Quantitative Analysis of MPLS Networks in AalWiNes

Shortest Trace Reachability Analysis of Weighted Pushdown Systems

Master Thesis

SV101F20

[illegible]

[illegible]

Dan Kristiansen
<dkris15@student.aau.dk>

Morten Konggaard Schou
<mschou15@student.aau.dk>

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerlöfsvej 300
Phone: +45 99 40 99 40
Fax: +45 99 40 97 98
http://www.cs.aau.dk

Title:

Quantitative Analysis of MPLS Networks in AalWiNes

Project:

Master Thesis

Project period:

01/02/2020 - 08/06/2020

Project group:

SV101F20

Authors:

Dan Kristiansen
Morten Konggaard Schou

Supervisor:

Jiri Srba
Peter Gjøl Jensen

Pagecount: 58

Hand-in date: 08/06/2020

Abstract:

Safe, reliable and efficient computer networks are essential for modern digital communication. Configuring these networks is a challenging task, which today is mostly performed manually. To aid in this task, the tool AalWiNes provides polynomial-time analysis of the behaviour of a type of network, called MPLS network, even under the scenario of multiple link failures. AalWiNes answers queries on MPLS networks by translating them into reachability queries on pushdown systems.
We extend AalWiNes with quantitative properties of network traces and implement an efficient algorithm to find traces minimizing these properties. We also define an algorithm for automatically generating MPLS networks, and we implement algorithms for pushdown system reachability in the library PDAAAL used by AalWiNes. Our experimental evaluation demonstrates that we outperform the state-of-the-art tool previously used by AalWiNes for pushdown reachability checking, and that our shortest trace algorithm for weighted pushdown systems only incurs a small overhead compared to unweighted reachability checking.

Acknowledgements

We would like to thank Bernhard Schrenk, University of Vienna for his work on the graphical user interface for AalWiNes. The GUI supports easy construction of queries and definition of weight expressions, and it provides an intuitive visualization of the network and output trace on a map.

We would also like to thank Stefan Schmid, University of Vienna for his feedback on our weight expressions for network traces. Given his expertise in computer networking this helps support the practical relevance of these weight expressions.

Finally we thank Henrik T. Jensen NORDUnet for providing their network configurations, which we have used as a case study for testing both correctness and performance of our implementations of the verification algorithms.

Summary

Large computer networks that facilitate modern digital communication are complex structures, on which many activities rely. Network operators face a challenging task of configuring their networks in a way so they are both efficient, safe and reliable. To achieve this, they need to understand not only how the networks behave in normal conditions, but also how their various fail-safe mechanisms affect the behaviour of the network under multiple hardware failures. In this thesis we focus on modelling the behavior of the network type Multiprotocol Label Switching (MPLS), which uses a routing technique of short path labels to route traffic rather than long network addresses. This avoids complex look-ups in routing tables, which makes the network traffic more flexible and faster.

The tool AalWiNes [1], developed at Aalborg University and the University of Vienna, attempts to analyse the behaviour of MPLS networks under the possibility of multiple link failures. We model the behaviour of an MPLS network as a pushdown automaton (PDA), which is a formal model that consists of a state machine and a label stack, and where the transitions of the state machine can push, pop or swap labels on the top of the stack. AalWiNes uses a special query language based on regular expressions to ask whether a given kind of trace through the network is possible if up to k links fail. The query together with the network is translated into a PDA, on which reachability analysis is performed through the library PDAAAL.

In this thesis we define quantitative properties of networks traces and solve the problem of finding a trace that satisfies a query while minimizing a weight expression of these quantitative properties. We implement efficient algorithms for reachability analysis in PDAAAL including an algorithm for finding shortest trace in weighted pushdown systems.

Finally we define an algebra for combining network topologies to generating large, realistic networks. We define an algorithm that generates a routing table for any network topology, such that there is label-switched paths between all pairs of routers, and fast reroute paths for all links to quickly recover from failures. The experimental evaluation demonstrates that PDAAAL outperforms the state-of-the-art tool Moped [2], and that minimizing the weight properties in the trace increases the likelihood of generating valid traces.

Contents

Introduction 1

Large computer networks that facilitate modern digital communication are complex structures, on which many activities rely. Network operators face a challenging task of configuring their networks in a way so they are both efficient, safe and reliable. To achieve this, they need to understand not only how the networks behave in normal conditions, but also how their various fail-safe mechanisms affect the behaviour of the network under multiple hardware failures.

Given the complexity of this task, it is evident that there is a need for automated tools for analysing and verifying networks - even under multiple failures. There exist several well-known tools [3, 4, 5, 6, 7] for network analysis, but the support for analysis of behaviour under failures is limited.

To solve the complexity issue, we apply techniques from formal verification to the analysis of computer networks. In particular we analyse Multiprotocol Label Switching (MPLS) [8] networks. MPLS is a network routing technique that uses short path labels to route traffic rather than long network addresses. This avoids complex look-ups in routing tables, which makes the network traffic more flexible and faster. MPLS is used for creating fast wide area network (WAN) [9] and virtual private networks (VPN) [10], and can be used by network providers to quickly tunnel IP-traffic through their sub-network. To handle failures, MPLS employs a local protection technique called fast rerouting, where a label is pushed on top of the existing labels and used to route the packet via an alternative path. When the packet arrives back on the original path, the extra label is popped.

The tool AalWiNes [1] (formerly named P-Rex [4]) being developed at Aalborg University and the University of Vienna attempts to analyse the behaviour of MPLS networks under the possibility of multiple failures.

The behaviour of an MPLS network is modelled using a pushdown automaton (PDA), which is a formal model that consists of a state machine and a label stack, and where the transitions of the state machine can push, pop or swap labels on the top of the stack. AalWiNes uses a special query language based on regular expressions to ask whether a given kind of trace through the network is possible, if up to k links fail. The query together with the network is translated into a PDA, on which reachability analysis is performed.

Jensen et al. [4] present a comparison of the approach of P-Rex and existing tools; NetKAT [3], Header Space Analysis (HSA) [5], VeriFlow [6], and Anteater [7], that all support reachability checking within a network model. NetKAT is a static verification tool with reachability and loop detection through Kleene algebra. HSA is like NetKAT, a static verification tool that covers basic reachability queries, but models each packet header as a point in a header space. Both VeriFlow and Anteater covers data plane information by converting it to boolean functions for a SAT solver to identify violations. VeriFlow is a debugging tool added to the configuration of an OpenFlow network, preventing any faulty rules to cause anomalous network behavior. Anteater analyses forwarding tables and checks if a network conforms to a set of invariants. The tool further provide a counter example of any violations.

All the compared tools can check for compliance of network properties, like reachability and loop-detection, and provide useful information to validate the correctness of that network. However, neither of the aforementioned tools support analysis under failure and they only support a fixed number of packet headers. Verification of MPLS networks, require arbitrarily large header sizes, so they cannot in general be verified by the mentioned tools.

The main contributions of this thesis are listed and explained below:

1. the pushdown automata reachability library PDAAAL,
2. weighted reachability extension for PDAAAL to find shortest trace in a network,
3. minimization of the global property of the maximum stack size,
4. early termination check to improve the performance of the library,
5. network topology manipulation that allows network topologies to be extended through the concatenation and injection operations,
6. generation of MPLS routing tables with failover based on the shortest path computation,
7. reduction of operation sequences in routing tables, and
8. a final experimental evaluation to prove the performance impact of the contributions.

We introduced the reachability library (1) as a performance improvement to AalWiNes, that formerly used the tool Moped [2] to perform unweighted nondeterministic PDA reachability analysis and trace generation. The implementation of the library PDAAAL allows the reachability analysis to be an integrated part of the tool, rather than an external Moped binary that introduces performance overhead when parsing. The weighted extension (2) models pushdown systems with weights, allowing for shortest trace queries using a combination of different quantitative properties of the network traces as weights. Queries can be annotated to minimize these properties, the supported properties to minimize in the trace are: number of hops, number of failures, number of tunnels, and the physical distance. A non-trivial problem is to minimize the global property of the maximum stack size during a trace (3).

The early termination check (4) is introduced as a performance improvement in the reachability algorithms of PDAAAL. The early termination check, is implemented as a constant time lookup in the current system, terminating if the goal state is already reached, rather than building the complete set of reachable states. Network manipulation (5) is a contribution to the existing AalWiNes implementation, allowing us to create large, realistic network topologies for testing. The manipulation works as an algebra for combining network topologies with concatenation and injection. Routing generation (6) create for any network topology a routing table, such that there exist label-switched paths between all pairs of routers, and fast reroute paths for all edges. If any edge in the network fails, such a fast reroute path is applied rather than the failing edge. Another performance improvements is introduced by reducing the operation sequence (7). A set of operation sequence reduction rules are defined, such that operation sequences are reduced to smaller sequences with the same behavior.

Finally we conduct an experimental evaluation to compare the performance of our implementation to Moped. The networks used in this experiment are constructed from five different network topologies from Topology Zoo [11], each manipulated to six different sizes, and annotated with generated routing tables. The experiment is executed on a cluster provided by Aalborg University, and include 200 instances of queries for each of the 30 different networks. The performance experiments capture the verification time of Moped and PDAAAL, and directly compare them in each test case. From the experiment can we conclude that PDAAAL outperforms the state-of-the-art tool Moped. Another experiment include the weighted extension to PDAAAL, and captures the results of verifying the same aforementioned networks and queries. The results show the improvements of annotating the system with weights, e.g. to minimize the number of links in the trace, such that the weighted extension improve the likelihood of a valid trace being found, while there is only little performance overhead of the weighted reachability algorithm.

MPLS Network 2

Multiprotocol Label Switching (MPLS) is of great importance when creating a fast deployable, flexible and efficient package forwarding architecture [8]. A virtual internet service provider (ISP) can organize a virtual network with provider edge-routers (PE-router) and provider-routers (P-routers) over its physical network layer routers, where PE-routers are the edge-routers connecting traffic between different virtual subnets, and P-Routers are inside the virtual network. The ISP can extend the virtual network to fit the requirements, handle traffic engineering, and ensure quality of service (QoS) inside the virtual network [8].

MPLS replace the IP lookup forwarding process [12] with a simpler label switching process. MPLS can run on top of several different network protocols, hence the multiprotocol part of the name. It encapsulates the network packet by prepending a header. The MPLS header contains a label stack, a traffic class, bottom of stack flag, and time to live. The packet is forwarded in the virtual network by swapping the top label of the stack. If the packet enters a different subnet another label is pushed on top of the stack, and popped when leaving the subnet [13].

We here present a formal model of MPLS networks and their behaviour. This model is used to describe the verification algorithms presented later. The model is based on the model in [4], but we introduce some differences: we separate the definition of network and network topology, we define routing and traces over links rather than over interfaces on routers, and we simplify the definition of the network topology by using a directed multigraph.

Definition 2.1. A network topology is a directed multigraph of routers, defined as a tuple (V, E, s, t), where:

- V is a finite set of routers,
- E is a finite set of links connecting routers,
- $s : E \to V$ assigning the source router to each link, and
- $t : E \to V$ assigning the target router to each link.

To ease the presentation, we may sometimes write $(v, u) \in E$ to mean $e \in E$ such that $s(e) = v$ and $t(e) = u$.

Most real-world networks have bidirectional links. We can model this property as the existence of a bijection $p : E \to E$ such that for all $e \in E$, $s(p(e)) = t(e)$ and $t(p(e)) = s(e)$. If this is the case, we call (V, E, s, t) a bidirectional network topology.

A bidirectional network topology can be modelled by an undirected multigraph, but we need the direction of edges when defining routing tables later. However, when depicting bidirectional network topologies in figures, we can often just draw an isomorphic undirected multigraph, since it carries the same information.

Figure 2.1 shows an example of a network topology with routers $V = \{v_0, v_1, v_2, v_3, v_4\}$ and bidirectional edges as depicted.

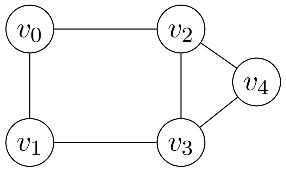


Figure 2.1. Example of a network topology.

Let L be a nonempty set of MPLS labels in headers of packets in an MPLS network and define the allowed MPLS operations on packet headers as $Op = \{\texttt{swap}(\ell) \mid \ell \in L\} \cup \{\texttt{push}(\ell) \mid \ell \in L\} \cup \{\texttt{pop}\}$.

Definition 2.2. An MPLS network is a tuple $N = (V, E, s, t, L, \tau)$, where:

- (V, E, s, t) is a network topology,
- $L = L_M \uplus L_M^{\perp} \uplus L_{IP}$ is the set of label stack symbols where,
  - L_M is the MPLS label set,
  - $L_M^{\perp}$ is the set of MPLS labels with the bottom of the stack bit set to *true*,
  - L_{IP} is a set of labels for IP routing information.
- $\tau : E \times L \to (2^{E \times Op^*})^*$ is the global routing table. For every link $e \in E$ and a top (left-most) label, it returns a sequence of traffic engineering groups $O_0 O_1 \ldots O_n$, where each O_i is a traffic engineering group $O = \{(e_1, \omega_1), \ldots, (e_m, \omega_m)\}$, where e_j is the outgoing link such that $t(e) = s(e_j)$, $\omega_j \in Op^*$ is a sequence of operations to perform on the packet header, and the group O_i has a higher priority than O_{i+1}.

  The global routing table can be represented as a collection of local routing tables $\tau_v : E_v \times L \to (2^{\overline{E_v} \times Op^*})^*$ for each router $v \in V$ where $E_v = \{e \in E \mid t(e) = v\}$ and $\overline{E_v} = \{e \in E \mid s(e) = v\}$.

In a network some links between routers may fail. We call the set of failed links F, where $F \subseteq E$. We say that a link e is *active* if $e \in E \setminus F$.

From the example network topology in Figure 2.1, we add a router θ to represent the outside world, and connect it to the PE-routers v_0 and v_3. For this network topology (V', E', s', t'), where

- $V' = V \cup \{\theta\}$,
- $E' = E \cup \{(\theta, v_0), (v_0, \theta), (\theta, v_3), (v_3, \theta)\}$

Figure 2.2 shows an example of a routing table with labels $L = \{ip_1, 10, 11, 20, 21, 30\}$

$$\begin{aligned}
&\tau((\theta, v_0), ip_1) = \{((v_0, v_1), \texttt{push}(10)), ((v_0, v_2), \texttt{push}(20))\}\\
&\tau((v_0, v_1), 10) = \{((v_1, v_3), \texttt{swap}(11))\}\\
&\tau((v_1, v_3), 11) = \{((v_3, \theta), \texttt{pop})\}\\
&\tau((v_0, v_2), 20) = \{((v_2, v_3), \texttt{swap}(21))\} \circ \{((v_2, v_4), \texttt{swap}(21) \circ \texttt{push}(30))\}\\
&\tau((v_2, v_4), 30) = \{((v_4, v_3), \texttt{pop})\}\\
&\tau((v_2, v_3), 21) = \{((v_3, \theta), \texttt{pop})\}\\
&\tau((v_4, v_3), 21) = \{((v_3, \theta), \texttt{pop})\}
\end{aligned}$$

Figure 2.2. Example routing table for the network topology in Figure 2.1.

The example network topology in Figure 2.1 and the associated example routing table in Figure 2.2 create a network $N = (V, E, s, t, L, \tau)$. The routing table allows dataflow in the network from v_0 to v_3, such that when a packet arrives at v_0 with label ip_1, the `push` operation encapsulates it in an MPLS label, which is used to route it through the network. When the packet leaves v_3 the MPLS label is popped. The routing table of v_0 contains nondeterminism, since $\tau((\theta, v_0), ip_1)$ can use either v_1 and v_2 to reach v_3. Rule $\tau((v_0, v_2), 20)$ include traffic engineering groups, such that if we have $(v_2, v_3) \in F$ then a fast-reroute path (v_2, v_4) and (v_4, v_3) become active to reach v_3.

2.1 Label Headers

An MPLS network routes traffic by prepending a header to the data. This header contains a sequence of labels, which is interpreted as a stack, so we can only read and write on the top (left-most in our notation) part of the stack. Using a stack of labels means that we can encapsulate routing labels by pushing a new label on top. This makes it simple to implement the fast rerouting technique, which we discuss in more detail in Section 2.4.

Definition 2.3 (Valid Header [4])**.** For a given network $N = (V, E, s, t, L, \tau)$ the set of valid headers $H \subseteq L^*$ is defined as $H = L_{IP} \cup \{\alpha\ell_1\ell_0 \mid \alpha \in {L_M}^*,\ \ell_1 \in L_M^{\perp},\ \ell_0 \in L_{IP}\}$.

MPLS operations manipulate the label-stack header by modifying the topmost label. A valid label-stack header is structured with MPLS labels on top of the underlying tunneled L_{IP} header. The semantics of the operations must ensure that the result of operations performed on a valid header is itself a valid header.

