that Fixer O1 uses a better strategy for eliminating overloads.

In order to test the overload fixers, we need test instances where there are overloaded cases and, therefore, we must use a more congested network by considering more demands and/or higher demand values. There is the possibility of generating invalid solutions (*i.e.*, solutions whose maximum load of the default scenario exceeds the defined value). In order to minimize the number of invalid solutions, we allow default scenarios with 100% maximum load by setting the load tolerance to 100%. Also, we will only consider link failure scenarios (to avoid any micro-loops) and the service fixer is disabled during the optimization process. The base configuration used in the following tests is:

- Demand percentage: 110%
- Load tolerance: 100%
- Link failures weight: 1
- Node failures weight: 0

We have considered seven different test cases: O1 Test 1; O2 Test 1; O2 Test 2; O3 Test 1; O3 Test 2; O4 Test 1 and O4 Test 2. O1 Test 1 tests the overload fixer O1 using costs between 1 and 100 and a time to leave of 0 child solutions. The stopping criteria for this test is 10000000 cost changes. This test is intended to prove the efficiency of fixer O1 on eliminating overloads. The two tests for both fixers O2 and O3 will show how they perform when using different intervals of costs. In the first test of fixer O4 (O4 Test 1), we are expecting to obtain better results than when using fixers O2 and O3. O4 Test 2 tests the effects of using a bigger time to leave. All tests for fixers O2, O3 and O4 use a stopping criteria of 500000 cost changes.

4.1.2.1 O1 Test 1: Costs between 1 and 100; Time to leave 0

Table 4.6 - O1 Test 1: optimization statistics

Total execution time	0:01:38
Total number of iterations	1557
Total number of tested solutions	408559
Iterations per second	15,9
Solutions per iteration	262,4

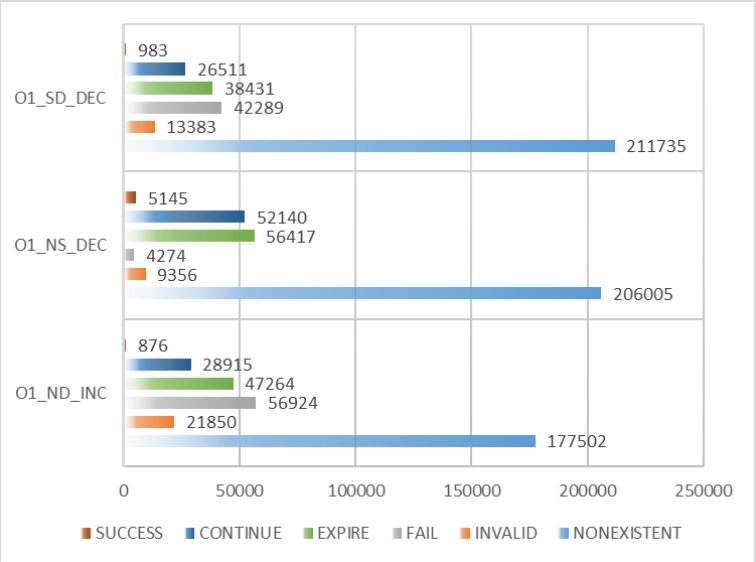

Figure 4.10 - O1 Test 1: change report

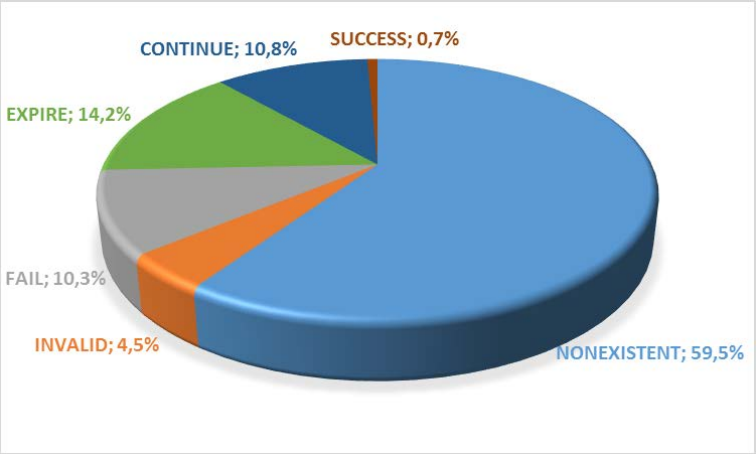

Figure 4.11 - O1 Test 1: change results

To improve the overload ratio of a solution, fixer O1 tries a lot more cost changes than fixer M1. While fixer M1 only tries to eliminate the first repair responsible for each micro-loop, fixer O1 tries to eliminate each repair that traverses an overloaded link. Eliminating a repair involves requesting a maximum of three cost changes, which may severely increase the number of solutions per iteration. If we ran this test more times, we might notice a lot more variation on the results comparing with the micro-loop tests, since we will be running much less iterations. Figure 4.10 suggests that disabling a backup next-hop by decreasing the distance between the corresponding neighbor and the source router (O1_NS_DEC) is more likely to eliminate the overloads. Figure 4.11 shows that this fixer was able to produce better solutions for 10.8%+0.7%=11.5% of the requested cost changes, but only 0.7% eliminated all overloads. Although it can be difficult to eliminate overloads, especially when considering highly congested networks, choosing more carefully which

repairs to remove could have increased the success rate. For instance, imagine we always start by disabling the backup next-hop that is responsible for most of the repair bandwidth in the overloaded link. This might eliminate the overloads more quickly, performing less cost changes, but this fixer was not implemented because it would require more memory and a lot more processing before each cost change (to determine which backup to disable). Also, a higher success rate of the fixer does not necessarily mean that we find better solutions: disabling a backup repairing less traffic can be sufficient to eliminate an overload, which would unprotect less service.

Fixers O2, O3 and O4 perform less precise cost changes, so they should consider some 'time to leave' to be more efficient. The number of nonexistent changes is expected to be lower than previous tests, which might increase the total execution time (more solutions are processed). In order to reduce execution time, the following tests consider 500000 cost changes.

4.1.2.2 O2 Test 1: Costs between 1 and 100; Time to leave 1

Table 4.7 - O2 Test 1: optimization statistics

Total execution time	0:02:28
Total number of iterations	53142
Total number of tested solutions	620613
Iterations per second	359,1
Solutions per iteration	11,7

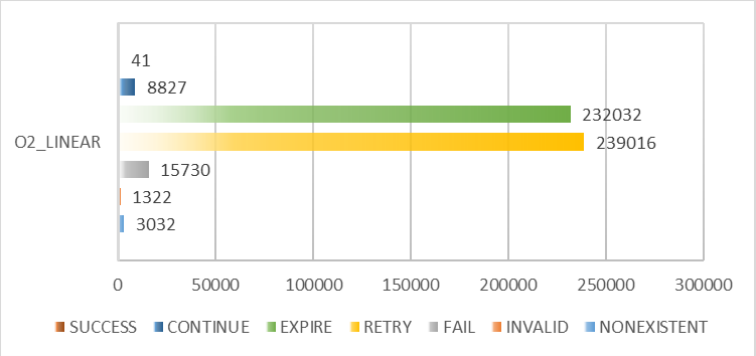

Figure 4.12 - O2 Test 1: change report

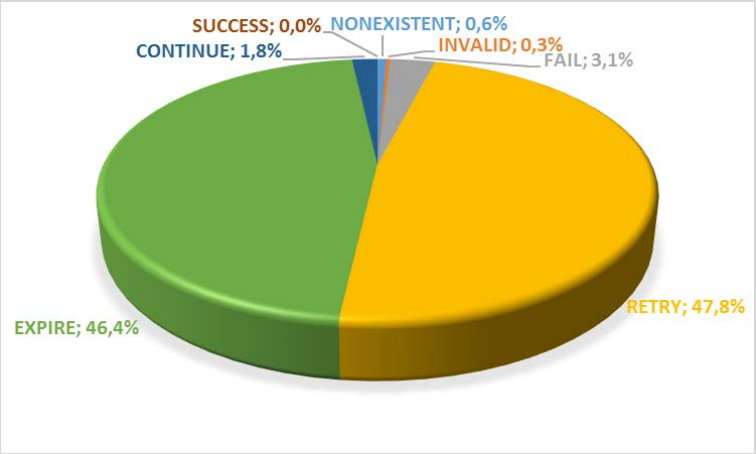

Figure 4.13 - O2 Test 1: change results

Fixer O2 increases the cost of an overloaded link by one unit. It is not expected to perform well on larger topologies and/or when considering a wider costs range. Figure 4.12 shows that using costs between 1 and 100 and a time to leave of 1, this fixer was able to eliminate the overload from only 41 of the 500000 total changes it has performed. Figure 4.13 shows that 94.2% (RETRY+EXPIRE) of all changes resulted in solutions that did not change the overload ratio, which reflects the poor performance of fixer O2 when considering large cost intervals. Increasing the time to leave for this fixer using large cost intervals should be insufficient to improve the fixer efficiency, since the number of RETRY reports would gain percentage in relation to other results. A better strategy would be to increment costs using a larger unit, like 10% of the maximum allowed cost. To test this strategy, we run fixer O2 using costs between 1 and 10.

4.1.2.3 O2 Test 2: Costs between 1 and 10; Time to leave 1

Table 4.8 - O2 Test 2: optimization statistics

Total execution time	0:02:09
Total number of iterations	38191
Total number of tested solutions	544084
Iterations per second	296,1
Solutions per iteration	14,2

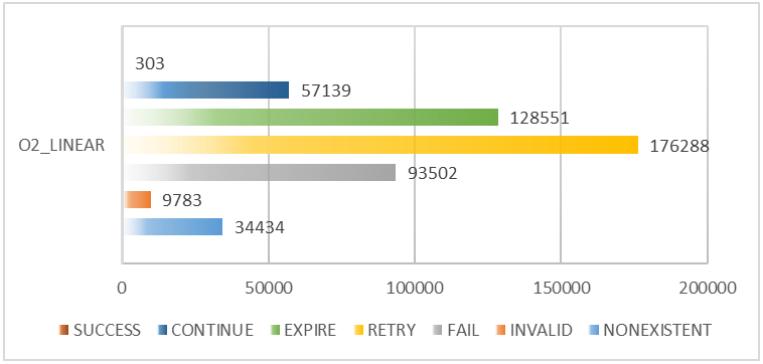

Figure 4.14 - O2 Test 2: change report

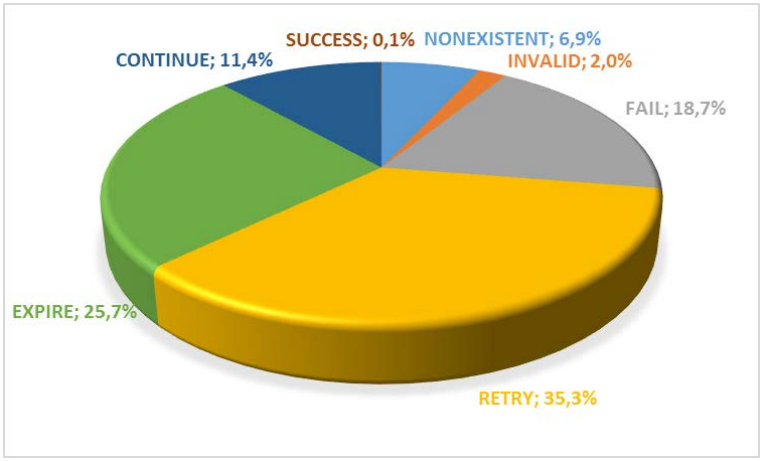

Figure 4.15 - O2 Test 2: change results

By using a smaller interval of costs, we were able to improve the effectiveness of this fixer. Figure 4.14 shows that the fixer succeeded to eliminate all overloads with 303 changes against the 41 in previous test. In Figure 4.15, we see a much smaller percentage of RETRY and EXPIRE reports, which means that the fixer was more capable of influence traffic in the network. The general increase of the percentages of the other outcomes is the consequence of this increased influence.

4.1.2.4 O3 Test 1: Costs between 1 and 10; Time to leave 1

Table 4.9 - O3 Test 1: optimization statistics

Total execution time	0:01:21
Total number of iterations	54930
Total number of tested solutions	395770
Iterations per second	678,1
Solutions per iteration	7,2

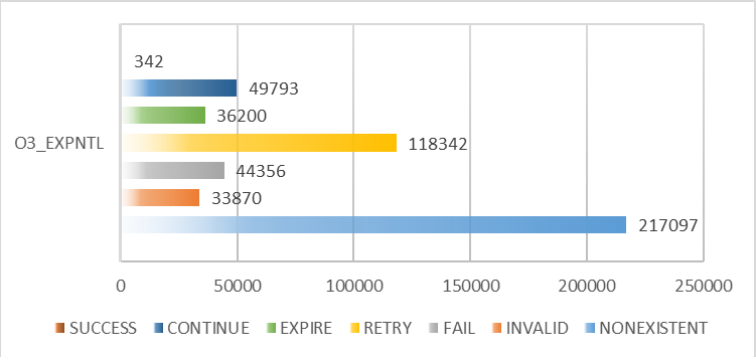

Figure 4.16 - O3 Test 1: change report

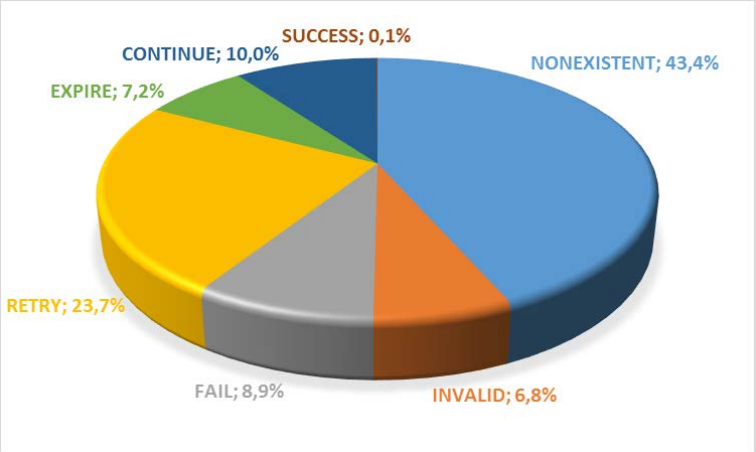

Figure 4.17 - O3 Test 1: change results

Fixer O3 doubles the cost of an overloaded uplink. If the interface attached to that link has a cost above 5, and since the maximum cost is 10, then the requested change will return nonexistent costs. This is why we have a NONEXISTENT percentage close to 50% (Figure 4.17). The global success rate of fixer O3 is very similar to the one registered for fixer O2 when using a cost interval between 1 and 10. The main difference between these fixers is the time taken to find better solutions. Fixer O2 took 2 minutes and 9 seconds to test 500000 cost changes (Table 4.8), while fixer O3 took only 1 minute and 21 seconds, performing much more iterations (Table 4.9).

