Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Remo Balaguer

Flow Embedding Algorithms for Software Defined Audio Networks

Master Thesis MA-2014-14
15th of September 2015 to 13th of March 2015

Supervisor: Prof. Dr. Bernhard Plattner
Academic Advisors: Dr. Panagiotis Georgopoulos, Vasileios Kotronis
Industrial Advisor: Peter Glaettli (R&D Director, Studer, Regensdorf)

Abstract

Audio over IP is becoming increasingly more important in the professional audio industry. The interoperabilty standard AES67 further promotes this trend by defining a common standard on how to transform digital audio data to a stream for IP networks. Consequently, the intelligent management of networks is the next step towards the widespread deployment of Audio over IP in the entertainment industry. The emerging approach of Software Defined Networking provides a global view and control on networks by abstraction, which enables optimized flow embedding. In professional applications multimedia streams have to run over paths, which satisfy a certain maximum latency requirement and a minimal bandwidth requirement. If the network paths do not satisfy both requirements, the stream cannot run successfully. As there may be multiple paths for a stream to be routed from a source to a target node in the network, the flow embedding has to be in such a way that as many other streams as possible can run on the same network concurrently, in order to maximize the end-user's benefit from the network.

The result of this thesis is the design, implementation and evaluation of three different flow embedding algorithms to address this routing problem.

We found that for some scenarios a Genetic Algorithm (given optimal parameters) can perform in the same acceptance ratio range as a Linear Program, which is a common technique to solve flow embedding problems. In addition, the Genetic Algorithm can deliver partial solution during the process of solving, which is a great advantage of the Genetic Algorithm. However, there are certain bottlenecks regarding the Genetic Algorithm, which are described in the thesis.

As far as the Online Algorithm is concerned, it performs slower than the Linear Program for small graphs but faster for large graphs. Yet, the larger the graph the lower the acceptance ratio (a few percent less) of the Online Algorithm compared to the Linear Program.

Acknowledgments

In the first place I would like to thank Prof. Dr. Bernhard Plattner for providing the academic environment for this work. Also, I wish to express great thanks to my advisors Dr. Panagiotis Georgopoulos and Vasileios Kotronis for their constant support.

I express my gratitude to Peter Glaettli, R&D Director at Studer [1], whom I'd like to thank for the idea and support of this thesis. Further, many thanks to Christos Liaskos at the Institute of Computer Science (ICS) of the Foundation for Research and Technology - Hellas (FORTH) for his helpfulness and advice regarding Genetic Algorithms. Last but not least, I would like to thank Bruno Loetscher and Thomas Wicki from SRG (Swiss Broadcasting Corporation) for providing their top level network topology and for interesting discussions.

Contents

1 Introduction **7**
1.1 Problem Statement and Motivation . . . 7
1.1.1 Why Audio over IP? . . . 7
1.1.2 Audio over IP today . . . 7
1.1.3 Audio over IP Limitations . . . 7
1.1.4 Towards Software Defined Audio Networking . . . 8
1.2 Goals . . . 9
1.3 Overview . . . 9

2 Background and Related Work **11**
2.1 Background . . . 11
2.1.1 Audio Network Management . . . 11
2.1.2 Software Defined Networking . . . 11
2.1.3 AES67 . . . 12
2.2 Related Work . . . 13
2.2.1 Resource Allocation in Communication Networks . . . 13
2.2.2 Audio / Video Bridging and Time Sensitive Networking . . . 13
2.2.3 Commercial Solutions . . . 14
2.2.4 Linear Programming for Routing Optimization Problems . . . 14
2.2.5 Genetic Algorithms . . . 15

3 Software Defined Audio Networking Algorithms **17**
3.1 Context . . . 17
3.2 Algorithm Requirements and Design . . . 18
3.2.1 The Embedding Problem . . . 18
3.2.2 Online case . . . 19
3.2.3 Offline case . . . 20
3.3 Algorithm Implementation . . . 21
3.3.1 Online Algorithm . . . 21
3.3.2 Offline Algorithm: Linear Program . . . 23
3.3.3 Offline Algorithm: Genetic Algorithm . . . 26

4 Evaluation Methodology for Software Defined Audio Networking Algorithms **31**
4.1 Graphs from Network Topologies . . . 31
4.1.1 Elementary Graphs . . . 31
4.1.2 SRG Graphs . . . 32
4.1.3 Butterfly Graph . . . 33
4.1.4 Internet Topology Zoo Graphs . . . 34
4.1.5 Overview of the Experimentation Graphs . . . 35
4.2 Demand Sets . . . 35
4.3 Experiments . . . 37

5 Evaluation Results and Discussion **39**
5.1 Linear Program Runtime versus Problem Size . . . 40
5.2 Search All Paths Problem . . . 40
5.3 Suitable Genetic Algorithm parameters . . . 41

5.4 Acceptance Ratio Comparison:
Linear Program versus Genetic Algorithm 43
5.4.1 Butterfly Graph 43
5.4.2 Redbest Graph 44
5.4.3 Cogent Graph 45
5.5 Genetic Algorithm: Robustness 46
5.6 Linear Program versus Online Algorithm 47
5.7 Conclusion 49

6 Summary and Outlook **51**
6.1 Summary 51
6.2 Outlook 52
6.2.1 System Aspect 52
6.2.2 Multicast 53
6.2.3 Demand Grouping 53
6.2.4 Fail-over Paths 53
6.2.5 Number Of Channels Per Stream 53
6.2.6 QoS Applications 53
6.2.7 Genetic Algorithm Refinements 53

Chapter 1

Introduction

1.1 Problem Statement and Motivation

1.1.1 Why Audio over IP?

Today's audio engineers are confronted with an enormous number of audio channels during an event. Broadcasting large live audio performances like for example the Super Bowl [2] requires the handling of up to 2000 different audio channels. It is obvious that a setup with single-channel point-to-point connections would end up in a big clutter considering all the separate cables. Also in recording studios a big number of point to point connections leads to the time-consuming process of manual signal routing and manual channel assignment with respect to their source. In order to keep the number of cables low, multichannel technologies like MADI [3], which corresponds to the AES10 Standard that can transmit up to 64 channels, are commonly used. But MADI still needs special hardware and is a point to point technology that does not support sophisticated network topologies. That is why MADI ends up in relatively high costs for hardware and cabling effort. This is the motivation behind the development of Audio over IP (AOIP).

1.1.2 Audio over IP today

AOIP technology can be used with common Ethernet cables, switches and routers and therefore mitigates the problem involving the need for special hardware and the requirement to install many cables. However, the problem is that, in the past decade, a variety of different proprietary AOIP technologies like Livewire [4], Dante [5] or WheatNet [6], which are not interoperable, were developed and deployed [7]. These well-established technologies support multichannel uncompressed audio transmission at ultra low latencies (1-2 milliseconds [8] [9]) and are used in many studios all around the world. They are wide spread even though they are not interoperable because of differences in sample packetization, time synchronization and transport protocols. A successful open standard for professional AOIP applications has not been released until now.
In 2013 the Audio Engineering Society (AES) published the AES67 Standard "AES standard for audio applications of networks - High-performance streaming audio-over-IP interoperability" [10]. The aim of that document is to define an open AOIP standard that is very close to the existing technologies so that manufacturers are motivated to create new versions of their protocols, which are AES67 compliant. Ravenna by ALC NetworX already supports AES67 and others announced new AES67 compliant versions of their protocols [7] [11].

1.1.3 Audio over IP Limitations

The industry now has strong motivation to make modern AOIP systems interoperable through AES67, which would be a huge advantage for audio production and broadcasting. The AES67 Standard promotes the formation of large AOIP networks, however it does not address the problem of AOIP network control and management. The uprising and remaining challenge then is to manage an AOIP network, since the core element of AES67 lies in the transmission of audio samples over a network but neither the routing of streams through the network nor the discovery and control of devices is considered.

Nowadays an audio engineer has to decide, if a network is suitable for a specific task, based on the network topology, latency and throughput, which requires a lot of manual labour. If necessary, additional Ethernet cables, switches or routers have to be added. Also the configuration has to be done manually, which is expensive and exhausting.
Consider the following situation: A major radio station in Switzerland owns a network that connects different cities (Fig. 1.1). An outdoor event is taking place at Basel where analog audio signals get converted to digital samples in the stagebox, which transmits the audio to the main studio in Basel on MADI. From there a stream A is sent to the broadcast center in Zurich, which at the same time also has to send a stream B to Genf for another program. The question that arises, is, whether there is enough capacity in the network for both streams to be routed at the same time and which routes they should take. Also the situation may change if there is an extraordinary event at Lugano, which causes a stream C to be routed to Basel. Additionally, the network should react very quickly and smartly if there is a link failure.

1.1.4 Towards Software Defined Audio Networking

A way to cope with such challenges is ”Software Defined Audio Networking” (SDAN), which is introduced by this project and is based on ”Software Defined Networking” (SDN). It enables dynamic traffic management and central network control by software, which can be designed to optimize the network performance. Besides the complete network topology, the controller has knowledge of the demands that the audio engineer would like to be routed in the network. Demands reflect streams by source ID, target ID, bandwidth requirement and latency requirement which result in network flows if embedded by the controller. Embedding (also called the solution to the embedding problem) in this context denotes the process of finding a path through the network by the use of a routing algorithm and pushing the flow rules to the switches. Allocation stands for the process storing an embedding in a database, such that the controller can calculate the residual capacities on each network link before embedding a new demand. In this way, it is possible to manage the available link capacities in the network without overloading individual links or bottlenecks.
In case the network is comprised of SDN capable switches, an OpenFlow-based controller can therefore centrally orchestrate the network, intelligently determine the optimal route for each stream, determine backup routes in case of link failure for each demand and push flow rules to the switches, so that the network performs optimally. Furthermore the controller should warn the network operator or audio engineer if starting streams are going to overload the network in such a way that other streams would break. It must be kept in mind that the dynamicity of such networks (compared to the Internet) is limited, as all streams are known by a central entity and the network topology is persistent.
Besides the pure routing and flow rule generation, the discovery of devices is also an important aspect of an SDAN controller for audio engineers, because they have to mix the different channels in the end and need to know what is on which channel. But since the AES67 Standard already proposes a few discovery protocols, it is assumed that this will be standardized in the following months or years.

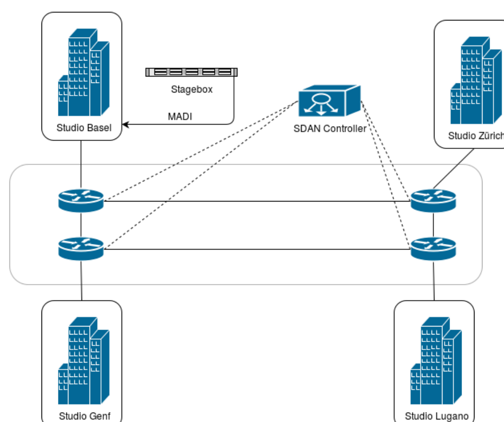

Figure 1.1: Simplified depiction of a hypothetical radio station network in Switzerland

1.2 Goals

In order to accomplish intelligent audio network management, in this thesis we will investigate optimal resource allocation routing algorithms for professional SDN Audio over IP networks. This is why we chose to set the following goals:

1. Define the specifications of SDAN resource allocation routing algorithms based on the needs of the industry.
2. Implement possible resource allocation routing algorithms.
3. Evaluate different algorithms and compare them against each other.

1.3 Overview

This thesis is structured as follows:

- Chapter 2 introduces SDN and the AES67 standard as two emerging elements in the world of AOIP and as the background of the thesis. Furthermore, it introduces network management in the context of SDAN and related work.
- Chapter 3 describes two different types of SDAN resource allocation routing algorithms based on the requirements of a professional AOIP network. It then introduces the design of three algorithms and provides implementation insights.
- Chapter 4 illustrates how the different implemented algorithms were evaluated by means of test scenarios, in order to give an overview on the algorithm performance.
- Chapter 5 provides results of the algorithm performance evaluation. It states conclusions and gives insights with respect to task of the project.
- Chapter 6 provides an overview on the problem, possible solutions and the contributions. It also suggests continuative work based on the findings in this project as an outlook.

Chapter 2

Background and Related Work

The following two sections outline the background and related work of this thesis. Firstly, network management is motivated and explained. Since the idea of SDAN originates from the emergence of SDN [12] and AES67 [10], the remaining sections provide a short introduction on both. In order to build on preexisting findings, methods and technologies the related work section presents related scientific works, technologies and commercial solutions.

2.1 Background

2.1.1 Audio Network Management

Computer networks are basically telecommunication networks that allow devices to exchange data packets amongst each others. The various applications of them are ubiquitous. Very often people make use of the World Wide Web or E-Mail. One rather special application is professional audio networking. The main difference of this application compared to the Internet is the very strict bandwidth and latency requirements of the media streams as well as the reliability. Streams can not successfully run without enough bandwidth (typically around 3 Mbps per stream, depending on packet size, audio sampling rate, audio bit depth and number of audio channels per stream [4]) and are useless for the application when introducing too much latency (sometimes down to 1-2 milliseconds [8] [9]). That is why admission control and traffic routing is essential in professional audio networks.
Admission control manages which streams are allowed to enter the network, in order to prevent congestion and overloading of the network. Traffic routing determines the best paths for the streams that are allowed to run and enables maximum network utilization subject to all stream requirements.

2.1.2 Software Defined Networking

The principal concept of Software Defined Networking (SDN) [12] is to separate the control and the data plane of a network. Central controllers on the control plane orchestrate the logical aspects of the network such as routing and push rules to the forwarding devices (switches) on the data plane. As a consequence, the routing logic of the network can reside in one central controller whereas the forwarding hardware implements the given routing decision. Moreover, the SDN approach facilitates sophisticated networking monitoring by one central entity. Above all, the total abstraction of a network is the main advantage of SDN compared to legacy networks as this allows for any kind of network control and monitoring.
Figure 2.1 depicts the SDN Stack with the applications in the north, the SDN controllers in the middle and the forwarding hardware in the south. While the southbound API between the controllers and the switches is commonly the standardized and popular OpenFlow protocol [13], the northbound API between the controller and the application depends on the used SDN controller framework (e.g. POX [14] or Floodlight [15]).
The great advantage of this architectural approach is that the application can learn about the complete network topology by the controllers and therefore get a global view on the network.

Sophisticated algorithms inside the applications can be applied to find the optimal routing policies for the network. The policies can then be transformed to flow rules by the applications and be pushed to each switch by the controller. This allows for efficient network management as each forwarding device implements optimal routing rules determined by the applications. For this reason it is reasonable to use SDN for AOIP networks.

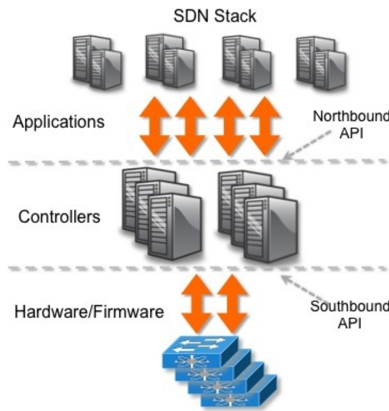

Figure 2.1: SDN Stack [16]

2.1.3 AES67

The AES67 Standard [10] was developed by the Audio Engineering Society (AES) and published in September 2013. It is an AOIP protocol on layer 3 and is designed to allow interoperabilty between previously competing AOIP systems by different professional audio manufacturers.
The key components of the protocol are:

- **Synchronization:** A master clock distributes the time over the whole network by Precision Time Protocol IEEE 1588-2008 [17]. This allows each node to generate a local media clock, which is used to synchronously playback or record audio samples.
- **Network and Transport:** In order to transport the samples over the network, AES67 makes use of RTP/UDP and supports IPv4 in unicast or multicast mode with IGMPv2.
- **Audio Sample Encoding:** The resolution of the audio samples is defined as 16 or 24 bit linear and the sampling frequency can be 44.1, 48 or 96 kHz. One stream carries up to 8 channels.
- **Audio Sample Packetization:** The maximum payload size in AES67 is defined as 1440 bytes, which allows for 6, 12, 16, 48, 192 audio samples per packet depending on the number of channels and sampling frequency.
- **Quality of service:** To prioritize time sensitive packets in the network, AES67 uses DiffServ with 3 different traffic classes (DSCP).
- **Connection management:** Session Description Protocol (SDP) and Session Initiation Protocol (SIP) are used to set up audio streams in the AES67 network.
- **Discovery:** In order to find other AES67 compliant devices in the network there are some recommendations given (i.e. ZeroConf [18], SAP [19], and others) but discovery is not fully defined in the standard.

The concept behind the design of AES67 was to create a new AOIP protocol, which is based on the commonalities of preexisting proprietary AOIP protocols like Livewire, Dante or Wheatnet, in order to minimize the effort to update preexisting protocols to be AES67 compliant. The goal is that any AOIP protocol will be AES67 compliant in the future so that every manufacturer gets interoperable with all others by means of AOIP [7] [20].