Definition 2.4. The semantics of MPLS operations is defined as a partial header rewrite function $\mathcal{H} : H \times Op^* \hookrightarrow H$ where $\omega, \omega' \in Op^*$, $\ell \in L$, $h \in H$ and ϵ is the empty sequence of operations:

$$\mathcal{H}(\ell h, \omega) := \begin{cases} \ell h & \text{if } \omega = \epsilon \text{ and } \ell \in L \\ \mathcal{H}(\ell' h, \omega') & \text{if } \omega = \texttt{swap}(\ell') \circ \omega' \text{ and } \ell' h \in H \\ \mathcal{H}(\ell'\ell h, \omega') & \text{if } \omega = \texttt{push}(\ell') \circ \omega' \text{ and } \ell'\ell h \in H \\ \mathcal{H}(h, \omega') & \text{if } \omega = \texttt{pop} \circ \omega' \text{ and } \ell \in L_M \cup L_M^{\perp} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

From the definition we observe that the header rewrite function preserves the valid structure of the label-stack symbols, otherwise undefined.

As an example, let $L_M = \{10, 20\}$, $L_M^{\perp} = \{30, 40\}$ and $L_{IP} = \{ip_1\}$. When applying the sequence of operations $\texttt{pop} \circ \texttt{swap}(40) \circ \texttt{push}(20)$ to the header $10 \circ 30 \circ ip_1$, we get: $\mathcal{H}(10 \circ 30 \circ ip_1,\ \texttt{pop} \circ \texttt{swap}(40) \circ \texttt{push}(20)) = 20 \circ 40 \circ ip_1$, which is also a valid header.

2.2 Network Traces

For a network $N = (V, E, s, t, L, \tau)$ and a set of failed links $F \subseteq E$, a trace is defined as a routing of a packet that consist of a sequence of active links together with the corresponding label-stack headers that the packet arrives with.[1]

For a traffic engineering group $O = \{(e_1, \omega_1), (e_2, \omega_2), \ldots, (e_m, \omega_m)\}$ let $E(O) = \{e_1, e_2, \ldots, e_m\}$ denote the set of all links in the traffic engineering group. The group O is said to be *active* if it contains at least one active link, i.e. $E(O) \setminus F \neq \emptyset$. Further, define $\mathcal{A}(O_0 O_1 \ldots O_n) = \{(e, \omega) \in O_j \mid e \text{ is an active link}\}$ where j is the lowest index, such that O_j is an active traffic engineering group, and define $\mathcal{A}(O_0 O_1 \ldots O_n) = \emptyset$ if no such j exists. This defines effectively the active routing entries for a sequence of traffic engineering groups and a set of failed links.

Definition 2.5 (Network Trace)**.** A trace in a network $N = (V, E, s, t, L, \tau)$ with a set of failed links $F \subseteq E$ is any finite sequence $(e_1, h_1), \ldots, (e_n, h_n) \in ((E \setminus F) \times H)^*$ of link-header pairs, where for all i, $1 \leq i < n$, we have $h_{i+1} = \mathcal{H}(h_i, \omega)$ for some $(e_{i+1}, \omega) \in \mathcal{A}(\tau(e_i, \mathit{head}(h_i)))$ where $\mathit{head}(h_i)$ is the top (left-most) label of h_i.

There may be more than one active link in an active traffic engineering group, in which case any active link can be (non-deterministically) picked. We say in this case that the routing is non-deterministic.

Here is an example of a network trace using the network defined in Figure 2.1 and Figure 2.2 and failed link $(v_2, v_3) \in F$:

$$((\theta, v_0), ip_1) \circ ((v_0, v_2), 20 \circ ip_1) \circ ((v_2, v_4), 30 \circ 21 \circ ip_1) \circ ((v_4, v_3), 21 \circ ip_1) \circ ((v_3, \theta), ip_1)$$

[1] This is based on the definition in [4]. However they define a trace as a sequence of interfaces on routers rather than a sequence of links.

2.3 Network Topology Manipulation

We introduce in this section the two functions CONCAT and INJECT to manipulate network topologies. The motivation for this is to create large realistic network topologies that we can use for testing the performance of the verification algorithms presented later in this thesis. From the network topologies created by CONCAT and INJECT we can create full realistic MPLS networks by generating routing tables as described in Section 2.4.

2.3.1 Network concatenation

Definition 2.6. For two disjoint network topologies $N = (V, E, s, t)$ and $N' = (V', E', s', t')$ (i.e. where $V \cap V' = \emptyset$ and $E \cap E' = \emptyset$), and a set of router pairs $\mathcal{C} \subseteq V \times V'$ connecting N and N' we define:

$$\textsc{Concat}(N, N', \mathcal{C}) = (V \cup V',\ (E \cup E') \uplus \mathcal{C}, s'', t'')$$

where:

- $s''(e) = \begin{cases} s(e) & \text{if } e \in E \\ s'(e) & \text{if } e \in E' \\ v & \text{if } e = (v, u) \in \mathcal{C} \end{cases}$
- $t''(e) = \begin{cases} t(e) & \text{if } e \in E \\ t'(e) & \text{if } e \in E' \\ u & \text{if } e = (v, u) \in \mathcal{C} \end{cases}$

The concatenated network topology N depicted in Figure 2.3 is defined as $N = \textsc{Concat}(N', N'', \mathcal{C})$ where N' and N'' are the example network topology from Figure 2.1 and $\mathcal{C} = \{(v_2, v'_0), (v_3, v'_1), (v'_0, v_2), (v'_1, v_3)\}$.

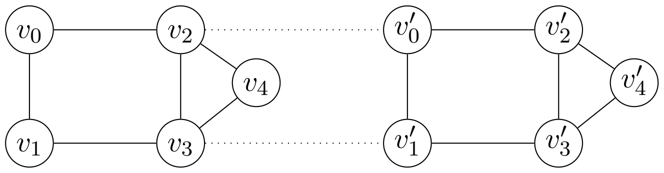


Figure 2.3. Concatenation of two copies of the example network topology from Figure 2.1. The dotted edges between nodes v_2 and v'_0, and v_3 and v'_1, are new connecting links defined in $\mathcal{C}$.

2.3.2 Network injection

Given two network topologies $N = (V, E, s, t)$ and $N' = (V', E', s', t')$, we define an injection tuple that contains the edge to be removed from N, allowing the injection of N', and two connecting routers from N' that will be inserted instead of the removed edge, i.e. $(e, v, u) \in E \times V' \times V'$.

Definition 2.7. For two disjoint network topologies $N = (V, E, s, t)$ and $N' = (V', E', s', t')$ and a set of injection tuples $\mathcal{T} \subseteq E \times V' \times V'$ we define:

$$\textsc{Inject}(N, N', \mathcal{T}) = (V \cup V',\ (E \setminus E_-) \cup E' \cup E_+,\ s'',\ t'')$$

where:

- $E_- = \{e \mid (e, v, u) \in \mathcal{T}\}$ are the removed links,
- $E_+ = \biguplus_{(e,v,u)\in\mathcal{T}} \{(s(e), v), (u, t(e))\}$ are the added links,
- $s''(e) = \begin{cases} s(e) & \text{if } e \in E \setminus E_- \\ s'(e) & \text{if } e \in E' \\ v & \text{if } e = (v, u) \in E_+ \end{cases}$
- $t''(e) = \begin{cases} t(e) & \text{if } e \in E \setminus E_- \\ t'(e) & \text{if } e \in E' \\ u & \text{if } e = (v, u) \in E_+ \end{cases}$

Figure 2.4 shows $N = \textsc{Inject}(N', N'', \mathcal{T})$, where N' and N'' is the example network topology from in Figure 2.1, and $\mathcal{T} = \{((v_2, v_3), v'_0, v'_3), ((v_3, v_2), v'_3, v'_0)\}$, so $E_- = \{(v_2, v_3), (v_3, v_2)\}$, and $E_+ = \{(v_2, v'_0), (v'_3, v_3), (v_3, v'_3), (v'_0, v_2)\}$.

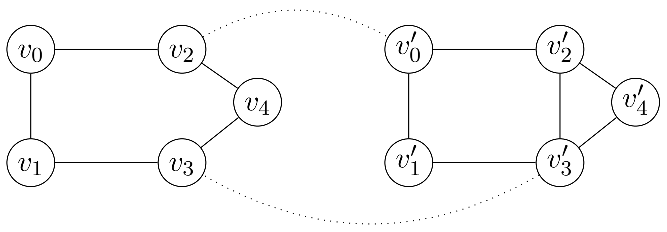


Figure 2.4. Injection of the network topology from Figure 2.1 into a copy of itself. The dotted edges denote the links in E_+ and the removed edges in E_- are between v_2 and v_3.

2.4 Generation of Routing Tables

In the previous section we defined the functions Concat and Inject to build network topologies from other topologies. This section defines a method for generating a realistic routing table for any network topology, and thereby producing a full MPLS network.

The method for generating routing tables, defined in Algorithm 1, consists of two phases. First for every pair of routers we create a label-switched path between them to encapsulate an IP-label. The path is found using a shortest path algorithm, where the weight of each link is some suitable measure like the physical distance or the inverse of the bandwidth capacity.

Since the connections to the outside IP-network is not part of the MPLS network model, we simulate it with an auxiliary router θ. The label-switched path from v to u starts at an edge going in to v (from θ), where it pushes an bottom-of-stack MPLS label on top of the IP-label, and ends with an edge going out of u (to θ), where the MPLS label is popped. This corresponds to a label-switched path for routers connected to an external IP-network.

Algorithm 1 Pseudocode for generating routing tables

```
    Input A network topology $N = (V, E, s, t)$
    Output An MPLS network with routing $N = (V', E', s', t', L, \tau)$
 1: function MakePath($e_1 \dots e_n$, $\ell_{e_1}$, $reroute$)
 2:     for $i \in \{2, \dots, n\}$ do
 3:         $L \leftarrow L \uplus \{\ell_{e_i}\}$          ▷ $\ell_{e_i} \in L_M$ if $reroute$, $\ell_{e_i} \in L_M^\perp$ otherwise
 4:         $\tau(e_{i-1}, \ell_{e_{i-1}}) := \{(e_i,\ \texttt{swap}(\ell_{e_i}))\}$
 5:     return $\ell_{e_n}$
 6: function MakeFlow($e_{start}$, $e_{end}$, $\ell_{ip}$)
 7:     $e_1 \dots e_n \leftarrow$ Dijkstra($t'(e_{start})$, $s'(e_{end})$, $(V, E)$)
 8:     $L \leftarrow L \uplus \{\ell_{e_1}\}$          ▷ $\ell_{e_1} \in L_M^\perp$ is a fresh label
 9:     $\ell_{last} \leftarrow$ MakePath($e_1 \dots e_n$, $\ell_{e_1}$, $false$)
10:     $\tau(e_{start}, \ell_{ip}) := \{(e_1,\ \texttt{push}(\ell_{e_1}))\}$, $\tau(e_n, \ell_{last}) := \{(e_{end},\ \texttt{pop})\}$
11: function MakeReroute($e$)
12:     $e_1 \dots e_n \leftarrow$ Dijkstra($s(e)$, $t(e)$, $(V, E \setminus \{e\})$)
13:     $L \leftarrow L \uplus \{\ell_{push}\}$          ▷ $\ell_{push} \in L_M$ is a fresh label
14:     for $e', \ell, i, \omega$ where $\tau(e', \ell) = O_0 \dots O_m$, $(e,\ \omega) \in O_i$ do
15:         $\tau(e', \ell) := O_0 \dots O_i(O_{i+1} \cup \{(e_1,\ \omega \circ \texttt{push}(\ell_{push}))\}) \dots O_m$
16:     $\ell_{last} \leftarrow$ MakePath($e_1 \dots e_{n-1}$, $\ell_{push}$, $true$)
17:     $\tau(e_{n-1}, \ell_{last}) := \{(e_n,\ \texttt{pop})\}$
18:     for $\ell$ where $\tau(e, \ell) \neq \epsilon$ do
19:         $\tau(e_n, \ell) := \tau(e, \ell)$
20:
21: $E' \leftarrow E$, $s' \leftarrow s$, $t' \leftarrow t$, $L \leftarrow \emptyset$, $\tau \leftarrow \lambda(e, \ell).\epsilon$
22: $V' \leftarrow V \uplus \{\theta\}$          ▷ $\theta$ represents the outside world
23: for $(v, u) \in V \times V$ where $v \neq u$ do
24:     $E' \leftarrow E' \uplus \{(\theta, v), (u, \theta)\}$
25:     $s'((\theta, v)) := \theta$, $s'((u, \theta)) := u$, $t'((\theta, v)) := v$, $t'((u, \theta)) := \theta$
26:     MakeFlow($(\theta, v), (u, \theta), \ell_{ip}$)          ▷ $\ell_{ip} \in L_{IP}$ is the ip address to the outside
27: for $e \in E$ do
28:     MakeReroute($e$)
29: return $(V', E', s', t', L, \tau)$
```

Secondly, for every link that may fail, we create a fast reroute path that encapsulates the header by pushing an MPLS label on top of it, and routes the traffic on an alternative path. To ensure this path is only used in case of link failure, the routing is added to a traffic engineering group with lower priority (on line 15) than the group that contains the direct routing through the failing link (where line 14 iterates over all such groups). The rerouting label is popped from the header on the last hop of the alternative label-switched path, and (on line 19) the routing tables of the destination are modified, so the traffic can continue from there, as if the link did not fail.

The routes are calculated using a standard shortest path algorithm $\textsc{Dijkstra}(v,\ u,\ G)$, which finds the shortest path from node v to node u in the graph $G = (V, E)$, using some suitable edge-weight function. For the real-world topologies from Topology Zoo we use coordinates to calculate the great-circle distance between routers using the haversine formula. The distance between routers is used as edge-weight to generate realistic shortest path routes.

Some technical details are omitted from the pseudocode in Algorithm 1. If the call to $\textsc{Dijkstra}$ does not find a path, then the calling function terminates, in which case the label-switched path (resp. fast reroute) will not be created for that router pair (resp. failing link). On line 14 if $i = m$, then by abuse of notation on line 15 we create a new traffic engineering group O_{i+1} and append it to the rest.

A generated routing table example with a data flow between all pairs of routers is too large to show here. Instead we use Algorithm 1 to show how the routing table in Figure 2.2 can be generated. Note that the routing table in Figure 2.2 only contains data flow for the router pair v_0 and v_3, and only a reroute on edge (v_2, v_3). $\textsc{MakeFlow}((\theta, v_0), (v_3, \theta))$ is called twice, where for the sake of this example $\textsc{Dijkstra}(v_0,\ v_3,\ (V, E))$ produces two different paths $(v_0, v_1)(v_1, v_3)$ and $(v_0, v_2)(v_2, v_3)$ with the same edge-weight. In $\textsc{MakeReroute}((v_2, v_3))$, $\textsc{Dijkstra}(v_2,\ v_3,\ (V, E \setminus \{(v_2, v_3)\}))$ produces the path $(v_2, v_4)(v_4, v_3)$.

The routing table is produced in $\textsc{MakePath}(\mathit{path}, \ell)$ for both $\textsc{MakeFlow}$ and $\textsc{MakeReroute}$, where in line 4 the routing table is extended with a unique label and `swap` operation to each e_{i-1} in *path*, such that for $\textsc{MakeFlow}$ with $\mathit{path} = (v_0, v_2)(v_2, v_3)$ and $\ell_{e1} = 20$ then $\tau((v_0, v_2), 20) = \{(v_2, v_3), \texttt{swap}(21)\}$. At line 10 the routing table is further extended to encapsulate the underlying IP-label, where $\tau((\theta, v_0), ip_1) = \{(v_0, v_2), \texttt{push}(20)\}$, and $\tau((v_2, v_3), 21) = \{(v_3, \theta), \texttt{pop}\}$. At this point the routing table is:

$$\tau((\theta, v_0), ip_1) = \{(v_0, v_2), \texttt{push}(20)\}$$
$$\tau((v_0, v_2), 20) = \{(v_2, v_3), \texttt{swap}(21)\}$$
$$\tau((v_2, v_3), 21) = \{(v_3, \theta), \texttt{pop}\}$$

The routing table is similarly extended for $\mathit{path} = (v_0, v_1)(v_1, v_3)$ and $\ell_{e_1} = 10$. In $\textsc{MakeReroute}((v_2, v_3))$ with $\mathit{path} = (v_2, v_4)(v_4, v_3)$ and $\ell_{push} = 30$, all rules containing (v_2, v_3) are modified before creating the failover tunnel, such that the tunnel is encapsulated. The final routing table is the one shown in Figure 2.2.