4.1.2.5 O3 Test 2: Costs between 1 and 100; Time to leave 1

Table 4.10 - O3 Test 2: optimization statistics

Total execution time	0:01:19
Total number of iterations	55255
Total number of tested solutions	423774
Iterations per second	699,7
Solutions per iteration	7,7

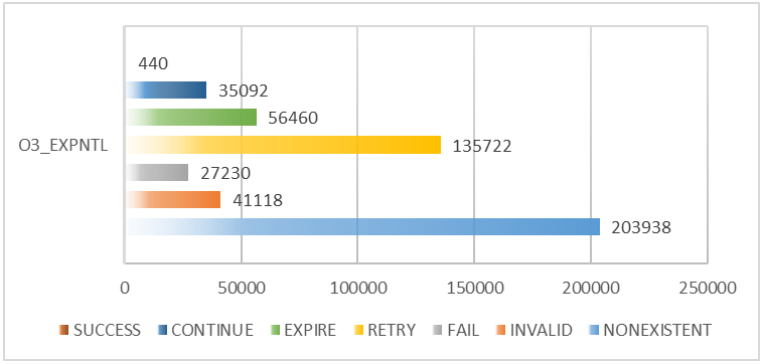

Figure 4.18 - O3 Test 2: change report

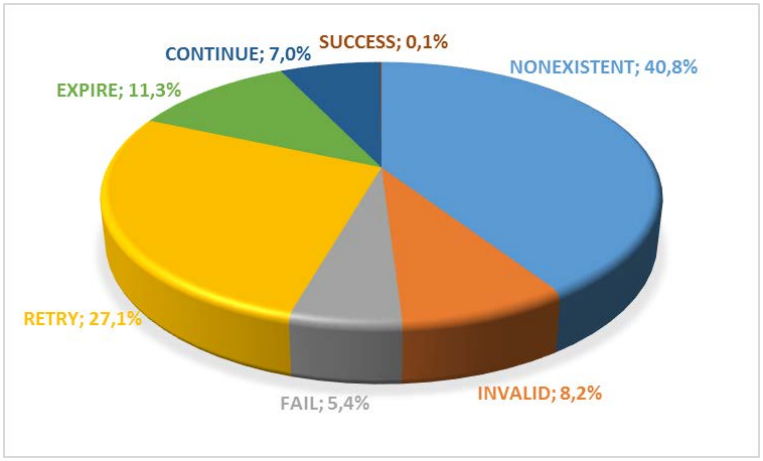

Figure 4.19 - O3 Test 2: change results

By comparing Figure 4.19 with Figure 4.17, we can conclude that the performance of fixer O3 is not influenced by the chosen costs range. Fixer O3 does not need to increase the magnitude of the change to perform well with larger cost intervals like fixer O2. The only problem with fixer O3 is that it may not be able to explore some solutions offering even better performance parameters because it does not make finer cost adjustments. There is a large percentage of nonexistent costs that would exist if we have incremented the interface cost instead of doubling its value. Fixer O4 mixes the speed of O3 with the accuracy of fixer O2.

4.1.2.6 O4 Test 1: Costs between 1 and 100; Time to leave 1

Table 4.11 - O4 Test 1: optimization statistics

Total execution time	0:01:36
Total number of iterations	26135
Total number of tested solutions	434350
Iterations per second	272,2
Solutions per iteration	16,6

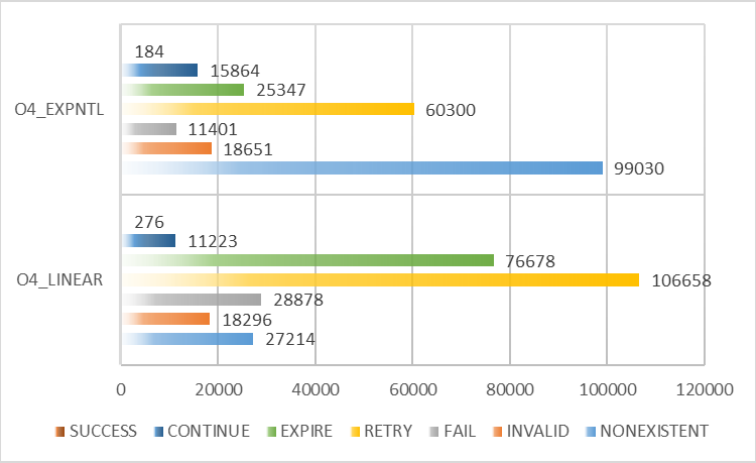

Figure 4.20 - O4 Test 1: change report

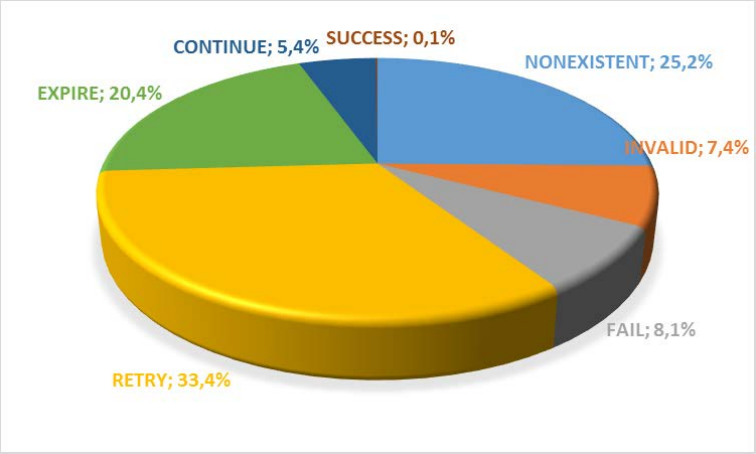

Figure 4.21 - O4 Test 1: change results

Fixer O4 keeps doubling the cost interface of an overloaded uplink at first, but once the change is nonexistent or results in an invalid or worse child solution, it starts incrementing an amount corresponding to one tenth of the maximum allowed cost value. Figure 4.21 shows a smaller percentage of nonexistent costs in relation to the percentage registered using fixer O3. Comparing with fixer O3, and despite spending more time in a single iteration, the success rate of fixer O4 is practically the same.

Analyzing Figure 4.20, we see that this fixer was able to eliminate more overloads while performing linear changes (276) than with exponential changes (184). Still, linear changes could benefit from using a higher time to leave value, since a total of 76678 of those have expired. The following test will run fixer O4 with a time to leave of 5.

4.1.2.7 O4 Test 2: Costs between 1 and 100; Time to leave 5

Table 4.12 - O4 Test 2: optimization statistics

Total execution time	0:01:33

Total number of iterations	13255
Total number of tested solutions	419109
Iterations per second	142,5
Solutions per iteration	31,6

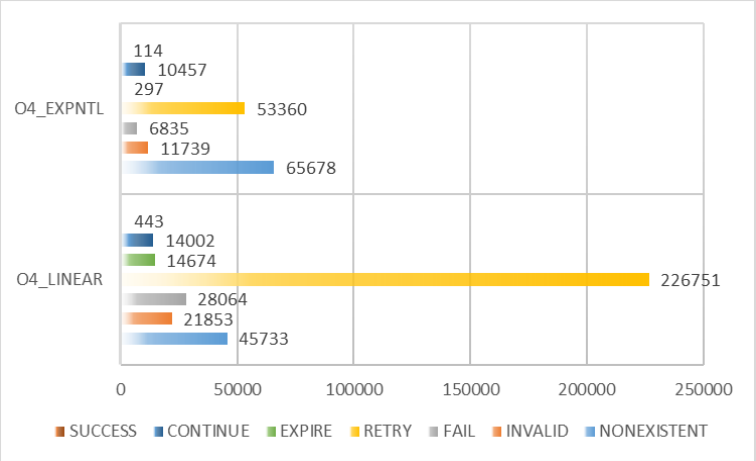

Figure 4.22 - O4 Test 2: change report

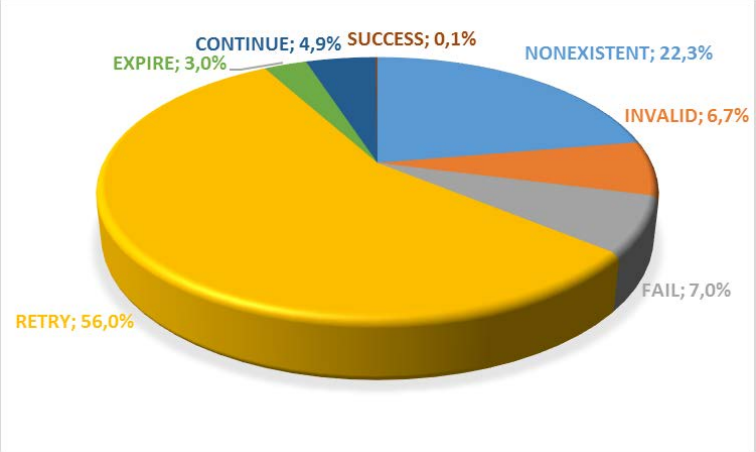

Figure 4.23 - O4 Test 2: change report

Figure 4.22 shows that fixer O4 has performed more linear increments. Using a higher time to leave value, doubling costs is much more likely to return nonexistent costs (65678) than to expire (297), which will make the fixer switch to linear increments more often. Linear increments, on the other hand, can retry more times before returning nonexistent costs. Observing Figure 4.23, it is difficult to say if we have improved the efficiency of fixer O4 by increasing the time to leave value: the CONTINUE percentage has decreased from 5.4% to 4.9%, but the FAIL percentage has also decreased from 8.1% to 7.0%. The high RETRY percentage (especially due to repeated increments) has contributed for a slightly overall decrease of other percentages, but the EXPIRE percentage has

dropped from 20.4% to 3% in relation to the previous test. Using a higher time to leave value for this fixer will increase the depth of the optimization tree of each iteration. Table 4.12 confirms that we have much more solutions per iteration when using a time to leave of 5 than when using a time to leave of 1 (31.6 against 16.6). By retrying the linear increments more times, the fixer has explored more solutions, being able to find 443 solutions without any overloads against the 276 solutions found previously (Figure 4.22). Fixer O4 achieved better results using a time to leave of 5, despite running about half the iterations.

Nevertheless, comparing the results of this fixer with the results of fixer O1, there is no doubt that fixer O1 is far more capable of eliminating overloads.

4.1.3 Service Fixers

Two fixers were implemented to try to increase the served bandwidth by enabling backup next-hops. The first fixer, S1, simply tries to force neighbors to meet the loop-free criterion while the second fixer, S2, has a more conservative approach by using the downstream path criterion whenever there is a risk of creating a micro-loop if a simple link-protecting loop-free alternate was enabled. Fixer S1 uses Algorithm 3, while fixer S2 may sometimes use Algorithm 4. For testing these fixers, we will avoid any invalid solutions and overloads (similar to what was done for the micro-loop fixer tests). The base configuration is as follows:

- Costs range: 1-100
- Demand percentage: 1%
- Load tolerance: 100%
- Time to leave: 0

Fixer S1 was tested considering link failure scenarios and node failure scenarios separately to confirm that it may be less efficient when dealing with node failures (where micro-loops can occur). Fixer S2 was only tested considering node failure scenarios, since it uses the same strategy as fixer S1 when dealing with link failures. All tests used a stopping criteria of 1000000 cost changes.

4.1.3.1 S1 Test 1: Link weight 1; Node weight 0

Table 4.13 - S1 Test 1: optimization statistics

Total execution time	0:04:09
Total number of iterations	60
Total number of solutions tested	994024
Iterations per second	0,2
Solutions per iteration	16567,1

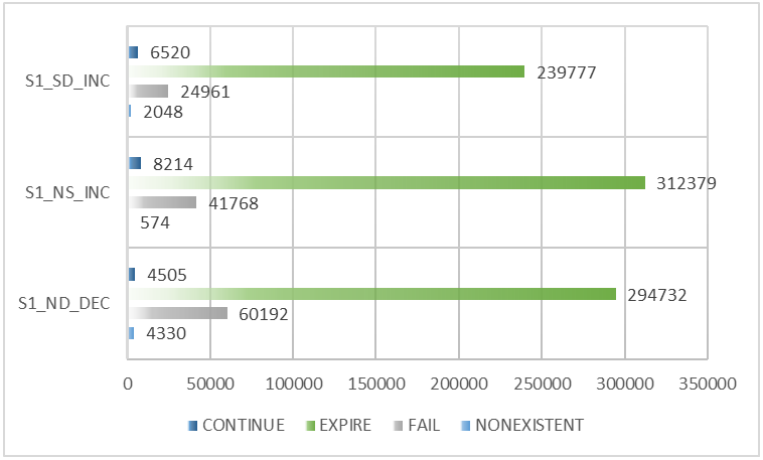

Figure 4.24 - S1 Test 1: change report

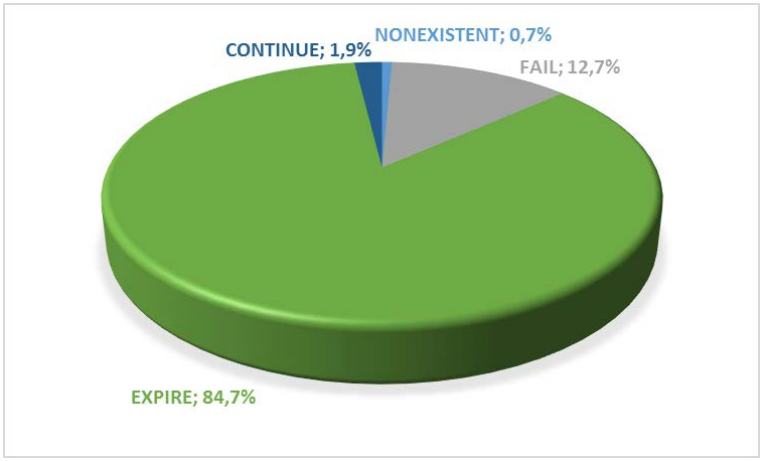

Figure 4.25 - S1 Test 1: change results

Since it is impossible to achieve 100% LFA coverage in Abilene network, it is no surprising that the fixer did not succeed in protecting service in full. However, the large number of expired solutions suggests we may be close to some kind of upper bound of served bandwidth. Next, we test this fixer when considering node failures. By analyzing the bar chart in Figure 4.24, we can conclude that the increase of the distance between the neighbor and the source (S1_NS_INC) is more likely to protect service. This has some logic to it, since the easiest and most obvious way of making a neighbor loop-free is to increase the cost of the directed link(s) connecting that neighbor to the source, so that it chooses other routers to reach the destination.