2.2 Related Work

2.2.1 Resource Allocation in Communication Networks

The dissertation on the Resource Allocation In Networks (RAIN) problem by Frei [21] [22] is closely related to this thesis, as it is also about algorithm design for optimal constraint-based routing. Actually, Frei addressed the problem of QoS routing, which can not efficiently be solved by common shortest path routing, because it leads to poor network utilization or even congestion. Frei aimed for managing bandwidth in the network so that a better use of available network resources is possible.
The network is depicted as a connected graph with nodes and edges. Nodes represent processing units like switches and routers. Edges correspond to communication media such as Ethernet or optical fibers. Each edge is characterized by its currently available bandwidth capacity. As the network is meant to fullfill communication needs between pairs of nodes, Frei introduces *demands*. A demand reflects one single communication need and is defined by the triple $d_u = (x_u, y_u, \beta_u)$, where x_u and y_u are distinct source and target nodes and β_u is the bandwidth requirement for the demand d_u. Therefore Frei illustrates the RAIN problem in Figure 2.2 and describes the RAIN problem as follows:

- **Given** a network composed of nodes and links, each link with a given resource capacity, and a set of demands to allocate,
- **Find** one route for each demand so that the bandwidth requirements of the demands are simultaneously satisfied within the resource capacities of the links.

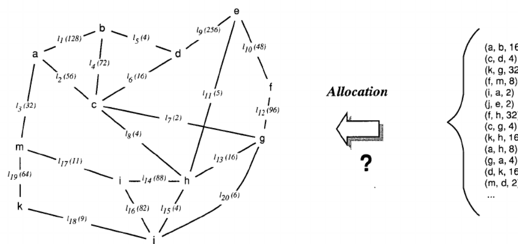

Figure 2.2: Rain Problem [21]

One important consideration to note is that it is not possible to divide demands among multiple routes. This means that all packets per flow have to take the same path as different paths introduce varying latencies and jitter, which could cause stalls in real time applications where network buffers have to be small. This makes the RAIN problem NP-hard.
Frei showed how an abstraction technique, namely "Blocking Islands", allowed applying the well-known Constrain Satisfaction Problem (CSP) technique to solve the problem with manageable complexity.
However, there was unfortunately no implementation of RAIN available and the main part of his work does not include latency requirements per demand.

2.2.2 Audio / Video Bridging and Time Sensitive Networking

Another approach to implement audio network management is Audio / Video Bridging (AVB). AVB is a layer 2 technology by the IEEE comprising the Stream Reservation Protocol (SRP), which implements admission control to guarantee QoS [23] [24]. Although AVB is an established technology in the multimedia industry, Layer 3 protocols are more widely used as they do not need any specialized hardware. Additionally, SRP does not optimize for network utilization in the first place.
Time sensitive networking (TSN) is a set of standards by the IEEE that defines extensions to the IEEE 802.1Q Virtual LANs. The TSN Task Group stems from the AVB Group after renaming them in 2012. The main purpose of TSN is to accomplish very low transmission latency and a high availability in networking by SRP. It can be used for audio and video applications or

real-time control streams in industrial facilities [25] [26]. However, it does not optimize the flow embedding such that as many streams as possible concurrently run on the same network and the end-user's benefit from the network would be maximized.

2.2.3 Commercial Solutions

Network management can evidently be carried out manually, meaning that a network engineer configures all switches in such a way that the network can successfully transport all streams concurrently. This might be a sufficient solution in small network. For rather big networks there are commercial and proprietary control and management solutions. For instance, the L-S-B Virtual Studio Manager (VSM) [27] is an example for an IP-based broadcast control and monitoring system, which abstracts the physical hardware from the control interfaces with all kinds of equipment. Many different protocols are integrated and hardware devices like video routers, audio routers, audio consoles and intercoms are supported. VSM is a server application that acts as a system controller that provides monitoring and control of the whole multimedia network and all connected devices. It can be operated on many special proprietary hardware panels that are well-suited for studios large broadcasters [28].

2.2.4 Linear Programming for Routing Optimization Problems

Linear Programming (LP) is a frequently used technique for solving routing optimization problems and has been applied in this thesis as well (Section 3.3.2). Linear Programming in the case of Mixed Integer Programming (introducing integer and binary variables in the program) is a widespread method for optimized flow embedding and routing. Bley [32], for instance, used the concept of Linear Programming for the minimum congestion unsplittable shortest path routing problem in IP networks whereas Kotronis et al. have optimized flow embedding in terms of maximizing the number of embeddable flows in the context of IXP multigraph [33].

Generally, LP is a method to find the best solution with respect to a linear objective function in a mathematical model. In fact, it is an optimization of a linear objective function subject to linear equality and linear inequality constraints [29]. A simple introductory example for a linear program is demonstrated in the following:

- **linear function to be maximized**
  $f(x_1, x_2) = c_1x_1 + c_2x_2$
- **constraints**
  $a_{11}x_1 + a_{12}x_2 \leq b_1$
  $a_{21}x_1 + a_{22}x_2 \leq b_2$
  $a_{31}x_1 + a_{32}x_2 \leq b_3$
- **non-negative variables**
  $x_1 \geq 0$
  $x_2 \geq 0$

Having a description of the Linear Program, a solver is needed in order to find the best solution to the problem. There are open-source solvers like LpSolve [30] or proprietary ones like Gurobi [31] that solve Linear Programs.
However, when introducing integer variables in a Linear Program, which is then called a Mixed Integer Programming (MIP), the solving turns into a NP-hard problem due to combinatorial characteristics of the Mixed Integer Linear Program [34].

2.2.5 Genetic Algorithms

In networking, Genetic Algorithms have been used for finding shortest paths in networks [36] [37]. Also Kumar et al. has used the concept of Genetic Algorithm to solve the shortest path problem routing problem for IP networks, and found that Genetic Algorithms are well suited for cases where the solution space is huge and the Dijkstra algorithm gets slower [38].
In general, Genetic Algorithms are search heuristics for optimization problems [39]. They were invented by John Holland at the University of Michigan [40] in the 70s and is a particular class of the evolutionary algorithms. Evolutionary algorithms mimic the characteristics of biological evolution such as mutation, inheritance and selection that enable finding a solution to a problem by iteration.
In a Genetic Algorithm there are populations of individuals where each individual represents a candidate solution of the optimization problem in an encoded manner (e.g. bit string). After having created an initial population, also called "first generation", the evolution starts in an iterative way. In each generation, every individual is evaluated and assigned a fitness value by an objective function. Afterwards, the individuals are recombined and randomly mutate in order to create a new generation. Commonly, the algorithm terminates after having reached a satisfactory fitness level or a maximum number of generations.
The process of each Genetic Algorithm is characterized by key parameters like the number of individuals per population or the mutation rate per evolutionary step. The time it takes for the evolution to reach better solution candidates is dependent on the value of these parameters and should therefore be tuned. A possible method of tuning the parameters is to use a second level Genetic Algorithm optimizing the parameters of an underlying Genetic Algorithm [41].
There are more sophisticated versions of Genetic Algorithms for Multi-objective Optimization such as the Non-Dominated Sorting Genetic Algorithm II (NSGA-II) [42] or Strength Pareto Evolutionary Algorithm (SPEA) [43]. In his work Zitzler showed that Genetic Algorithm can be well applied to Multi-objective Optimization [44].
To the best of our knowledge we are the first to use Genetic Algorithms to address the flow embedding optimization problem with bandwidth and latency requirements on a per flow basis.

Chapter 3

Software Defined Audio Networking Algorithms

3.1 Context

In this project we introduce Software defined Audio Networking (SDAN) as a proposal for efficient management of professional audio networks. The components of such a network comprise an SDN/OpenFlow enabled network for the transport of audio data and an SDN controller, which instructs the switches to establish designated routes for each stream. The information on the exact route of each stream is the output of an algorithm on top of the controller (Fig. 3.1) that determines the best possible allocation of resources (also called optimal embedding) in the network, such that as many streams as possible can run concurrently and therefore the end-user's benefit from the network is maximized. Another way to express this aim is to set the goal to maximize the so-called acceptance ratio $\frac{\text{number of embedded demands}}{\text{total number of demands}}$, which is a measure for the quality of embedding with respect to the end-user's benefit from the network in the context of SDAN. Given a network and a number of demands, it is possible that not all demands can be embedded at the same time. Therefore, the algorithm does not only decide on the routing of each demand but also on which demands are embedded in order to maximize the acceptance ratio. Furthermore, it is essential that each solution satisfies each stream's minimal bandwidth and maximum latency requirement and that the residual bandwidth capacity of each link is non-negative. We call this the SDAN algorithm problem. To solve the problem for a given demand set of streams, an algorithm is also given the whole network topology including bandwidth and latency values on each edge, which can by done by the approach of SDN.

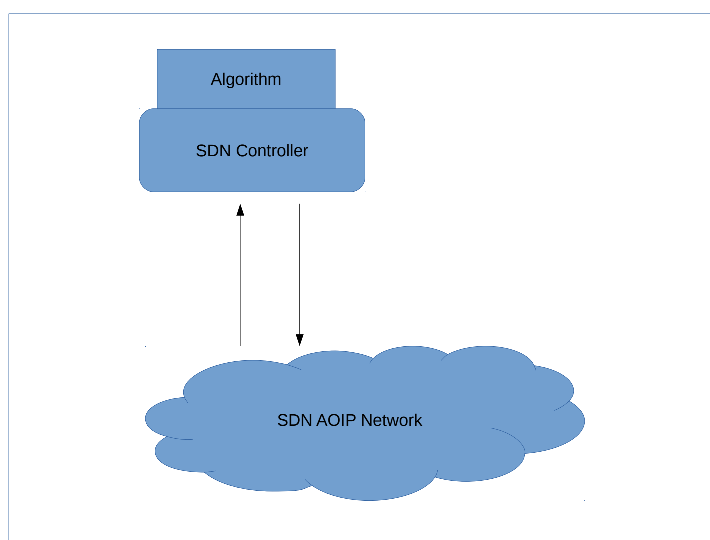

Figure 3.1: Software Defined Audio Networking Overview

3.2 Algorithm Requirements and Design

3.2.1 The Embedding Problem

In order to illustrate the challenge of optimal resource allocation in terms of network utilization, Table 3.1 presents an example demand set and Figure 3.2 and 3.3 show a suboptimal and an optimal embedding in the network for the same demand set.
Figures 3.2 and 3.3 depict an identical graph with residual bandwidth capacity as edge labels. The two figures demonstrate that when having multiple possible embedding possibilities, one of them can be more efficient than the other in terms of acceptance ratio regarding the whole demand set.

time	demand ID	source ID	target ID	bandwidth requirement
1	1	1	4	0.4
2	2	1	4	0.4
3	4	1	4	0.8

Table 3.1: Example demand set

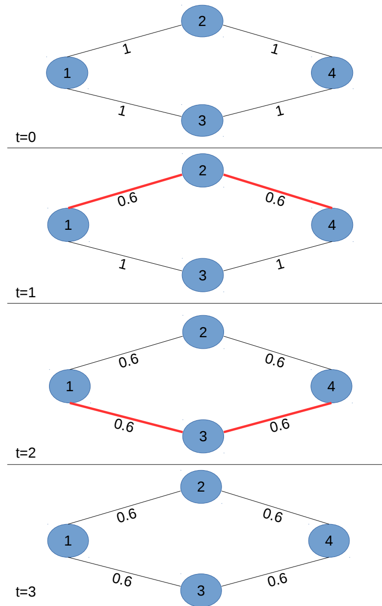

Figure 3.2: Suboptimal Allocation for example demand set of Table 3.1

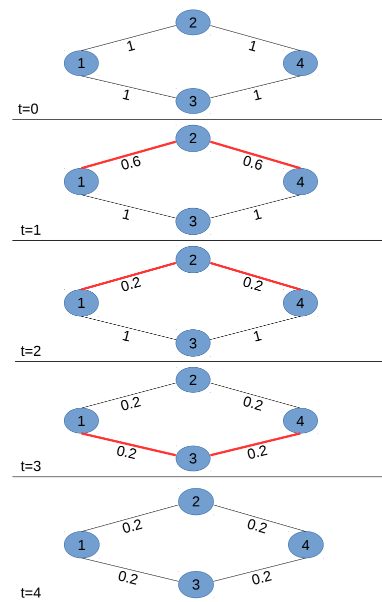

Figure 3.3: Optimal Allocation for example demand set of Table 3.1

At $t = 1$ the first demand is embedded on the edges $(1, 2)$ and $(2, 4)$ in both Figures. At $t = 2$ the second demand is embedded on the edges $(1, 3)$ and $(3, 4)$ in Figure 3.2, whereas in Figure 3.3 it is embedded on edges $(1, 2)$ and $(2, 4)$. Both embeddings are valid as they do not violate the bandwidth capacity on the edges, but it turns out that for this demand set one embedding is more efficient than the other. In the case of Figure 3.2 there is no capacity left for the third demand at $t = 3$ while in Figure 3.3 the demand can be embedded on $(1, 3)$ and $(3, 4)$ which leads to a higher acceptance ratio.
It is important to note that in many scenarios the SDAN Algorithm has to cope with latency requirements as well. Furthermore, like in the RAIN problem described by Frei [21], it is not possible to split streams. This means that all packets of the same stream have to take the same path as different paths introduce varying latencies and jitter, which could cause stalls in real time applications where network buffers have to be small.

In order to tackle the problem of optimal embedding, namely, optimal resource allocation, we divide SDAN Algorithms into two classes. **Online algorithms** determine the path for each new demand on the fly in a short amount of time and therefore embed demands one after another. **Offline algorithms** in contrast are allowed to use more calculation time and try to find a set of best paths for all known demands in advance before the embedding. The following subsections present an overview on the two classes:

3.2.2 Online case

SDAN for radio and television broadcasting faces the challenge of unforeseen events. For instance, it can happen that there is need for a live-broadcast to a faraway place due to a catastrophe, which is reported in the news show. In such cases it is necessary to be able to embed demands within a few minutes or less. The disadvantage of this situation is that there is no possibility to optimize the total acceptance ratio because future demands are not known and previously allocated demands may or may not be rerouted due to different latency and jitter on different routes. We call this type of problems Online case and list its key characteristics as follows.

- **Scenario**
  Demands are unknown in advance and are embedded one by one on the fly
- **Objective**
  Find a feasible path fast
- **Principle**
  Prune graph for bandwidth, use Dijkstra to find shortest path satisfying latency req.
- **Advantage / Disadvantage**
  fast, but embedding is not optimal

3.2.3 Offline case

Another common application of professional audio infrastructure including AOIP are live events like concerts or TV shows. The demands can be pre-scheduled in this cases, which allows for maximizing the acceptance ratio by dedicated flow embedding algorithms that find the best possible solution. We call this type of problems Offline case.
An additional application of such application is the dimensioning of a network. Since these algorithms find the best possible solution, a network can be checked for the ability of transporting a set of streams. If the network provides too little capacity, the offline algorithms allows for investigating the effect of additional switches and lines.
Moreover, an offline algorithm can be used to allow a performance evaluation of an online algorithm. By finding the best possible embedding for a set of demands, which have been previously embedded by an online algorithm, the offline algorithm provides a reference for the comparison between the embedding performed by the online algorithm and the best possible embedding.
The offline algorithm can even assist the online algorithm by solving sub-problems (find a solution for a subset of demands in a loaded network) and implementing reallocation mechanisms (Section 3.3.1).
The key characteristics of the Offline case are as follows.

- **Scenario**
  - Demands are known in advance and embedded at once
  - Dimensioning a network
  - Rating the performance of an online algorithm
  - Performing reallocation in the online case
- **Objective**
  - Find paths for as many demands as possible, maximize acceptance ratio
- **Principle**
  - Linear Mixed Integer Program
  - Genetic Algorithm
- **Advantage / Disadvantage**
  - Linear Program: Embedding is optimal, but solving a Mixed Integer Linear Program is NP-hard [34]. Therefore this approach gets much slower as the graph size increases (Section 5.1). In addition, it is not possible to get partial solution during the solving process.
  - Genetic Algorithm: Embedding is optimal, too. Yet, it is possible to get partial solution during the solving process. On the other hand, it is difficult to tune parameters for optimal performance [41].

3.3 Algorithm Implementation

As a proof of concept we implemented SDAN algorithms as described in Section 3.2 and tested them on different network topologies and demand sets. Network topologies are represented by graphs. A graph with a corresponding demand set form a scenario or 'SDAN problem'. This Section presents the functional principles of one online algorithm and two different offline algorithms, which are evaluated later in Section 4.
All three algorithms take a graph, which represents a network and a list of demands as input, and try to find routes for all demands such that the acceptance ratio is maximized. If not all demands can be embedded in the network, the algorithm has to decide which demands to reject and not to embed.
The implementation was done in Python with the help of some preexisting modules. The most important ones are NetworkX [45] and Graphviz [46]. NetworkX is a Python package providing data structures for modeling graphs and methods for handling graph problems. It comes with various interfaces for building, importing and exporting graphs as well as methods for finding paths in the graph and producing graph-theoretical statistics. Further, Graphviz is an open-source software package suitable for drawing graphs. Its layout algorithms are very sophisticated such that even large and complex graphs are clearly presented.