2.4.1 Reduction of operation sequences

In some cases, different sequences of operations in a routing table may result in the same behavior of the MPLS network. If this is the case, using the shorter sequence when creating routing tables as described above will reduce the size of the resulting table.

To avoid strange edge-cases, the following discussion assumes non-trivial label sets: $|L_{IP}| > 1$, $|L_M^\perp| > 1$ and $|L_M| > 1$.

Definition 2.8. A function $r : Op^* \to Op^*$ is a valid reduction of operations if for all valid headers $h \in H$ and all sequences of operations $\omega \in Op^*$, it holds that $\mathcal{H}(h, \omega) = \mathcal{H}(h, r(\omega))$ and $|r(\omega)| \leq |\omega|$.

Define the function $r : Op^* \to Op^*$:

$$r(\omega) := \begin{cases} \omega' \circ \texttt{swap}(\ell) \circ \omega'' & \text{if } \omega = \omega' \circ \texttt{swap}(\ell') \circ \texttt{swap}(\ell) \circ \omega'' \\ \omega' \circ \texttt{push}(\ell) \circ \omega'' & \text{if } \omega = \omega' \circ \texttt{push}(\ell') \circ \texttt{swap}(\ell) \circ \omega'' \\ \omega' \circ \texttt{pop} \circ \omega'' & \text{if } \omega = \omega' \circ \texttt{swap}(\ell) \circ \texttt{pop} \circ \omega'' \\ \omega' \circ \omega'' & \text{if } \omega = \omega' \circ \texttt{push}(\ell') \circ \texttt{pop} \circ \omega'' \\ \omega & \text{otherwise} \end{cases}$$

Repeated application of r on ω will eventually reach a fixed point. We call this value $r^*(\omega)$. At this point either $|r^*(\omega)| \leq 1$ or all adjacent operations $r^*(\omega) = \omega' \circ op_1 \circ op_2 \circ \omega''$ are on the form:

$$(op_1, op_2) \in \{(\texttt{pop}, \texttt{pop}), (\texttt{pop}, \texttt{swap}(\ell)), (\texttt{pop}, \texttt{push}(\ell)), (\texttt{swap}(\ell), \texttt{push}(\ell')), (\texttt{push}(\ell), \texttt{push}(\ell'))\}$$

This means that $r^*(\omega)$ will be on the form described by the following regular expression of operations, where ℓ_i iterates over labels:

$$\texttt{pop}^* \circ \texttt{swap}(\ell)^? \circ \texttt{push}(\ell_i)^*$$

Lemma 2.1. The function r^* is a valid reduction of operations.

Proof. By straightforward case analysis we see that the function r is a valid reduction. Repeated application of a valid reduction is also a valid reduction. □

Theorem 2.2. Assuming non-trivial label-sets, the reduction r^* is the unique minimal valid reduction, i.e. for any ω and ω' such that for all valid headers h, $\mathcal{H}(h, \omega) = \mathcal{H}(h, \omega')$, we have $|r^*(\omega)| \leq |\omega'|$, and if $|r^*(\omega)| = |\omega'|$ then $r^*(\omega) = \omega'$.

Proof. Take any ω and ω' such that for all valid headers h, $\mathcal{H}(h, \omega) = \mathcal{H}(h, \omega')$. If $\omega' = r^*(\omega)$ then the theorem follows directly, so take $\omega' \neq r^*(\omega)$. By Lemma 2.1 $\mathcal{H}(h, r^*(\omega)) = \mathcal{H}(h, \omega')$ for all valid headers h. The result of $r^*(\omega)$ is on the form $\texttt{pop}^n \circ \texttt{swap}(\ell)^s \circ \texttt{push}(\ell_i)^m$, where n and m are non-negative integers and $s \in \{0, 1\}$.

Define $lcs(h, h')$ as the largest common suffix (bottom sub-part) of h and h'. Since we assume non-trivial label-sets, there must exist a valid header h such that $|h| - |lcs(h, \mathcal{H}(h, r^*(\omega)))| = n + s$ and $|\mathcal{H}(h, r^*(\omega))| - |lcs(h, \mathcal{H}(h, r^*(\omega)))| = s + m$. Since $\mathcal{H}(h, r^*(\omega)) = \mathcal{H}(h, \omega')$, we have $|h| - |lcs(h, \mathcal{H}(h, \omega'))| = n + s$ and $|\mathcal{H}(h, \omega')| - |lcs(h, \mathcal{H}(h, \omega'))| = s + m$.

If $s = 1$ then ω' must contain at least $n + 1$ `pop` operations, or at least n `pop` operations followed by a `swap`. Also ω' must contain at least $m + 1$ `push` operations, or at least a `swap` operation followed by m `push` operations. Since $\omega' \neq r^*(\omega)$, we have $|\omega'| > n + s + m = |r^*(\omega)|$.

If $s = 0$, then ω' must contain at least n `pop` operations, or at least $n-1$ `pop` operations followed by a `swap`. Also ω' must contain at least m `push` operations, or at least a `swap` operation followed by $m-1$ `push` operations. Since $\omega' \neq r^*(\omega)$ and the number of `push` minus number of `pop` operations must be $m-n$ the only possibility with $|\omega'| \leq |r^*(\omega)|$ is $\omega' = \texttt{pop}^{n-1} \circ \texttt{swap}(\ell) \circ \texttt{push}^{m-1}(\ell_i)$. However, there exists a valid header h' such that $\mathcal{H}(h', \texttt{pop}^{n-1} \circ \texttt{swap}(\ell) \circ \texttt{push}^{m-1}(\ell_i))$ is a valid header, while $\mathcal{H}(h', \texttt{pop}^{n} \circ \texttt{push}^{m}(\ell_i)) = \mathcal{H}(h', \omega)$ is undefined. So this sequence will not be a valid reduction of ω. $\square$

The following example shows a repeated application of r to a sequence of operations ω.

$$\begin{aligned}
\omega &= \texttt{swap}(1) \circ \texttt{push}(2) \circ \texttt{swap}(3) \circ \texttt{pop} \circ \texttt{pop} \circ \texttt{push}(4) \circ \texttt{swap}(5)\\
& \texttt{swap}(1) \circ \texttt{push}(2) \circ \texttt{swap}(3) \circ \texttt{pop} \circ \texttt{pop} \circ \texttt{push}(5)\\
& \texttt{swap}(1) \circ \texttt{push}(2) \circ \texttt{pop} \circ \texttt{pop} \circ \texttt{push}(5)\\
& \texttt{swap}(1) \circ \texttt{pop} \circ \texttt{push}(5)\\
r^*(\omega) &= \texttt{pop} \circ \texttt{push}(5)
\end{aligned}$$

We cannot reduce $\texttt{pop} \circ \texttt{push}(5)$ further to $\texttt{swap}(5)$, since for a header h of size $|h| = 1$, $\mathcal{H}(h, \texttt{pop} \circ \texttt{push}(5))$ is undefined, while $\mathcal{H}(h, \texttt{swap}(5))$ is well defined. So $r^*(\omega) = \texttt{pop} \circ \texttt{push}(5)$ is the minimal valid reduction of ω.

2.5 AalWiNes and Queries

The tool AalWiNes (Aalborg Wien Network suite) [1] - formerly named P-Rex [4] - performs what-if analysis of MPLS networks such as reachability and policy-compliance properties under an arbitrary number of failures. It defines a query language and the main goal of the tool is to answer whether a given query is satisfied for a given network. If this is the case, the tool also produces a network trace as proof that the query is satisfied.

Definition 2.9 (Regular Expression)**.** A regular expression over the alphabet Σ ranged over by the symbols $s_1, s_2, \dots$ is given by the abstract syntax:

$$a ::= \cdot \mid [s_1, \dots, s_n] \mid [\hat{}\, s_1, \dots, s_n] \mid a_1 | a_2 \mid a_1 a_2 \mid a^* \mid a^+ \mid a^?$$

where the language of the regular expression $Lang(a) \subseteq \Sigma^*$ is given by:

$$\begin{aligned}
&Lang(\cdot) = \Sigma && Lang(a_1 a_2) = Lang(a_1) \circ Lang(a_2)\\
&Lang([s_1, \dots, s_n]) = \{s_1, \dots, s_n\} && Lang(a^*) = Lang(a)^*\\
&Lang([\hat{}\, s_1, \dots, s_n]) = \Sigma \setminus \{s_1, \dots, s_n\} && Lang(a^+) = Lang(a) \circ Lang(a^*)\\
&Lang(a_1|a_2) = Lang(a_1) \cup Lang(a_2) && Lang(a^?) = Lang(a) \cup \{\varepsilon\}
\end{aligned}$$

Definition 2.10 (Query [4])**.** A reachability query q is based on regular expressions and has the following structure:

$$< a > \; b \; < c > k$$

where, for a given network $N = (V, E, s, t, L, \tau)$, a and c are regular expressions over the labels L, b is a regular expressions over links E, and $k \geq 0$ specifies the maximum number of failed links to be considered.

For specifying labels in a and c, the following abbreviations can be used:

- `ip` $= [ip_0, \dots, ip_n]$ where $L_{IP} = \{ip_0, \dots, ip_n\}$,
- `mpls` $= [\ell_0, \dots, \ell_n]$ where $L_M = \{\ell_0, \dots, \ell_n\}$, and
- `smpls` $= [\ell_0^\perp, \dots, \ell_n^\perp]$ where $L_M^\perp = \{\ell_0^\perp, \dots, \ell_n^\perp\}$.

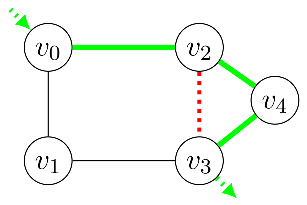
The input format for specifying networks to AalWiNes has identifiers on routers and on the interfaces that connect routers to links, rather than having identifiers on the links themselves. Each interface is associated with a router and identifies a link. To handle this, the query language uses the following syntax to specify links in the regular expression b based on those identifiers:

If v and u are routers, then v # u matches any edge e such that $s(e) = v$ and $t(e) = u$. If in_1 identifies the interface on router v with outgoing link e, and in_2 identifies the interface on router u with ingoing link e, then $v.in_1$ # $u.in_2$ matches exactly the edge e. Similarly we have: v # $u.in_2$ matches e if $s(e) = v$, and $v.in_1$ # u matches e if $t(e) = u$. The dot-syntax is extended so that x # $\cdot = \bigcup_{v \in V} x$ # v and $\cdot$ # $x = \bigcup_{v \in V} v$ # x, where either $x = u$ or $x = u.in$ for some router $u \in V$ and interface in on u with semantics as specified above.

Problem 2.1 (Query Satisfiability Problem [4])**.** For a network N and a query $< a > b < c > k$, does there exist a trace $\sigma = (e_1, h_1), \dots, (e_n, h_n)$ in the network with a set of failed links F such that $|F| \leq k$, $h_1 \in Lang(a)$, $e_1 \dots e_n \in Lang(b)$, and $h_n \in Lang(c)$? If this is the case, we say that the query is *satisfied* and that σ is a *witness trace* for the query.

A query thus allows property validation within the network. These properties operate on the routing table, allowing it to find bugs. A test suite of properties can be configured for an MPLS network, such that the properties enforce correctness. Such correctness can include connectivity (all routers are connected to a router), and avoidance of certain links (check connectivity without using a certain links). The connectivity proves useful, if e.g. there exists some internal data hosting in the MPLS network, and all parties of the network need access to such host. The avoidance of some link can e.g. be used to ensure traffic engineering and apply tunnels to transfer data rather than the shortest and possibly over-loaded path.

An example of a query and resulting witness trace is shown in Figure 2.5 on the network from Figure 2.1 and Figure 2.2. The query checks if there exist any trace with at most 1 failure ($k = 1$), starting with a single IP-label on the stack ($Lang(a) = \{ip_1\}$), taking a path with an initial edge to v_0, with any number of intermediate edges avoiding router v_1 and link v_2 # v_3, and finally an edge from v_3 ($b = [\cdot$ # $v_0]$ $[\hat{}\ \cdot$ # $v_1,\ v_2$ # $v_3]^*$ $[v_3$ # $\cdot]$), and a ending with a single IP-label on the stack ($Lang(c) = \{ip_1\}$).



Query: $< \texttt{ip} > [\cdot \texttt{ \# } v_0]\ [\hat{}\ \cdot \texttt{ \# } v_1,\ v_2 \texttt{ \# } v_3]^*\ [v_3 \texttt{ \# } \cdot] < \texttt{ip} > 1$

$F = \{(v_2, v_3)\}$

Trace: $((v_0, \theta), ip_1)$
$((v_0, v_2), 20 \circ ip_1)$
$((v_2, v_4), 30 \circ 21 \circ ip_1)$
$((v_4, v_3), 21 \circ ip_1)$
$((v_3, \theta), ip_1)$

Figure 2.5. An example of a query and a witness trace on the network topology from Figure 2.1 with the label-set and routing table from Figure 2.2.

The links used in the trace in Figure 2.5 is highlighted with bold and green, and the failed link is marked with a dotted red line. The first applied rule use the current header ip_1 to `push` 20 on to the header, resulting in a new link and header pair: $((v_0, v_2), 20 \circ ip_1)$.

The shortest path from v_2 to v_3 is through $(v_2, v_3) \in E$ but since $(v_2, v_3) \in F$ the rerouting path is used and encapsulate the tunnel with `swap` 21, and `push` 30. The final two rules decapsulate the tunnel and the underlying IP-label by `pop` operations so only ip_1 remains on the stack when leaving the network. Finally $|F| \leq k$ since $|F| = 1$, so the query is satisfied. Note that if the query instead has $k = 0$ then it cannot be satisfied.

2.6 Weight Expressions for Shortest Trace Selection

Sometimes it is not enough to get a network trace that witnesses a query, if we specifically want a trace that minimizes some property among all the witness traces. For specifying these quantitative properties of network traces, we define a weight language. Evaluating an expression from the weight language on a network trace $\sigma = (e_1, h_1), \ldots, (e_n, h_n) \in ((E \setminus F) \times H)^*$ gives a value, which can then be compared to the values for other network traces. The problem then becomes to find a query-satisfying trace that is shortest with respect to a given weight expression.

The weight language is build up of the following syntactic categories and production rules:

$p \in \mathbf{P}$	Atomic properties
$le \in \mathbf{LinExp}$	Linear expressions
$oe \in \mathbf{OrdExp}$	Ordered expressions

$$le ::= p \mid a * le \mid le_1 + le_2, \text{ where } a \in \mathbb{N}$$

$$oe ::= (le) \mid (le, oe)$$

An atomic property can in principle be any function $p : ((E \setminus F) \times H)^* \to \mathbb{N}_0$. That is for any network trace σ it evaluates to a non-negative integer $p(\sigma) \in \mathbb{N}_0$. However as described later in Section 4.3, not all properties allow for easy solutions to the shortest trace problem.

We define the following atomic properties for a network trace $\sigma = (e_1, h_1), \ldots, (e_n, h_n)$:

- $\textsc{Links}(\sigma) = n$ is the length of the trace.
- $\textsc{Hops}(\sigma) = |\{e \in \{e_1, \ldots, e_n\} \mid s(e) \neq t(e)\}|$ is the number of hops, where we avoid counting links that are self-loops.
- $\textsc{Distance}(\sigma) = \Sigma_{i=1}^{n} d(e_i)$ for any distance function $d : E \to \mathbb{N}_0$. This can be physical distance, latency, inverse bandwidth capacity, etc.
- $\textsc{LocalFailures}(\sigma) = \Sigma_{i=1}^{n-1} |\{e \mid (e, \omega) \in O_{i_k},\ 0 \leq k < j\}|$, where $\tau(e_i, \mathit{head}(h_i)) = O_{i_0} \ldots O_{i_m}$ and j is the lowest index such that O_{i_j} is an active traffic engineering group.
- $\textsc{Failures}(\sigma) = |F|$, where F is the smallest set of failed links such that σ is a trace in the network N.
- $\textsc{Tunnels}(\sigma) = \Sigma_{i=1}^{n-1} \min\{\#\texttt{push}(\omega) \mid \omega \in \mathit{Op}^*,\ (e_{i+1}, \omega) \in \mathcal{A}(\tau(e_i, \mathit{head}(h_i))) \wedge \mathcal{H}(h_i, \omega) = h_{i+1}\}$, where $\#\texttt{push}(\omega)$ is the number of `push` operations in ω. An MPLS tunnel is started with a push operation, so finding the trace on which the minimal number of `push` operations was applied corresponds to minimizing the number of tunnels.
- $\textsc{StackSize}(\sigma) = \max_i |h_i|$ is the maximum size of the header during the trace.