4.1.3.2 S1 Test 2: Link weight 0; Node weight 1

Table 4.14 - S1 Test 2: optimization statistics

Total execution time	0:01:28
Total number of iterations	1437
Total number of solutions tested	990850

Iterations per second	16,3
Solutions per iteration	689,5

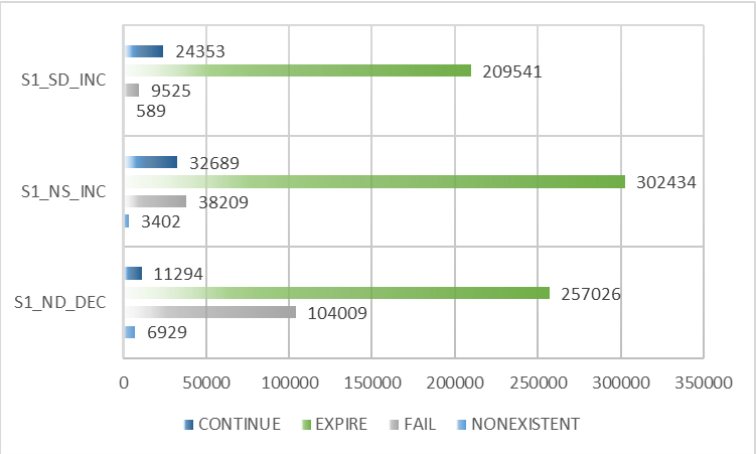

Figure 4.26 - S1 Test 2: change report

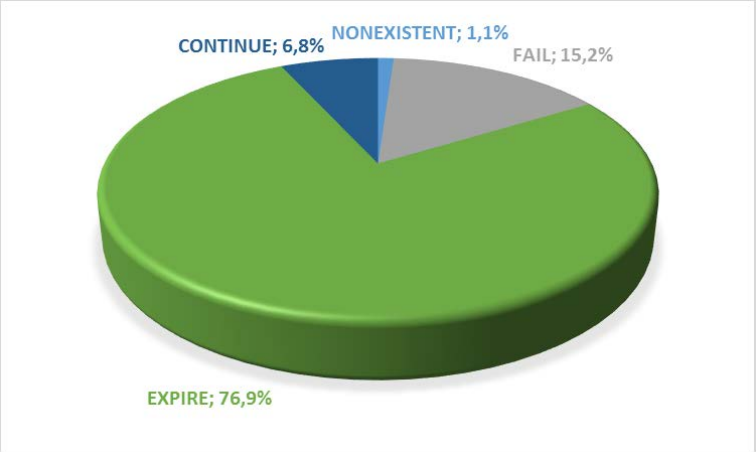

Figure 4.27 - S1 Test 2:change results

By considering node failures, some solutions might contain micro-loops. Figure 4.27 shows that 15.2% of all child solutions had worse performance parameters than their parents. This percentage is slightly higher than the 12.7% registered when running link failure scenarios (where micro-loops do not occur), which suggests that some of the changes performed by this fixer could have enabled link-protecting loop-free alternates that originated some micro-loop due to a node failure. Fixer S2 avoids changes that offer any risk of creating micro-loops.

4.1.3.3 S2 Test 1: Link weight 0; Node weight 1

Table 4.15 - S2 Test 1: optimization statistics

Total execution time	0:00:43
Total number of iterations	767

Total number of solutions tested	478459
Iterations per second	17,8
Solutions per iteration	623,8

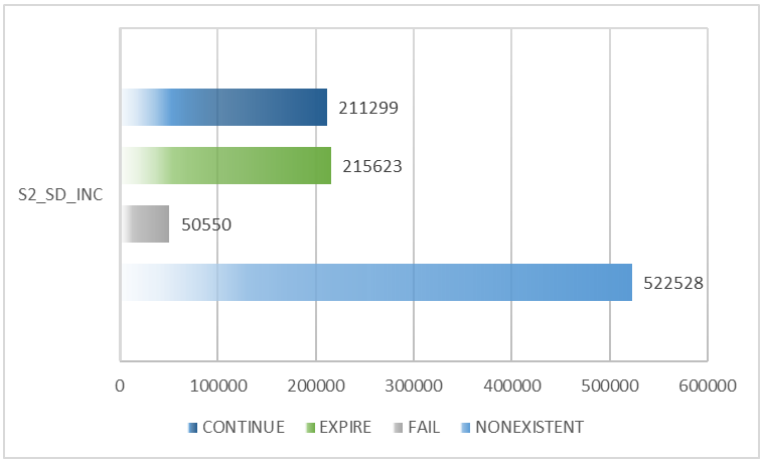

Figure 4.28 - S2 Test 1: change report

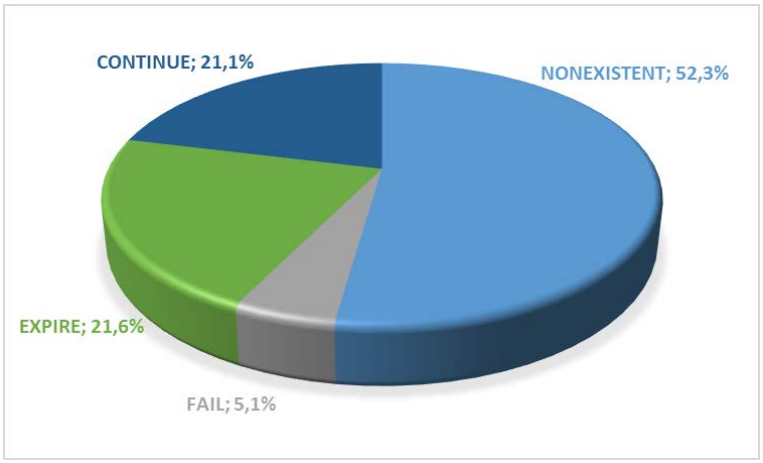

Figure 4.29 - S2 Test 1: change results

When considering node failures, S2 tries to enable downstream paths to avoid any micro-loops. As expected, the percentage of changes originating worse solutions is reduced to 5.1% using this fixer (Figure 4.29). While for previous test using fixer S1 only 6.8% of the changes were able to improve service, fixer S2 improved service with 21.1% of the changes. The relatively high number of nonexistent changes (52.3%) may very well be related with the restrictive nature of downstream paths.

4.1.4 Conclusion of preliminary tests

The preliminary tests allowed us to quickly define which fixers and which configurations might improve the efficiency of future optimization runs, which will increase the probability of finding better solutions in shorter runtimes.

Fixer M1 proved to be very efficient in eliminating micro-loops presenting success rates above 30%. Fixer O1 showed much better results than fixers O2, O3 and O4, having a success rate of 0.7% for the considered traffic. Fixer S2 is more efficient than fixer S1 when protecting service from node failures. The service fixers did not succeed on fully protecting service due to the inherent topology constraints.

We have decided to use costs between 1 and 100 since it gives enough room for cost changes. The time to leave is set to zero, because the strategies used by fixers M1, O1 and S2 pre-compute the magnitude of each change.

An optimization run is expected to eliminate micro-loops quickly. The time taken to eliminate the overloads should depend on the congestion levels of the network. The optimization is then expected to spend most of its time trying to improve service.

4.2 Optimization results

In this section, longer optimization runs will be performed using all fixers combined to try to find solutions that potentiate the use of LFAs, by protecting service without causing any micro-loops or overloads.

Twelve different test cases are considered. They combine different demand values with different failure scenarios and configurations in order to achieve different results and conclusions. All these cases are presented in Table 4.16.

Table 4.16 - Optimization test cases

Case	Demand Values Type	Demand Percentage (%)	Node Failures Weight	Memorize Costs?	Maximum Load (%)
A	1	1	0		100.0
B	1	1	1		100.0
C	2	1	0		100.0
D	2	1	1		100.0
E	1	110	0		100.0
F	1	110	1		100.0
G	2	110	0		100.0
H	2	110	1		100.0
I	1	110	1	X	100.0
J	2	110	1	X	100.0
K	1	110	1		89.6
L	2	110	1		90.5

Both demand values type 1 and 2 are listed in Appendix B. Table 4.17 shows the parameters for the default solution using both traffic matrices with a demand percentage of 110% and the same weight for link and node failures. The type 2 of demand values produces a slightly more congested network than type 1.

Table 4.17 - Default solution parameters

Demand Values Type	Def. Scenario Max. Load (%)	Def. Scenario Avg. Load (%)	Micro-loop Ratio (%)	Overload Ratio (%)	Served Bandwidth (%)
1	84.6	24.7	0.0	31.7	95.3
2	85.5	25.0	0.0	31.7	95.1

Cases A to D decrease the demand values so that it is impossible to create any invalid solutions or overloaded scenarios. Cases A and C only try to protect traffic, without worrying about the formation of micro-loops and overloads, so we hope to achieve good levels of LFA protection for both these cases. Cases B, D, F and cases H to L consider node failures so they must deal with possible micro-loops. Cases E to L use congested networks to increase the chance for repair traffic to overload links. Cases I and J use the *BoundedCostsFactory* (*i.e.*, they memorize generated costs, so that they are not re-evaluated). Case K and case L reduce the number of valid solutions by limiting the acceptable maximum load for default scenarios.

These different cases aim to test the overall quality of the optimization algorithm used against different traffic patterns and to evaluate the influence that LFA might have in the network performance. The expectation is that higher LFA protection levels may harm the overall performance of the network during convergence time. The same base configuration is used for all cases (as determined in the preliminary tests):

- Costs range: 1-100;
- Time to leave: 0;
- Fixers: M1, O1 and S2.
- Link failures weight: 1;
- Stopping criteria: 15 minutes (900 seconds).

Table 4.18 resumes the optimization results obtained for each case. The last four columns show some of the best parameters found. The columns for the micro-loop and overload ratios are omitted because the optimization was able to eliminate all micro-loops and overloads. Note that only the served bandwidth parameter is common to all best costs. The other three parameters, since they were not considered in the objective function, belong to the first best solution found and are purely indicative.

Table 4.18 - Optimization results

Case	No. of Iterations	No. of Solutions	No. of Costs found	Default Scenario Max. Load (%)	Link Failure Coverage (%)	Node Failure Coverage (%)	Served Bandwidth (%)
A	89	3538539	1406	-	62.1	50.8	93.0
B	29	2796716	420	-	63.6	54.5	96.7
C	175	3928349	48	-	62.1	49.2	93.6
D	22	2824889	459	-	64.4	57.6	96.0
E	136	3466546	4761	88	63.6	56.8	90.7
F	72	2526625	1379	86	53.8	50.0	87.0
G	141	3698518	9424	89	62.9	53.8	89.9
H	66	2715448	454	89	59.1	55.3	83.9
I	61	1782194	9304	88	57.6	50.8	87.0
J	54	1761071	1160	91	56.1	53.0	83.6
K	256	2545979	1650	86	56.8	53.4	87.1
L	237	2643530	2100	87	57.6	47.7	83.6

The analysis of each case is presented separately in the following subsections and contains, for each case, one or two line charts showing the evolution of each parameter throughout the optimization time. The x-axis is in milliseconds, but parameters are registered with nanosecond precision.

4.2.1 Case A

Case A uses very low demand values with respect to link capacities, so it becomes impossible to create invalid solutions and overloads. This case only considers link failure scenarios, which assures that micro-loops do not occur as well. Then, the optimization algorithm will only use the service fixer to try to increase the served bandwidth.