3.3.1 Online Algorithm

Functionality and features

By experimentation we found that a possible way for finding paths that satisfy the bandwidth and latency requirement, in a short amount of time ('on the fly'), is the combination of the following two steps:

1. **Pruning the graph for bandwidth**
   By pruning the graph for bandwidth, all the edges that can not offer enough bandwidth for the demand in question are removed. The pruned graph is then stored in a temporary variable for further steps. The pruning ensures that all paths found by the following step have enough bandwidth.

2. **Looking for low latency paths**
   The second step is to look for all paths that fullfill the latency criterion. Afterwards the list of all paths, that provide a latency that is lower than the demand's constraint, are sorted by the number of hops. In order to keep the consumption of network resources as small as possible (e.g. the number of edges in the path for a demand multiplied by its bandwidth requirement), the path having the least number of hops is chosen. If no path is found, the demand has to be rejected or a reallocation procedure can be initiated as explained in the following section.
   The search for low latency paths can be performed by a modified depth-first search algorithm [47], which is also implemented in the NetworkX [45] method 'all_simple_paths'.

Reallocation Procedure

Although the aforementioned technique finds existing paths that satisfy the constraints, the fact that it allocates resources in the network on the fly yields a greedy character of the algorithm since it can not take upcoming demands into consideration. Thereby, it can happen that after having embedded n demands, d_n (the (n+1)-th demand) can not be allocated anymore due to a lack of network resources, although all n+1 demands could coexist with a more intelligent embedding. To overcome this downside we developed a reallocation mechanism in the algorithm, which tries to reallocate (allocate demands on alternative paths) some of the previously allocated demands such that all n+1 streams can coexist in the network, if possible.
The basic idea is to let an offline algorithm solve the allocation problem for a subset of the demands. This subset comprises one currently non-allocatable demand n+1 and a specific number of selected demands d_u's that have been previously allocated.
Due to possible jumps in path latency and jitter during rerouting, delicate streams (e.g. live streams) can stall or break because of buffer underrun and timeout at the stream's receiver. In

order to avoid this, the end user can set a flag for each demand, defining whether a stream is allowed to be rerouted or not. Hence, the subset of demands for the offline algorithm must not contain demands that are not allowed to be rerouted.
The remaining component of this mechanism is the selection of promising d_u's. Demands that are more likely to release network resources for the embedding of d_n are preferred for the subset of demands. The selection process for the d_u's goes as follows:

1. Get the paths for d_n in the empty (no other demands embedded) graph as the optimal path to assure that the demand is feasible and prepare the search for reroutable d_u's
2. Check all paths of the currently allocated and reroutable demands for intersections in the graph with the optimal path for d_n and store all demands as d_u's that intersect by at least one edge
3. Sort d_u's based on bandwidth first and on the number of alternative paths secondly in a descending manner. By taking demands with high bandwidth requirements first, the probability of releasing enough bandwidth for d_n by reallocation is increased. On the other hand, sorting on the number of alternative paths increases chances of a successful reallocation process, where all d_u's and d_n can be embedded.
4. Take the first k demands in the sorted list of d_u's and d_n to formulate a sub-problem. Afterwards, let an offline algorithm solve this sub-problem. The number k can be adjusted, as larger amounts decelerate the processing.
5. If there is a feasible path for each demand of the subset: allocate new paths for all d_u's and d_n, otherwise reject d_n

Figure 3.4 depicts the procedure of the offline algorithm including the reallocation mechanism.

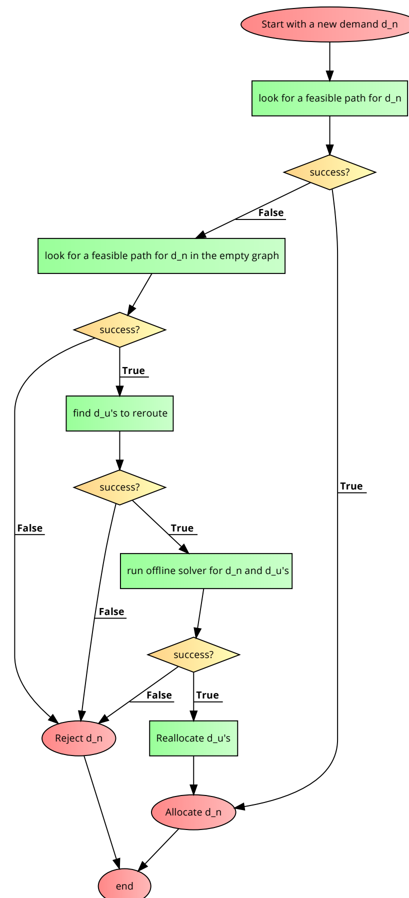

Figure 3.4: Online Algorithm Flow Chart

3.3.2 Offline Algorithm: Linear Program

Linear Programming is a widespread method [32] for optimising flow embedding and routing. Also for the SDAN problem we can use Linear Programming with the acceptance ratio being the objective function and the bandwidth / latency requirements as constraints. Hence, the following Linear Program description is a formal way to formulate the SDAN problem. It is strongly motivated by the work of Kotronis et al. that have solved a similar problem in a different context (Internet Exchange Point multigraph) [33].

Linear Program: Variables

Graph	
V_G	Set of nodes representing switches and routers
E_G	Set of edges representing links between switches and routers
$G = (V_G, E_G)$	Directed graph representing a network
$\delta^+(v)$	outgoing edges of a vertex v
$\delta^-(v)$	incoming edges of a vertex v
$e \in E_G$	edge of graph G
$bw_e \in \mathbb{R}_{\geq 0}$	Bandwidth corresponding to edge e
$lat_e \in \mathbb{R}_{\geq 0}$	Latency corresponding to edge e
Demands	
$\mathbb{D}$	Set of demands representing streams to embed in the network
D	One specific demand of the demand set
s_D	Source of demand D
t_D	Target of demand D
bw_D	Bandwidth requirement of demand D
lat_D	Latency requirement of demand D
Solution	
P_D	Path corresponding to demand D
x_D	Binary variable that indicates whether demand D is embedded or not
P_D^e	Indication whether edge e is used by demand D or not

Table 3.2: Linear Program: Variables

It should be noted that there are binary variables in this Linear Program due to the fact that streams can not be split in the network (Section 2.2.1). This makes solving the problem a lot more difficult for a Linear Program solver than compared to real-valued solutions, as the problem turns into a combinatorial one with binary variables. On the other hand, managing unsplittable flows is a basic requirement for the algorithms, as we explained in Section 3.2.1.

Linear Program: Model description

Objective Function to maximize:

$$(0) \quad max \sum_{D \in \mathcal{D}} x_D$$

subject to:

$$(1) \quad x_D = \sum_{e \in \delta^+(s_D)} P_D^e - \sum_{e \in \delta^-(s_D)} P_D^e \qquad \forall D \in \mathcal{D}$$

$$(2) \quad 0 = \sum_{e \in \delta^+(v)} P_D^e - \sum_{e \in \delta^-(v)} P_D^e \qquad \forall D \in \mathcal{D} \text{ and } v \in V_G \setminus \{s_D, t_D\}$$

$$(3) \quad bw_e \geq \sum_{D \in \mathcal{D}} bw_D \cdot P_D^e \qquad \forall e \in E_G$$

$$(4) \quad lat_D \geq \sum_{D \in \mathcal{D}} lat_e \cdot P_D^e \qquad \forall D \in \mathcal{D}$$

$$(5) \quad x_D \in \{0, 1\} \qquad \forall D \in \mathcal{D}$$

$$(6) \quad P_D^e \in \{0, 1\} \qquad \forall D \in \mathcal{D}, e \in E_G$$

If a demand D is embedded, subject (1) and (2) induce a flow from the source to the target by the law of flow conservation. Subject (3) assures that the bandwidth capacity of a link is not exceeded whereas (4) avoids violations of the latency requirements of each demand.
The variables that determine the actual embedding decisions are x_D of (5) that indicate whether demand D is embedded or not. If a specific demand D is embedded, P_D^e of (6) then denotes whether edge e is part of the path of demand D.

Linear Program: Implementation

Gurobi provides a Python interface for building and optimizing mathematical models. When Gurobi is installed on a machine the module called *gurobipy* can be imported in any python script. This way, the optimization capabilities of Gurobi can be used in a convenient way. In this project we used the Gurobi version 5.6.3. In the context of SDAN algorithms we implemented the Linear Program presented in Section 3.3.2 in a Python module by applying the following steps:

1. Convert the NetworkX Graph to a list of edges and a dictionary containing the bandwidth and latency for each edge.
2. Convert set of demands to a list of source nodes, target nodes, bandwidth requirements and latency requirements.
3. Create a Gurobi optimization model.

```
m = Model('sdan')
```

4. Create boolean embedding variables x_D of (5).

```
for r in Requests:
    x[r] = m.addVar(vtype=GRB.BINARY, name='x%s' % (r))
```

5. Create boolean membership edge variables P_D^e of (6).

```
for r in Requests:
    for i,j in edges:
P[r,i,j] = m.addVar(vtype=GRB.BINARY, name='P%s_%s_%s' % (r, i, j))
```

6. Update and input variables to the Gurobi model.

```
m.update()
```

7. Set the objective of (0).

```
acceptance = quicksum(x[r] for r in Requests)
m.setObjective(acceptance, GRB.MAXIMIZE)
```

8. Induce flow (conservation) of (1) and (2).

```
for r in Requests:
outgoing=quicksum(P[r,e,w] for e,w in edges.select(s[r],'*'))
incoming=quicksum(P[r,e,w] for e,w in edges.select('*',s[r]))
m.addConstr(x[r] == outgoing - incoming)
```

   as well as

```
for r in Requests:
for k in nodes:
if k!=s[r] and k!=t[r]:
outgoing=quicksum(P[r,e,w] for e,w in edges.select(k,'*'))
incoming=quicksum(P[r,e,w] for e,w in edges.select('*',k))
m.addConstr(0 == outgoing - incoming)
```

9. Implement bandwidth and latency requirement of (3) and (4).

```
for e,w in edges:
m.addConstr(bw_e[e,w] >= quicksum(bw_r[r]*[P[r,e,w]][0] for r in Requests))
```

   as well as

```
for r in Requests:
m.addConstr(lat_r[r] >= quicksum(lat_e[e,w]*[P[r,e,w]][0] for e,w in edges))
```

10. Perform optimization.

```
m.optimize()
```

11. Read out variables x_D of (5) and P_D^e of (6) and convert to a path for each embedded demand.

3.3.3 Offline Algorithm: Genetic Algorithm

Mapping the allocation problem to a GA

Because the Mixed Integer Program is NP-hard [34] and gets slower in an over-proportional manner as the graph size increases (Section 5.1), we were looking for an alternative, which is faster for large graphs. As it turned out Genetic Algorithms (Section 2.2.5) are worth investigating as the use of Genetic Algorithms is reasonable in cases where a straightforward solution is impossible or too complex to derive and implement. NP-Hard (as the Mixed Integer Linear Program of Section 3.3.2) problems fall into the first case [35]. In addition, we had strong evidence from related work, which is presented in the following paragraph, that Genetic Algorithms can yield benefits to problems of similar nature.
Gonen et al., for instance, used Genetic Algorithms to find a shortest path for OSPF routing within a limited time. He suggests to apply a Genetic Algorithm in the beginning of searching the space for shortest paths and then switch to other techniques towards the end of the process to speed up the whole procedure [36]. As a further example, Behzadi et al. successfully applied Genetic Algorithms on the shortest path problem in transportation on urban road maps and came up with specific mutation methods for this application of Genetic Algorithms [37]. Similar to the SDAN problem, the aforementioned related works all deal with graph problems in an abstracted way. This similarity made us even more confident on the potential applicability of Genetic Algorithms in this project.
Generally, Genetic Algorithms can be well suited for problems with huge search space, which is exactly the case when having an SDAN problem on a large graph. Last but not least we realized, how diverse the applications of Genetic Algorithms are [44]. Tobias Siegfried et al. as an example solved a multi-objective groundwater management problem [49] and others under the supervision of Eckart Zitzler at ETH Zurich, solved a physical block stacking problem using Genetic Algorithms [50]. Therefore we expected some potential in using Genetic Algorithms also in the context of the SDAN problem as well.
To make use of the principles of Genetic Algorithms described in Section 2.2.5, we abstracted the SDAN problem to a decision-finding problem. Given a graph and a set of demands, all possible paths in the graph can be determined for each demand based on the source, target, bandwidth and latency requirement. This information is stored in the gene pool, which consequently contains all possible paths per demand. The remaining and important task is to decide whether a certain demand is allocated or not and which of the possible paths it gets assigned to when embedding it. We found that this process can be performed by a Genetic Algorithm.
The following simple example (Fig. 3.5) demonstrates the operating principle of the mapping. Assume we have a graph with 5 nodes and 6 edges and a demand set containing 4 demands. The second demand, for instance, has source node 3 and target node 2. Therefore, the possible paths in the gene pool are: [3,2], [3,4,5,2] and [3,4,5,1,2]. For the sake of simplicity this example does not take specific latency and bandwidth requirements into account.

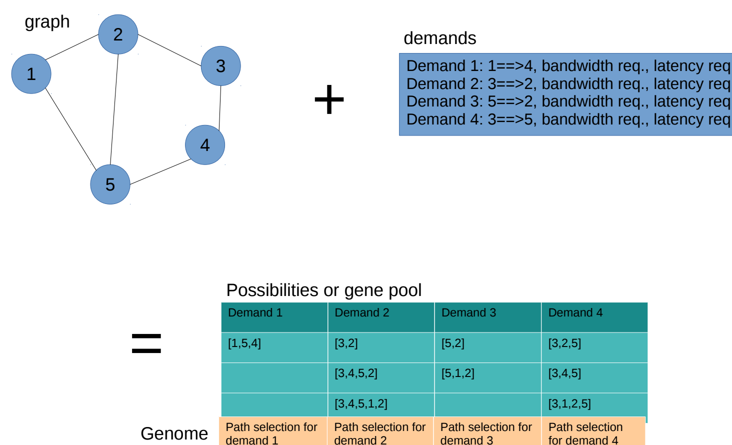

Figure 3.5: Example: Mapping the SDAN problem to a Genetic Algorithm

In order to achieve optimal embedding, the algorithm has to make a choice for each demand. This choice can either be 'non-allocated' or set to one of the paths in the pool of possible paths for that specific demand. We defined the genome to contain this choice for each demand. The genome is then evolved by the Genetic Algorithm in order to find the optimal genome that represents the optimal solution.

Implementation components

We implemented the basic elements required for a Genetic Algorithm tackling the SDAN problem in Python from scratch. The main components are presented in the following paragraphs to outline the implementation without going into the details of the code.
The principal concept was to implement an evolution class as the main class that takes a graph and a demand set as input and returns a path for each embedded demand. Besides finding and choosing paths, also the decision whether a demand is embedded or not, is carried out by the evolution process in order to maximize the acceptance ratio.
The evolution class is responsible for instantiating a population and hosting the process of evolution by calling methods on the population and monitoring the fitness values of each solution candidate. Eventually it determines the end of the evolution process, stops it and returns the best solution candidate to the caller.
The population is an object instantiated by an evolution object and is described by the corresponding population class. It carries a constant number of genome objects as individuals, which reflect candidate solutions. The population is responsible for instantiating all individuals and keep fitness values assigned to each of them. Based on the fitness values it establishes a ranking from the fittest to the least fit individuals. In addition, it calls methods on the genome object for mutating and determining the fitness values.
A genome object is described by the corresponding class and is responsible for carrying one candidate solution (path choices for demands for the SDAN problem). It provides methods to the population that enables mutation of and fitness assignment to the genome. Mutation in this context denotes the process of randomly selecting a path out of a pool of possible paths for a certain number of demands. The pool of possible paths for each demand can be determined in advance (before the evolution has started) by looking for all feasible paths in the unloaded network graph. A feasible path for a certain demand has to be shorter than the latency requirement and offer enough bandwidth regarding the bandwidth requirement. The number of demands for which a mutation is carried out is denoted by the mutation rate, which is the ratio between the number of mutations in the genome and total number of demands and is managed by the evolution object. Eventually, each genome object calculates its fitness values by determining the acceptance ratio of the solution candidate as well as the bandwidth usage $\frac{\text{total_requested_bw}}{\text{total_alloc_bw}}$ such that the population object is able to rank all individuals in the population by the acceptance ratio first and bandwidth usage secondly. Thereby, the acceptance ratio is the dominant fitness value whereas the bandwidth usage enables finding the best genome under a set of genomes with equal acceptance ratios.