For the denotational semantics of linear expressions we define $[\![le]\!] : ((E \setminus F) \times H)^* \to \mathbb{N}_0$ for $le \in \textbf{LinExp}$ by

1. $[\![p]\!](\sigma) = p(\sigma)$
2. $[\![a * le]\!](\sigma) = a \cdot [\![le]\!](\sigma)$
3. $[\![le_1 + le_2]\!](\sigma) = [\![le_1]\!](\sigma) + [\![le_2]\!](\sigma)$

This is a linear combination of atomic properties.

For the denotational semantics of ordered expressions we define $[\![oe]\!] : ((E \setminus F) \times H)^* \to \mathbb{N}_0^k$ for $oe \in \textbf{OrdExp}$ and some $k \geq 1$ by

1. $[\![(le)]\!](\sigma) = ([\![le]\!](\sigma))$
2. $[\![(le, oe)]\!](\sigma) = ([\![le]\!](\sigma), [\![oe]\!](\sigma))$

Point 1 evaluates to a singleton tuple, and point 2 evaluates to a tuple of size $k+1$ if the inner oe evaluates to a tuple of size k.

We use the standard lexicographical ordering of tuples of integers: $(a_1, \ldots, a_k) \leq (b_1, \ldots, b_k)$ iff $a_1 < b_1 \vee (a_1 = b_1 \wedge (a_2, \cdots, a_k) \leq (b_2, \cdots, b_k))$ with the empty sequence being $\leq$ all elements of $\mathbb{N}_0^*$. A total order of network traces imposed by an ordered expression oe is defined as:

$$\sigma_1 \sqsubseteq_{oe} \sigma_2 \text{ iff } [\![oe]\!](\sigma_1) \leq [\![oe]\!](\sigma_2)$$

Problem 2.2 (Weighted Trace Problem)**.** For a network, a query and a weight expression oe, if the query is satisfied, find a witness trace σ such that for any other witness trace σ', we have $\sigma \sqsubseteq_{oe} \sigma'$.

As an example consider the witness trace σ in Figure 2.5. The value of some of the atomic properties of this example trace are: $\textsc{Links}(\sigma) = 5$, $\textsc{LocalFailures}(\sigma) = 1$, and $\textsc{StackSize}(\sigma) = 3$.

2.7 Overview of the Solution

The query satisfiability problem (Problem 2.1) is solved by the tool AalWiNes, which is based on the previous tool P-Rex [4]. The contribution of this thesis is to solve the weighted trace problem (Problem 2.2) by extending AalWiNes with the weight language, extending the construction of pushdown automata (PDA) with weights, and implementing weighted reachability analysis in the library PDAAAL (PushDown Automata AALborg), as well as overall improvements to the library.

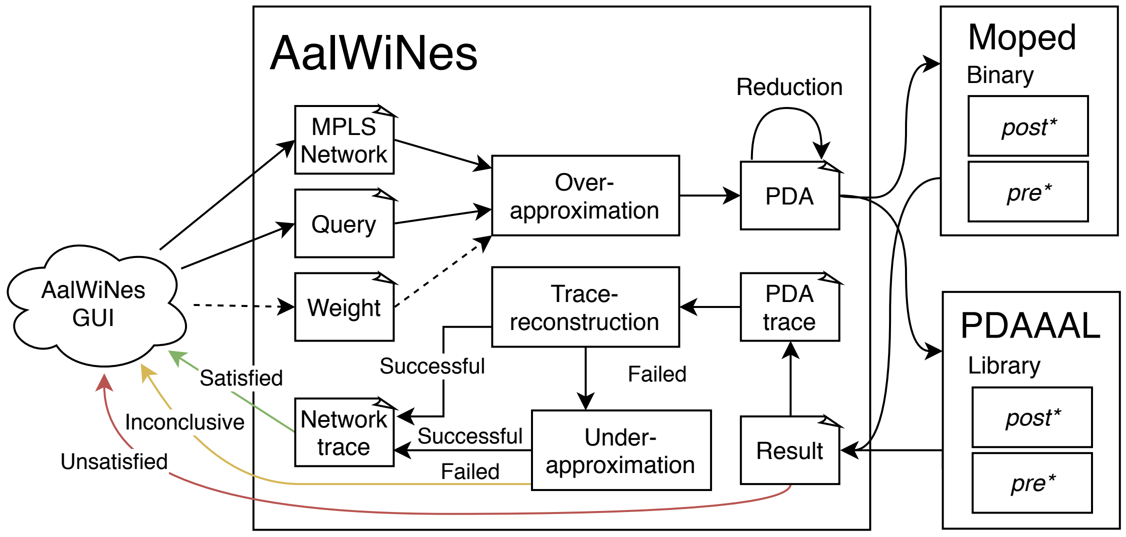


Figure 2.6. Overall flow including the contributions of this thesis. The result can be satisfied, unsatisfied, or inconclusive, such that an inconclusive answer is given if trace reconstruction fail in both over- and under-approximation.

Figure 2.6 provide an overview to the workflow of AalWines. Starting from the graphical user interface [14], we get an MPLS network, a query, and, if we want to perform shortest trace analysis, a weight expression. AalWiNes uses these to construct a PDA by means of over-approximation. Exact analysis has been deemed too expensive, since it requires an exponential number of states to keep track of the set of failed links. The over-approximation is formally defined in [4]. Intuitively, over-approximation counts the number of failed links locally at each step of the way, rather than keeping a global set of failed links. This means that the constructed PDA can contain traces that does not correspond to a valid witness trace in the network. If the over-approximation does not provide a conclusive result, the tool tries with an under-approximation instead. The under-approximation adds a failed link counter to each state (thus multiplying the number of states by k), and uses this counter to keep track of how many failed links the trace has encountered. This is an under-approximation, since the model can count the same failed link twice.

Once a PDA is constructed, it can be reduced by the top-of-stack reductions defined in [4]. Reducing the size of the PDA can speed up the reachability analysis, however the reduction algorithm also takes time. This time trade-off is explored further in the experimental evaluations in Chapter 5.

The reachability analysis of the PDA was previously performed by the external tool Moped [2]. Doing this involves writing the PDA to a file that Moped parses, performs analysis, and finally prints the result and trace to a file, which is read by AalWiNes. To avoid this overhead of printing and parsing, we have implemented reachability analysis in the library PDAAAL. This was implemented by one of the authors in an assignment prior to this thesis. If a weight expression is specified, then weighted PDAAAL performs shortest trace reachability analysis of the weighted PDA. Implementing this is one of the major contributions of this thesis. The result of PDA reachability analysis is a binary answer and, in the case of positive result, a trace witnessing the reachability.

The trace-reconstruction needs to make sure that the resulting network trace actually satisfies the query. This check uses two sets of links: *active* and *failed*. When we use a link for the network trace, it is added to *active*. If a link from a traffic engineering group with lower priority is added to *active*, we add the links from the higher priority traffic engineering groups to *failed*. If at some point *active* and *failed* have a non-empty intersection, or if the size of *failed* gets larger than k i.e. $\mathit{active} \cap \mathit{failed} \neq \emptyset$ or $|\mathit{failed}| > k$, then the trace-reconstruction fails. With the addition of weighted analysis, the under-approximation presented in [4] has become unnecessary, since the weighted expression (LocalFailures) finds any feasible trace that the under-approximation can find.

Reachability in Pushdown Systems 3

MPLS networks and the queries implemented in AalWiNes can be modelled as pushdown systems in which the query becomes a simple reachability query. Reachability in pushdown systems is a well studied area. We will here present the notation and core algorithms based on [15]. We also propose a simple early termination check to speed up the reachability queries.

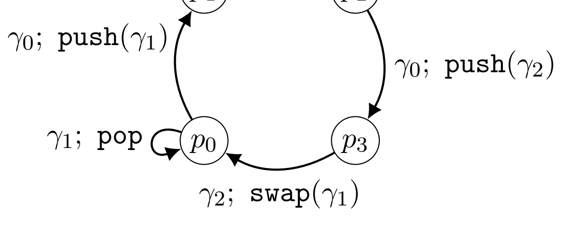
3.1 Pushdown Systems

Definition 3.1. A pushdown system (PDS) is defined as $\mathcal{P} = (P, \Gamma, \Delta)$, where:

- P is the set of control locations,
- Γ is the stack alphabet, and
- the rules Δ is a finite subset of $(P \times \Gamma) \times (P \times \Gamma^*)$.

A rule $((p, \gamma), (p', w)) \in \Delta$ can be written on the form: $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$. A configuration of the pushdown system is a pair $\langle p, w\rangle$, where $p \in P$ and $w \in \Gamma^*$. The set of all configurations is denoted $\mathit{Conf}(\mathcal{P})$. The semantics of a pushdown system $\mathcal{P}$ is described by the unique transition system $\mathcal{T}_{\mathcal{P}} = (\mathit{Conf}(\mathcal{P}), \Rightarrow_{\mathcal{P}})$, where the transition relation $\Rightarrow_{\mathcal{P}}$ is defined based on the rules in Δ, such that if $r = \langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$ then $\langle p, \gamma w'\rangle \overset{r}{\Rightarrow}_{\mathcal{P}} \langle p', ww'\rangle$ for all $w' \in \Gamma^*$. When we are not interested in the rule r, we just write $\langle p, \gamma w'\rangle \Rightarrow_{\mathcal{P}} \langle p', ww'\rangle$. The reflexive, transitive closure of $\Rightarrow_{\mathcal{P}}$ is denoted $\Rightarrow^*_{\mathcal{P}}$.

We present an example pushdown system with $P = \{p_0, p_1, p_2, p_3\}$, $\Gamma = \{\gamma_0, \gamma_1, \gamma_2\}$ and Δ defined in Figure 3.1.


γ_1; swap(γ_0)
p_1
p_2
γ_0; push(γ_1)
γ_0; push(γ_2)
γ_1; pop
p_0
p_3
γ_2; swap(γ_1)

$$\begin{aligned}
\Delta &= \{r_1, r_2, r_3, r_4, r_5\}\\
r_1 &= \langle p_0, \gamma_0\rangle \hookrightarrow \langle p_1, \gamma_1\gamma_0\rangle\\
r_2 &= \langle p_1, \gamma_1\rangle \hookrightarrow \langle p_2, \gamma_0\rangle\\
r_3 &= \langle p_2, \gamma_0\rangle \hookrightarrow \langle p_3, \gamma_2\gamma_0\rangle\\
r_4 &= \langle p_3, \gamma_2\rangle \hookrightarrow \langle p_0, \gamma_1\rangle\\
r_5 &= \langle p_0, \gamma_1\rangle \hookrightarrow \langle p_0, \varepsilon\rangle
\end{aligned}$$

Figure 3.1. A simple example PDS used throughout this chapter, with the edge notation: Top-of-stack; Operation.

3.2 $\mathcal{P}$-automaton

For reachability analysis of a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$, we wish to represent a regular set of configurations in $\mathcal{P}$. For this we define the $\mathcal{P}$-automaton.

Definition 3.2. For a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$, a $\mathcal{P}$-automaton is a tuple $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$, where:

- $Q \supseteq P$ is a finite set of states,
- the stack alphabet of the pushdown system, Γ, is the alphabet of the $\mathcal{P}$-automaton,
- $\rightarrow \subseteq Q \times (\Gamma \cup \{\varepsilon\}) \times Q$ is the set of transitions,
- the control locations of the pushdown system, P, is the set of the initial states of the $\mathcal{P}$-automaton, and
- $F \subseteq Q$ is the set of final states.

$\mathcal{A}$ accepts a configuration $\langle p, w\rangle$ if the labelled transition system $(Q, \Gamma, \rightarrow, p)$ satisfies $p \xrightarrow{w}{}^{*} q$ for some $q \in F$. The set of configurations accepted by $\mathcal{A}$ is denoted by $Lang(\mathcal{A})$. If a set of configurations is accepted by a $\mathcal{P}$-automaton it is called a regular set. For a transition $(p, \gamma, q) \in \rightarrow$ we sometimes write it as $p \xrightarrow{\gamma} q$.

In most cases, such as for input to the *pre** and *post** algorithms presented subsequently, we restrict the $\mathcal{P}$-automaton to not contain ε-transitions, i.e. $\rightarrow \subseteq Q \times \Gamma \times Q$. The *post** algorithm may introduce ε-transitions, hence they are included in the definition here.

3.3 Reachability Analysis

Problem 3.1 (Pushdown Reachability Problem)**.** For a PDS $\mathcal{P}$ and two regular sets of configurations C and C', does there exist $c \in C$ and $c' \in C'$ such that $c \Rightarrow_{\mathcal{P}}^{*} c'$?

A simpler version of this problem, which is sufficient for the use in AalWiNes, is the case when $|C| = |C'| = 1$.

Two algorithms for reachability analysis of pushdown systems are presented in [15, 16]: *pre** and *post**, which compute the reflexive transitive closure of respectively the predecessors and successors of a regular set of configurations C. The algorithms exploits the fact that the set of predecessors or successors of a regular set of configurations is itself a regular set. This means that even though the set is potentially infinite, it has a finite representation in the form of a $\mathcal{P}$-automaton. Both algorithms take as input a $\mathcal{P}$-automaton $\mathcal{A}$ that accepts C. The output is a $\mathcal{P}$-automaton $\mathcal{A}_{pre^*}$ (or $\mathcal{A}_{post^*}$) that accepts *pre*$^*(C)$ (or *post*$^*(C)$). For *post** we allow ε-transitions in the resulting automaton. The concrete algorithms require without loss of generality that the input $\mathcal{P}$-automaton has no ε-transitions and no transitions into the initial states P.

3.3.1 Construction of pre^*

Input: $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$

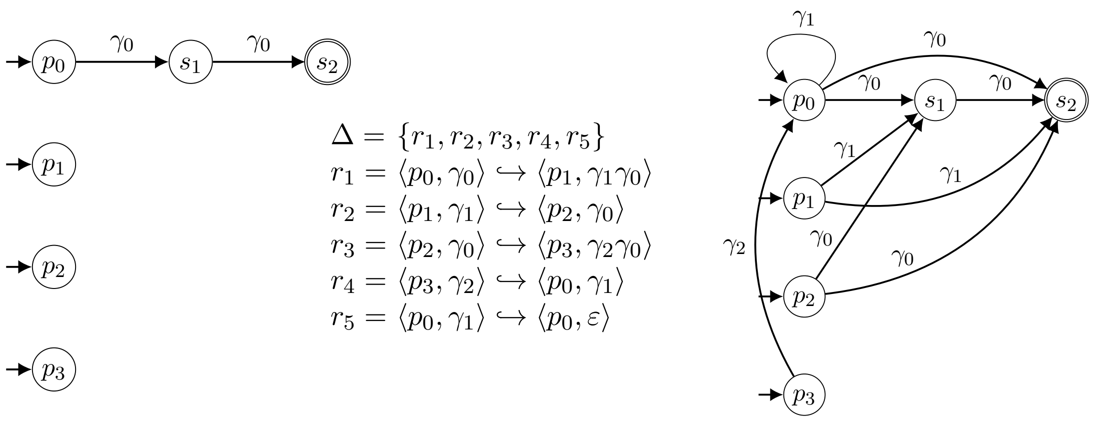
Output: $\mathcal{A}_{pre^*} = (Q, \Gamma, \rightarrow, P, F)$, where $\rightarrow$ is constructed by adding transitions to $\rightarrow_0$ according to the following saturation rule:

$$\boxed{\text{If } \langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle \text{ and } p' \xrightarrow{w}{}^* q \text{ in the current automaton, add a transition } p \xrightarrow{\gamma} q.}$$

Theorem 3.1 ([15])**.** This algorithm satisfies that $Lang(\mathcal{A}_{pre^*}) = pre^*(Lang(\mathcal{A}))$, and can be executed in $O(|Q|^2 \cdot |\Delta|)$ time and $O(|Q| \cdot |\Delta| + |\rightarrow_0|)$ space.

For two regular sets C and C' recognized by $\mathcal{A}$ and $\mathcal{A}'$ respectively, there exists $c \in C$ and $c' \in C'$ such that $c' \Rightarrow^*_{\mathcal{P}} c$ iff there exists $c' \in C'$ such that $c' \in pre^*(C)$, or in other words if $Lang(\mathcal{A}') \cap Lang(\mathcal{A}_{pre^*}) \neq \emptyset$. If $|C'| = 1$ this is simply checking whether $\mathcal{A}_{pre^*}$ accepts c', which can be solved by e.g. a depth-first-search.