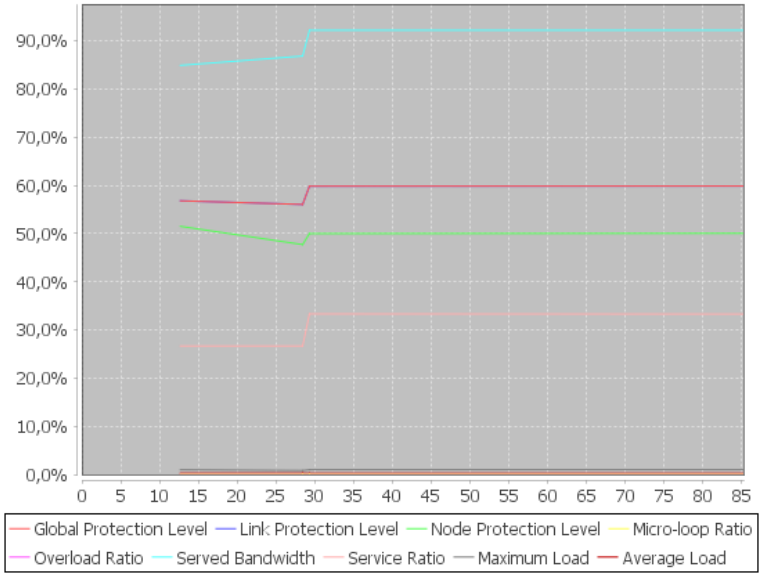

Figure 4.30 - Case A: early stages of the optimization

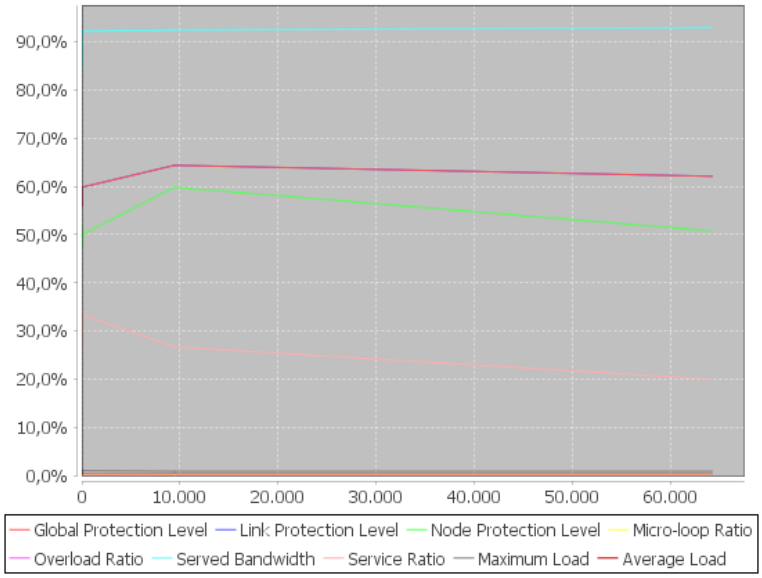

Figure 4.31 - Case A: optimization progress

The optimization algorithm was able to improve the served bandwidth. At the very beginning of the optimization, approximately 85% of total demand bandwidth was being served (Figure 4.30). After just 28 milliseconds the served bandwidth reached about 92% as consequence of increasing the Link Protection Level (which matches the Global Protection Level in this case) from about approximately 57% to 60%. After this period, the served bandwidth increased at a very slow rate. Since the nature of the network imposes a LFA coverage limit, it should become harder to improve service protection at some point.

At approximately 1 minute and 5 seconds of the optimization time, there is a slight decrease of the Link Protection Level (Figure 4.31). This can be easily explained by, for instance, disabling two backups that were repairing two low bandwidth flows and enabling other backup to repair a higher bandwidth flow. The decrease of the service ratio from about 27% to 20% supports this observation. The objective function only considers the served bandwidth parameter, which gives priority to protecting flows with higher bandwidth values rather than the demands themselves. Note that the ECMP also plays an important part on this, since we may be protecting only parts of a demand. It is then completely possible for the served bandwidth to increase while the service ratio decreases.

The maximum link coverage of approximately 65% (registered at second 10 of the optimization) may not be the maximum achievable level for the Abilene topology. Although we can still expect some correlation between the served bandwidth and the protection levels, we are not using demands for all source destination pairs (and with the same exact demand values). Some primary next-hops may not be used to forward traffic, so it becomes irrelevant if they have a backup or not. This makes it possible to register a served bandwidth increase while the LFA protection decreases.

4.2.2 Case B

Case B uses the same demand values as case A, but also considers node failure scenarios. It is still impossible for invalid solutions or overloads to happen, but the optimization algorithm may need to use the micro-loop fixer to eliminate some micro-loops before protecting service and protecting service must avoid creating micro-loops.

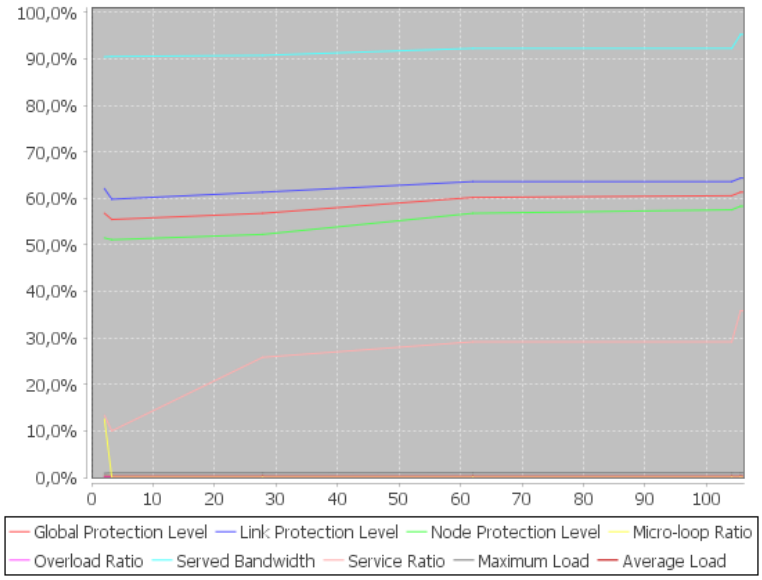

Figure 4.32 - Case B: early stages of the optimization

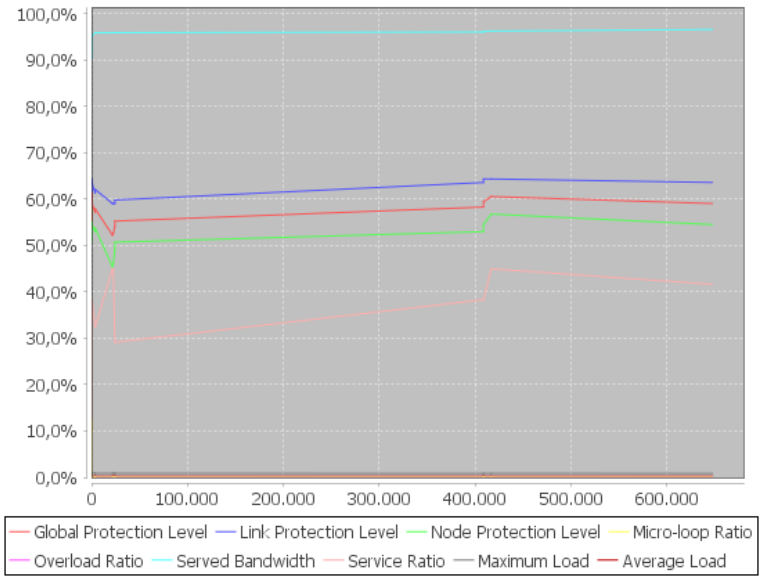

Figure 4.33 - Case B: optimization progress

'Luckily', the first best solution had a micro-loop ratio of 12.5% (Figure 4.32), which corresponds to exactly one node failure scenario with at least one micro-loop (using formula (5), (0/15 + 1/4)/(1 + 1)). This micro-loop was rapidly eliminated by decreasing the Link Protection Level from 62% to 60%. The optimization continues finding solutions that increase the served bandwidth while avoiding micro-loops.

Though we could have been expecting less served bandwidth comparing with previous case, we must remember we are considering 4 extra node failure scenarios with the same weight as 15 link failure scenarios, so these cases aren't really comparable. The Node Protection Level, however, is expected to increase, since we need to protect service from node failures as well. The chart lines for the Link Protection Level and Node Protection Level present a very similar behavior which can be a consequence of using the same weight for both failure types.

The final best solution was found approximately eleven minutes after the optimization started (Figure 4.33). Processing all scenarios and calculating performance parameters should take a bit longer than previous case, since we consider 4 more failure scenarios. Also, we should be generating much more cost changes which delays the average execution time of a single iteration. Table 4.18 shows that case B ran less iterations than case A, which decreases the probability of finding better solutions in shorter periods of time.

4.2.3 Case C

Case C is equivalent to case A, but it uses different demand values.

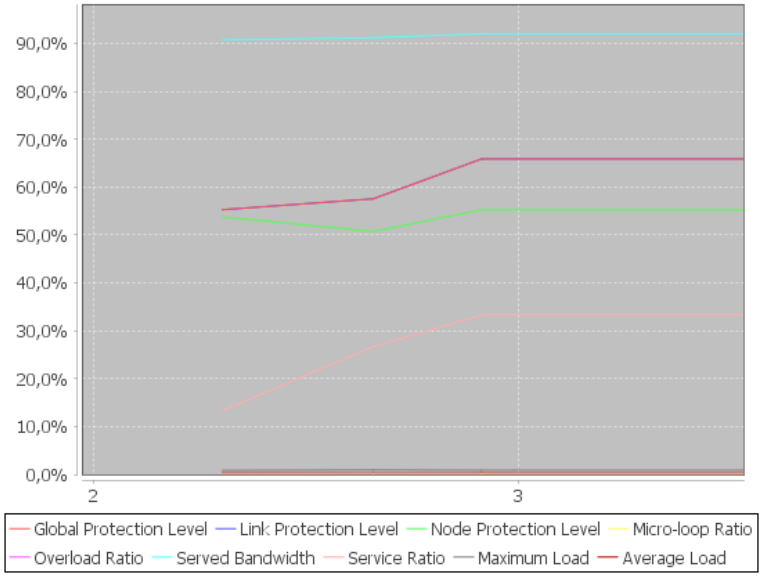

Figure 4.34 - Case C: early stages of the optimization

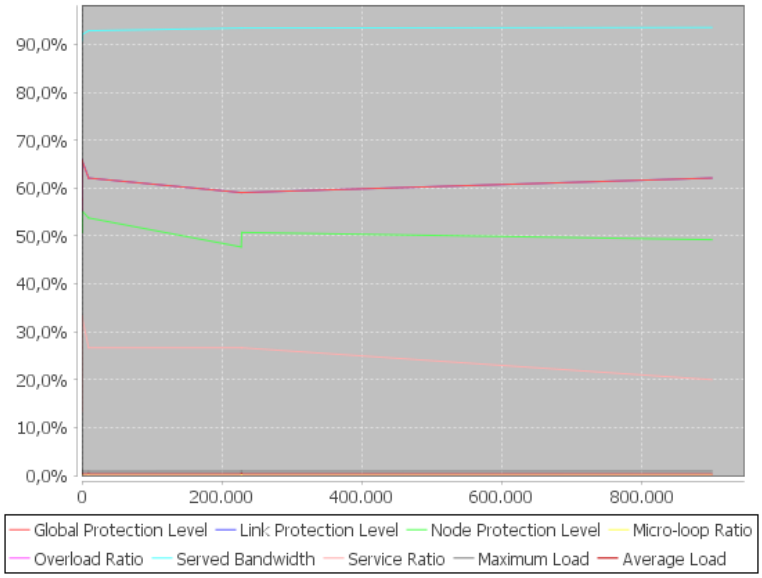

Figure 4.35 - Case C: optimization progress

Case C and case A present similar results. Though using different demand values, those value differences are too small to make much difference.

4.2.4 Case D

Case D is equivalent to case B, but it uses different demand values.

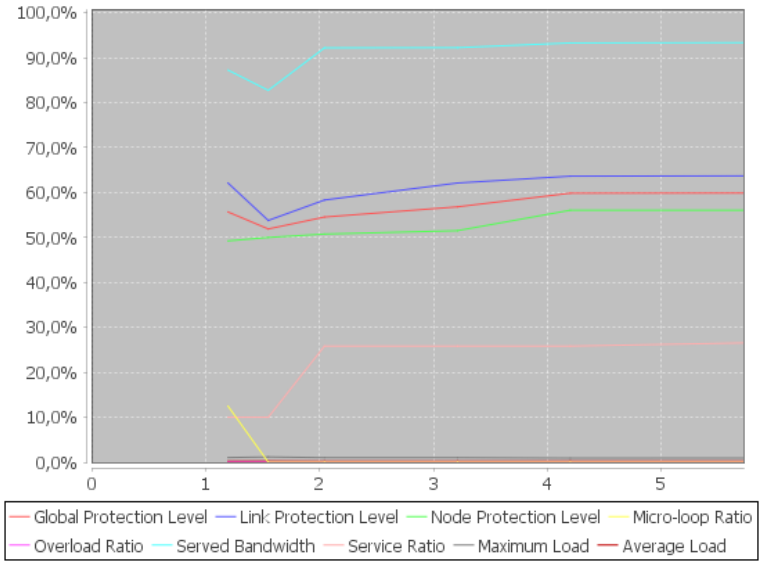

Figure 4.36 - Case D: early stages of the optimization

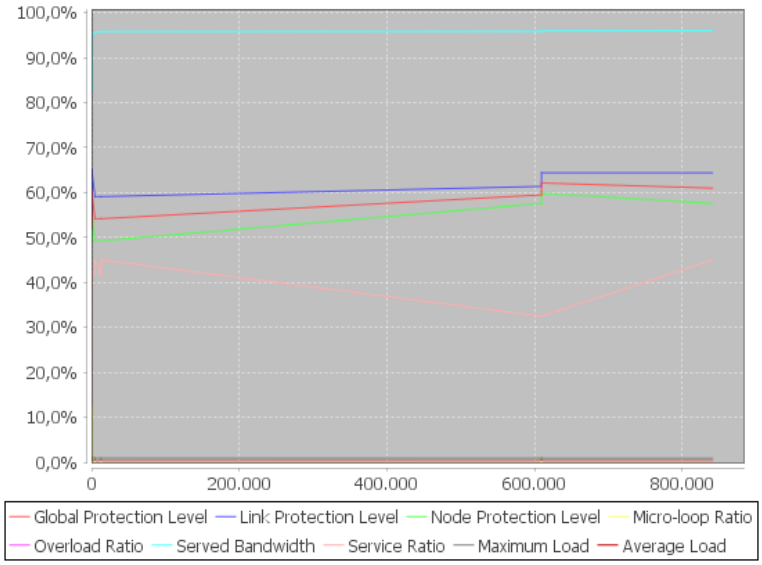

Figure 4.37 - Case D: optimization progress

Like in the previous subsection, case D and case B also present similar results since the demand value differences are too small to make much difference.