Genome Class

- **Variables**
  - **Demand set**
    Contains information about source, target, bandwidth requirement and latency requirement of each demand.
  - **Genome**
    Denotes the path choice for each demand of the demand set.
    The choice is either 'None' for non-allocated or a feasible path as a node list.
- **Methods**
  - **mutate**(`mutation_rate`, `non_allocation_probability`)
    Randomly selects `mutation_rate`-many positions in the Genome.
    At all selected positions in the Genome, set the choice to 'None' with a probability of `non_allocation_probability` or randomly choose another path from the gene pool

- **rate**(`graph`)
  Returns the acceptance ratio of the genome in the `graph` as well as the bandwidth usage $\frac{total_requested_bw}{total_alloc_bw}$ as a fitness value tuple

Population Class

- **Variables**
  - **Graph**
    Represents the network on which the demands are embedded.
  - **Demand set**
    Contains information about source, target, bandwidth requirement and latency requirement of each demand.
  - **Population size**
    Determines the number of genomes in one population to enable parallel evolution. The number of genomes is constant throughout the whole process.
  - **Amount of 'clergy' genomes**
    Defines the amount of 'clergy' genomes that are treated better than the rest to implement evolutionary advantage. 'Clergy' genomes are preserved unchanged for the next generation and have an adjustable amount of children.
  - **Amount of 'clergy' children genomes**
    Adjusts the amount of children of clergy genomes in the next generation.
  - **Non allocation probability**
    Denotes the probability that a mutation operation results in 'non-allocated' instead of a path variant.
  - **Individuals**
    Contains all genomes of the population.
  - **Fitness**
    Contains the assigned fitness value tuples of each genome in the population.
  - **Ranking**
    Contains the ranking based on the fitness of all genomes in the population. Sorted from the fittest to the least fit.
- **Methods**
  - **mutate**_all(`mutation_rate`)
    Calls the mutate method of each genome in the population and passes the `mutation_rate`.
  - **rate**_all()
    Calls the rate method of each genome in the population.
  - **rank**()
    Updates the ranking of all genomes in the population from the fittest to the least fit. The ranking is based on the fitness value tuple. It is achieved by sorting on acceptance ratio in the first place and for bandwidth usage in the second place.
  - **get_best_genome**()
    Returns the best genome and the according fitness value based on the ranking.
  - **evolve**(`mutation_rate`)
    Produces the next generation (step in evolution) out of the current population of genomes.
    In order to do so, it calls the mutate method of each Genome. It organizes genomes with respect to the parameters of 'clergy' genomes.
    Furthermore, it calls the rate_all() and rank() method of the population to update the fitness and ranking of all genomes for the current generation respectively population. During the whole process of evolution, this method preserves the best genome in each step such that the best genome in a population can only get better over time.

Evolution Class

- **Variables**
  - **Population**
    Contains a population object.
- **Methods**
  - **init**(`graph, demand set, population size, amount of 'clergy' genomes, amount of children by preferred genomes, non-allocation probability, target acceptance ratio, timeout`)
    This method instantiates a population with Genomes for the evolution.
  - **run_evolution**(`target_ratio, timeout`)
    This method is the core of the evolutionary process.
    It calls the 'evolve' method on the population object iteratively until the best genome has either achieved the target ratio (as an acceptance ratio) or the timeout is reached. In the end, it returns the best solution. In addition, During the process of evolution it adapts the mutation rate according to the one-fifth rule by Ingo Rechenberg [48].

Having implemented the classes described above, the evolution class can be used to run several evolution processes on the same problem on multiple threads (e.g. number of CPU cores of the machine). This parallelism increases to probability to get a solution of required fitness by a shorter amount of time as each evolution process is driven by random. At the end of the evolutionary process, the solution of the best thread can be extracted.

Parameter Choice

A downside of Genetic Algorithms is the fact that parameters like the population size or the mutation rate have to be tuned. Generally speaking, there is no generic way to tune this parameters. One way to go is to let a second level Genetic Algorithm find the optimal parameters Greffensette [41]. However, this would arise the challenge of tuning the parameters of this second level Genetic Algorithms again. Eventually, we evaluated the performance of the Genetic Algorithm for a different parameter sets by the means of brute force and looked for the best performing one. This involved selecting a pool of parameters for brute forcing by experimentation. The results of this brute force search is presented in Section 5.3.
There are two performance indices for SDAN algorithms. The first performance index denotes how long it takes for the algorithm to reach a solution with the best possible acceptance ratio (runtime performance). The second performance index is denoted by the acceptance ratio of the solution after a fixed amount of time (acceptance ratio performance). As a reference the best possible acceptance ratio can be determined by the Linear Program.
The implementation in this thesis comprises the following parameters:

- **pop_size_pool**
  Defines the number of genomes per population.
- **clergy_size_pool**
  Defines the amount (decimal fraction) of 'clergy' genomes.
  'Clergy' genomes are preserved for the next generation and have an adjustable number of children genomes.
- **clergy_children_pool**
  Defines the number of children for 'clergy' genomes.
- **start_mut_pool**
  Defines the starting mutation rate by percent of the maximum rate (number of demands).
- **non_prob_pool**
  Defines the percental probability of not choosing any path but setting the choice for a demand to 'non-allocated' by a mutation process.

Chapter 4

Evaluation Methodology for Software Defined Audio Networking Algorithms

The SDAN algorithms presented in Chapter 3 take a graph and a demand set as input and produce a list of paths for all embedded demands such that the acceptance ratio is maximized. As we wanted to test and evaluate the performance of the algorithms we had to create network graphs and demand sets as input. The following sections present the underlying network topologies that were used to create example graphs and demand sets for the performance evaluation.

4.1 Graphs from Network Topologies

All graphs represent networks where the numbers in the nodes stand for the switch IDs. Each edge connecting two nodes in the plots represents two bidirectional edges reflecting the data links. Furthermore, all edges are labelled by their characteristics in the following way: 'latency / bandwidth'.

4.1.1 Elementary Graphs

For debugging purposes and preliminary tests we have created a couple of very simple graphs depicted in Fig. 4.1, 4.2 and 4.3.

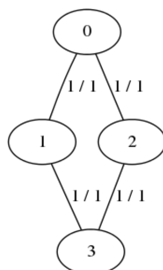

Figure 4.1: Basic diamond shaped graph

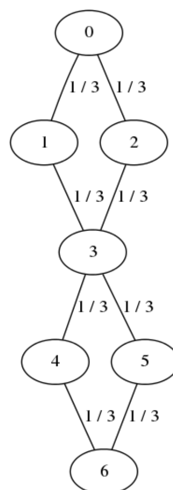

Figure 4.2: Figure of eight graph

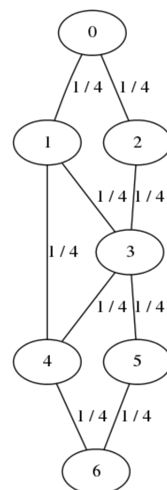

Figure 4.3: Modified figure of eight graph

4.1.2 SRG Graphs

In order to have more realistic Wide Area Network topologies the SRG (Swiss Broadcasting Corporation) [51] provided a top-level view (Fig. 4.4) of their network.

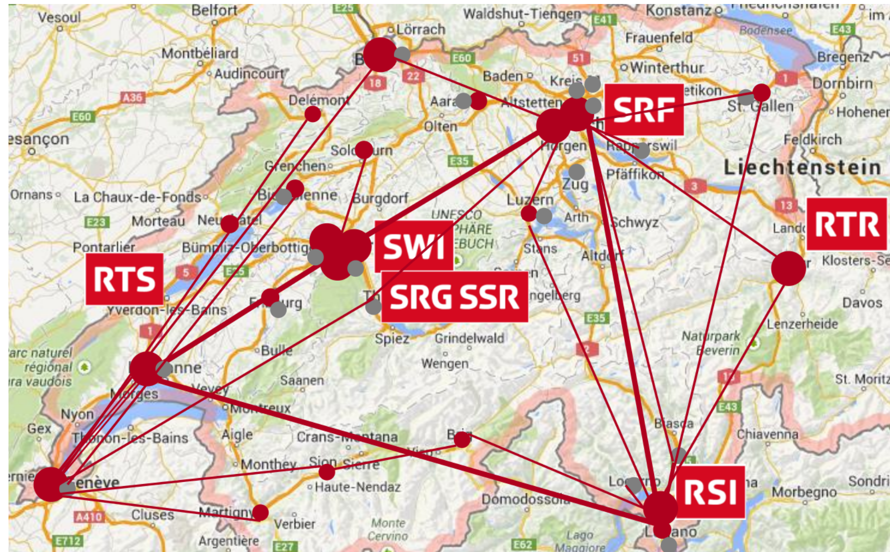

Figure 4.4: SRG network top level view

We abstracted this network to a numbered graph (Fig. 4.5) with hypothetical latencies and bandwidths because we could not get the exact facts and figures on each link. The latencies are proportional to the geographic distance (geographical distance divided by the speed of light). As for bandwidth there are a couple of devised bandwidth classes (Fig. 4.6) according to the thickness of the lines in Fig. 4.4. Also, it has to be kept in mind that this graph is not very detailed as it is only a top level view. Nevertheless, it is interesting for evaluating SDAN algorithms because it is the part of a network where the variety of possible paths for certain demands stems from.

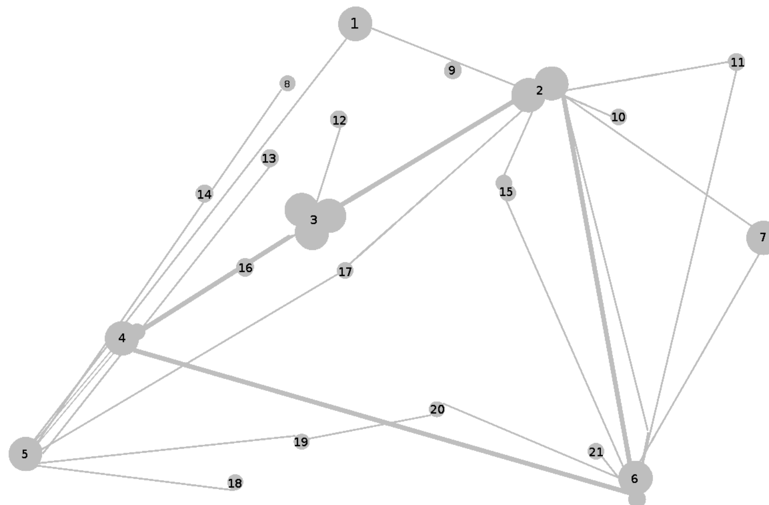

Figure 4.5: Abstracted SRG graph

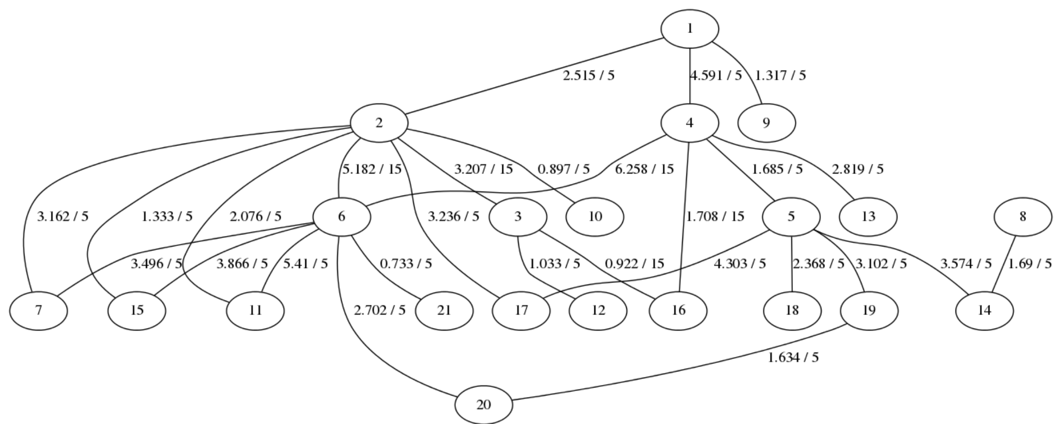

Figure 4.6: SRG graph with edge labels drawn by Graphviz.
Edge label legend: [latency in 1e-4 seconds / bandwidth in Gbit/s]
Node legend: 1: 'Basel', 2: 'Zurich', 3: 'Bern', 4: 'Lausanne', 5: 'Genf', 6: 'Lugano', 7: 'Chur', 8: 'Delemont', 9: 'Aarau', 10: 'Rapperswil', 11: 'St. Gallen', 12: 'Solothurn', 13: 'Biel', 14: 'Neuenburg', 15: 'Luzern', 16: 'Friburg', 17: 'Thun', 18: 'Martigny', 19: 'Sion', 20: 'Brig', 21: 'Locarno'

4.1.3 Butterfly Graph

Based on the SRG graph (Fig. 4.6) we built another graph that we call 'butterfly' due to its shape. It consists of the SRG graph being put in place four times next to each other as sub-graphs. The shape of the four SRG sub-graphs are identical but they differ in their node IDs in order to avoid ambiguity. The four sub-graphs are then interconnected by 16 links (latency=10, bandwidth=100) between randomly picked nodes.
This way, the butterfly graph represents four interconnected networks (e.g. countries). The fact that there are more routes within the sub-graphs than between them, introduces bottlenecks on the linkage between sub-graphs. This property forces SDAN algorithms more than in the other graphs to make a decision on which demands to embed and which not. As the bottleneck links must be shared optimally by multiple flows, without overloading the links, this is a sub-case of the knapsack problem, which is NP-Hard [52]. It has been showed that Genetic Algorithms can perform well in solving the knapsack problem [53]. Therefore, it is interesting how the Genetic Algorithm performs as compared to the Linear Program in this case.

4.1.4 Internet Topology Zoo Graphs

In order to have a set of different sized graphs we took four graphs from the Internet Topology Zoo [54]. We looked through the whole data-set and picked the 'SwitchL3' graph (Fig. 4.7) from Switzerland with 42 nodes and 126 edges, the 'RedBestel' graph (Fig. 4.8) from Mexico with 84 nodes and 186 edges, the 'Cogent' graph (Fig. 4.9) from USA/Europe with 197 nodes and 486 edges, the 'Kdl' graph (Fig. 4.10) from the USA with 754 nodes and 1790 edges. These graphs provide real topologies of real networks and were therefore used for experimentation in this thesis.

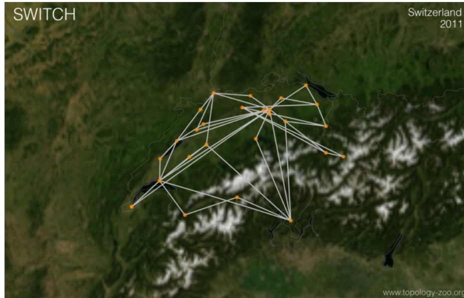

Figure 4.7: SwitchL3 graph

Figure 4.8: Redbest graph

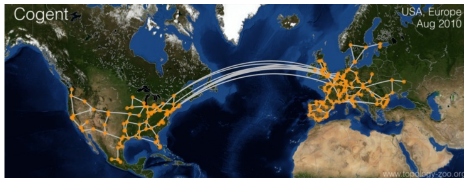

Figure 4.9: Cogent graph

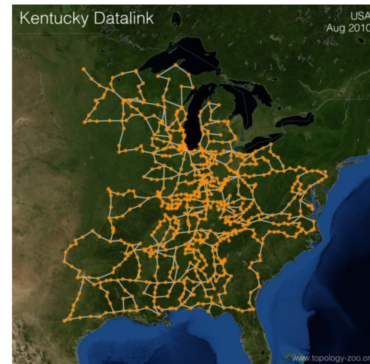

Figure 4.10: kdl graph

We note that there is information missing on the link latency provided in the data sets by The Internet Topology Zoo. Also bandwidth is only indicated on some links. This is why we had to set latencies and bandwidth manually. We chose to assign latencies and bandwidths randomly from a pool of variants. For bandwidth we chose randomly from 1, 4 or 8 bandwidth units (e.g. Gbit/s) and for latency 5, 10, 20 or 40 time units (e.g. milliseconds).

4.1.5 Overview of the Experimentation Graphs

To provide an overview on the presented graphs we list the main features of the graphs.

graph	diamond	figure of eight	mod. figure of eight
number of nodes	4	7	7
number of directed edges	8	16	18

Table 4.1: Graph properties

graph	SRG	switch	redbest	cogent	kdl	butterlfy
number of nodes	21	42	84	197	754	336
number of directed edges	54	126	186	486	1790	896

Table 4.2: Graph properties

4.2 Demand Sets

As we did not have realistic audio stream scenarios at our disposal we produced sets of random demands for the Internet Topology Zoo graphs with the following coefficients, which have been found by experimentation.
Since the network's capacity rises with the number of edges in the graph, we chose to make the number of demands linearly dependent. The bandwidth requirements are independent on the graph, whereas the latency requirement distribution is dependent on the graph's diameter to account for the size of the graph.
We produced a strict and a loose demand set for each graph in order to evaluate the algorithms on demand sets which allow acceptance ratios around 60% (strict) and close to 100% (loose).