We extend the $\mathcal{A}_{pre^*}$ example from [15] to match with our contribution; the example is depicted in Figure 3.2.



$$\begin{aligned}
\Delta &= \{r_1, r_2, r_3, r_4, r_5\} \\
r_1 &= \langle p_0, \gamma_0 \rangle \hookrightarrow \langle p_1, \gamma_1\gamma_0 \rangle \\
r_2 &= \langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_0 \rangle \\
r_3 &= \langle p_2, \gamma_0 \rangle \hookrightarrow \langle p_3, \gamma_2\gamma_0 \rangle \\
r_4 &= \langle p_3, \gamma_2 \rangle \hookrightarrow \langle p_0, \gamma_1 \rangle \\
r_5 &= \langle p_0, \gamma_1 \rangle \hookrightarrow \langle p_0, \varepsilon \rangle
\end{aligned}$$

Figure 3.2. On the left is an input $\mathcal{P}$-automaton $\mathcal{A}$ that accepts configuration $\langle p_0, \gamma_0\gamma_0 \rangle$. On the right is the constructed $\mathcal{A}_{pre^*}$ from applying the pre^* algorithm to $\mathcal{A}$. All initial states are marked with an empty in-going edge.

Let $\mathcal{A}$ be a $\mathcal{P}$-automaton for a pushdown system with $P = \{p_0, p_1, p_2, p_3\}$, Δ shown in Figure 3.2 such that $\mathcal{A}$ accepts $C = \{\langle p_0, \gamma_0\gamma_0 \rangle\}$. After adding all transitions to $\rightarrow$ we have the final $\mathcal{A}_{pre^*}$ also depicted in Figure 3.2. The step-by-step construction of the example $\mathcal{A}_{pre^*}$ is described below:

1. The first match to the saturation rule is r_5 and thus the first transition we add to $\rightarrow$ is $p_0 \xrightarrow{\gamma_1} p_0$.
2. With the new set of transitions we get a match for r_4 and add $p_3 \xrightarrow{\gamma_2} p_0$.
3. With r_3 we can now saturate path $p_3 \xrightarrow{\gamma_2} p_0 \xrightarrow{\gamma_0} s_1$, and thus, add $p_2 \xrightarrow{\gamma_0} s_1$.
4. From r_2 and the current path $p_2 \xrightarrow{\gamma_0} s_1$ we add $p_1 \xrightarrow{\gamma_1} s_1$.
5. From r_1 and the current path $p_1 \xrightarrow{\gamma_1\gamma_0} s_2$ we add $p_0 \xrightarrow{\gamma_0} s_2$.
6. With the new path $p_3 \xrightarrow{\gamma_2\gamma_0} s_2$ we can again use r_3 and, hence, add $p_2 \xrightarrow{\gamma_0} s_2$.
7. Similar with path $p_2 \xrightarrow{\gamma_0} s_2$ we can use r_2 to add $p_1 \xrightarrow{\gamma_1} s_2$

3.3.2 Construction of $post^*$

Input: $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$

Output: $\mathcal{A}_{post^*} = (Q', \Gamma, \rightarrow, P, F)$, where Q' is obtained by adding new states to Q. For each pair (p', γ'), such that $\mathcal{P}$ contains at least one rule of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$, a new state $q_{p',\gamma'}$ is added.

Then $\rightarrow$ is constructed by adding transitions to $\rightarrow_0$ according to the following saturation rules:

(i) If $\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle$ and $p \xrightarrow{\gamma}{}^* q$ in the current automaton, add a transition $p' \xrightarrow{\varepsilon} q$.

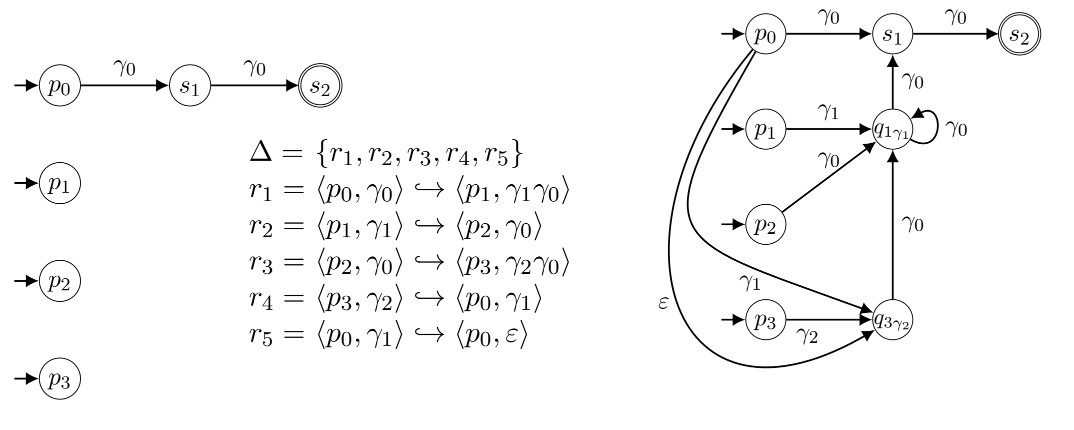
(ii) If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ and $p \xrightarrow{\gamma}{}^* q$ in the current automaton, add a transition $p' \xrightarrow{\gamma'} q$.

(iii) If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ and $p \xrightarrow{\gamma}{}^* q$ in the current automaton, first add $p' \xrightarrow{\gamma'} q_{p',\gamma'}$ and then $q_{p',\gamma'} \xrightarrow{\gamma''} q$.

Theorem 3.2 ([15])**.** This algorithm satisfies that $Lang(\mathcal{A}_{post^*}) = post^*(Lang(\mathcal{A}))$, and can be executed in $O(|P|{\cdot}|\Delta|{\cdot}(n_1{+}n_2){+}|P|{\cdot}|{\rightarrow_0}|)$ time and space, where $n_1 = |Q \backslash P|$ and n_2 is the number of different pairs (p, γ) such that there is a rule of the form $\langle p', \gamma' \rangle \hookrightarrow \langle p, \gamma\gamma'' \rangle$ in Δ.

For two regular sets C and C' recognized by $\mathcal{A}$ and $\mathcal{A}'$ respectively, there exists $c \in C$ and $c' \in C'$ such that $c \Rightarrow^*_{\mathcal{P}} c'$ iff there exists $c' \in C'$ such that $c' \in post^*(C)$, or in other words if $Lang(\mathcal{A}') \cap Lang(\mathcal{A}_{post^*}) \neq \emptyset$. If $|C'| = 1$ this is simply checking whether $\mathcal{A}_{post^*}$ accepts c', which can be solved by e.g. a depth-first-search.

An example similar to Figure 3.2 is made for $\mathcal{A}_{post^*}$ in [15] and extended in Figure 3.3.



$$\begin{aligned}
\Delta &= \{r_1, r_2, r_3, r_4, r_5\} \\
r_1 &= \langle p_0, \gamma_0 \rangle \hookrightarrow \langle p_1, \gamma_1\gamma_0 \rangle \\
r_2 &= \langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_0 \rangle \\
r_3 &= \langle p_2, \gamma_0 \rangle \hookrightarrow \langle p_3, \gamma_2\gamma_0 \rangle \\
r_4 &= \langle p_3, \gamma_2 \rangle \hookrightarrow \langle p_0, \gamma_1 \rangle \\
r_5 &= \langle p_0, \gamma_1 \rangle \hookrightarrow \langle p_0, \varepsilon \rangle
\end{aligned}$$

Figure 3.3. On the left is an input $\mathcal{P}$-automaton $\mathcal{A}$ that accepts configuration $\langle p_0, \gamma_0\gamma_0 \rangle$. On the right is the constructed $\mathcal{A}_{post^*}$ from applying the $post^*$ algorithm to $\mathcal{A}$. All initial states are marked with an empty in-going edge.

After adding all saturating transitions to $\rightarrow$ and states to Q of $\mathcal{A}$ then we have the final $\mathcal{A}_{post^*}$ depicted in Figure 3.3. The example process of constructing $\mathcal{A}_{post^*}$ is iterated below:

1. From r_1 and r_3 we add two additional states: $q_{1\gamma_1}$ and $q_{3\gamma_2}$, respectively.
2. The left-hand side of r_1 matches the current transition $p_0 \xrightarrow{\gamma_0} s_1$, thus, we use saturation rule (iii) to add the two following transitions: $p_1 \xrightarrow{\gamma_1} q_{1\gamma_1}$ and $q_{1\gamma_1} \xrightarrow{\gamma_0} s_1$.
3. Then with r_2 and saturation (ii) we add $p_2 \xrightarrow{\gamma_0} q_{1\gamma_1}$.
4. With the new transition $p_2 \xrightarrow{\gamma_0} q_{1\gamma_1}$ and r_3 we add two more transitions $p_3 \xrightarrow{\gamma_2} q_{3\gamma_2}$ and $q_{3\gamma_2} \xrightarrow{\gamma_0} q_{1\gamma_1}$.
5. From $p_3 \xrightarrow{\gamma_2} q_{3\gamma_2}$ and r_4 we saturate (ii) and add $p_0 \xrightarrow{\gamma_1} q_{3\gamma_2}$.
6. Then we use the just added transition $p_0 \xrightarrow{\gamma_1} q_{3\gamma_2}$ and r_5 to saturate (i) and add the ε-edge $p_0 \xrightarrow{\varepsilon} q_{3\gamma_2}$
7. The final transition is added from $p_0 \xrightarrow{\gamma_0}{}^* q_{1\gamma_1}$, and r_1 with saturation rule (iii), because $p_1 \xrightarrow{\gamma_1} q_{1\gamma_1}$ already exist, we just add $q_{1\gamma_1} \xrightarrow{\gamma_0} q_{1\gamma_1}$.

3.4 Early Termination

To improve the running time of the *pre** and the *post** algorithms, we can add an early termination check, which will be described in this section. The reasoning is the same for both *pre** and *post**.

During the saturation process, let $\rightarrow_i$ be the transition relation after adding i transitions to $\rightarrow_0$ using the saturation rule(s), and let $\mathcal{A}_i$ be the corresponding intermediate automaton. During the saturation process we get a sequence of automata $\mathcal{A}_0, \ldots, \mathcal{A}_n$, where it holds that the languages of the automata increases, i.e. $\mathit{Lang}(\mathcal{A}_i) \subseteq \mathit{Lang}(\mathcal{A}_{i+1})$. When the saturation process stops, we are left with the final automaton $\mathcal{A}_n$ which is $\mathcal{A}_{pre^*}$ (or $\mathcal{A}_{post^*}$).

When solving the pushdown reachability problem, we need to determine whether any configuration $c' \in C'$ is an element of $pre^*(C)$ (or $post^*(C)$). If $C' \cap \mathit{Lang}(\mathcal{A}_i) \neq \emptyset$ for some intermediate automaton $\mathcal{A}_i$, it is not necessary to compute the final automaton $\mathcal{A}_n$, since the saturation rule(s) are only increasing the accepted language. If this is the case, the reachability algorithm can terminate early giving a positive result. Otherwise the algorithm needs to continue.

The purpose of early termination is to speed up the algorithm by terminating as soon as the answer is known. For this to be efficient, the overhead of checking $C' \cap \mathit{Lang}(\mathcal{A}_i) \neq \emptyset$ may not be too large.

Problem 3.2 (Early Termination Problem)**.** Let $\mathcal{A}'$ be the $\mathcal{P}$-automaton accepting C', and $\mathcal{A}_i$ the current intermediate automaton. The early termination problem is to determine whether $\mathit{Lang}(\mathcal{A}') \cap \mathit{Lang}(\mathcal{A}_i) \neq \emptyset$.

The general problem of deciding intersection non-emptiness between two finite automata, is NL-complete [17]. It can be solved by a well known product construction [18]. However, this is be too expensive to do in every iteration of the saturation rule. Instead it can be done for every k iterations, for a sufficiently large k, and possibly executed in parallel of the saturation algorithm. This line of inquiry is not followed further, since the use in AalWiNes restricts itself to a particular special case, which has a simple and easy solution.

The special case is when $C' = \{\langle p, \gamma \rangle\}$, i.e. C' only contains a single configuration, in which the stack has one element. The early termination problem is solved for this special case by checking whether a transition $p \xrightarrow{\gamma} q$, where $q \in F$ is a final state, is added to $\mathcal{A}_i$. This is simply a constant-time check, when adding transitions using the saturation rule.

The correctness of this early termination check is obvious for the *pre** algorithm. For the *post** algorithm the check is obviously sound, but it is not complete, since the saturation rules in *post** may add ε-transitions. If $\mathcal{A}_i$ contains the transitions $p \xrightarrow{\varepsilon} q'$ and $q' \xrightarrow{\gamma} q$, with $q \in F$, then $\langle p, \gamma \rangle$ is accepted by $\mathcal{A}_i$ even though no $p \xrightarrow{\gamma} q$, with $q \in F$, was added. In reality this is not a problem, since the concrete algorithm for *post** [15] will also later add $p \xrightarrow{\gamma} q$ in this case. In the implementation of *post** in PDAAAL, we use a last-in-first-out stack for transitions, which means that $p \xrightarrow{\gamma} q$ will in fact be the next transition to be added, so early termination will occur only one iteration later.

Weighted Pushdown Systems 4

The previous chapter introduced pushdown systems, the reachability problem, and the *pre** and *post** algorithms to solve it. This is enough to solve the query satisfiability problem (Problem 2.1) using the reduction to the pushdown reachability problem (Problem 3.1) presented in [4].

In this chapter we extend pushdown systems with weights in order to solve the weighted trace problem (Problem 2.2). Weighted pushdown systems and their *pre** and *post** algorithms are presented in [19] for their application to interprocedural dataflow analysis. To this end they define the weight domain in a rather abstract way based on a semiring. This definition will become relevant when discussing the weight encoding of the maximum stack-size for a trace. For the other weight properties, it is sufficient to restrict the weight domain to be totally ordered, as initially discussed in [15], which will allow for a more efficient algorithm.

Recall that a monoid is an algebraic structure $(M, \bullet)$, where M is a set and $\bullet : M \times M \to M$ is a binary operator on M, such that the following two properties hold:

- Associativity: $\forall a, b, c \in M,\ (a \bullet b) \bullet c = a \bullet (b \bullet c)$,
- Identity: $\exists e \in M,\ \forall a \in M,\ e \bullet a = a = a \bullet e$. Here e is called the identity element.

In a commutative monoid, the following additional property holds:

- Commutativity: $\forall a, b \in M,\ a \bullet b = b \bullet a$

Definition 4.1. A well-ordered bounded semiring is a tuple $\mathcal{S} = (D, \sqcap, \oplus, \top, \bot)$ where D is a set, $\top$ and $\bot$ are elements in D, and $\sqcap$ and $\oplus$ are binary operators on D, such that:

- $(D, \sqcap)$ is a commutative monoid with the identity element $\top$.
- $(D, \oplus)$ is a monoid with the identity element $\bot$.
- $\oplus$ distributes over $\sqcap$: $\forall a, b, c \in D$ we have $a \oplus (b \sqcap c) = (a \oplus b) \sqcap (a \oplus c)$ and $(a \sqcap b) \oplus c = (a \oplus c) \sqcap (b \oplus c)$.
- $\top$ is an annihilator for $\oplus$: $\forall a \in D, a \oplus \top = \top = \top \oplus a$.
- $\bot$ is an annihilator for $\sqcap$: $\forall a \in D, \bot \sqcap a = \bot$.
- The order $\sqsubseteq$ defined by: $\forall a, b \in D, a \sqsubseteq b$ iff $a \sqcap b = a$ is a well-order (i.e. a total order with no infinite descending chains).

The first four conditions constitutes the definition of a semiring, and the first five conditions that of a bounded semiring [20]. By Lemma 3 in [20], this is also an idempotent semiring, so $\forall a \in D, a \sqcap a = a$. By Lemma 2 in [20], D provided with the natural order $\sqsubseteq$ is monotonic, so for all $a, b, c \in D$: $a \sqsubseteq b \implies (a \oplus c \sqsubseteq b \oplus c) \wedge (c \oplus a \sqsubseteq c \oplus b)$. This implies that for all $a, b \in D$: $a \sqsubseteq a \oplus b$ and $a \sqsubseteq b \oplus a$.

Lemma 4.1. Given Definition 4.1, the following properties hold:

(a) $(D, \sqsubseteq)$ is a meet-semilattice and its binary meet (greatest lower bound) operator is exactly $\sqcap$.
(b) $\top$ is the top element of $(D, \sqsubseteq)$.
(c) $\bot$ is the bottom element of $(D, \sqsubseteq)$.

Proof. (a) This follows from $\sqcap$ being associative, commutative and idempotent.
(b) Since $\top$ is the neutral element in the commutative monoid $(D, \sqcap)$, we have $a \sqcap \top = a$ and thus $a \sqsubseteq \top$ for all $a \in D$, so $\top$ is the top element of $(D, \sqsubseteq)$.
c. Since $\forall a \in D, \bot \sqcap a = \bot$, we have $\bot \sqsubseteq a$ for all $a \in D$, so $\bot$ is the bottom element of $(D, \sqsubseteq)$. □

Lemma 4.1 justifies using the notation of $\sqcap$, $\top$ and $\bot$, rather than the usual notation for an arbitrary semiring $(D, \oplus, \otimes, \mathbf{0}, \mathbf{1})$.