4.2.5 Case E

Case E uses 110% of the demand values of type 1 to simulate a congested network and increase the probability for overloaded scenarios to occur. This case only considers link failure scenarios. The optimization will use the overload fixer to eliminate any existing overloads before using the service fixer to increase the served bandwidth. Improving service protection must not create any overloads.

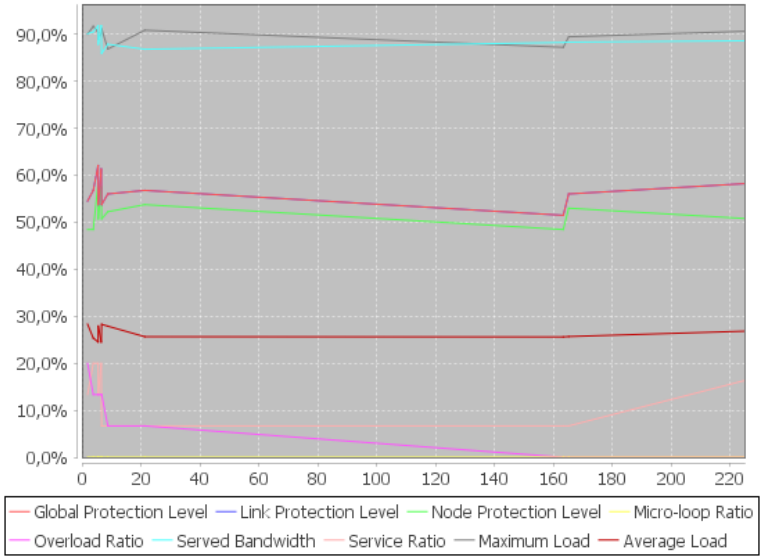

Figure 4.38 - Case E: early stages of the optimization

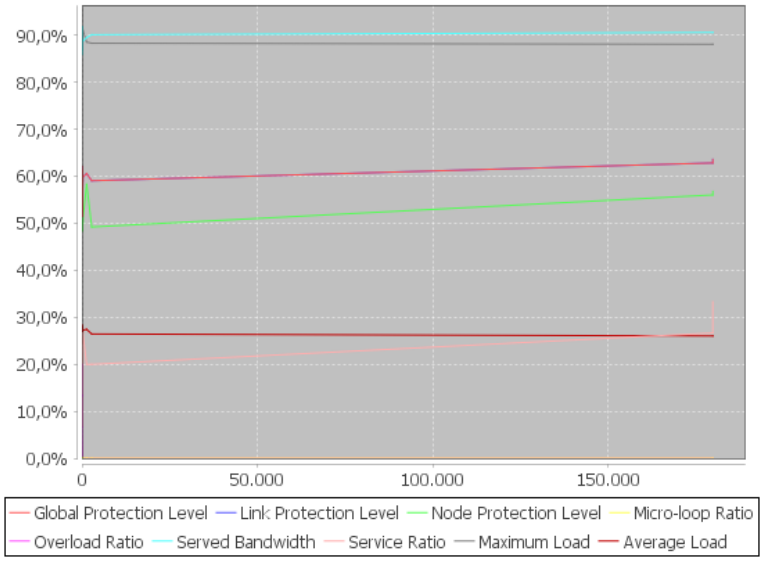

Figure 4.39 - Case E: optimization progress

Figure 4.38 shows that case E had to eliminate the overloads from three link failure scenarios at the very beginning of the optimization ($20\% \times 15 = 3$) by decreasing the Link Protection Level (and Served Bandwidth). Note that the algorithm took a bit longer to fix the last overloaded scenario at the cost of decreasing the Link Protection Level from about 57% to 51%. This clearly illustrates the strategy used by the overload fixer and the difficulty to eliminate overloads in a congested network.

After eliminating all the overloads, the service fixer was able to find other less congested repair paths and increase the served bandwidth to 90.7% (Figure 4.39). A curious observation is that it was able to recover the Link Protection Level to even higher values than for case A, but the served

bandwidth is a bit lower, so it still had to sacrifice some service protection to avoid the overloads. As explained before for case A, the LFA protection levels and the served bandwidth are not strongly related.

4.2.6 Case F

Case F uses the same demand values as case E, but it also considers node failure scenarios. The micro-loop fixer may occasionally be used to eliminate micro-loops. Once again, service protection must avoid causing micro-loops and overloads.

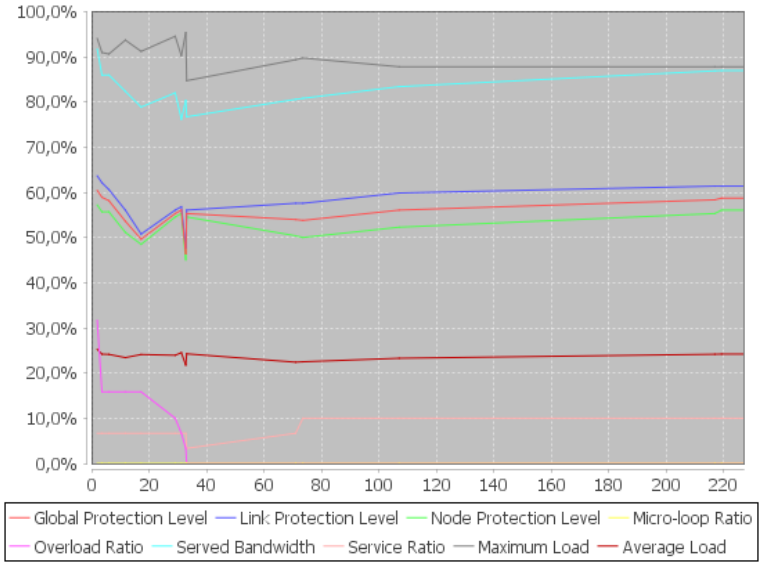

Figure 4.40 - Case F: early stages of the optimization

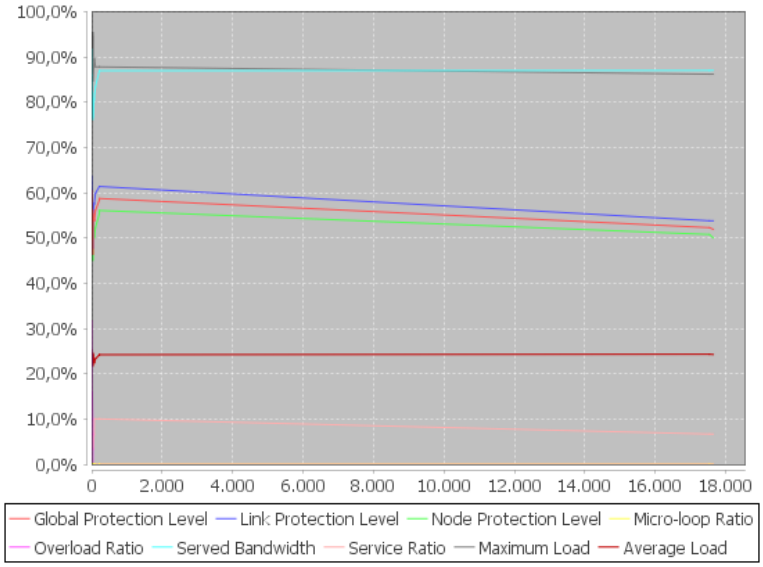

Figure 4.41 - Case F: optimization progress

Case F clearly shows worse results than the previous case. Node failures usually mean the failure of several links. This can lead to the invocation of more repair paths in a network with less available resources. Then, links can become overloaded much easier than when dealing only with link failures. The micro-loop prevention, as we have seen in case B, is much less restrictive, since micro-loops can only occur on node failure scenarios when meeting very special conditions (when traffic is repaired through more than one link-protecting LFA).

When comparing with the best parameters found in test case B, the served bandwidth has decreased from 96.7% to 87% (Table 4.18). Figure 4.41 also shows a decrease of the Global Protection Level from almost 60% to about 52% followed by a small decrease of the service ratio.

4.2.7 Case G

Case G is equivalent to case E, but it uses the type 2 of demand values. When comparing to case E, it is expected a more conservative approach on enabling loop-free alternates, since this case uses a slightly more congested network, which increases the probability for overloads to occur.

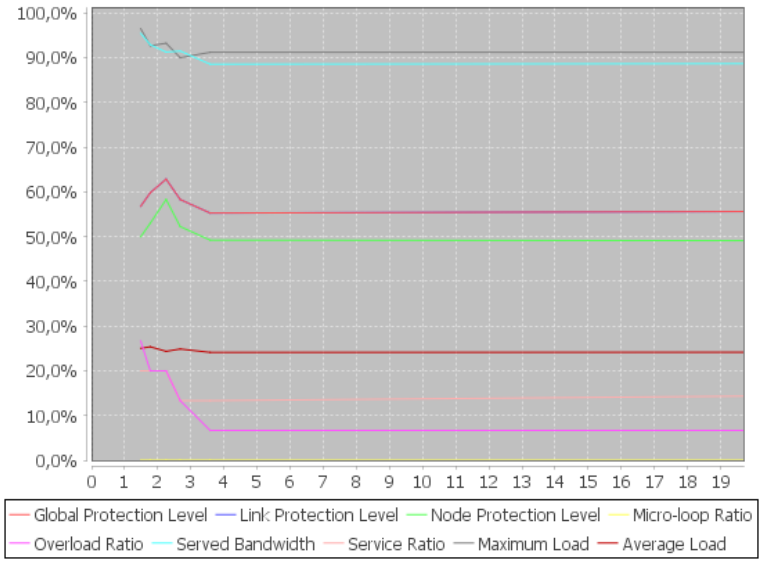

Figure 4.42 - Case G: early stage of the optimization

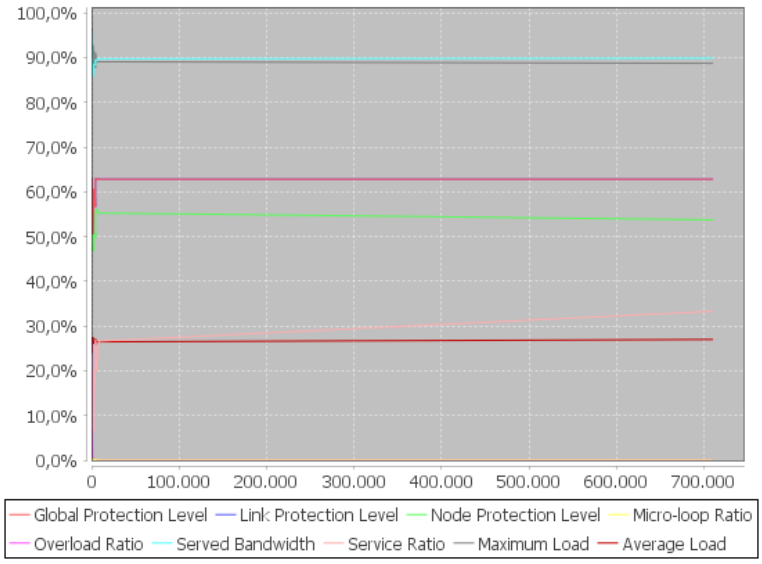

Figure 4.43 - Case G: optimization progress

The served bandwidth for this case is a bit lower than for case E. Table 4.18 shows that, for case E, the served bandwidth is 90.7%, while for case G it is 89.9%. This difference becomes more accentuated if we consider node failure scenarios as will be shown in the next subsection.

4.2.8 Case H

Case H is equivalent to case F, but it uses the type 2 of demand values.

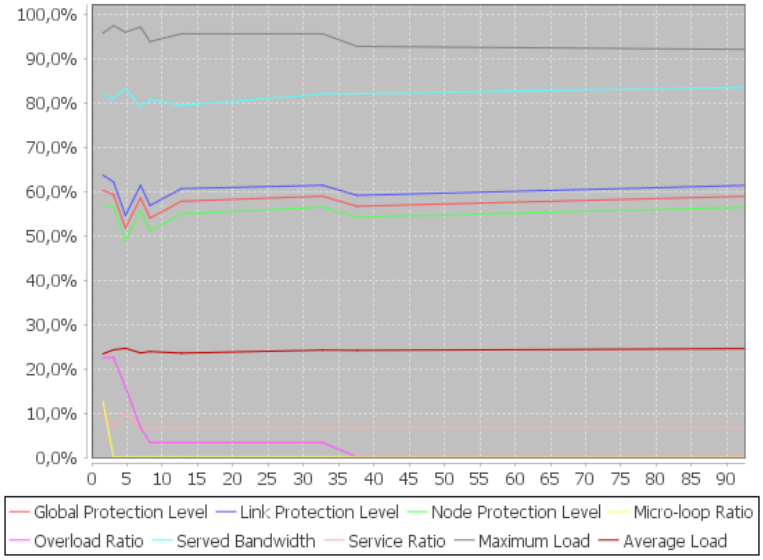

Figure 4.44 - Case H: early stages of the optimization

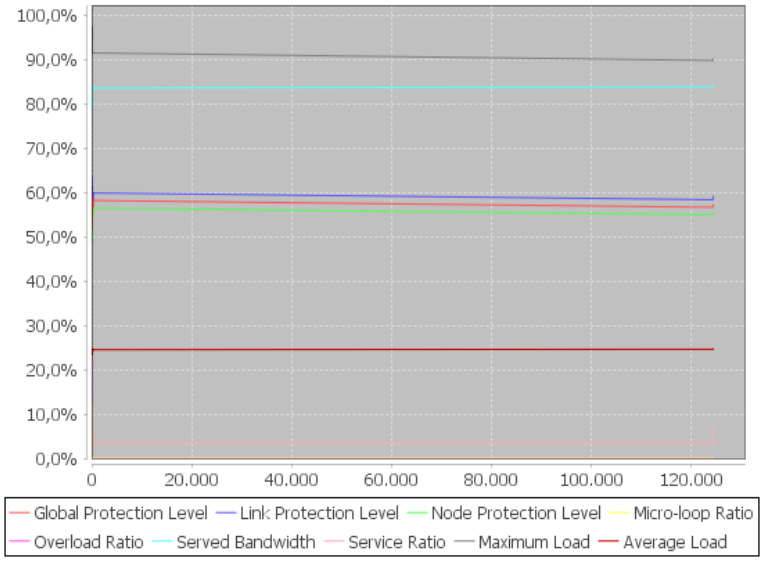

Figure 4.45 - Case H: optimization progress

As we have already seen from comparing case F with case E, considering node failures will make overloads more probable, which limits even more the amount of service that is possible to protect. When comparing case H to case F, we confirm that a slightly more congested network will decrease the served bandwidth, especially when considering node failure scenarios. Table 4.18 shows that, for case F, the served bandwidth is 87.0%, while for case H, the served bandwidth is only 83.9%. Different traffic demands may produce different solutions with different paths to guarantee the maximum load constraint. Different links can become overloaded, with different overload amounts, leading to optimization runs where different LFAs are disabled to remove overloads and others may be enabled to improve service.