- Strict demand set for Internet Topology Zoo graphs
  - number of demands:
    `0.4*number_of_graph_edges`
  - bandwidth requirement:
    random choice of `[0.2, 0.2, 0.4, 0.6, 1]`
  - latency requirement:
    random choice out of 4-valued linearly spaced list of
    `[10, graph_diameter*mean_graph_latency*10]`
- Loose demand set for Internet Topology Zoo graphs
  - number of demands:
    `0.3*number_of_graph_edges`
  - bandwidth requirement:
    random choice of `[0.2, 0.2, 0.4, 0.6]`
  - latency requirement:
    random choice out of 5-valued linearly spaced list of
    `[5, graph_diameter*mean_graph_latency]`

For the butterfly graph we focused on having one strict set of demands.

- Butterfly graph
  - number of demands:
    `20`
  - bandwidth requirement:
    uniform random choice between
    `[minimum_graph_bandwidth/100, maximum_graph_bandwidth/100]`
  - latency requirement:
    uniform random choice between
    `[minimum_graph_latency*20, maximum_graph_latency*20]`

It is important that all demands in the demand set are checked for feasibility before being added to the set. In other words, each demand must be actually embeddable if it had access to all resources in the idle network, respectively, empty graph. If a demand, for instance, has a bandwidth requirement higher than any link in the network, the demand is considered as unembeddable. Also, it might be generally impossible to find a path, which is short enough with respect to the latency requirement of a demand, even in the idle network. If unembeddable demands were in the demand set, this would distort the measurement as unembeddable demands decrease the acceptance ratio without the algorithm being responsible for it. Therefore, the acceptance ratio would no longer reflect the performance of the algorithm, which had to be avoided in the process of evaluation.
Also it is essential to note that the characteristics of the demand set have a great impact on the best possible acceptance ratio and the performance of the different algorithms. The following list provides some examples.

- Best possible acceptance ratio increases,
  - when the bandwidth requirements of the demands decrease
  - when the latency requirements of the demands increase
  - when the number of demands decreases
- Best possible acceptance ratio decreases,
  - when the bandwidth requirements of the demands increase
  - when the latency requirements of the demands decrease
  - when the number of demands increases

The best possible acceptance ratio in turn has an effect on the runtime performance of the Genetic Algorithm. Because the Genetic Algorithm starts out with an empty genome it has a high chance to get the best allocation in a short amount of time if the best possible acceptance ratio is very low. Moreover, the graph topology itself has an influence on the algorithm performance as well. If, for instance, the number of possible paths per demand is very low, the genetic algorithm logically needs less time to go through the search space and find the best solution.
Therefore, the selection of graphs and creation of demand sets for SDAN algorithm performance evaluation is rather delicate.

4.3 Experiments

In order to evaluate the performance for the different algorithms on the different scenarios (graphs with associated demand sets), we implemented a simulator that carries out experiments and collects measurement data.
An experiment consists of a certain graph and a demand set, which correspond to an SDAN problem. This problem is handed over to the Online Algorithm and the two Offline Algorithms (Linear Program and Genetic Algorithm) by the simulator and being solved sequentially and independently by all three algorithms. The simulator logs the runtimes used by each algorithm as well as the acceptance ratios of the provided solutions. Afterwards the simulator plots these results in one plot.
Regarding the Genetic Algorithm there are some particularities. The Genetic Algorithm is given a parameter set by the simulator, since it needs parameters (e.g. population size) in addition to the graph and the demand set. In order to find the best suitable parameters for the Genetic Algorithm for a certain scenario, we implemented a parameter set pool, which is used for brute forcing by the simulator to find the parameter set with the best acceptance ratio after a fixed amount of time. If runtime was not limited for the Genetic Algorithm, it would have run for days under some circumstances while the Linear Program solved the problem within less than a minute. This can occur when parameters of the Genetic Algorithm lead to an extremely slow progress in the evolution of the solution. For instance, a mutation rate, which is extremely low naturally turns the evolution very slow. Also implementation related effects like a very high population size slows down the software by introducing more operation per generation and defer a good solution.
As the Genetic Algorithm includes randomness in the solution process, we chose to account for this by repeating experiments 10 times and take the average of all result values, which is also done by the simulator.
Eventually, it is important to note that all runtime values depend on the hardware used, that is why it is important to list the features of the machines used:

```
Operating System: Ubuntu 14.04.1 LTS
CPU: Intel(R) Core(TM) i7-2600K CPU at 3.40GHz CPU
RAM: 16GiB system memory
```

Chapter 5

Evaluation Results and Discussion

Chapter 4 describes the test scenarios for the designed algorithms, which are presented in Chapter 3. The current chapter provides results and insights stemming from the carried out algorithm evaluation of this thesis. Firstly, Section 5.1 demonstrates the fact, that the Linear Program gets much slower as the problem size increases, which was one of the reasons why we chose to investigate Genetic Algorithms. However, in the mapping of the SDAN problem to a Genetic Algorithm (Section 3.3.3) there is a bottleneck regarding runtime performance. In particular, the time necessary for finding all possible paths per demand (respectively filling the gene pool) increases rapidly as the graph size increases. This effect and its consequences are described in Section 5.2. Besides a populated gene pool, the parameters for the Genetic Algorithm play an essential role to its performance. In order to find suitable parameters, we used a brute force approach and searched for the best parameter set in a pool of possible parameter sets. This procedure is presented in Section 5.3. Given the gene pools and parameter sets for every scenario we compare the performance of the Genetic Algorithm versus the Linear Program in Section 5.4. Furthermore, Section 5.5 investigates on the parameter robustness of the Genetic Algorithm. Eventually, the Online Algorithm (Section 3.3.1) is compared with the Linear Program in Section 5.6. The chapter is then completed up by a brief conclusion of the presented results.

5.1 Linear Program Runtime versus Problem Size

In order to demonstrate the steep rise of the time necessary to solve the NP-hard Mixed Integer Linear Program [34], we plotted the solving time versus the problem size. Figure 5.1 presents a measurement example for different graph sizes and demand sets. The graphs are 'switch', 'redbest, 'cogent' and 'kdl', which are taken from the Section 4.1.4. We measured the solving time for the Linear Program for the switch, redbest, cogent and kdl graphs (Section 4.1.4) using the loose and strict demand sets (Section 4.2).

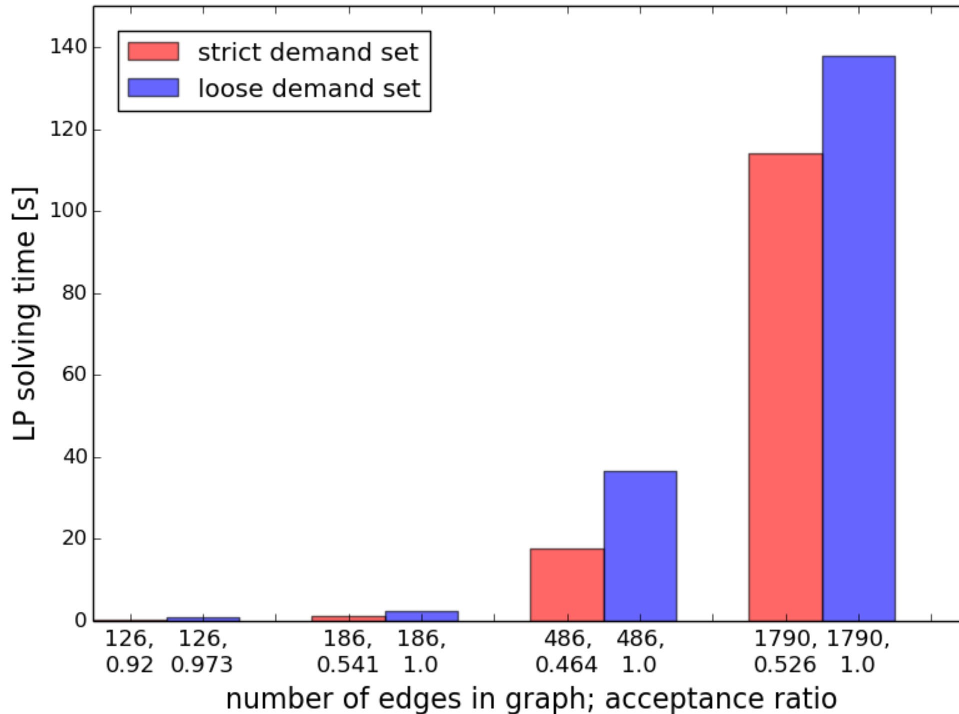

Figure 5.1: LP performance for the switch, redbest, cogent and kdl graph from Section 4.1.4

In Figure 5.1, it can be seen that with a relatively small rise of the graph size, the solving time increases highly. This therefore demonstrates the need for considering algorithms that might perform better in large graphs and for big demand sets. Furthermore, it can be seen that the solving time depends on the demand set. Since there are more possible embeddings for the loose demand set, the search space is larger and the Linear Program takes more time to explore it.

5.2 Search All Paths Problem

The time for searching all possible paths, to populate the gene pool explodes as the graph size increases. Table 5.1 provides some examples to illustrate this fact. The underlying method is the NetworkX method 'all_simple_paths', which uses a modified depth-first search. Since the complexity of finding all paths can be very large, e.g. $\mathcal{O}(n!)$ in the complete graph of order n, it task takes a vast amount of CPU time and memory for large graphs [58]. The table shows the calculation time for different graphs with different sizes and maximum path length, which limits the search depth. The specific value was taken based on experiments where we let the Linear Program and the Genetic Algorithm solve the SDAN problem for the same graph and demand set. We then took the maximum path length appearing in the solution of the Linear Program as the limit. Of course, the values depend on the characteristics of the graph and demand set. For instance, by increasing the number of edges (per node) in a graph, the number of possible paths between two random nodes is likely to increase. This again enlarges the search space and therefore increases the time needed to find all paths. A similar effect can appear by increasing

the latency requirements of the demands. As the latency requirement is higher, it is probable that feasible paths can get longer, which leads to more possible paths and enlarges the search space again. These effects are most probably the reason for the fact that the time to find all paths for some graphs with more nodes (e.g. 84) is lower than for the graphs with less nodes (e.g. 42).

graph name	switch	redbest	cogent	butterfly	kdl
nodes	42	84	197	336	754
edges	126	186	486	896	1790
max path length	11	23	22	19	54
calculation time [s]	**0.56**	**0.24**	**39.8**	**16.4**	**not known**

Table 5.1: Finding all paths

The measurements were taken on a PC with Ubuntu 14.04.1 LTS, a Intel(R) Core(TM) i7-2600K CPU at 3.40GHz CPU and 16GiB system memory. Unfortunately, it was not possible to get the runtime for the graph kdl with 754 nodes and 1790 as it always ended in a segmentation fault after 2 demands and 10 hours. This was possibly due to memory constraints on the experimental computer. The vast amount of possible paths is most likely the reason for this behaviour. However, to mitigate the problem the caching of all possible paths can simply be distributed and parallelized on multiple machines that determine all possible paths for subsets of the demands. In our experiments we performed the caching of all possible paths once and stored them in a file. Thereby, we could evaluate the actual evolutionary process of the Genetic Algorithm without caching all paths every time. Therefore, the following sections describe the evaluation of the Genetic Algorithm without the caching.

5.3 Suitable Genetic Algorithm parameters

As the performance of the Genetic Algorithm is dependent on the parameters and the scenario, we had to find suitable parameter sets for each scenario. Therefore we defined the following parameter pool as candidate parameter sets by experimentation and consultation from literature [39] and Christos Liaskos from FORTH, Greece, who has had research experience with Genetic Algorithms.

variable	**parameter pool**
population size	[10, 20, 50, 100]
amount of clergy genomes	[0,20]
amount of clergy children	[2,4]
starting mutation rate	[10,20,50]
non allocation probability	[5,10,20,40,80]

Table 5.2: Parameter pool used for finding decent parameters

Is has to be noted that this pool could be extended to increase the probability to find the best possible parameter set. However, brute forcing for the best parameters is very time consuming (in the order of several weeks), which is the reason why we have a limited amount of parameter sets in the pool.
Table 5.3 shows the results of the brute force search. For each scenario we let the Linear Program solve the problem and store the acceptance ratio of the solution and the time needed for solving the problem. Afterwards, we let the Genetic Algorithm run on the same problem for the same amount of time. In doing so, we could select the parameter set, which achieved the best acceptance ratio. In order to factor in the randomness of the Genetic Algorithm we repeated each run ten times and calculated the average acceptance ratio. Based on this, we selected the parameter set with the highest average acceptance ratio .

We note that the use of genome classes including 'clergy' genomes is beneficial in some cases. In addition, the values for the butterfly scenario clearly differ from the other scenarios, as the graph shape is different on purpose (Section 4.1.3).
Eventually, Table 5.3 shows that the connection between the scenario and the corresponding parameter set is not trivial.

	population size	clergy size	clergy number of children	start mutation rate	non allocation probability
switch, tight	20	0	0	10	5
switch, loose	20	20	4	10	40
redbest, tight	10	0	0	10	5
redbest, loose	20	20	4	10	5
cogent, tight	10	0	0	10	5
cogent, loose	10	0	0	10	20
butterfly	100	20	4	50	10

Table 5.3: Best parameter sets out of the pool 5.2

As the Genetic Algorithm is implemented in Python and could be up to 10x faster with a more efficient C-based implementation, we did the brute forcing again by giving the Genetic Algorithm 10x the solving time of the Linear Program. The results are presented in Table 5.4 and were used in all experiments of Sections 5.4 and 5.5 since there we let the Algorithm run 10x longer than the Linear Program. However, if the Algorithm is given the exact same amount of time as the Linear Program used, the parameters of Table 5.3 are suitable.

	population size	clergy size	clergy number of children	start mutation rate	non allocation probability
switch, tight	10	0	0	10	10
switch, loose	20	20	2	20	20
redbest, tight	10	0	0	10	10
redbest, loose	20	0	0	20	10
cogent, tight	10	0	0	50	5
cogent, loose	20	0	0	10	5
butterfly	50	20	4	50	20

Table 5.4: Best parameter sets given the parameter pool of Table 5.2

5.4 Acceptance Ratio Comparison: Linear Program versus Genetic Algorithm

This section compares the resulting acceptance ratio of the Linear Program and the Genetic Algorithm. Given a suitable parameter set (Section 5.3) and a precalculated pool of all paths (Section 5.2), the Genetic Algorithm performance can be considered with regard to the evolutionary process.
The Genetic Algorithm as well as the Linear Program were given the same graphs and demand sets as input. Generally, it has to be noted that the comparison is somehow unfair since the Genetic Algorithm is implemented in Python in the course of 3 months during this project, whereas Gurobi is compiled and highly optimized by a purpose specific company over the course of the last 6 years. A C++ implementation of the Genetic Algorithm could probably be around 10x faster [55] as comparisons of different programming languages for various problems show [56] [57]. For this reason, we let the Genetic Algorithm run 10x longer (excluding caching the gene pool, which can be done in advance) than the Linear Program. We let the Linear Program run first and stored the acceptance ratio of its solution with the corresponding runtime used. Afterwards, the Genetic Algorithm was given the same graph and demand set with a timeout of 10x the runtime of the Linear Program.
Figure 5.2 and 5.3 show the solution finding process of the Linear Program and the Genetic Algorithm. The performance of the Linear Program is plotted using two dashed lines, which indicate the acceptance ratio of its solution as well as the time needed for solving the problem. The colored lines connect partial solutions of the Genetic Algorithm that are produced after each iteration of the evolution, when a new generation has been created. In fact, we implemented the Genetic Algorithm such that it can utilize all eight cores on the test machine. Therefore we ran the evolution eight-fold in parallel as eight threads. For the analysis in this section we let the Genetic Algorithm run for six different parameter sets. The parameter sets are based on the best parameter set found for the corresponding scenario (Table 5.4) and varied in one parameter to demonstrate its impact on the performance. We chose to show the effect of the population size and non allocation probability as they are the most important. The population size determines the degree of parallelism in the evolution on the one hand. On the other hand, it limits the speed of the evolution as the number of operations per iteration increases by increasing the number of genomes per population. The non allocation probability can be seen as the 'tenacity' of the algorithm to embed a demand by a mutation process. As the non allocation probability increases, the probability of creating unfit solutions (solutions that can not be embedded because they exceed the bandwidth capacity of at least one link) decreases. However, also the speed of convergence is lowered by increasing the non allocation probability because new demands get allocated less frequently.
Each experimental run included the evolution process with eight threads running in parallel. Regarding the plots we only show the results of the best thread per run. The following subsections present these plots for the different topologies.