Given a total order $(D, \sqsubseteq)$ the meet operator follows directly as $a \sqcap b = \begin{cases} a & \text{if } a \sqsubseteq b \\ b & \text{otherwise} \end{cases}$

An example of a weight domain is the set of vectors of non-negative integers: $\mathcal{S} = (\mathbb{N}_0^n \cup \infty^n, \min, +, \infty^n, 0^n)$ for some $n > 0$, where the total-order $(\mathbb{N}_0^n, \leq)$ is the lexicographical ordering of vectors, and $+$ is element-wise vector addition. This definition satisfies the requirements of a well-ordered bounded semiring.

4.1 Weight Extension for Pushdown Systems

With the weight domain defined, we are now ready to extend our definition of pushdown systems with weights.

Definition 4.2 (Weighted Pushdown System [19][1]). A weighted pushdown system is defined as a tuple $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, where:

- $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system,
- $\mathcal{S} = (D, \sqcap, \oplus, \top, \bot)$ is a well-ordered bounded semiring, and
- $f : \Delta \to (D \setminus \{\top\})$ is a function that assigns a value from D to each rule in $\mathcal{P}$.

Let $\sigma \in \Delta^*$ be a sequence of rules, and for any $\sigma = r_1 \dots r_k$ define $v(\sigma) = f(r_1) \oplus \cdots \oplus f(r_k)$. For any two configurations c and c' of $\mathcal{P}$ let *path*(c, c') denote the set of all sequences of rules $r_1 \dots r_k$ that transforms c into c' by $c \overset{r_1}{\Rightarrow}_{\mathcal{P}} \cdots \overset{r_k}{\Rightarrow}_{\mathcal{P}} c'$. Let $c \xRightarrow[d]{\sigma}{}^{\oplus} c'$ denote the existence of a sequence $\sigma \in$ *path*(c, c'), where $d = v(\sigma)$. If the sequence is not relevant for the discussion, we may just write $c \xRightarrow[d]{}{}^{\oplus} c'$. For technical convenience define $\bigsqcap \emptyset = \top$, and if *path*$(c, c') = \emptyset$ then say $c \xRightarrow[\top]{}{}^{\oplus} c'$.

[1]This follows the definition in [19], except from the requirement of the weight domain to be totally ordered, which is not the case in [19].

Definition 4.3 (Shortest Pushdown System Distance)**.** For a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$, define for any two configurations $c, c' \in P \times \Gamma^*$ the shortest pushdown system distance: $\delta(c, c') = \bigsqcap\{d \mid c \overset{\oplus}{\underset{d}{\Longrightarrow}} c'\}$.

T. Reps et al. [19] defines two reachability problems: finding shortest distance to respectively the predecessors and the successors of a regular set of configurations. Given the assumption of a totally ordered weight domain we can generalise these two problems into the following reachability problem:

Problem 4.1 (Weighted Pushdown Reachability Problem)**.** For a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$, and regular sets $C \subseteq P \times \Gamma^*$ and $C' \subseteq P \times \Gamma^*$, find:

- the shortest distance $\delta = \bigsqcap\{\delta(c, c') \mid c \in C,\ c' \in C'\}$, and
- a witness path $\omega \in \bigcup_{c \in C, c' \in C'} path(c, c')$ such that $v(\omega) = \delta$.

The weighted predecessors and successors problems are solved in [19] for the general case, where the weight domain is not necessarily totally ordered, using weighted versions of respectively the *pre** and *post** algorithm. The output of those algorithms is a pair of a $\mathcal{P}$-automaton and a labelling function. For convenience we make the following definition:

Definition 4.4. For a weighted PDS $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ with $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{S} = (D, \sqcap, \oplus, \top, \bot)$, we define a $\mathcal{W}$-automaton (or weighted $\mathcal{P}$-automaton) as a pair $\mathcal{V} = (\mathcal{A}, l)$, where:

- $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$ is a $\mathcal{P}$-automaton, and
- $l : (Q \times (\Gamma \cup \{\varepsilon\}) \times Q) \rightarrow D$ is a function labelling transitions in $\mathcal{A}$ with weights, such that $l(t) = \top$ if and only if $t \notin \rightarrow$.

For a $\mathcal{W}$-automaton $(\mathcal{A}, l)$ we are not only interested in whether $\mathcal{A}$ accepts a configuration $\langle p, w \rangle$, but also in the total weight of the accepting path in the $\mathcal{W}$-automaton, where l is the weight function. For any $q, q' \in Q$ and $w \in \Gamma^*$, we let $q \xrightarrow[d]{w}{}^{\oplus} q'$ denote the existence of states $q_1, \ldots, q_{n-1}$ and $\gamma_1, \ldots, \gamma_n \in (\Gamma \cup \{\varepsilon\})$ such that $l(q_{n-1}, \gamma_n, q') \oplus \cdots \oplus l(q, \gamma_1, q_1) = d$ and $\gamma_1 \ldots \gamma_n = w$. By convention we say $q \xrightarrow[\bot]{\varepsilon}{}^{\oplus} q$ for any $q \in Q$.

Definition 4.5 (Shortest Accepting Path)**.** For a $\mathcal{W}$-automaton $\mathcal{V} = (\mathcal{A}, l)$, and any configuration in $\mathcal{W}$: $\langle p, w \rangle \in P \times \Gamma^*$, we define $\rho(\langle p, w \rangle) = \bigsqcap\{d \mid \exists q_f \in F : p \xrightarrow[d]{w}{}^{\oplus} q_f\}$, which is the length of a shortest w-labelled path from p to a final state in $\mathcal{A}$.

4.2 Efficient Weighted *post** Algorithm

In this section we present an algorithm that solves the weighted pushdown reachability problem, by constructing a $\mathcal{W}$-automaton $\mathcal{V}_{post^*} = (\mathcal{A}_{post^*}, l)$ in a way such that a shortest distance δ can be reconstructed. It does this more efficiently than the general algorithm presented in [19] by exploiting the total order of the weight domain. The algorithm in [19] may process each transition multiple times, whereas our algorithm only processes each transition once.

In Subsection 4.2.2 we present as an example a family of weighted pushdown systems for which our algorithm has asymptotically smaller run-time than the algorithm in [19].

We use a priority queue, with the ability to pop minimum element, and on insertion of element t with priority d, $queue[t] \coloneqq d$, checks if t is already present and in that case only updates the priority of t, if the new priority is less than the existing, or if t is not present, it is inserted normally.

The following function is used in Algorithm 3. The variables *queue*, *cost*, l and $\rightarrow$ are part of Algorithm 3 and are implicitly passed by reference.

Algorithm 2 Update function

```
1: function UPDATE($t = p \xrightarrow{\gamma} q$, $d$)
2:     if $t \notin \rightarrow$ then
3:         $l(t) \coloneqq l(t) \sqcap d$
4:         if $q \in Q$ then
5:             $queue[t] \coloneqq d$
6:         else
7:             $queue[t] \coloneqq cost(q) \oplus d$
```

Algorithm 3 implements the saturation process described in Subsection 3.3.2. In short, the saturation rule can be applied for some p, γ, q, if $p \xrightarrow{\gamma}^* q$ in the current automaton and there is a matching rule with $\langle p, \gamma \rangle$ on the left hand side. This is implemented by going through the transitions $t = p \xrightarrow{\gamma} q$ one by one, and finding all matching rules in Δ. The application of the saturation rule adds new transitions to the workset *queue*, so they can be processed later. Since the process can add ε-transitions, we may have $p \xrightarrow{\gamma}^* q$ due to the transitions $p \xrightarrow{\varepsilon} q'$ and $q' \xrightarrow{\gamma} q$. To handle this case, we contract ε-transitions thus adding $p \xrightarrow{\gamma} q$ - this is what lines 24 and 27 are for. Only transitions going from a state in P can match with a rule in Δ, so transitions going from a state in $Q' \setminus P$ are added directly to $\rightarrow$, which happens on line 6 and 18.

The description so far applies also to the unweighted version in [15]. What makes Algorithm 3 able to find the shortest trace, is that the workset, *queue*, of transitions still to be considered, is implemented as a priority queue, where the priority of transition t is the cost of the current shortest path from an initial configuration, accepted by $\mathcal{A}$, to a configuration, whose accepting path starts with t.

If we later attempt to add the transition t again, due to a shorter path, we update the priority with this new cost. When processing a transition with priority d popped from the *queue* at line 8, the new transitions added to *queue*, cannot have a priority less than d. This means that once a transition is popped from the priority queue on line 8, its priority d is the final shortest path cost for that transition.

The cost of the shortest path in the pushdown system from an initial configuration, accepted by $\mathcal{A}$, to a configuration c, is encoded in the $\mathcal{P}$-automaton as the cost of the shortest path accepting c. When processing the saturation rules, we only need the first transition $p \xrightarrow{\gamma} q$ of this path, but for computing the transitions priority in the queue, we need the total cost of the path. If $q \in Q$ this is $\perp$, since q must be on an accepting path

Algorithm 3 Shortest trace *post** algorithm for weighted pushdown systems

```
Require: $\mathcal{A}$ has no $\varepsilon$-transitions, no transitions into initial states and all transitions are
    part of an accepting path.
1: function ShortPost*($\mathcal{W} = ((P, \Gamma, \Delta), (D, \sqcap, \oplus, \top, \bot), f)$, $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$)
2:     $Q' := Q \cup \{q_{p',\gamma'} \mid \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \in \Delta\}$
3:     $queue[t] := \bot,\ \forall t \in (\rightarrow_0) \cap (P \times \Gamma \times Q)$ ▷ priority queue: $(P \times (\Gamma \cup \{\varepsilon\}) \times Q') \rightarrow D$
4:     $l(t) := \begin{cases} \bot & \text{if } t \in \rightarrow_0 \\ \top & \text{if } t \in (Q' \times (\Gamma \cup \{\varepsilon\}) \times Q') \setminus \rightarrow_0 \end{cases}$
5:     $cost(q) := \top \quad \forall q \in (Q' \setminus Q)$
6:     $\rightarrow := (\rightarrow_0) \cap ((Q \setminus P) \times \Gamma \times Q)$
7:     while $queue \neq \emptyset$ do
8:         $(t = p \xrightarrow{\gamma} q, d) :=$ PopMin($queue$)          ▷ Gets element and priority
9:         $\rightarrow := \rightarrow \cup \{t\}$
10:        if $\gamma \neq \varepsilon$ then
11:            for $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$ do
12:                Update($p' \xrightarrow{\varepsilon} q,\ l(t) \oplus f(r)$)
13:            for $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta$ do
14:                Update($p' \xrightarrow{\gamma'} q,\ l(t) \oplus f(r)$)
15:            for $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \in \Delta$ do
16:                $t_1 = p' \xrightarrow{\gamma'} q_{p',\gamma'},\ t_2 = q_{p',\gamma'} \xrightarrow{\gamma''} q$
17:                $l(t_2) := l(t_2) \sqcap (l(t) \oplus f(r))$
18:                $\rightarrow := \rightarrow \cup \{t_2\}$
19:                if $d \oplus f(r) \sqsubset cost(q_{p',\gamma'})$ then
20:                    $cost(q_{p',\gamma'}) := d \oplus f(r)$
21:                    Update($t_1$, $\bot$)
22:                else if $l(t_2)$ was changed on line 17 then
23:                    for $t' = (p'', \varepsilon, q_{p',\gamma'}) \in \rightarrow$ do
24:                        Update($p'' \xrightarrow{\gamma''} q,\ l(t_2) \oplus l(t')$)
25:        else
26:            for $t' = (q, \gamma', q') \in \rightarrow$ do
27:                Update($p \xrightarrow{\gamma'} q',\ l(t') \oplus l(t)$)
28:    return $((Q', \Gamma, \rightarrow, P, F), l)$          ▷ Return a $\mathcal{W}$-automaton
```

in the original $\mathcal{P}$-automaton $\mathcal{A}$, which by definition has cost $\bot$. For states $q \in Q' \setminus Q$ we store the cost of the current shortest path from q to a final state in the function $cost(q)$. The *cost* values can be updated on-the-fly (line 20), and allows us to efficiently compute the priority of the transition (line 7 in Algorithm 2).

Finally the algorithm makes sure to store the transition weights in the function l (on line 3 in Algorithm 2 and line 17 in Algorithm 3). The if-else conditions on lines 19 - 24 makes sure we only update transitions in the priority queue, when there is a possibility of a shorter path to them.

After this informal explanation of the algorithm, we move on to prove its correctness in Theorem 4.2. For a set $C \subseteq P \times \Gamma^*$ and a configuration $c' \in P \times \Gamma^*$ we define: $\delta_C(c') = \bigsqcap\{\delta(c, c') \mid c \in C\}$, which is the shortest distance to c' from any configuration in C.

Theorem 4.2 ([15])**.** For a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$, and a regular set of configurations $C \subseteq P \times \Gamma^*$ accepted by $\mathcal{A}$, let $\mathcal{V}_{post^*} = (\mathcal{A}_{post^*}, l)$ be the output of running Algorithm 3 on $\mathcal{W}$ and $\mathcal{A}$. For any configuration $c' \in P \times \Gamma^*$, we have $\delta_C(c') = \rho(c')$ in $\mathcal{V}_{post^*}$.

Proof. In [15] a procedure for computing shortest trace using *post** is described and proved correct, i.e. that $\delta_C(c') = \rho(c')$. To prove Theorem 4.2, we only need to show that Algorithm 3 corresponds to this procedure. The key difference is that the procedure in [15] does not explicitly use a priority queue to determine which transition to pick in each iteration. So we need to prove that the priority of the transition popped at line 8 in Algorithm 3 is the same as the value defined declaratively in the procedure in [15]. This is proved in Lemma 4.3 using the invariant in Lemma 4.4 and the auxiliary Lemma 4.5. $\square$

First we need to define some notation. Let $queue_i$, l_i, $cost_i$, $\rightarrow_i$ denote the corresponding values in Algorithm 3 before iteration i of the while-loop on lines 7 to 27, for $i \in \{1, 2, \dots\}$.

The intermediate $\mathcal{W}$-automaton before iteration i is $\mathcal{V}_i = (\mathcal{A}_i, l_i)$, where $\mathcal{A}_i = (Q', \Gamma, \rightarrow_i \cup queue_i, P, F)$, where we use transitions both in $\rightarrow_i$ and in $queue_i$. It is easy to verify that $l_i(t) = \top$ if and only if $t \notin \rightarrow_i \cup\, queue_i$, so $\mathcal{V}_i$ does conform to the definition of a $\mathcal{W}$-automaton.

Let $p \xrightarrow[d]{w}{}^{\oplus}_{i}\, q$ denote that the property holds in $\mathcal{V}_i$. Let d_i be the priority of the transition popped from $queue_i$ at line 8 in iteration i of the while-loop.

It is easy to check that $\rightarrow_i \cup\, queue_i \subseteq (P \times (\Gamma \cup \{\varepsilon\}) \times (Q' \setminus P)) \cup ((Q' \setminus P) \times \Gamma \times (Q' \setminus P))$ for all i, i.e. that no transitions go into states in P, and all ε-transitions go from a state in P. This fact is used without reference in the remainder of these proofs.

Define $\eta_i(p \xrightarrow{\gamma} q) = \bigsqcap\{d_2 \oplus d_1 \mid p \xrightarrow[d_1]{\gamma}{}^{\oplus}_{i}\, q \xrightarrow[d_2]{w}{}^{\oplus}_{i}\, q_f,\ q_f \in F,\ w \in \Gamma^*\}$, which before round i is the shortest accepting path starting from p with a word starting with γ. Note that this path starts either with transition $p \xrightarrow{\gamma} q$ or with transitions $p \xrightarrow{\varepsilon} q' \xrightarrow{\gamma} q$ for some $q' \in Q' \setminus P$.

Define $\kappa_i(q) = \bigsqcap\{d \mid q \xrightarrow[d]{w}{}^{\oplus}_{i}\, q_f,\ q_f \in F,\ w \in \Gamma^*\}$ which is the shortest accepting path from q before round i.

First we state some facts that follow directly from unfolding the definitions above:
For all $p \xrightarrow{\gamma} q \in P \times \Gamma \times Q'$, for all i:

$$\eta_i(p \xrightarrow{\gamma} q) = \kappa_i(q) \oplus \Big(\bigsqcap\{l_i(q' \xrightarrow{\gamma} q) \oplus l_i(p \xrightarrow{\varepsilon} q') \mid q' \in (Q' \setminus P)\} \sqcap l_i(p \xrightarrow{\gamma} q)\Big) \tag{4.1}$$

For all $p \xrightarrow{\varepsilon} q \in P \times \{\varepsilon\} \times Q'$, for all i:

$$\eta_i(p \xrightarrow{\varepsilon} q) = \kappa_i(q) \oplus l_i(p \xrightarrow{\varepsilon} q) \tag{4.2}$$

For all $q, q' \in Q' \setminus P$ and $\gamma \in \Gamma$, for all i:

$$\kappa_i(q') \sqsubseteq \kappa_i(q) \oplus l_i(q' \xrightarrow{\gamma} q) \tag{4.3}$$

To proof Theorem 4.2, we need to show:

Lemma 4.3. The priority d_i of the transition $p \xrightarrow{\gamma} q$ popped on line 8 in round i is: $d_i = \eta_i(p \xrightarrow{\gamma} q)$.