4.2.9 Case I

The difference between case I and case F is that case I uses the *BoundedCostsFactory* to guarantee the return of different combinations of costs.

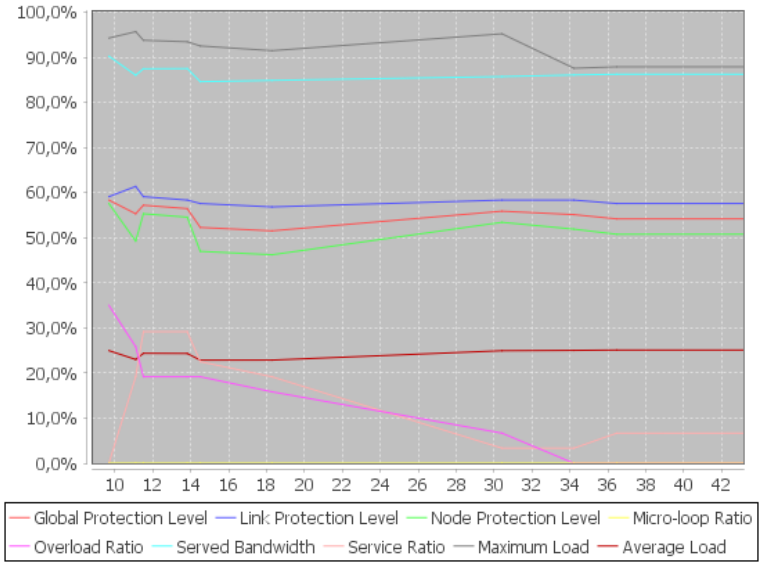

Figure 4.46 - Case I: early stages of the optimization

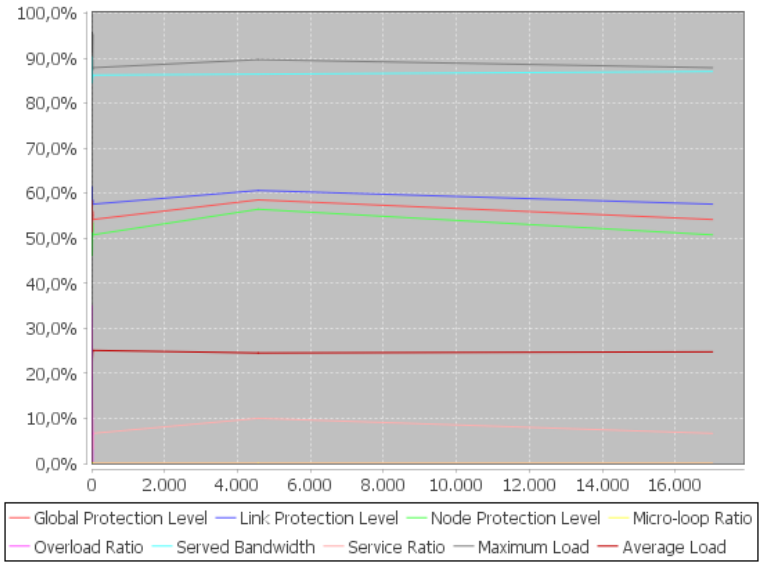

Figure 4.47 - Case I: optimization progress

Though showing greater LFA coverage, the served bandwidth is exactly the same as for case F. Both cases present optimized solutions with a served bandwidth of 87% (Table 4.18). This suggests that, for the considered topology and costs interval, always generating different sets of costs does not have much influence on the obtained results.

4.2.10 Case J

The difference between case J and case H is that case J uses the *BoundedCostsFactory* to return different combinations of costs.

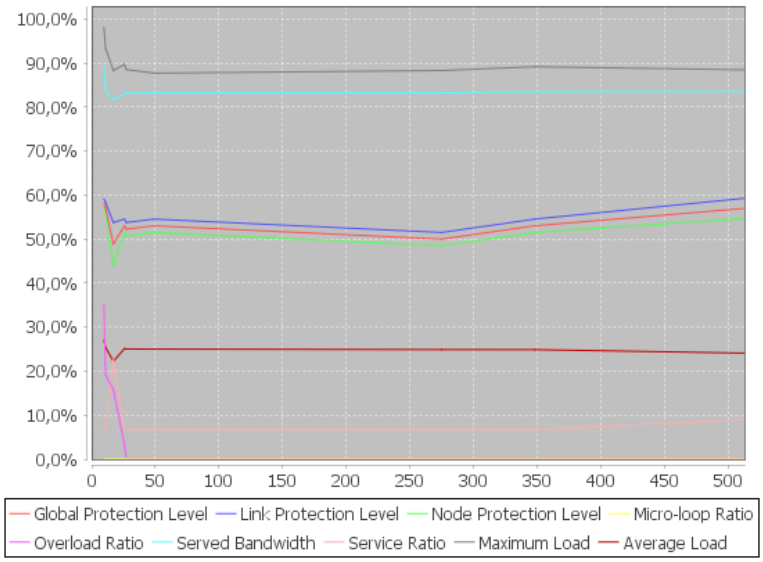

Figure 4.48 - Case J: optimization early stages

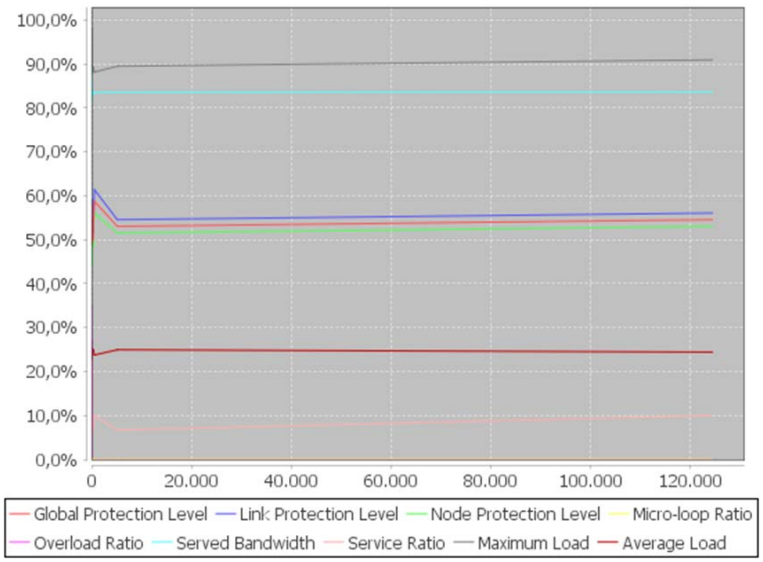

Figure 4.49 - Case J: optimization progress

This case produced slightly worse results than case H. Table 4.18 shows that case J protects 83.6% of the service, while case H protected 83.9%. Note that, conceptually, this optimization setting should produce better results, if the same number of iterations is run. Nevertheless, since we have given the same runtime limit for all optimizations, observe that this case has executed only 54 iterations, while case H has executed 66, and this is the reason for the slightly worse results. The

BoundedCostsFactory is slower in generating new costs than the normal *CostsFactory* (besides the additional execution overhead associated with the synchronization between different threads) and it will decrease the overall speed of the optimization.

4.2.11 Case K

The difference between cases K and F is that case K imposes a more constraining maximum load value to the default scenario.

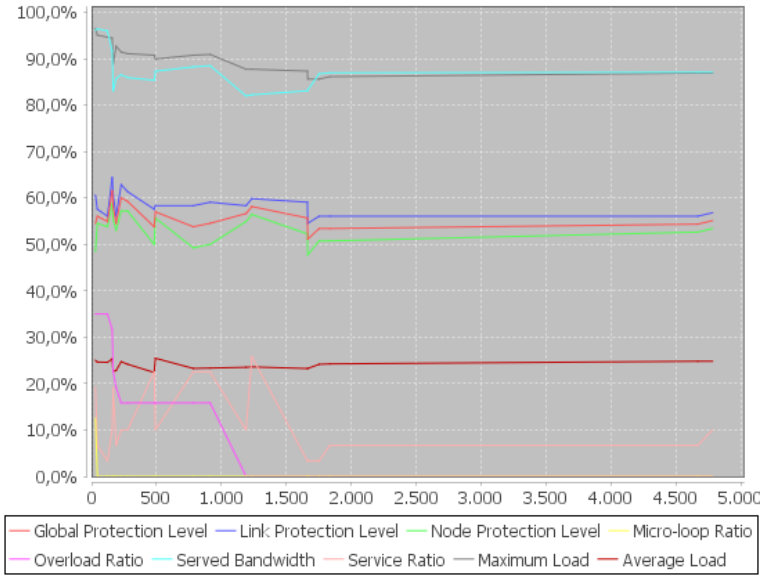

Figure 4.50 - Case K: optimization progress

Though it took more time to eliminate overloads, probably struggling on finding valid solutions, the heuristic algorithm was able to produce even better results than case F (Figure 4.50). This happened mostly because case K ran 256 iterations against the 72 of case F (Table 4.18). The iterations of case K have finished sooner because of the greater number of invalid solutions, in which only the default scenario is processed. By running much more iterations, it ended up increasing its chances on finding better solutions.

4.2.12 Case L

Like in the previous case, the difference between cases L and H is that case L imposes a more constraining maximum load value to the default scenario.

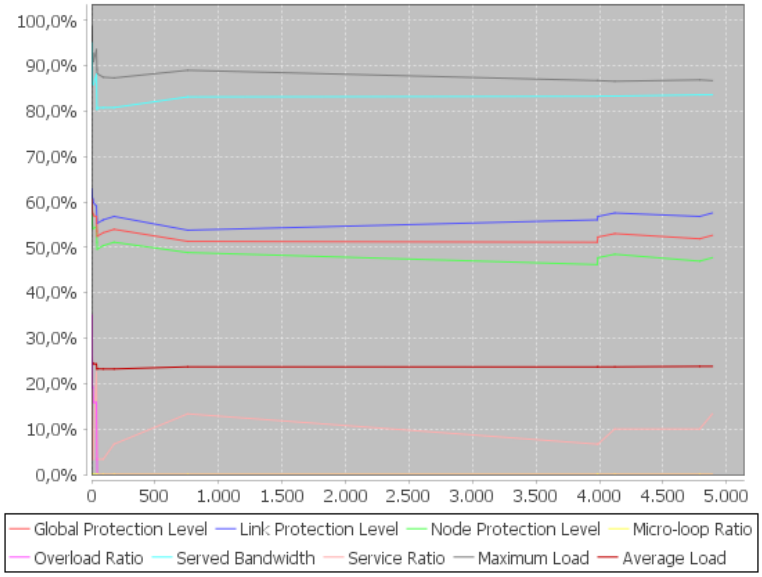

Figure 4.51 - Case L: optimization progress

Despite running much more iterations than case H, case L did not find a better solution. It could have been bad luck, but the fact is that this case uses a slightly more congested network. It may already be difficult to find solutions without any overloads and limiting the number of valid solutions will make it even more difficult. The maximum load constraint for this case should then be relaxed.

5 Conclusions

This chapter summarizes this work and highlights the most important conclusions about employing LFAs to improve service uptime of ISP networks. It discusses the results obtained from trying to optimize the overall performance of Abilene network considering different traffic patterns. Finally, it presents some theoretical ideas about more elaborate methods that might improve the heuristic algorithm efficiency, and consequently, the quality of the solutions.

5.1 Summary

The time taken for a network to recover from a failure can degrade service that is sensible to delay and packet loss. LFA is a simple IPFRR mechanism that provides alternative paths for traffic affected by a single failure as soon as that failure is detected. In this way, when detecting failures at the physical layer, this mechanism is able to reduce service disruption to no more than a hundred milliseconds, which is acceptable for most sensible traffic.

By changing link costs, it is possible to increase the LFA coverage of a network. However, real networks topologies impose a maximum level of protection that can be achieved. The node failure coverage, in particular, is usually below 60%. If a link-protecting LFA is used to reroute traffic affected by a node failure, there is a possibility of creating micro-loops. Also, repair traffic can cause links to become overloaded. Both these situations can severely deteriorate the network performance during convergence time.

Using the developed application in test scenarios based on the Abilene network, it was possible to find costs that have increased service protection without creating micro-loops and overloads. The implemented heuristic proved to be successful in finding better solutions. This heuristic consists on continuously applying strategic cost changes, through the so-called fixers, in current solutions, to try to produce better solutions. The optimization process comprehends three steps in sequence:

1. Eliminate micro-loops;
2. Eliminate overloads;
3. Increase the amount of served bandwidth.

The cost changes implemented by the fixers that try to eliminate micro-loops and overloads aimed to disable the responsible repair paths by removing the corresponding backup next-hops, while the cost changes implemented by the fixer that tries to increase the served bandwidth aimed to enable other possible repair paths.