5.4.1 Butterfly Graph

In order to show the effect of different parameters we took the best possible parameter set for the butterfly graph (Table 5.4) and the corresponding demand set. For the plot we varied one parameter of the parameter set to show its impact on the performance. Figure 5.2 shows the effect of different non-allocation probabilities. It can be seen that a very high non allocation probability leads to weak solutions as the algorithm embeds fewer demands per iteration. In the extreme case, no demands are allocated at all when the non allocation probability is 100%. In addition, it can be seen that a too low non allocation probability is counterproductive as the algorithm tries to embed a lot of demands too hard. There seems to be an optimum around 10%.
In contrast, Figure 5.3 shows the effect of the population size. It is evident that small populations evolve faster as one iteration step takes less time because there are fewer operations to perform by the machine. The population size of 400 resulted in very long calculation times, that even the first generation takes nearly 100 seconds to end and it was not able to stop at 10x the time of the Linear Program. However, it is notable that the Genetic Algorithm came up with such a good solution in just one iteration due to parallelism.

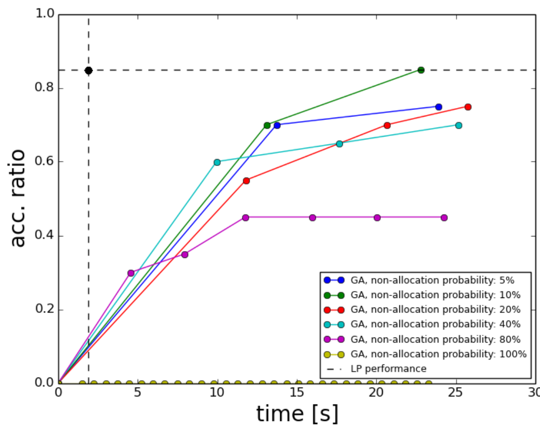

Figure 5.2: Acceptance ratio versus time, Linear Program and Genetic Algorithm (using a set of different non-allocation probabilities), solving the butterfly scenario with the corresponding loose demand set.

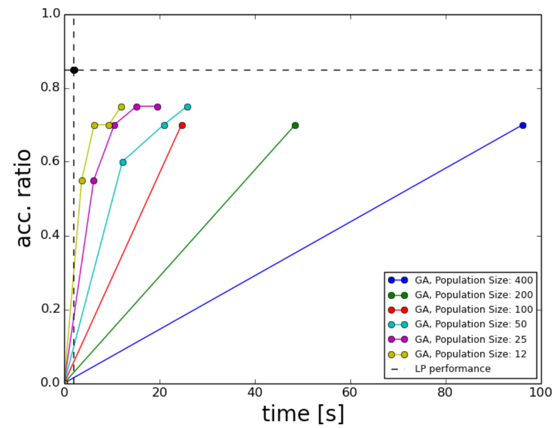

Figure 5.3: Acceptance ratio versus time, Linear Program and Genetic Algorithm (using a set of different population sizes), solving the butterfly scenario with the corresponding demand set.

5.4.2 Redbest Graph

Similar to the Figures 5.2 and 5.3 for the Butterfly graph, Figures 5.4 and 5.5 show the progress of the solution finding by the Linear Program and the Genetic Algorithm. Again, Figure 5.4 shows that there seems to be an optimum around 10% for the non allocation probability as trying to allocate too many demands per generation can be counterproductive regarding the acceptance ratio. On the other side a too low non allocation probability makes the convergence very slow, as demands are only embedded very rarely. Because the Genetic Algorithm had more time to run, since we set its timeout to 10x the time that the Linear Program used, the effect of smaller population sizes in Figure 5.5 is even more evident compared to the Figure 5.3 of the butterfly scenario. The smaller population sizes allow for more iterations as there are less operations per generation. This leads to a higher acceptance ratio of the solution by the timeout. In contrast, when the population size is extremely high, one iteration takes very long. This led to the fact, that the Genetic Algorithm was not even able to stop within a reasonable amount of time after the timeout (e.g. in Figure 5.5 for population size 200) since it can only stop after an iteration.

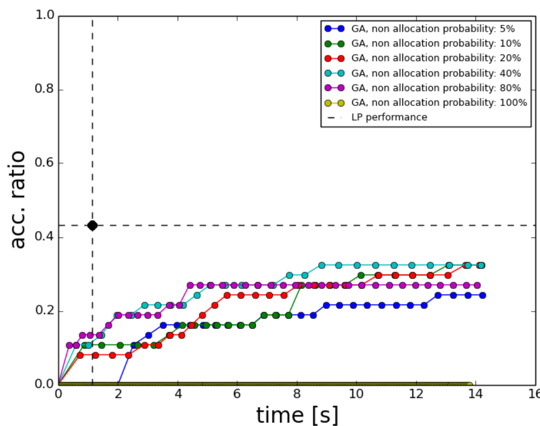

Figure 5.4: Acceptance ratio versus time, Linear Program and Genetic Algorithm (using a set of different non-allocation probabilities), solving the redbest scenario with the corresponding loose demand set.

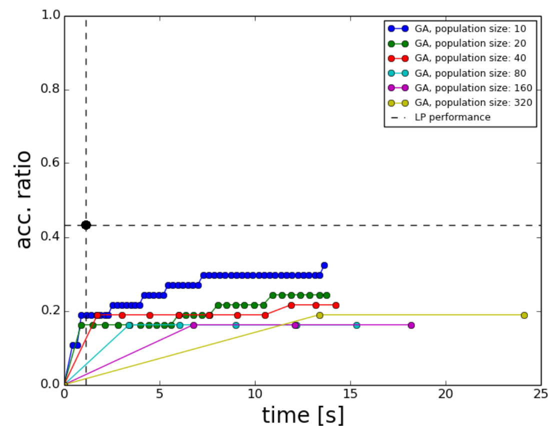

Figure 5.5: Acceptance ratio versus time, Linear Program and Genetic Algorithm (using a set of different population sizes), solving the redbest scenario with the corresponding loose demand set.

5.4.3 Cogent Graph

The fundamental relations between the parameters and the performance of the Genetic Algorithm explained in Sections 5.4.1 and 5.4.2 remain the same for the cogent graph scenario. However, it can be seen that the Genetic Algorithm could not reach the same region of acceptance ratio within the time given. One possible reason for this is that the search space increases a lot when the graph size increases and there are much more possible paths in the gene pool to choose from. In addition, it takes more effort by the genetic algorithm to determine the fitness of each genome in large graphs as there are more edges to check. We note that the Linear Program still finds the best possible solution within a reasonably short amount of time.

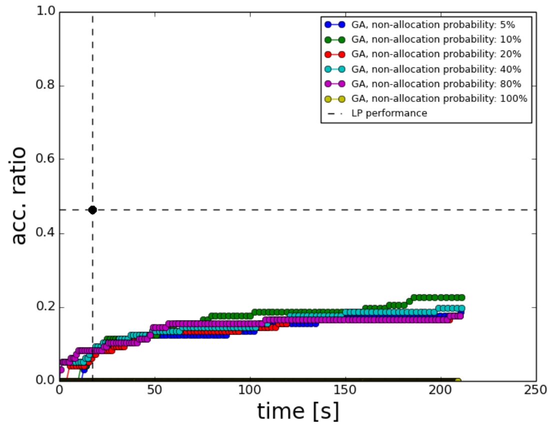

Figure 5.6: Acceptance ratio versus time, Linear Program and Genetic Algorithm (using a set of different non-allocation probabilities), solving the cogent scenario with the corresponding loose demand set.

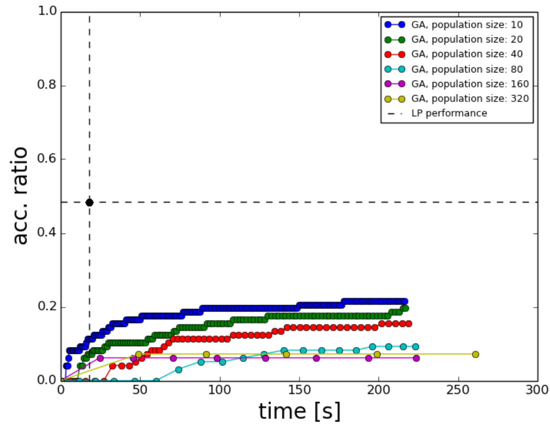

Figure 5.7: Acceptance ratio versus time, Linear Program and Genetic Algorithm (using a set of different population sizes), solving the cogent scenario with the corresponding loose demand set.

5.5 Genetic Algorithm: Robustness

This section presents the performance of the Genetic Algorithm with respect to parameter robustness. Figures 5.8 and 5.9 show the acceptance ratio of the Genetic Algorithm for the butterfly graph scenario. The Genetic algorithm was given 10x the time that the Linear Program needed for the same task. The parameters of the Genetic Algorithms were set to the optimal values found by brute forcing (Table 5.4). Although, one parameter is varied to show its impact on the performance. In order to factor in the randomness of the Genetic Algorithm we repeated the experiments 10 times and created boxplots.
Figure 5.8 shows a boxplot of the acceptance ratio versus the non allocation probability. The non allocation probability is introduced in Section 3.3.3 and determines how likely a mutation process ends up with a path choice as opposed to 'None', which would mean that the demand is not allocated. It can be seen that the Genetic Algorithm is reasonably robust within 5% to 20% non allocation probability but significantly drops for higher values, as the algorithm then tries to embed demands too rarely.
In contrast, the Genetic algorithm shows a more robust performance with respect to population size as Figure 5.9 demonstrates. The reason for this lies most probably in the fact that the higher degree of parallelism with large populations compensates for the additional computation time per generation. Furthermore, we note that for some parameters the performance is very constant over several runs such that the boxplot does only show very little variance in the acceptance ratio.

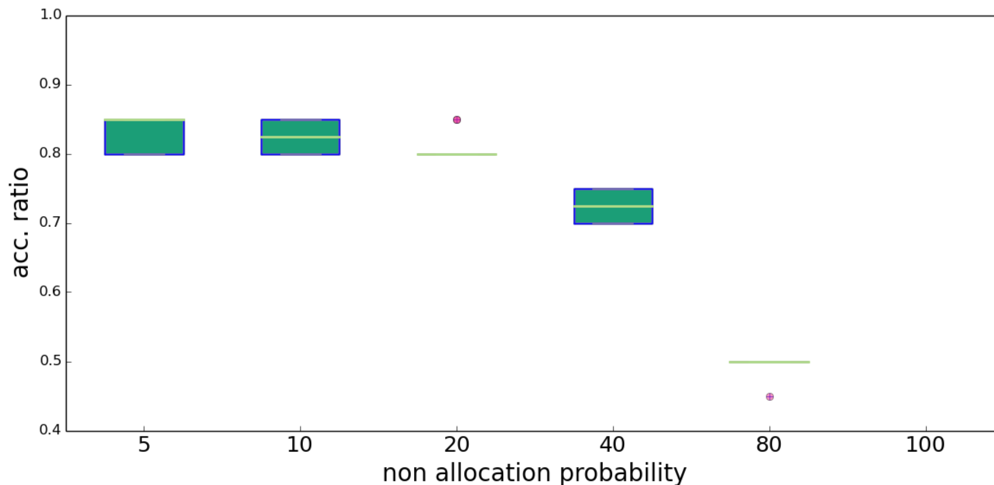

Figure 5.8: Genetic Algorithm robustness against non allocation probability

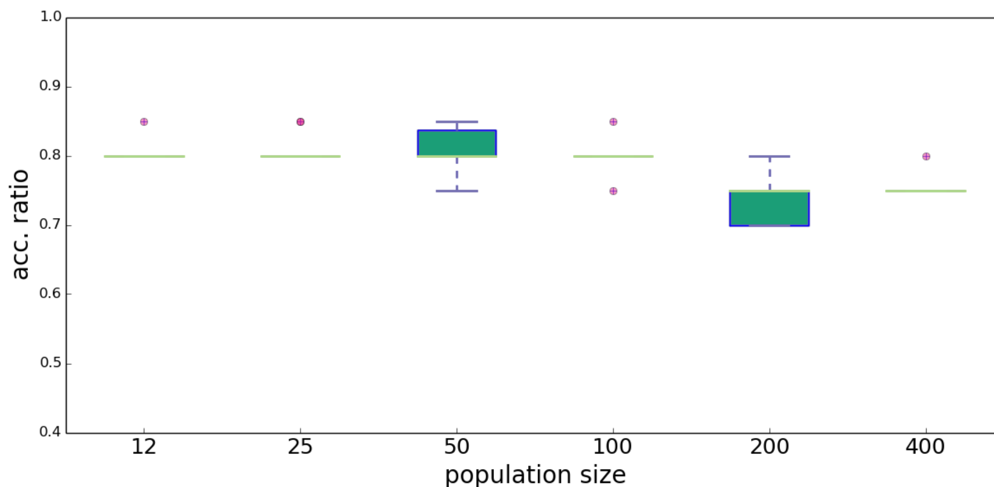

Figure 5.9: Genetic Algorithm robustness against population size

5.6 Linear Program versus Online Algorithm

This section compares the performance of the Linear Program versus the Online Algorithm. To show the advantages and disadvantages of the algorithms, we let the Linear Program and Online Algorithm solve the same scenario and compare the acceptance ratio and solving time needed for the solution. As the Online Algorithm was designed to allocate demands fast, we show a version with and a version without the reallocation mechanism, since the reallocation mechanism is an extension to the Online Algorithm. The reallocation mechanism is described in Section 3.3.1. This mechanism tries to reallocate previously allocated demands to reorganize resources such that a new demand can be embedded although there was no feasible path initially due to occupied links. To achieve this, the Online Algorithm let the Linear Program solve a sub-problem and find a way to reallocate previously allocated demands and embed the new demand at the same time. The size of this sub-problem is set to 10 demands in this evaluation. However, this parameter can be adjusted. A bigger sub-problem size increases the probability to reach a higher acceptance ratio. Yet, it introduces more overhead by the reallocation mechanism and therefore increases the solving time. In fact, the reallocation mechanism improves the acceptance ratio performance of the algorithm. However, the reallocation is to the disadvantage of the solving time as it introduces a lot of overhead, which contradicts the concept of Online Algorithms (Section 3.2.2).
We let each algorithm solve three scenarios sequentially and independently and plot the time needed for solving and the acceptance ratio of the solution. Figures 5.10, 5.11 and 5.12 indicate the performance of the Linear Program by circles, the performance of the Online Algorithm without the reallocation mechanism by triangles and the performance of the Online Algorithm with the reallocation mechanism by squares. As the performance of the Linear Program is not dependent on the order of the demands it was run once per scenario. However, to take into account that the embedding of the Online Algorithm depends on the order of the demands, we shuffled the demands and plotted the results for 10 runs per scenario.

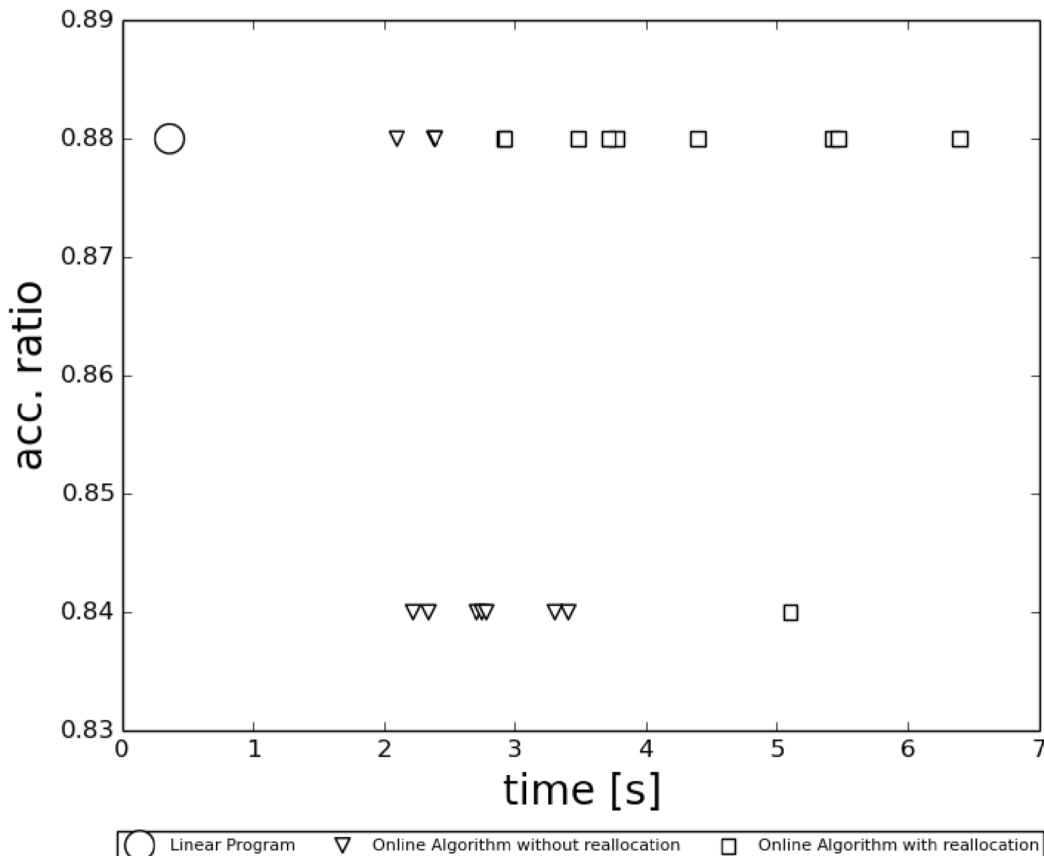

Figure 5.10: Acceptance ratio vs time for the Online (with and w/o reallocation mechanism) versus the LP on the switch graph with the corresponding 'loose' demand set

Figure 5.10 shows the performance evaluation of the three algorithm for the switch graph and the corresponding loose demand set. It demonstrates that the Linear Program is at least 5 times faster than the Online Algorithm in this scenario. Nevertheless, the Online Algorithm manages to get to the same acceptance ratio as the Linear Program in 12 cases whereas only 8 of the 20 in total ended up with lower acceptance ratio. Generally, we note that the Online Algorithm without reallocation performs worse but is tendentially faster. An important insight is that the Online Algorithm with reallocation does not always perform better than the version without reallocation. The reason for this could be that the order of the demands is such that the best acceptance ratio can also be achieved without reallocation. It can even happen that the Online Algorithm with reallocation performs worse than the Online Algorithm without reallocation. This is the worst case since the extra effort of the reallocation mechanism results in a longer calculation time and a rearrangement of network resources, which by accident deteriorates the embedding regarding the resulting overall acceptance ratio.