Proof. We use the invariant from Lemma 4.4. If $\eta_i(p \xrightarrow{\gamma} q) = \kappa_i(q) \oplus l_i(p \xrightarrow{\gamma} q)$ then it follows directly from the invariant. Otherwise, assume $\eta_i(p \xrightarrow{\gamma} q) \neq \kappa_i(q) \oplus l_i(p \xrightarrow{\gamma} q)$ with the purpose of arriving at a contradiction. Due to Equation 4.1 and Equation 4.2 there must be a $q \in Q' \setminus P$ such that $\eta_i(p \xrightarrow{\gamma} q) = \kappa_i(q) \oplus l_i(q' \xrightarrow{\gamma} q) \oplus l_i(p \xrightarrow{\varepsilon} q')$ and $l_i(q' \xrightarrow{\gamma} q) \oplus l_i(p \xrightarrow{\varepsilon} q') \sqsubset l_i(p \xrightarrow{\gamma} q)$. From this and Equation 4.3 we have:

$$\kappa_i(q') \oplus l_i(p \xrightarrow{\varepsilon} q') \sqsubseteq \kappa_i(q) \oplus l_i(q' \xrightarrow{\gamma} q) \oplus l_i(p \xrightarrow{\varepsilon} q') \sqsubset \kappa_i(q) \oplus l_i(p \xrightarrow{\gamma} q)$$

From $\kappa_i(q') \oplus l_i(p \xrightarrow{\varepsilon} q') \sqsubset \kappa_i(q) \oplus l_i(p \xrightarrow{\gamma} q)$ and the invariant, we know that $p \xrightarrow{\varepsilon} q'$ must have been popped from the queue in an earlier iteration $j < i$.

If $q' \xrightarrow{\gamma} q \notin \rightarrow_j$ then it is added as t_2 on line 18 in an iteration k, $j < k \leq i$, but then by Lemma 4.5 the only transition going into q' in round k is t_1, which contradicts that $p \xrightarrow{\varepsilon} q'$ was popped from the queue in round j. So we must have $q' \xrightarrow{\gamma} q \in \rightarrow_j$.

Hence on line 27 we update $p \xrightarrow{\gamma} q$ so that $l_{j+1}(p \xrightarrow{\gamma} q) \sqsubseteq l_j(q' \xrightarrow{\gamma} q) \oplus l_j(p \xrightarrow{\varepsilon} q')$. Since $p \xrightarrow{\varepsilon} q' \in \rightarrow_{j+1}$, it is not later updated, so we have $l_j(p \xrightarrow{\varepsilon} q') = l_i(p \xrightarrow{\varepsilon} q')$.

If $l_j(q' \xrightarrow{\gamma} q) = l_i(q' \xrightarrow{\gamma} q)$, then $l_i(p \xrightarrow{\gamma} q) \sqsubseteq l_i(q' \xrightarrow{\gamma} q) \oplus l_i(p \xrightarrow{\varepsilon} q')$ contradicting $l_i(q' \xrightarrow{\gamma} q) \oplus l_i(p \xrightarrow{\varepsilon} q') \sqsubset l_i(p \xrightarrow{\gamma} q)$.

So $q' \xrightarrow{\gamma} q$ must have been updated as t_2 on line 17 in an iteration k, $j \leq k < i$. Consider the last such k. We have $l_i(q' \xrightarrow{\gamma} q) = l_{k+1}(q' \xrightarrow{\gamma} q) = l_k(t) \oplus f(r)$. If $d_k \oplus f(r) \sqsubset cost_k(q')$ then by Lemma 4.5 the only transition going into q' in round k is t_1, which is a contradiction. So $cost_k(q') \sqsubseteq d_k \oplus f(r)$, and on line 24 we update $p \xrightarrow{\gamma} q$ so that $l_{k+1}(p \xrightarrow{\gamma} q) \sqsubseteq l_{k+1}(q' \xrightarrow{\gamma} q) \oplus l_k(p \xrightarrow{\varepsilon} q')$. Since k was the last change to $q' \xrightarrow{\gamma} q$ we have $l_i(p \xrightarrow{\gamma} q) \sqsubseteq l_i(q' \xrightarrow{\gamma} q) \oplus l_i(p \xrightarrow{\varepsilon} q')$ contradicting $l_i(q' \xrightarrow{\gamma} q) \oplus l_i(p \xrightarrow{\varepsilon} q') \sqsubset l_i(p \xrightarrow{\gamma} q)$. This completes the proof. □

The following proofs of Lemma 4.4 and Lemma 4.5 use some insights from less formal explanations in [15].

Lemma 4.4. Before round i:

(a) The priority of each $p \xrightarrow{\gamma} q \in P \times (\Gamma \cup \{\varepsilon\}) \times Q'$ in $queue_i$ is $\kappa_i(q) \oplus l_i(p \xrightarrow{\gamma} q)$.
(b) For all $q \in Q' \setminus Q$: $cost_i(q) = \kappa_i(q)$.

Proof. By strong induction on i. However we need to use a more fine-grained notion of iteration than the outer while-loop. For the induction we iterate over the inner for-loops, so the induction hypothesis holds before each iteration of a for-loop, and the induction step carries out one such iteration. Since $\mathcal{V}_i$ is still a valid $\mathcal{W}$-automaton before each for-loop iteration, most of the notation generalises to this fine-grained definition of i. Only d_i needs special attention, since d_i is only defined at the beginning of the while loop, i.e. before the first for-loop iteration. In other cases d_i will refer to d_j, where $j \leq i$ is the latest iteration in which d_j is defined. Due to the induction hypothesis we have $d_i = d_j = \kappa_j(q) \oplus l_j(p \xrightarrow{\gamma} q)$. Since $p \xrightarrow{\gamma} q$ is added to $\rightarrow$ in iteration j, we have $l_j(p \xrightarrow{\gamma} q) = l_i(p \xrightarrow{\gamma} q)$. The only possibility of a change to $\kappa_j(q)$ is a change to $l(t_2)$, but then $t_2 = q \xrightarrow{\gamma''} q$ is a self loop with non-negative weight, so this cannot decrease the shortest accepting path. So $\kappa_j(q) = \kappa_i(q)$, and $d_i = \kappa_i(q) \oplus l_i(p \xrightarrow{\gamma} q)$.

Basis. $i = 1$ (before starting the first round). (a) *queue* only contains transitions from $\rightarrow_0$ with priority $\bot$. All transitions t in $\rightarrow_0$ must be part of an accepting path and have $l(t) = \bot$ at line 4.
(b) All states in $Q' \setminus Q$ are new and thus not part of transitions in $\rightarrow_0$, so there are no accepting paths from them. On line 5 $cost(q) = \top$ is initialized correctly.

Inductive step. Assume (a) and (b) holds for all j, $j \leq i$, then we show that they hold for $i + 1$:
The weight function l is only updated in two places, so the weight of transitions from states in P change at line 3 in Algorithm 2, and the weight of transitions from $Q' \setminus Q$ change at line 17 in Algorithm 3. Transitions from states in $Q \setminus P$ have weight $\bot$ which never changes.

If $l(t_2)$ changes on line 17, then the new value is $l_i(p \xrightarrow{\gamma} q) \oplus f(r)$. The value $\kappa_{i+1}(q_{p',\gamma'})$ is either unchanged, which by induction hypothesis (b) is $\kappa_i(q_{p',\gamma'}) = cost(q_{p',\gamma'})$, or it is lower due to the change of $l(t_2)$. Using the induction hypothesis we have $\kappa_i(q) \oplus l_{i+1}(t_2) = \kappa_i(q) \oplus l(p \xrightarrow{\gamma} q) \oplus f(r) = d_i \oplus f(r)$, so:

$$\kappa_{i+1}(q_{p',\gamma'}) = \kappa_i(q_{p',\gamma'}) \sqcap (\kappa_i(q) \oplus l_{i+1}(t_2)) = cost(q_{p',\gamma'}) \sqcap (d_i \oplus f(r))$$

This is checked by the if-else conditions at lines 19 to 24. Consider the two possibilities at line 19:

- If $d_i \oplus f(r) \sqsubset cost_i(q_{p',\gamma'})$, then at line 20 we correctly set $cost_{i+1}(q_{p',\gamma'}) = d_i \oplus f(r) = \kappa_{i+1}(q_{p',\gamma'})$, which concludes part (b) of the induction step. By Lemma 4.5, the only transition going into $q_{p',\gamma'}$ is t_1, so for all $t' \neq t_1$, the invariant also holds for t' at $i + 1$. The priority of t_1 is set correctly in the call to UPDATE at line 21 concluding the induction step for (a).
- If $cost_i(q_{p',\gamma'}) \sqsubseteq d_i \oplus f(r)$, then $\kappa_i(q_{p',\gamma'}) = cost_i(q_{p',\gamma'}) \sqsubseteq d_i \oplus f(r) = \kappa_i(q) \oplus l_{i+1}(t_2)$. Since $\kappa_i(q_{p',\gamma'}) \sqsubseteq \kappa_i(q) \oplus l_{i+1}(t_2) \sqsubset \kappa_i(q) \oplus l_i(t_2)$ there is already a shortest accepting path from $q_{p',\gamma'}$ not containing t_2. Since t_2 is the only transition in $Q \times \Gamma \times Q$ that changes in round i, we conclude $\kappa_{i+1}(q_{p',\gamma'}) = \kappa_i(q_{p',\gamma'})$. This concludes the induction step.

It is easy to prove that when $l_{i+1}(p \xrightarrow{\gamma} q)$ is updated on line 3 of Algorithm 2, then the priority of $p \xrightarrow{\gamma} q$ in $queue_{i+1}$ is updated accordingly. It is also easy to check that the priorities in $queue_{i+1}$ and the values in $cost_{i+1}$ is not changed in other places. □

Lemma 4.5. Assuming the induction hypothesis from Lemma 4.4 for all j, $j \leq i$, if $d_i \oplus f(r) \sqsubset cost_i(q_{p',\gamma'})$ at line 19, then t_1 is the only transition going into $q_{p',\gamma'}$ in $\mathcal{A}_i$.

Proof. From induction hypothesis part (b) we have $cost_i(q_{p',\gamma'}) = \kappa_i(q_{p',\gamma'})$, so:

$$d_i \sqsubseteq d_i \oplus f(r) \sqsubset cost(q_{p',\gamma'}) = \kappa_i(q_{p',\gamma'}) \sqsubseteq \kappa_i(q_{p',\gamma'}) \oplus l_i(t_1)$$

Since $d_i \sqsubset \kappa_i(q_{p',\gamma'}) \oplus l_i(t_1)$, which by induction hypothesis part (a) is the priority of t_1 in $queue_i$, we know that t_1 cannot have been popped at line 8 in round i or earlier, and therefore also that $t_1 \notin \rightarrow_i$. This implies that t_1 is the only transition leading into $q_{p',\gamma'}$ in $\mathcal{A}_i$, since transitions leading into $q \in Q' \setminus Q$ are only added if t at line 8 leads to q, $t' \in \rightarrow$ at line 26 leads to q, or if t_1 at line 16 leads to q. □

Corollary 4.5.1. Continuing from Theorem 4.2: In particular for a regular set $C' \subseteq P \times \Gamma^*$, the shortest distance as defined in Problem 4.1 is found by: $\delta = \bigsqcap\{\rho(c') \mid c' \in C'\}$ in $\mathcal{V}_{post^*}$.

Proof. Follows trivially from Theorem 4.2 and the definition of Problem 4.1. □

By Corollary 4.5.1, Algorithm 3 reduces the weighted pushdown reachability problem (Problem 4.1) to the problem of finding shortest accepting paths in a finite automaton. For the simple case where $C' = \{c'\}$, we have $\delta = \rho(c')$, which is a simple graph traversal problem solved by Dijkstra's algorithm.

The general case, where the regular set C' is accepted by a $\mathcal{P}$-automaton $\mathcal{A}'$, is a bit more complicated, since we need to find the minimum $\rho(c')$ for an element c' in the possibly infinite set of configurations C'. However its finite representation as $\mathcal{A}'$, allows us to represent the intersection $L(\mathcal{V}_{post^*}) \cap L(\mathcal{A}')$ by means of a product construction [18] of $\mathcal{V}_{post^*}$ and $\mathcal{A}'$, where we keep the weights from $\mathcal{V}_{post^*}$. Finding a shortest accepting path on this weighted product automaton solves the weighted pushdown reachability problem.

4.2.1 Computing the shortest trace

Algorithm 3, as presented here, only allows us to compute the cost of the shortest trace δ in the pushdown system. It does not have enough information to reconstruct the witness trace ω itself. Reps. et al. [19] describe how to compute a witness set in the general case by modifying the UPDATE procedure to construct a hypergraph of transition-weight pairs, where edges are annotated with rules.

In our algorithm, where the weight domain is totally ordered and the transitions are stored in a priority queue, each transition is only processed once, and there exists a trace with weight δ. Given these facts, we can use the simpler approach for pushdown systems without weights presented in [15] and further formalized by Morten Schou in an assignment [21] prior to this project. The following text and Algorithm 4 is based on [21].

We use a function α to annotate all transitions in $\mathcal{A}_{post^*}$. The annotation of a transition $p \xrightarrow{\gamma} q$ is either:

- $\alpha(p \xrightarrow{\gamma} q) = \bot$ if $p \xrightarrow{\gamma} q$ is in the original $\mathcal{A}$,
- $\alpha(p \xrightarrow{\gamma} q) = r$ with $r = \langle p', \gamma' \rangle \hookrightarrow \langle p, w \rangle$ if $p \xrightarrow{\gamma} q$ was added because of the rule r, or
- $\alpha(p \xrightarrow{\gamma} q) = \langle q' \rangle$ if $p \xrightarrow{\gamma} q$ was added because of the transitions $p \xrightarrow{\varepsilon} q'$ and $q' \xrightarrow{\gamma} q$.

The following additions to Algorithm 3 is needed to annotate the edges of the $\mathcal{P}$-automaton. We annotate with the rule used at line 12: $\alpha(p' \xrightarrow{\varepsilon} q) := r$, line 14: $\alpha(p' \xrightarrow{\gamma'} q) := r$, line 18: $\alpha(t_2) := r$, and line 21: $\alpha(t_1) := r$. And we annotate with the intermediate state at line 24: $\alpha(p'' \xrightarrow{\gamma''} q) := \langle q_{p',\gamma'} \rangle$ and line 27: $\alpha(p \xrightarrow{\gamma'} q') := \langle q \rangle$.

Given the annotations α, the trace can be reconstructed using Algorithm 4. It finds an accepting path for a configuration using Dijkstra's algorithm, and then it uses the annotations to reconstruct all the intermediate configurations and the rules that were applied. This sequence of configurations is the PDA-trace.

Algorithm 4 Trace generation for *post**

```
1:  function MakeConf($(p \xrightarrow{\gamma_1} q_1, \ldots, q_{n-1} \xrightarrow{\gamma_n} q_n)$)
2:      return $\langle p, \gamma_1\gamma_2 \ldots \gamma_n\rangle$
3:  function Trace($\langle p, w\rangle$, $\mathcal{V}_{post^*} = (\mathcal{A}_{post^*}, l)$, $\alpha$)
4:      let $\gamma_1\gamma_2 \ldots \gamma_n = w$
5:      if $\langle p, w\rangle \in L(\mathcal{A}_{post^*})$ then
6:          $(p, q_1, \ldots, q_n) \leftarrow$ compute accepting path for $\langle p, w\rangle$ in $\mathcal{V}_{post^*}$   ▷ Using Dijkstra
7:          $edges \leftarrow (p \xrightarrow{\gamma_1} q_1, \ldots, q_{n-1} \xrightarrow{\gamma_n} q_n)$
8:          $trace \leftarrow [\langle p, w\rangle]$
9:          $t_1 \leftarrow$ pop front of $edges$
10:         while $\alpha(t_1) \neq \perp$ do
11:             let $(p' \xrightarrow{\gamma'} q) = t_1$
12:             if $\alpha(t_1) = \langle q'\rangle$ then
13:                 push $(p' \xrightarrow{\varepsilon} q',\ q' \xrightarrow{\gamma'} q)$ to front of $edges$
14:             else $(\alpha(t_1) = \langle p, \gamma\rangle \hookrightarrow \langle p', w'\rangle)$
15:                 if $|w'| \leq 1$ then
16:                     push $(p \xrightarrow{\gamma} q)$ to front of $edges$
17:                 else $(|w'| = 2)$
18:                     $t_2 \leftarrow$ pop front of $edges$
19:                     let $(q \xrightarrow{\gamma''} q'') = t_2$
20:                     $(\alpha(t_2) = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle)$
21:                     push $(p \xrightarrow{\gamma} q'')$ to front of $edges$
22:                 push MakeConf($edges$) to front of $trace$
23:             $t_1 \leftarrow$ pop front of $edges$
24:         return $trace$
25:     return with no trace
```

4.2.2 Example of Asymptotic Improvement

Our solution to the weighted pushdown reachability problem (Algorithm 3) uses a priority queue to ensure that each transition of the resulting $\mathcal{P}$-automaton is only processed once in the algorithm. The corresponding algorithm in [19], which applies to a broader range of weight domains by not assuming it to be totally ordered, cannot use a priority queue. Instead, transitions are stored in a set and picked arbitrarily. To ensure correctness it needs to add transitions to the set whenever their weight changes, which can happen many times in the worst case. This sections presents an example of pushdown systems, where our algorithm is asymptotically faster than the one in [19].