5.2 Results

Before conducting any long optimization run for the Abilene topology, several short runs were executed to determine which strategies and configurations were suited for the optimization. The

chosen cost interval, for instance, has considerable influence in the algorithm efficiency. Smaller cost intervals, besides decreasing the number of different possible solutions, also increase the probability for a cost change to be rejected, especially if considering large topologies. This is related with two factors:

- The smaller the cost interval, the higher the probability for a randomly generated cost to be close to cost limits.
- The applied cost change techniques try to change shortest distances by increasing or decreasing the IGP cost of a single router interface by a minimum calculated value. The result of this operation can easily exceed cost limits using smaller cost intervals.

Other configurations that seemed a good idea at first, turned out to be less useful. Memorizing costs can be useful when considering a small cost interval and topologies with a small number of links but, usually, the number of possible solutions is so large that the probability of repeating costs becomes infinitively small. This does not compensate the extra execution overhead for synchronizing access to costs factory memory and running the hash function to check if the generated costs were already used or not.

Having decided the optimization setup, we have executed long runs for several test cases. These test cases considered different traffic demand values and different failure scenarios to produce different results. By analyzing and comparing those results, the following conclusions were drawn:

- When 100% LFA protection is impossible to achieve, different traffic demand values may benefit from different sets of backup next-hops. Consequently, depending on the number and value of demands, maximizing the served bandwidth does not necessarily maximize the LFA coverage. This is because flows with higher bandwidth can gain priority over flows with lower bandwidth. However, most of the times, increasing/decreasing the LFA coverage does increase/decrease the served bandwidth.
- Micro-loops do not happen often and are easy to avoid and to eliminate. Eliminating a micro-loop usually requires disabling a single backup next-hop.
- The more congested the network is, the more difficult it becomes to eliminate and avoid possible overloads. It usually requires disabling several backup next-hops.
- Node failure scenarios are much more problematic than link failures, since they involve more repair paths which increases the chance for overloads to occur.
- Increasing the chance for overloads to occur due to repair traffic (when networks are more congested) will decrease the maximum amount of service that is possible to protect, since it limits the LFA coverage of the network.
- Improving service while avoiding micro-loops is much less restrictive than when avoiding overloads, in terms of LFA protection. In fact, improving service may actually help to avoid micro-loops when it is possible to find a node-protecting LFA.

5.3 Final considerations and future improvements

Note that the developed application also considers connections between routers given by switching nodes (instead of only directed links). Nevertheless, due to time constraints, this type of network setups was not properly tested and, therefore, results for such cases are not included in this dissertation.

Some other important performance parameters could have been considered like the propagation delay. A valid solution, besides maximum load constraints, may also impose a maximum number of hops, since LFA is intended to protect delay sensitive traffic. Of course, this would limit the number of valid solutions, which makes it more difficult to find better solutions.

Although overloads can be difficult to eliminate, especially when considering networks close to congestion, choosing which backup next-hop(s) to eliminate, using the repair flow bandwidth as the criteria, could have provided higher success rates. But the effects of more elaborate strategies are not really predictable, since they usually require more memory and would most certainly impose more execution overhead. A more viable alternative would be to use an overload fixer that also considers the maximum overload parameter. This fixer would fix overloads with more precision, but it might spend more time trying to eliminate them.

The application should give the possibility to use test vectors. Instead of generating random initial solutions, the optimization would start each iteration with a predefined solution, making results more comparable.

For simplicity, a solution is fully processed before calculating the performance parameters. It may be possible to improve the optimization speed if we use a different approach when processing the failure scenarios of a solution. For instance, consider the optimization of a solution without any micro-loops. If a micro-loop is detected when executing a failure scenario of a child solution, this child solution is discarded right away (no need to execute other failure scenarios).

Developing applications for network optimization purposes can be very difficult. It usually requires implementing protocols and heuristics that may consume a lot of the machine resources. It is important to adopt a more defensive programming approach when developing this kind of applications.

Bibliographical References

Atlas, A. and Zinin, A. 2008. RFC 5286 - Basic Specification for IP Fast Reroute: Loop-Free Alternates. [Online] September 2008. http://tools.ietf.org/html/rfc5286.

Cisco Systems, Inc. 2012. Cisco IOS XR Routing Configuration Guide for the Cisco CRS Router, Release 4.2.x. *Cisco.* [Online] Setembro 2012. http://www.cisco.com/c/en/us/td/docs/routers/crs/software/crs_r4-2/routing/configuration/guide/b_routing_cg42crs/b_routing_cg42crs_chapter_0100.html.

Doyle, Jeff. 2007. Fast Reroute without MPLS? *Network World.* [Online] September 24, 2007. http://www.networkworld.com/article/2348436/cisco-subnet/fast-reroute-without-mpls-.html.

Fortz, Bernard, Rexford, Jennifer and Thorup, Mikkel. 2002. Traffic Engineering With Traditional IP Routing Protocols. *Communications Magazine, IEEE.* 2002, Vol. 40, 10, pp. 118-124.

Gjoka, Minas, Ram, Vinayak and Yang, Xiaowei. 2007. Evaluation of IP Fast Reroute Proposals. *Communication Systems Software and Middleware.* 2007, pp. 1-8.

Goyal, M., et al. 2011. Improving Convergence Speed and Scalability in OSPF: A Survey. *Communications Surveys & Tutorials.* 2011, Vol. 14, 2, pp. 443-463.

Internet World Stats. Internet Growth Statistics - the Global Village Online. *Internet World Stats.* [Online] http://www.internetworldstats.com/emarketing.htm.

Markopoulou, Athina, et al. 2008. Characterization of Failures in an Operational IP Backbone Network. *IEEE/ACM Transactions on Networking.* Fevereiro 25, 2008, Vol. 16, 4, pp. 749-762.

Moy, J. 1998. RFC 2328 - OSPF Version 2. [Online] April 1998. http://tools.ietf.org/html/rfc2328.

Neagle, Colin. 2012. 'Insatiable demand' for streaming video puts pressure on providers. *Network World.* [Online] March 15, 2012. http://www.networkworld.com/article/2186861/data-breach/-insatiable-demand--for-streaming-video-puts-pressure-on-providers.html?page=1.

Rétvári, Gábor, et al. 2011. IP Fast ReRoute: Loop Free Alternates Revisited. *INFOCOM, 2011 Proceedings IEEE.* Abril 10-15, 2011, pp. 2948-2956.

Rétvári, Gábor, et al. 2011. Optimizing IGP link costs for improving IP-level resilience. *Desing of Reliable Communications Networks.* 2011, pp. 62-69.

Shand, M. and Bryant, S. 2010. RFC5714 - IP Fast Reroute Framework. [Online] January 2010. http://tools.ietf.org/html/rfc5714.

SNDlib. SNDlib. *SNDlib.* [Online] http://sndlib.zib.de.

Appendix A: The Application

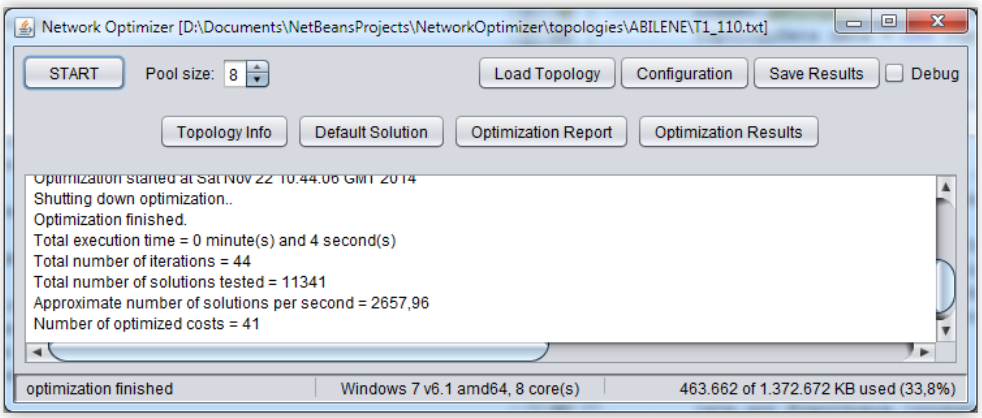

Figure A.1 - Application console window

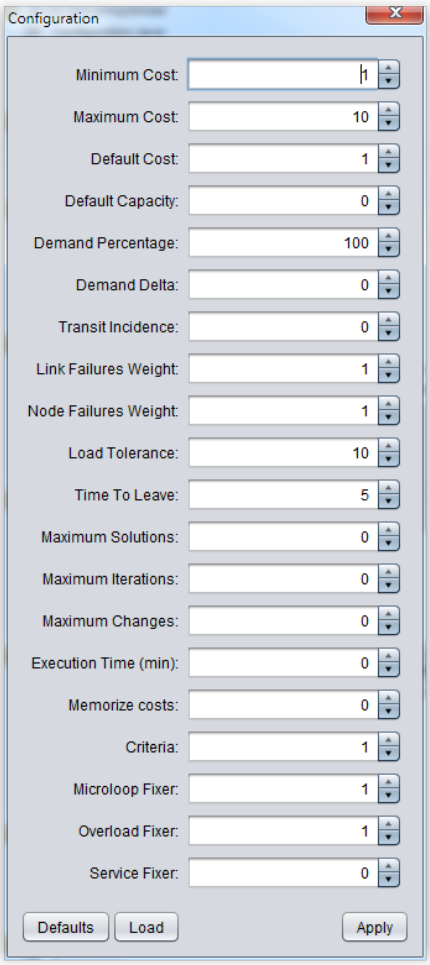

Figure A.2 - Configuration window

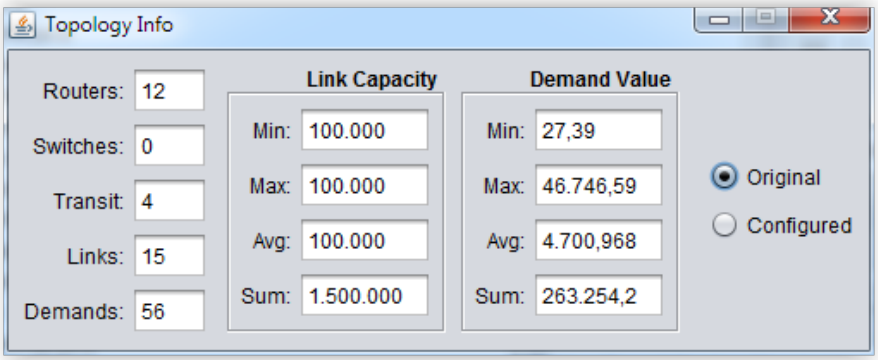

Figure A.3 - Topology information window

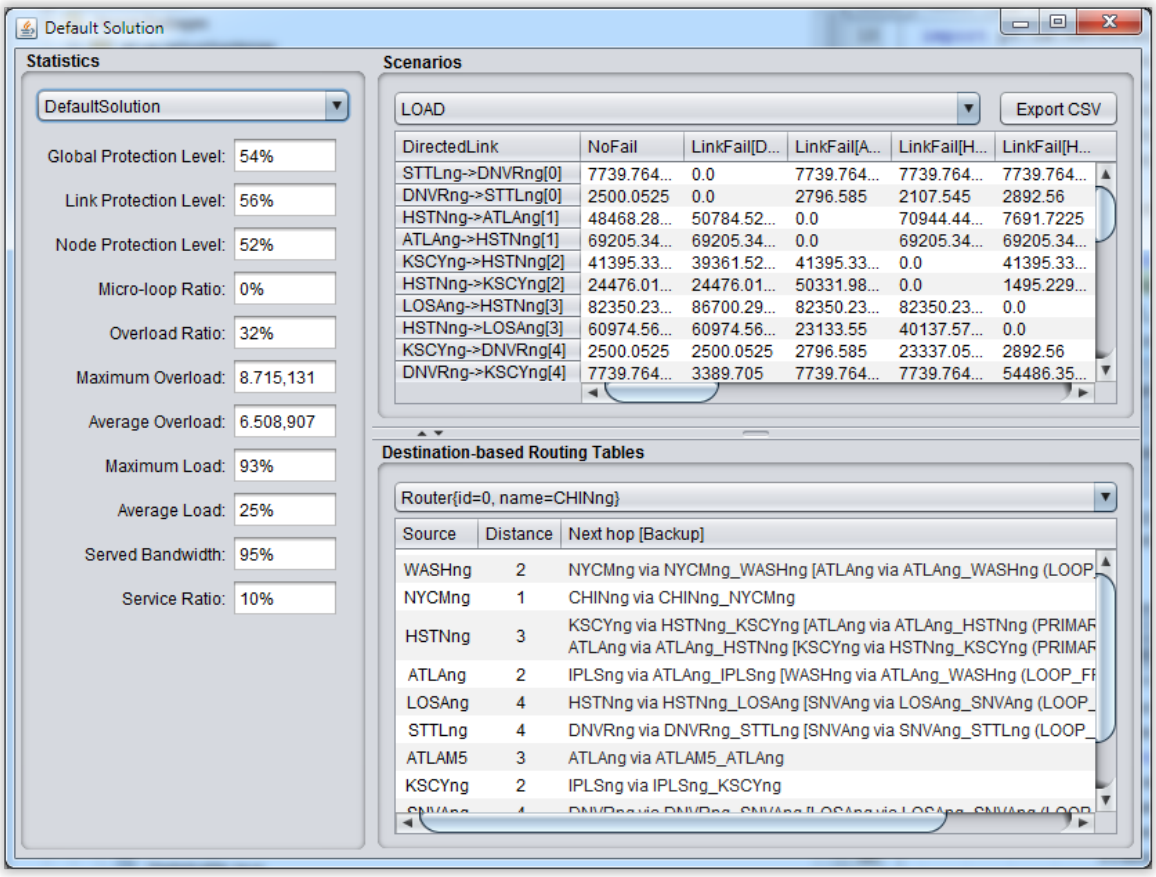

Figure A.4 - Default solution window

Optimization Report

	NONEXIS...	INVALID	FAIL	RETRY	RECOVER	EXPIRE	CONTINUE	SUCCESS
M1_ND_INC	16	0	0	0	0	3	1	12
M1_NS_DEC	21	0	0	0	0	1	2	8
M1_SD_DEC	21	0	0	0	0	0	1	10
O1_ND_INC	14223	1203	8963	0	0	434	2418	1
O1_NS_DEC	18416	1209	459	0	0	2346	4589	6
O1_SD_DEC	17381	620	6089	0	0	574	2357	0
O2_LINEAR	0	0	0	0	0	0	0	0
O3_EXPNTL	0	0	0	0	0	0	0	0
O4_LINEAR	0	0	0	0	0	0	0	0
O4_EXPNTL	0	0	0	0	0	0	0	0
S1_ND_DEC	0	0	0	0	0	0	0	0
S1_NS_INC	0	0	0	0	0	0	0	0
S1_SD_INC	0	0	0	0	0	0	0	0
S2_ND_DEC	115	1	3761	0	0	4236	187	0
S2_NS_INC	0	0	3643	0	0	4207	206	0
S2_SD_INC	979	572	4834	0	0	4034	333	0