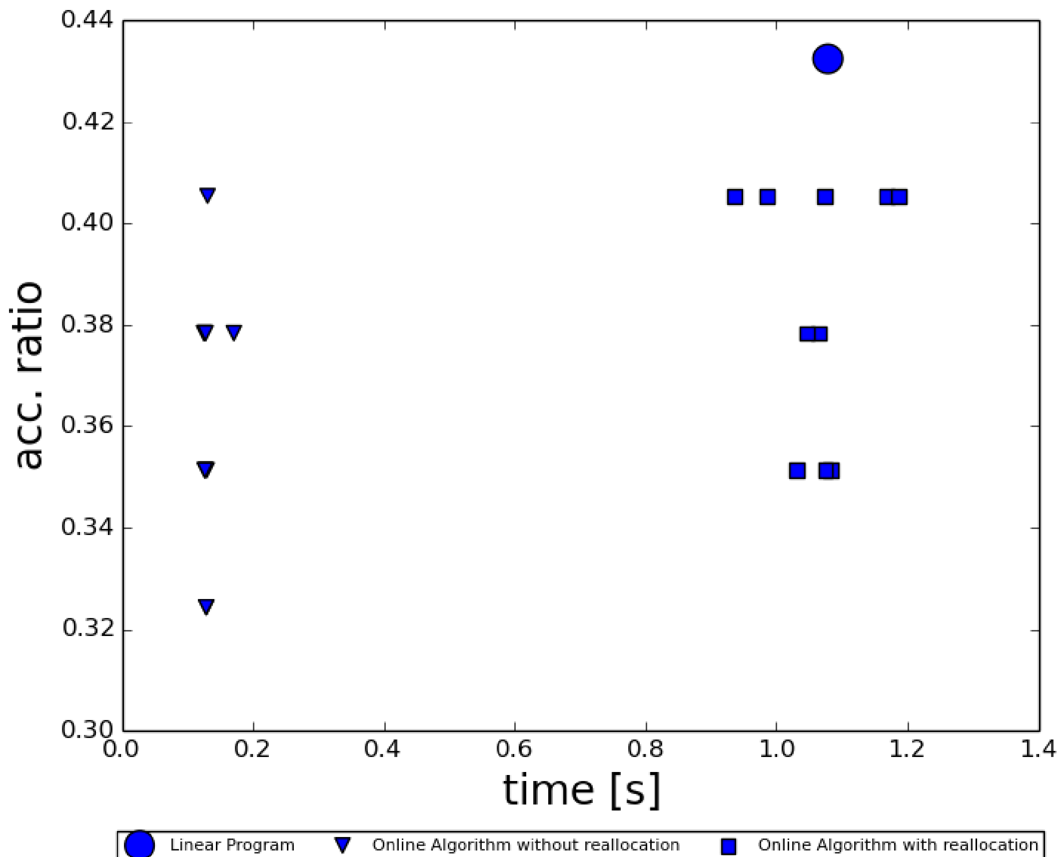

Figure 5.11: Acceptance ratio vs time for the Online (with and w/o reallocation mechanism) versus the LP on the redbest graph with the corresponding 'loose' demand set

Figure 5.11 presents the performance evaluation of the two algorithm for the redbest graph and the corresponding loose demand set. It can be seen that each algorithm performs in a different region. The Online Algorithm without reallocation is at least 5 times faster than the Linear Program or the Online Algorithm with reallocation. The reallocation mechanism produces a slightly better averaging acceptance ratio. However, the Online Algorithms could not reach the optimal acceptance ratio of the Linear Program. On the other hand the Online Algorithm was no more slower than the Linear Program like in Figure 5.10.

As for the cogent graph and the corresponding loose demand set, Figure 5.12 depicts the performance of the three algorithms. As in Figure 5.11 the Online Algorithm without allocation was fastest but the Online Algorithm with allocation shows a higher average acceptance ratio closer to the solution by the Linear Program. However, it was not possible for the Online Algorithms to reach the best allocation ratio determined by the Linear Program. In addition, the plot shows that the Online Algorithm with reallocation is even slower than the Linear Program, which contradicts

the concept of Online Algorithms (Section 3.2.2). The reason behind this behaviour is that the overhead of creating a sub-problem and let it solve by the reallocation mechanism increases a lot with graph size.

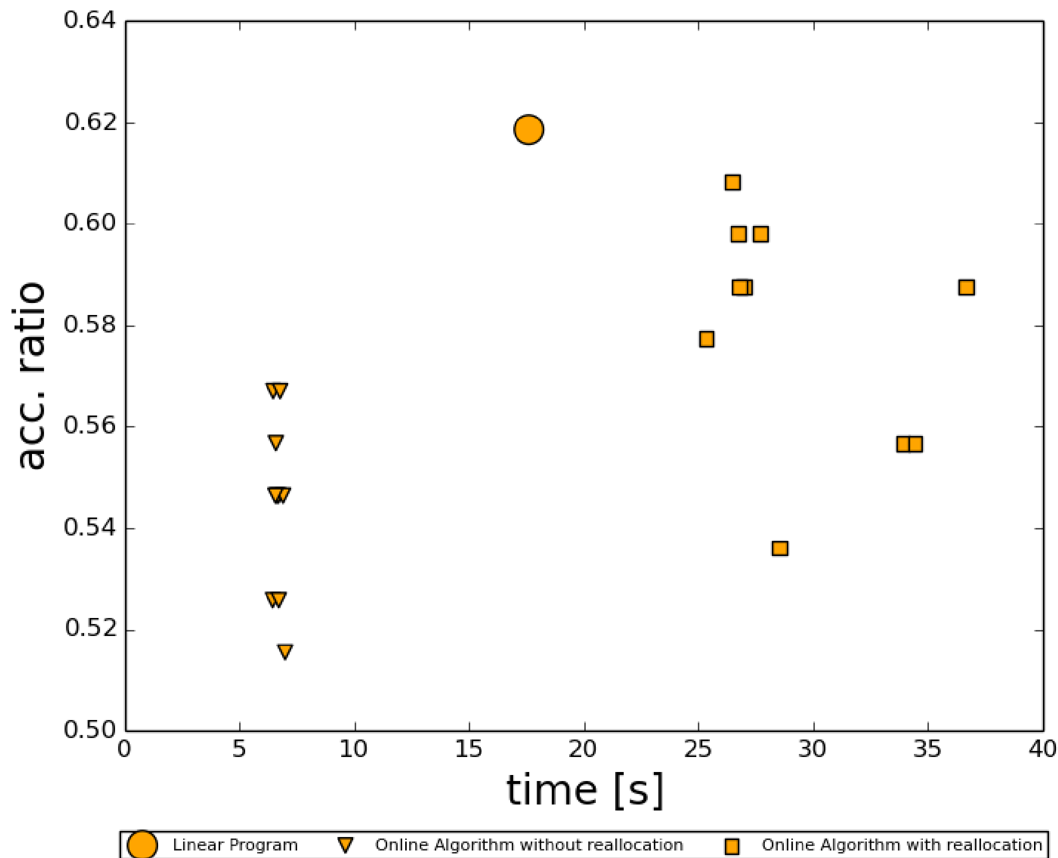

Figure 5.12: Acceptance ratio vs time for the Online (with and w/o reallocation mechanism) versus the LP on the cogent graph with the corresponding 'loose' demand set

5.7 Conclusion

This section condenses the results found in Chapter 5.
The Linear Program does get slower as the graph size increases, yet, it always returned a solution in our experiments independently from the graph or demand set characteristics. Also, for most cases it appears to be much more performant compared to the other algorithms of this project.
Regarding the Genetic Algorithm, it is not trivial at all to tune the parameters such that it performs optimally in every case. Also, for graphs and demand sets, which do not allow an acceptance ratio of 100%, there is no obvious break condition for the Genetic Algorithm to stop the evolution and return the solution. The most substantial disadvantage is the pronounced bottleneck of finding all paths. Yet, this problem can be addressed by finding and caching all possible paths in advance. Besides the disadvantages of the Genetic Algorithm, there is a strong advantage. In contrast to the Linear Program, the Genetic Algorithm is able to produce partial solutions (acceptance ratio not yet maximized but greater than zero) when an embedding is needed within a very short amount of time. In a large problem it is well possible, that the Genetic Algorithm has found a first solution within a couple of generations, when the Linear Program has not finished calculating yet. As it turned out, the Genetic Algorithm performed similar (when considering that a C++ implementation could be up to 10x faster) to the Linear Program in the case of the butterfly graph (Figure 5.2).
The butterfly graph comprises distinctive bottlenecks in the graph by connecting four SRG subgraphs by only a few links. These bottleneck links must be shared optimally by multiple de-

mands, without exceeding the capacity of each link. Actually, this is a sub-case of the knapsack problem, which is NP-Hard [52]. During our experiments we also validated related literature that showed that Genetic Algorithms are particularly efficient in solving the knapsack problem [53]. Therefore, it would be promising to further optimize the Genetic Algorithm implementation and parameter set.

By comparing the Linear Program and the Online Algorithm we could show that the Online Algorithm can perform close to the Linear Program by a few percent whereas the Linear Program always found the best solution in our scenarios. Yet, the reallocation mechanism helped (e.g. in Figures 5.11 and 5.12) to improve the acceptance ratio of the Online Algorithm. However, this is to the disadvantage of the runtime performance as the reallocation mechanism introduces a lot of overhead. Another important insight is the fact that the order of demands has a great effect on the performance of the Online Algorithm.

Chapter 6

Summary and Outlook

6.1 Summary

The main goal of this thesis was to solve the problem of optimal flow embedding for Audio over IP networks. The case was motivated by the lack of network management for the emerging Audio over IP Standard AES67 [10]. By network management we denote the optimal flow embedding of audio streams in a network. Each audio stream is represented by a demand carrying the information on the source, target, bandwidth requirement and latency requirement of the stream. The aim of the network management is to satisfy bandwidth and latency requirements on a per flow basis while maximizing the acceptance ratio. The acceptance ratio is defined as $\frac{\text{number of embedded demands}}{\text{total number of demands}}$.

To use such algorithms we propose an SDN-enabled Audio over IP network Software Defined Audio Networking (SDAN) with a SDN controller on top. The controller provides a global view on the network's topology by abstracting switches and links to nodes and edges of a graph. Given the graph representing the network and a set of demands, an SDAN algorithm can search for the best possible embedding.

In this thesis we defined two classes (online and offline) of SDAN algorithms for two typical types of scenarios. Online Algorithms handle scenarios where demands are unknown in advance and a feasible path has to be found fast and on the fly. However, the embedding is not optimal, which limits the achievable acceptance ratio. Offline Algorithms on the other hand deal with scenarios when a set of demands is known in advance and search for the best possible embedding for as much demands as possible. The benefit of this approach is an optimal embedding, yet the runtime of the algorithms scales poorly with the problem size.

We implemented an Online Algorithm as well as two Offline Algorithms in Python. The first offline algorithm is the implementation of a Mixed Integer Linear Program. Linear Programming is a common approach for solving flow embedding optimization problems. However, solving a Mixed Integer Linear Program is NP-hard and the time to solve a problem increases rapidly with the graph size. We had indications that a Genetic Algorithm might be applicable in this situation. The use of Genetic Algorithms is reasonable in cases where a straightforward solution is impossible or too complex to derive and implement. NP-hard problems as the Mixed Integer Linear Program fall into this case [35]. In addition, we had strong evidence from related work, that Genetic Algorithms can yield benefits to problems of similar nature [36] [37] [49]. We implemented a Genetic Algorithm from scratch in Python. It is essential to note that the Genetic Algorithm could be up to 10x faster if it was implemented in a more efficient language (e.g. C++) as comparisons of different programming languages for various problems show [56] [57] [55]. The disadvantage of the Genetic Algorithm is that there are parameters that have to be tuned per graph and demand set.

In order to evaluate the algorithm we created a set of scenarios. A scenario is comprised of a graph and a demand set. As network graphs we had the top level network graph provided by the SRG (Swiss Broadcasting Corporation) as well as four graphs from the Internet Topology

Zoo [54], which are real topologies of wide area networks. As it turned out, the creation of reasonable demand sets is not trivial as the number of demands as well as the bandwidth and latency requirements influence the best achievable acceptance ratio. Eventually we tuned parameters by experimentation to get a strict (allowing an acceptance ratio around 60%) and a loose demand set (allowing an acceptance ratio around 100%) for each graph.
With the help of these scenarios we could compare the runtime and acceptance ratio performance of each algorithm. We found that for some scenarios the Genetic Algorithm (given optimal parameters) can perform in the same acceptance ratio range as the Linear Program, when taking into account that it would be 10x faster if implemented in C++. In addition, the Genetic Algorithm can deliver partial solutions during the process of solving, which is a great advantage of the Genetic Algorithm. We also analysed the parameter robustness of the Genetic Algorithm and found that it is reasonably robust against population size whereas the non allocation probability in a mutation process has a great impact on the performance. However, the mapping of the SDAN problem to the Genetic Algorithm revealed a bottleneck. For running the Genetic Algorithm, a gene pool of all possible paths per demand has to be precalculated. For smaller graphs this is not an issue (as it takes a couple of minutes) whereas it can take several weeks for graphs with over thousand nodes.

As far as the Online Algorithm is concerned, it performs slower than the Linear Program for small graphs but faster for large graphs. Yet, the larger the graph the lower the acceptance ratio (a few percent less) of the Online Algorithm compared to the Linear Program. Yet, it performs in the same acceptance ratio range, too.

6.2 Outlook

6.2.1 System Aspect

The concept of SDAN is explained in Section 3.1. This thesis focuses on the algorithmic part of an SDAN controller. However, to make use of the algorithms in a real SDN-enabled network, they have to be integrated on the northbound interface of an SDN controller framework (e.g. POX [14] or Floodlight [15]). The integration includes the following essential components:

1. One component is responsible for providing a network overview by recognizing all switches and links, convert this information to a graph and feed it to the algorithm.
2. Another component has to convert the output of the algorithm (a path as a node list per embedded demand) to flow rules for each switch and push them to the switches by the southbound interface (e.g. OpenFlow [13]).
3. An interface for an end-user or a professional audio mixing console allowing the generation of demands. The same interface should also allow the deletion of demands and flow rules in order to release network resources.
4. A module that manages safety margins on the network lines to assure enough capacity for the streams (AVB for example does not load links more than 75% [61] [62]).
5. Eventually networking monitoring is inevitable and has to be integrated in the controller to ensure optimal network performance and recognize potential bandwidth over-consumption by streams.

A test network of SDN-enable switches (e.g. Open vSwitches [60]) and line simulators (e.g. Wanem [59] to simulate bandwidth capacity, latency, jitter and loss on a line) could be set up. Professional AOIP devices could then be attached to the test network to carry out Quality of Service and Quality of Experience experiments allowing a performance evaluation of the whole SDAN system.

6.2.2 Multicast

As it is common in audio broadcasting and during live events to have a signals being consumed at several end nodes in the network (e.g. the ambiance audio mix of a sports event that captures fan chants and acclamations is consumed by numerous TV broadcasters, which then mix their sports commentator's voice to it), it is essential to use multicast in order to minimize the network load per stream. In order to achieve this, the algorithms have to be adopted for multicast (e.g. the work of Ciro A. Noronha et al. [63]) and the SDN controller has to provide SDN-multicast capabilities [64].

6.2.3 Demand Grouping

In professional audio production it is prevalent that some audio streams logically belong together, for example all the microphone channels of a drum set. As all the streams have to be in sync at the end node it is reasonable to route them on the same path though the network. This is implementing demand grouping could be beneficial in such scenarios.

6.2.4 Fail-over Paths

As all the ongoing streams and the current topology is known by the controller, fail-over paths can be calculated and pushed to the switches in advance to provide a higher system reliability in case of a link or switch failure.
We suggest the approach of calculating two distinct paths (minimizing the number of common edges) for each demand, which are converted to flow rules and pushed to the switches. By this, a main stream and a backup stream can run concurrently. However, one could also choose to just install the flow rules of the fail-over path at the beginning (without actually having a backup stream running) and only switch to it, when the main path breaks. By this, less network capacity is occupied, yet the reaction time in case of failure increases, since the link failure has to be recognized and the backup stream has to be initiated first.

6.2.5 Number Of Channels Per Stream

As each stream introduces some overhead traffic in the network it is reasonable to have multiple audio channels per stream. In AES67 [10] one single stream can carry multiple audio channels. Depending on the sampling rate and packet time, there is a certain maximum number of channels defined in the standard. When a certain amount of audio channels has to be transported over a network, the question is, how many streams are needed and which channels should be embedded in each of them. Future work could therefore investigate on the optimal grouping of audio channels into streams such that optimal performance of the streams is guaranteed and the network load per stream is minimized.

6.2.6 QoS Applications

The SDAN algorithm can of course also be used for general QoS applications. With the help of rate limiters in each device on the network (because bulk traffic is bursty and not constant regarding bandwidth consumption like streams) it would be possible to build an SDN-enabled network, where each node of the network is given a link to other nodes with a fixed well-defined minimal bandwidth and maximum latency. This feature could be used to embed office or bulk traffic in real time media networks.

6.2.7 Genetic Algorithm Refinements

GA Parameter Tuning

One of the big challenges when using Genetic Algorithms is to find parameters such that the algorithm converges fast. A method to find these parameters is a second level Genetic Algorithm, which is described in work by John J. Grefenstette [41].

PyPy

In order to make the Genetic Algorithm faster it should be written in C++ or PyPy [65].

Implement Crossover

Crossover is, besides mutation, an important element in Genetic Algorithms [39]. Due to time constraints, the implementation of this thesis does not involve crossover but it could be considered in future work.

Multi-thread Multi-parameter Search

Another idea to accelerate the process of evolution is to instantiate several threads that run the same Genetic Algorithm whereas each Genetic Algorithm instance is running on different parameters. Since different parameters lead to different performance, the probability of finding the best solution faster increases when running different Genetic Algorithm instances on different parameters in parallel.

Tabu Search

Another possibility to speed up the Genetic Algorithm is the introduction of Tabu search [39]. This technique avoids the same solution candidate being visited multiple times by accident and therefore saves time in the process of optimization.

Multi-objective Optimization

SPEA [43] [44] and NSGA [42] are Genetic Algorithms for multi objective optimization, but the SDAN algorithm just optimizes for one single objective (maximizing acceptance ratio). Although SPEA and NSGA do not precisely match the task of an SDAN algorithm, it could be investigated on the performance of them with an additional secondary objective (minimizing bandwidth usage on each link).

Bibliography

[1] STUDER by HARMAN,
http://www.studer.ch/,
Date of the last visit: 9th of February 2015

[2] Wikipedia, "Super Bowl",
http://en.wikipedia.org/wiki/Super_Bowl,
Date of the last visit: 9th of February 2015

[3] Wikipedia: "MADI",
http://en.wikipedia.org/wiki/MADI,
Date of the last visit: 9th of February 2015

[4] S. Church, S. Pizzi,
"Audio Over IP: A Practical Guide to Building Studios with IP, including VoIP and Livewire"
Focal Pr, November 2009, ISBN-10: 0240812441, ISBN-13: 978-0240812441

[5] Audinate,
"Dante Audio Networking",
https://www.audinate.com/,
Date of the last visit: 3th of March 2015

[6] Wheatnet-IP,
"Technology Overview",
http://wheatstone.com/index.php/wheatnet-ip-technology-overview-corporate
Date of the last visit: 3th of March 2015

[7] A. Hildebrand,
"AES67 & beyond, Presentation at Broadcast Engineering Conference",
NAB, 2014

[8] Audio Engineering Society (AES),
"Best Practices in Network Audio" White Paper,
AESTD1003V1, 2009
http://www.aes.org/technical/documents/AESTD1003V1.pdf,
Date of the last visit: 9th of February 2015

[9] ALC Networx,
"Latency RAVENNA Features",
http://ravenna.alcnetworx.com/technology/features/latency.html,
Date of the last visit: 9th of February 2015

[10] Audio Engineering Society (AES),
"AES standard for audio applications of networks -
High-performance streaming audio-over-IP interoperability",
AES67-2013, Standards and Information Documents, 2013

[11] ALC NetworX GmbH,
"Ravenna V 1.0, White Paper",
http://ravenna.alcnetworx.com/infos-downloads/white-papers.html,
Date of the last visit: 9th of February 2015

[12] N. Feamster, J. Rexford, E. Zegura,
"The Road to SDN - An intellectual history of programmable networks",
Georgia Institute of Technology and Princeton University,
Acmqueue, Volume 11, issue 12, December 30, 2013,
http://queue.acm.org/detail.cfm?id=2560327,
Date of the last visit: 9th of February 2015

[13] N. McKeown,
"OpenFlow: enabling innovation in campus networks",
ACM SIGCOMM Computer Communication Review 38.2 (2008): 69-74.

[14] POX,
"SDN Controller",
http://www.noxrepo.org/pox/about-pox/,
Date of the last visit: 3th of March 2015

[15] Project Floodlight,
"Open Source Software for Building Software-Defined Networks",
http://www.projectfloodlight.org/floodlight/,
Date of the last visit: 3th of March 2015

[16] networkstatic.net,
"The Northbound API - A Big Little Problem",
http://networkstatic.net/the-northbound-api-2/,
Date of the last visit: 3th of March 2015

[17] Institute of Electrical and Electronics Engineers (IEEE),
"IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems",
1588-2002, IEEE Standard, 2002
http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1048550
Date of the last visit: 3th of March 2015

[18] The Internet Engineering Task Force (IETF),
"Zero Configuration Networking (zeroconf)",
https://datatracker.ietf.org/wg/zeroconf/charter/
Date of the last visit: 3th of March 2015

[19] The Internet Engineering Task Force (IETF),
"Session Announcement Protocol",
RFC2974, October 2000
http://tools.ietf.org/html/rfc2974
Date of the last visit: 3th of March 2015

[20] Wikipedia, "AES67",
http://en.wikipedia.org/wiki/AES67,
Date of the last visit: 9th of February 2015

[21] C. Frei, EPFL,
"Abstraction techniques for resource allocation in communication networks",
PhD These Nr. 2144,
http://infoscience.epfl.ch/record/32620

[22] C. Frei, B. Faltings, M. Hamdi,
"Resource Allocation in Communication Networks Using
Abstraction and Constraint Satisfaction",
IEEE Journal on Selected Areas in Communication, Vol 23, p. 304-320, February 2005,
http://liawww.epfl.ch/Publications/Archive/Frei2005.pdf,
Date of the last visit: 9th of February 2015

[23] Audio Video Bridging Task Group,
http://www.ieee802.org/1/pages/avbridges.html,
Date of the last visit: 9th of February 2015

[24] M. J. Teener, AVnu Alliance,
"No-excuses Audio/Video Networking:
The Technology Behind AVnu" White Paper, 8th of August 2009,
http://avnu.org/wp-content/uploads/2014/05/No-excuses-Audio-Video-Networking-v2.pdf,
Date of the last visit: 9th of February 2015

[25] Wikipedia, "Time-Sensitive Networking",
http://en.wikipedia.org/wiki/Time-Sensitive_Networking,
Date of the last visit: 9th of February 2015

[26] time sensitive networks,
http://www.802tsn.org/,
Date of the last visit: 9th of February 2015

[27] L-S-B,
Website "VSM - Broadcast Control and Monitoring System",
http://www.l-s-b.de/en/products/vsm.html,
Date of the last visit: 9th of February 2015

[28] L-S-B,
Brochure "VSM - Broadcast Control and Monitoring System",
http://www.l-s-b.de/fileadmin/content/LSB_NEU/Marketing/Brochures/Flyer_VSM_2014.pdf,
Date of the last visit: 9th of February 2015

[29] Wikipedia: "Linear programming",
http://en.wikipedia.org/wiki/Linear_programming,
Date of the last visit: 9th of February 2015

[30] lpsolve,
http://sourceforge.net/projects/lpsolve/,
Date of the last visit: 9th of February 2015

[31] Gurobi Optimization,
http://www.gurobi.com/,
Date of the last visit: 9th of February 2015

[32] A. Bley,
"An Integer Programming Algorithm for Routing Optimization in IP Networks",
Zuse Institute Berlin, Algorithms - ESA 2008, Lecture Notes in Computer Science, p. 198-209, Volume 5193, 2008,
http://link.springer.com/chapter/10.1007%2F978-3-540-87744-8_17#page-1,
Date of the last visit: 24th of February 2015

[33] V. Kotronis, R. Kloeti, M. Rost, P. Georgopoulos, B. Ager, S. Schmid, X. Dimitropoulos,
"Investigating the Potential of the Inter-IXP Multigraph for the Provisioning
of Guaranteed End-to-End Services",
ETH Zurich, Laboratory TIK, Nr. 360, February, 2015

[34] M. Juenger,
"50 Years of Integer Programming 1958-2008,
From the Early Years to the State-of-the-Art",
Springer-Verlag Berlin Heidelberg, p. 345, 2010
http://www.springer.com/gp/book/9783540682745 Date of the last visit: 3th of March 2015

[35] R. Fabregat and Y. Donoso,
"Multi-objective optimization in computer networks using metaheuristics",
CRC Press, 2007,
http://www.crcnetbase.com/isbn/9780849380846,
Date of the last visit: 3th of March 2015

[36] B. Gonen,
"Genetic Algorithm Finding the Shortest Path in Networks",
Department of Computer Science and Engineering, University of Nevada, Reno, Reno,

Nevada, U.S.A,
http://weblidi.info.unlp.edu.ar/worldcomp2011-mirror/GEM3983.pdf,
Date of the last visit: 9th of February 2015

[37] S. Behzadi, A. Alesheikh,
"Developing a Genetic Algorithm for Solving Shortest Path Problem",
WSEAS International Conference on Urban Planning and Transportation, Heraklion, Crete Island, Greece, July 22-24, 2008,
http://www.wseas.us/e-library/conferences/2008/crete/upt/upt04.pdf,
Date of the last visit: 9th of February 2015

[38] R. Kumar, M. Kumar,
"Exploring Genetic Algorithm for Shortest Path Optimization in Data Networks",
Global Journal of Computer Science and Technology, Vol. 10, Issue 11 (Ver. 1.0), p. 8-12, October 2010,
http://globaljournals.org/GJCST_Volume10/2-Exploring-Genetic-Algorithm-for-Shortest-Path-Optimization-in-Data-Networks.pdf,
Date of the last visit: 9th of February 2015

[39] S. Luke,
"Essentials of Metaheuristics - A Set of Undergraduate Lecture Notes",
Department of Computer Science, George Mason University,
Second Edition, Online Version 2.1, October 2014,
http://cs.gmu.edu/ sean/book/metaheuristics/Essentials.pdf

[40] J. Holland,
"Adaptive algorithms for discovering and using general patterns
in growing knowledge bases",
International Journal of Policy Analysis and Information Systems, 4(3), p. 245-268, 1980

[41] J. Grefenstette,
"Optimization of Control Parameters for Genetic Algorithms",
IEEE Transactions on Systems, Man, and Cybernetics,
VOL. SMC-16, No, 1, January/February 1986,
http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4075583,
Date of the last visit: 9th of February 2015

[42] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan,
"A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II",
IEEE Transactions On Evolutionary Computation, VOL. 6, No. 2, APRIL 2002,
http://www.iitk.ac.in/kangal/Deb_NSGA-II.pdf,
Date of the last visit: 9th of February 2015

[43] E. Zitzler, L. Thiele,
"An Evolutionary Algorithm for Multiobjective Optimization:
The Strength Pareto Approach",
TIK-Report, No. 43, May 1998,
http://www.tik.ee.ethz.ch/sop/publicationListFiles/zt1998a.pdf,
Date of the last visit: 9th of February 2015

[44] E. Zitzler, ETH TIK,
"Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications",
Diss. ETH No. 13398, November the 11th, 1999,
http://www.tik.ee.ethz.ch/sop/publicationListFiles/zitz1999a.pdf

[45] NetworkX,
https://networkx.github.io/,
Date of the last visit: 9th of February 2015

[46] Graphviz,
http://www.graphviz.org/,
Date of the last visit: 9th of February 2015

[47] R. Sedgewick,
"Algorithms in C, Part 5: Graph Algorithms",
Addison Wesley Professional, 3rd ed., 2001

[48] I. Rechenberg,
"Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution",
Fromman-Holzbook, Stuttgart, Germany, 1973

[49] Tobias Siegfried,
"Multiobjective Groundwater Management Using Evolutionary Algorithms".
IEEE Transactions On Evolutionary Computation, Vol. 13, No. 2, April 2009
http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4540060,
Date of the last visit: 9th of February 2015

[50] Eckart Zitzler,
"An Evolutionary Algorithm for the Block Stacking Problem",
Computer Engineering and Networks Laboratory (TIK), ETH Zurich.
http://www.tik.ee.ethz.ch/file/51778347a40443e27115d11aca1a8baa/hegb2008a.pdf,
Date of the last visit: 9th of February 2015

[51] SRG,
"Schweizerische Radio- und Fernsehgesellschaft",
http://www.srgssr.ch/,
Date of the last visit: 9th of February 2015

[52] A. Drexl,
"A Simulated Annealing Approach to the Multiconstraint Zero-one Knapsack Problem",
Computing, January 1988,
http://dl.acm.org/citation.cfm?id=46827,
Date of the last visit: 9th of February 2015

[53] F. Lin,
"Applying the Genetic Approach to Simulated Annealing in Solving Some NP-Hard Problems",
IEEE Transactions On Systems, Man, And Cybernetics, Vol. 23. No. 6, November December 1993,
http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=257766,
Date of the last visit: 9th of February 2015

[54] The Internet Topology Zoo,
http://www.topology-zoo.org/,
Date of the last visit: 9th of February 2015

[55] Python Software Foundation,
"Comparing Python to Other Languages",
https://www.python.org/doc/essays/comparisons/,
Date of the last visit: 23th of February 2015

[56] S. B. Aruoba, J. Fernandez-Villaverdez,
"A Comparison of Programming Languages in Economics",
University of Maryland, University of Pennsylvania, August 5, 2014,
http://economics.sas.upenn.edu/ jesusfv/comparison_languages.pdf,
Date of the last visit: 23th of February 2015

[57] I. Zahariev,
"C++ vs. Python vs. Perl vs. PHP performance benchmark",
/contrib/famzah, July 1, 2010,
http://blog.famzah.net/2010/07/01/cpp-vs-python-vs-perl-vs-php-performance-benchmark/,
Date of the last visit: 23th of February 2015

[58] NetworkX all_simple_paths,
http://networkx.github.io/documentation/networkx-1.9/reference/generated/networkx.algorithms.simple_paths
Date of the last visit: 9th of February 2015

[59] WANem,
"The Wide Area Network emulator",
http://wanem.sourceforge.net,
Date of the last visit: 3th of March 2015

[60] Open vSwitch,
"Open vSwitch, a production quality, multilayer virtual switch licensed under the open source Apache 2.0 license.",
http://openvswitch.org/,
Date of the last visit: 3th of March 2015

[61] D. Pannell,
"Audio Video Bridging Audio Video Bridging - AVB Assumptions",
IEEE 802.1 AVB Plenary, Sept 4, 2007, Stockholm,
http://www.ieee802.org/1/files/public/docs2007/avb-pannell-assumptions-0907-v8.pdf,
Date of the last visit: 3th of March 2015

[62] Biamp Systems,
"AVB Resource Guide - Covering the Basics of AVB",
http://c353616.r16.cf1.rackcdn.com/Biamp_AVB_Reference_Guide.pdf,
Date of the last visit: 3th of March 2015

[63] C. A. Noronha,
"Optimum routing of multicast audio and video streams in communications networks ",
Standford University, Technical Report: CSL-TR-94-618, 1994
http://i.stanford.edu/pub/cstr/reports/csl/tr/94/618/CSL-TR-94-618.ps,
Date of the last visit: 3th of March 2015

[64] P. Leu,
"SDN-assisted IP Multicast",
ETH, TIK, Semester Thesis SA-2014-30, September 2014 to December 2014,
ftp://ftp.tik.ee.ethz.ch/pub/students/2014-HS/SA-2014-30.pdf,
Date of the last visit: 9th of February 2015

[65] Pypy,
http://pypy.org/,
Date of the last visit: 9th of February 2015