Figure A.5 - Optimization report window

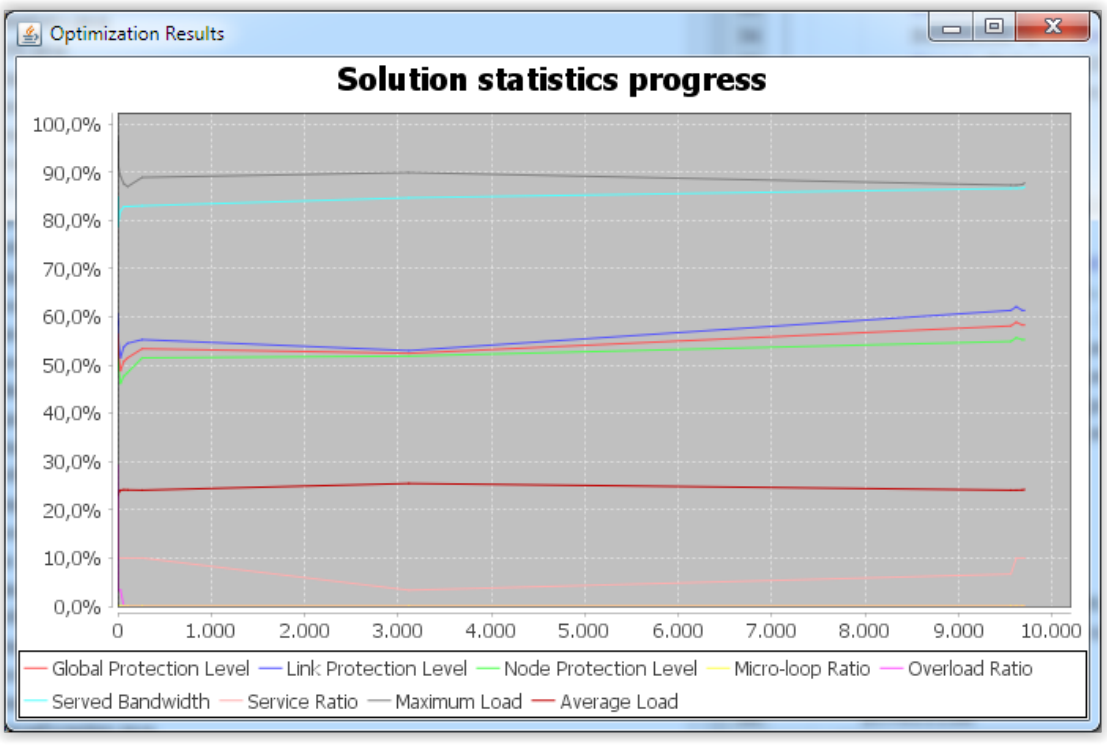

Figure A.6 - Solution parameters window

Appendix B: Abilene Input Files

Original Source (considering only 10% of the demand values):
http://sndlib.zib.de/coredata.download.action?objectName=abilene&format=xml&objectType=network

Table B.1 Demand values statistics

Demand Values Type	Minimum Demand Value	Maximum Demand Value	Average Demand Value	Demand Values Sum
1	24.9	42496.9	4273.6	239322
2	17.878	57033.011	4396.5	246204.936

B.1 Demand Values 1 Input File

15
DNVRng_STTLng STTLng DNVRng 100000,000000 1 1
ATLAng_HSTNng HSTNng ATLAng 100000,000000 1 1
HSTNng_KSCYng KSCYng HSTNng 100000,000000 1 1
HSTNng_LOSAng LOSAng HSTNng 100000,000000 1 1
DNVRng_KSCYng KSCYng DNVRng 100000,000000 1 1
NYCMng_WASHng WASHng NYCMng 100000,000000 1 1
ATLAng_WASHng WASHng ATLAng 100000,000000 1 1
CHINng_NYCMng NYCMng CHINng 100000,000000 1 1
CHINng_IPLSng IPLSng CHINng 100000,000000 1 1
LOSAng_SNVAng SNVAng LOSAng 100000,000000 1 1
DNVRng_SNVAng SNVAng DNVRng 100000,000000 1 1
ATLAM5_ATLAng ATLAng ATLAM5 100000,000000 1 1
SNVAng_STTLng STTLng SNVAng 100000,000000 1 1
IPLSng_KSCYng KSCYng IPLSng 100000,000000 1 1
ATLAng_IPLSng IPLSng ATLAng 100000,000000 1 1
56
CHINng_LOSAng CHINng LOSAng 38599,100000
STTLng_ATLAM5 STTLng ATLAM5 93,200000
NYCMng_ATLAM5 NYCMng ATLAM5 107,600000
STTLng_ATLAng STTLng ATLAng 4133,900000
WASHng_CHINng WASHng CHINng 3459,700000
HSTNng_CHINng HSTNng CHINng 1291,300000
NYCMng_ATLAng NYCMng ATLAng 1799,100000
WASHng_LOSAng WASHng LOSAng 4411,900000
WASHng_HSTNng WASHng HSTNng 2677,900000

ATLAng_LOSAng ATLAng LOSAng 6901,600000
STTLng_WASHng STTLng WASHng 1617,800000
CHINng_ATLAM5 CHINng ATLAM5 277,000000
LOSAng_ATLAM5 LOSAng ATLAM5 133,700000
NYCMng_LOSAng NYCMng LOSAng 3416,700000
CHINng_HSTNng CHINng HSTNng 32967,300000
ATLAM5_NYCMng ATLAM5 NYCMng 114,900000
ATLAM5_LOSAng ATLAM5 LOSAng 101,600000
LOSAng_STTLng LOSAng STTLng 968,400000
LOSAng_WASHng LOSAng WASHng 7119,700000
LOSAng_ATLAng LOSAng ATLAng 4448,400000
ATLAng_ATLAM5 ATLAng ATLAM5 214,600000
HSTNng_ATLAM5 HSTNng ATLAM5 83,200000
CHINng_WASHng CHINng WASHng 2189,600000
ATLAng_HSTNng ATLAng HSTNng 5606,700000
ATLAng_NYCMng ATLAng NYCMng 878,300000
LOSAng_CHINng LOSAng CHINng 42496,900000
NYCMng_CHINng NYCMng CHINng 12232,700000
ATLAng_WASHng ATLAng WASHng 3715,000000
CHINng_NYCMng CHINng NYCMng 2388,200000
HSTNng_NYCMng HSTNng NYCMng 684,000000
STTLng_CHINng STTLng CHINng 2437,100000
WASHng_NYCMng WASHng NYCMng 4006,900000
CHINng_ATLAng CHINng ATLAng 3673,700000
NYCMng_STTLng NYCMng STTLng 868,600000
NYCMng_HSTNng NYCMng HSTNng 3299,800000
ATLAM5_ATLAng ATLAM5 ATLAng 114,000000
STTLng_LOSAng STTLng LOSAng 3646,800000
ATLAM5_WASHng ATLAM5 WASHng 553,800000
WASHng_STTLng WASHng STTLng 793,000000
ATLAM5_CHINng ATLAM5 CHINng 312,800000
CHINng_STTLng CHINng STTLng 508,200000
STTLng_HSTNng STTLng HSTNng 775,400000
ATLAng_CHINng ATLAng CHINng 614,200000
LOSAng_HSTNng LOSAng HSTNng 16158,100000
ATLAng_STTLng ATLAng STTLng 260,400000
HSTNng_STTLng HSTNng STTLng 174,500000
LOSAng_NYCMng LOSAng NYCMng 1196,900000
HSTNng_LOSAng HSTNng LOSAng 1643,700000
WASHng_ATLAng WASHng ATLAng 2550,300000
STTLng_NYCMng STTLng NYCMng 1288,900000

ATLAM5_STTLng ATLAM5 STTLng 24,900000
ATLAM5_HSTNng ATLAM5 HSTNng 175,400000
WASHng_ATLAM5 WASHng ATLAM5 198,200000
HSTNng_WASHng HSTNng WASHng 1509,500000
HSTNng_ATLAng HSTNng ATLAng 2608,900000
NYCMng_WASHng NYCMng WASHng 4798,000000

B.2 Demand Values 2 Input File

15
DNVRng_STTLng STTLng DNVRng 100000,000000 1 1
ATLAng_HSTNng HSTNng ATLAng 100000,000000 1 1
HSTNng_KSCYng KSCYng HSTNng 100000,000000 1 1
HSTNng_LOSAng LOSAng HSTNng 100000,000000 1 1
DNVRng_KSCYng KSCYng DNVRng 100000,000000 1 1
NYCMng_WASHng WASHng NYCMng 100000,000000 1 1
ATLAng_WASHng WASHng ATLAng 100000,000000 1 1
CHINng_NYCMng NYCMng CHINng 100000,000000 1 1
CHINng_IPLSng IPLSng CHINng 100000,000000 1 1
LOSAng_SNVAng SNVAng LOSAng 100000,000000 1 1
DNVRng_SNVAng SNVAng DNVRng 100000,000000 1 1
ATLAM5_ATLAng ATLAng ATLAM5 100000,000000 1 1
SNVAng_STTLng STTLng SNVAng 100000,000000 1 1
IPLSng_KSCYng KSCYng IPLSng 100000,000000 1 1
ATLAng_IPLSng IPLSng ATLAng 100000,000000 1 1
56
CHINng_LOSAng CHINng LOSAng 57033,011128
STTLng_ATLAM5 STTLng ATLAM5 79,981794
NYCMng_ATLAM5 NYCMng ATLAM5 91,439048
STTLng_ATLAng STTLng ATLAng 3613,684356
WASHng_CHINng WASHng CHINng 3914,581448
HSTNng_CHINng HSTNng CHINng 1074,705077
NYCMng_ATLAng NYCMng ATLAng 1226,309010
WASHng_LOSAng WASHng LOSAng 6036,770606
WASHng_HSTNng WASHng HSTNng 2052,759145
ATLAng_LOSAng ATLAng LOSAng 6701,775108
STTLng_WASHng STTLng WASHng 1535,541853
CHINng_ATLAM5 CHINng ATLAM5 287,000942
LOSAng_ATLAM5 LOSAng ATLAM5 168,084214
NYCMng_LOSAng NYCMng LOSAng 2847,423447
CHINng_HSTNng CHINng HSTNng 17405,495674

ATLAM5_NYCMng ATLAM5 NYCMng 144,674949
ATLAM5_LOSAng ATLAM5 LOSAng 110,904458
LOSAng_STTLng LOSAng STTLng 791,308320
LOSAng_WASHng LOSAng WASHng 7572,877405
LOSAng_ATLAng LOSAng ATLAng 4231,871591
ATLAng_ATLAM5 ATLAng ATLAM5 252,620878
HSTNng_ATLAM5 HSTNng ATLAM5 61,511726
CHINng_WASHng CHINng WASHng 3045,922357
ATLAng_HSTNng ATLAng HSTNng 5701,265168
ATLAng_NYCMng ATLAng NYCMng 522,628984
LOSAng_CHINng LOSAng CHINng 34609,863377
NYCMng_CHINng NYCMng CHINng 15577,789584
ATLAng_WASHng ATLAng WASHng 3267,634930
CHINng_NYCMng CHINng NYCMng 3445,010779
HSTNng_NYCMng HSTNng NYCMng 470,684434
STTLng_CHINng STTLng CHINng 3287,630630
WASHng_NYCMng WASHng NYCMng 5526,488366
CHINng_ATLAng CHINng ATLAng 2055,757395
NYCMng_STTLng NYCMng STTLng 538,333659
NYCMng_HSTNng NYCMng HSTNng 4304,946022
ATLAM5_ATLAng ATLAM5 ATLAng 74,333912
STTLng_LOSAng STTLng LOSAng 3938,604515
ATLAM5_WASHng ATLAM5 WASHng 585,706890
WASHng_STTLng WASHng STTLng 504,097052
ATLAM5_CHINng ATLAM5 CHINng 166,662097
CHINng_STTLng CHINng STTLng 363,183433
STTLng_HSTNng STTLng HSTNng 629,536643
ATLAng_CHINng ATLAng CHINng 552,823753
LOSAng_HSTNng LOSAng HSTNng 24177,204838
ATLAng_STTLng ATLAng STTLng 184,355163
HSTNng_STTLng HSTNng STTLng 119,684025
LOSAng_NYCMng LOSAng NYCMng 1469,390378
HSTNng_LOSAng HSTNng LOSAng 2002,713463
WASHng_ATLAng WASHng ATLAng 1589,933611
STTLng_NYCMng STTLng NYCMng 1598,789333
ATLAM5_STTLng ATLAM5 STTLng 17,878379
ATLAM5_HSTNng ATLAM5 HSTNng 197,495214
WASHng_ATLAM5 WASHng ATLAM5 297,060072
HSTNng_WASHng HSTNng WASHng 1235,672733
HSTNng_ATLAng HSTNng ATLAng 1381,719415
NYCMng_WASHng NYCMng WASHng 5529,802931