Figure 4.1 gives a simple example of a pushdown system, where our Algorithm 3 has better worst-case performance than the algorithm in [19] when applying weighted *post** to the regular set of initial configurations $C = \{\langle 1, \gamma\rangle \mid \gamma \in \Gamma\}$. Note that while Figure 4.1 has $\Gamma = (A, \ldots, Z)$ it is straightforward to extend it to an arbitrarily large Γ.

$$\Gamma = (A, \dots, Z) \qquad C = \{(1, \gamma) \mid \gamma \in \Gamma\}$$

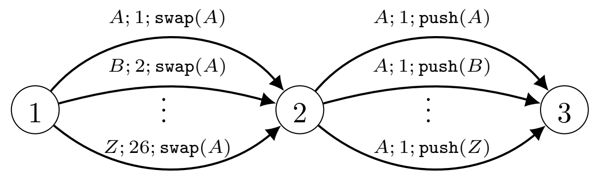


Figure 4.1. A pushdown system for which our Algorithm 3 in the worst case outperforms the algorithm in [19] by using a priority queue. As an example of the notation, the annotation $A; 1; \texttt{push}(B)$ on the edge from 2 to 3 corresponds to the rule $r = \langle 2, A\rangle \hookrightarrow \langle 3, BA\rangle$ with weight $f(r) = 1$.

To illustrate why, this example gives a poor worst-case performance for the algorithm in [19], we go through a possible execution of the algorithm. Initially the set of transitions to be considered is $\{1 \xrightarrow{A} 4, \dots, 1 \xrightarrow{Z} 4\}$. In the worst case the algorithm picks $1 \xrightarrow{Z} 4$, so when applying the saturation rule it adds $2 \xrightarrow[26]{A} 4$. Our Algorithm 3 will then proceed to pick the other transitions from state 1, since they all have weight $\perp$, leaving $2 \xrightarrow[1]{A} 4$ in the workset. However the algorithm in [19] does not know to pick these first, so it can in the worst case pick $2 \xrightarrow[26]{A} 4$, and then the saturation rule adds transitions $3 \xrightarrow[\perp]{A} 5 \xrightarrow[27]{A} 4$, $3 \xrightarrow[\perp]{B} 6 \xrightarrow[27]{A} 4$, ..., $3 \xrightarrow[\perp]{Z} 30 \xrightarrow[27]{A} 4$. Figure 4.2 illustrates the intermediate $\mathcal{W}$-automaton at this point.

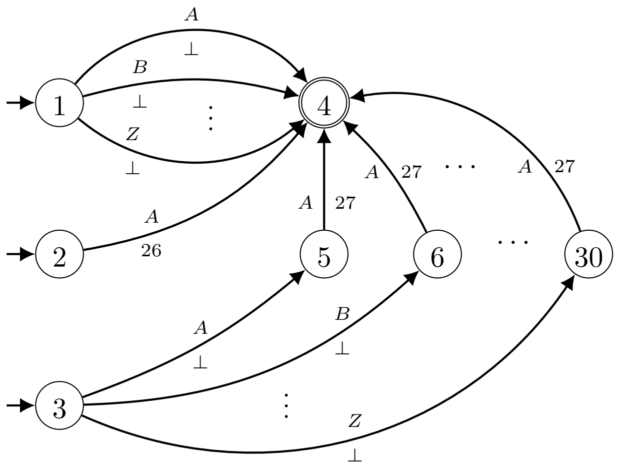


Figure 4.2. An intermediate $\mathcal{W}$-automaton during a possible execution of the weighted *post** algorithm in [19] applied to the pushdown system in Figure 4.1 with the initial configurations $C = \{\langle 1, \gamma\rangle \mid \gamma \in \Gamma\}$.

The algorithm can then in the worst-case go through all these transitions, and then pick transition $1 \xrightarrow[25]{Y} 4$, so the saturation rule adds $2 \xrightarrow[25]{A} 4$, which has a smaller weight than before and hence is added back to the workset. Picking $2 \xrightarrow[25]{A} 4$ next will add transitions $5 \xrightarrow[26]{A} 4, 6 \xrightarrow[26]{A} 4, \dots, 30 \xrightarrow[26]{A} 4$, which all have lower weight than before, so they are added back into the workset. This process can continue each round decreasing the weight by 1 until $1 \xrightarrow{A} 4$ is picked and the weights converge to their final value.

This example shows how in the worst case the algorithm in [19] adds each transition $5 \xrightarrow{A} 4, \dots, 30 \xrightarrow{A} 4$ to the workset once for each transition in $\{1 \xrightarrow{A} 4, \dots, 1 \xrightarrow{Z} 4\}$, which leads to a quadratic factor in the complexity of the algorithm for this problem instance. In the our Algorithm 3 each of the edges $5 \xrightarrow{A} 4, \dots, 30 \xrightarrow{A} 4$ are only added once, so the corresponding factor is only linear in the input size. Even considering the logarithmic overhead of the operations on the priority queue, our Algorithm 3 has a better asymptotic complexity then the algorithm in [19] for this family of pushdown systems.

4.3 Weight Encoding for Network Trace Properties

The weight expressions described in Section 2.6 apply to a full network trace σ, but the weight function in the weighted pushdown system assigns a weight to each pushdown rule. Therefore we need a way to decompose the weight expression for the network trace into single weights for each rule.

When constructing the pushdown system from the MPLS network, we create a pushdown rule for each operation in the operation sequence of a routing table entry. The details of this construction is presented in [4]. Here it is enough to know that we can identify which rules correspond to a routing going out on a link, and which are auxiliary rules carrying out the operation sequence. Given this, it is straightforward to decompose the following atomic properties into weights for each pushdown rule r:

- Links: $f(r) = 1$ if r corresponds to a routing along a link, 0 otherwise.
- Hops: $f(r) = 1$ if r corresponds to a routing along e, where $s(e) \neq t(e)$, 0 otherwise.
- Distance: $f(r) = d(e)$ if r corresponds to a routing along e, 0 otherwise, where $d : E \to \mathbb{N}_0$ is the distance function used, e.g. physical distance, latency, etc.
- LocalFailures: $f(r) = |\{e \mid (e, \omega) \in O_i,\ 0 \leq i < j\}|$ if r corresponds to a routing along e' for the routing table lookup $\tau(e'', \ell) = O_0 \dots O_n$, where j is the index of the traffic engineering group O_j containing e'.
- Tunnels: $f(r) = 1$ if r corresponds to a `push` operation in Op, 0 otherwise.

When using these decomposable atomic properties, it is trivial to extend the decomposition to linear expressions and ordered expressions. For StackSize however, decomposing it into weights for each pushdown rule is not trivial, since local information at each rule is not enough to determine its impact on the overall weight of the trace. In other words, StackSize is a global properties of the trace. Failures is another global property that is not covered in this thesis.

Figure 4.3 shows an example PDS where the StackSize property cannot be easily decomposed into weights on each rule. The example shows four traces all starting in state 1 with the first two ending in state 8 and the latter two ending in state 9. When performing the shortest trace *post** algorithm starting at state 1, we need to know which trace going to 5 is the shortest in terms of the weights. However, as the four traces in the example show, one needs to keep track of both possible traces from 1 to 5.

If at state 5 we choose that the trace directly from 1 to 5 is shorter than the one going through states 2, 3 and 4, then if we decide to proceed to state 8, we will have a stack size of 3, while the other possible trace has a maximum stack size of 2. If we instead at

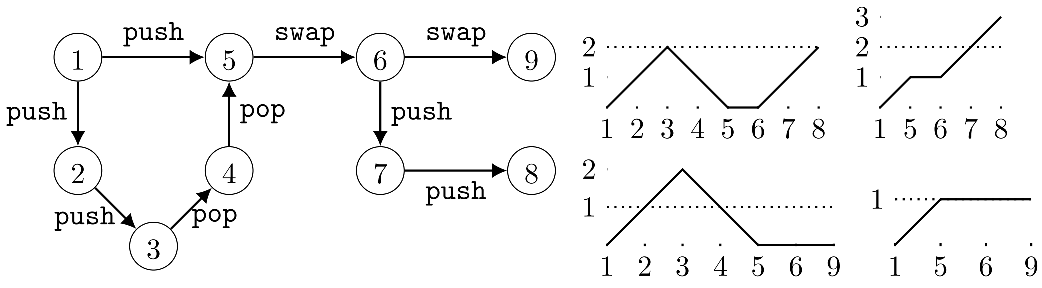


Figure 4.3. A pushdown system (PDS) (left), and four traces from node 1 to node 8 and node 9 (right).

state 5 choose that the shortest trace is the one going through states 2, 3 and 4, then if we decide to proceed to state 9, our trace will have a maximum stack size of 2, while the other possible trace has a maximum stack size of 1. This shows that it is impossible to make a local choice at state 5 about which trace is shortest, so the maximum stack size of a trace is a global property. The next section gives a solution for solving the weighted trace problem for the weight expression (StackSize).

4.3.1 Minimizing the maximum stack size in a trace

Some measures describe global properties of the trace. One such example is the maximal stack size of a configuration of the trace. For simplicity we here analyse the maximal stack size problem for stacks in pushdown systems, but it is analogous to minimizing the maximal MPLS stack size of a network trace.

The problem is, for a weighted pushdown systems $\mathcal{W}$ and two regular sets C and C' to find a trace that transforms a $c \in C$ into a $c' \in C'$ while minimizing the maximum stack size of configurations on this trace. However for the weighted *post** algorithm to work, the empty trace to an initial configuration must have weight 0, so we need to reformulate the problem as finding a trace for which the maximum increase in stack size compared to the initial stack is minimal. If all configurations in C have the same stack size this is equivalent to minimizing the maximal stack size.

Define a sub-trace as any contiguous sub-sequence of a trace. The relevant information for a sub-trace is the maximum increase in stack size during the trace, and the stack size at the end compared to the beginning of the trace. Both of these numbers can be easily computed for any given sub-trace. We will call these two numbers the weight of a sub-trace: $w(\sigma)$. Push rules have weight $(1, 1)$, pop rules have weight $(0, -1)$, and swap rules have weight $(0, 0)$.

When concatenating two sub-traces σ_1 and σ_2, the weight of the resulting sub-trace can be calculated from the weights of σ_1 and σ_2: $(a_1, b_1) \oplus (a_2, b_2) = (\max\{a_1, b_1 + a_2\},\ b_1 + b_2)$. Figure 4.4 shows a concatenation of two sub-traces with weight $(2, 1)$ and $(1, -1)$.

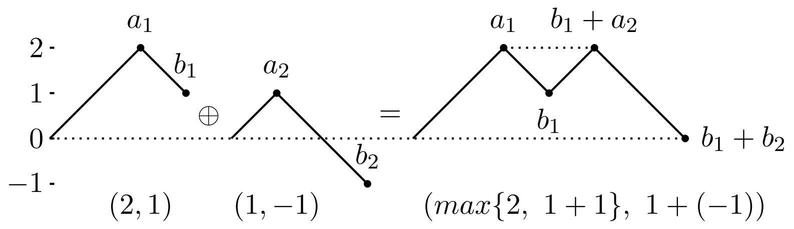


Figure 4.4. Concatenation of two sub-traces (left) σ_1 and σ_2 such that $w(\sigma_1) = (2,1)$ and $w(\sigma_2) = (1,-1)$. The trace (right) show the calculation of the final trace, such that $w(\sigma) = (2,0)$

The problem comes when comparing the weights of two sub-traces to find the one that will lead to a minimum weight trace in the end. For example at state 5 in Figure 4.3, where we need to choose between two different traces from 1 to 5. If in the end we have $((2,0) \sqcap (1,1)) \oplus (0,0)$, we prefer $(1,1)$ over $(2,0)$, but if we have $((2,0) \sqcap (1,1)) \oplus (2,2)$ we prefer $(2,0)$ over $(1,1)$. This example shows that there is no reasonable total ordering of the weights. Instead we need to define the meet operator $\sqcap$ in such a way that it keeps the information of both traces.

In general we want to consider the weight of a set of sub-traces. Initially these will be singleton sets, but when performing the meet operation, we take the union of two sets: $W_1 \sqcap W_2 = W_1 \cup W_2$, and filter out the elements for which there is a smaller element in the set with respect to the product order: $(a_1, b_1) \leq (a_2, b_2) \iff a_1 \leq a_2 \wedge b_1 \leq b_2$.

When combining two sets of sub-traces W_1 and W_2, we try all combinations of sub-traces from the two sets: $\{w_1 \oplus w_2 \mid w_1 \in W_1, w_2 \in W_2\}$, and then filter out the non-minimal elements.

4.3.2 Formal definition of StackSize weight

We extend the integer operators $+$ and $\leq$ to work on infinity in the standard way: for all $a \in \mathbb{Z} \cup \{\infty\}$, $a \leq \infty$ and $a + \infty = \infty + a = \infty$. Likewise $\max\{a, \infty\} = \infty$.

We define the binary operator $\oplus$ over $\mathbb{N} \times \mathbb{Z} \cup \{(\infty,\infty)\}$ by:

$$(a_1, b_1) \oplus (a_2, b_2) = (\max\{a_1, b_1 + a_2\},\ b_1 + b_2)$$

Definition 4.6. For any $W \in 2^{\mathbb{N}\times\mathbb{Z}\cup\{(\infty,\infty)\}}$ we define the pruning operator for filtering out non-minimal elements $\mathcal{R}(W) = \{(a,b) \in W \mid \nexists(a',b') \in (W \setminus \{(a,b)\}).\ a' \leq a \wedge b' \leq b\}$. We call $\mathcal{R}(W)$ the minimal representation of W.

Definition 4.7 (Obtainable Values)**.** For a domain D, a finite set of initial values $I \subseteq D$, and a finite set of binary operators $F = \{f_1, \ldots, f_n\}$, $f_i : D \times D \to D$, define the F-closure of I: $F^*(I) \subseteq D$ by the inductive definition:

- if $d \in I$ then $d \in F^*(I)$, and
- if $d_1, d_2 \in F^*(I)$ and $f \in F$ then $f(d_1, d_2) \in F^*(I)$.

Definition 4.8. $\mathcal{S}_{\textsc{StackSize}} = (D_{\textsc{StackSize}}, \sqcap, \otimes, \{(\infty,\infty)\}, \{(0,0)\})$ is the stack size weight domain, where

- $D_{\textsc{StackSize}} = F^*(I)$ with $I = \{\{(1,1)\}, \{(0,0)\}, \{(0,-1)\}, \{(\infty,\infty)\}\}$, $F = \{\sqcap, \otimes\}$,
- $W_1 \sqcap W_2 = \mathcal{R}(W_1 \cup W_2)$, and
- $W_1 \otimes W_2 = \mathcal{R}(\{x_1 \oplus x_2 \mid x_1 \in W_1, x_2 \in W_2\})$.

Lemma 4.6. For any obtainable weight $W \in D_{\textsc{StackSize}}$, the following invariants hold:

- $W \neq \emptyset$
- $\forall (a,b) \in W.\ a \geq b.$
- $\forall (a_1,b_1),(a_2,b_2) \in W.\ a_1 \leq a_2 \wedge b_1 \leq b_2 \implies (a_1,b_1) = (a_2,b_2)$

Proof sketch. By structural induction: Base case: The invariants hold for each initial value in $I = \{\{(1,1)\}, \{(0,0)\}, \{(0,-1)\}, \{(\infty,\infty)\}\}$. Inductive step: The invariants are preserved by the operations $\sqcap$ and $\otimes$. □

As noted, $\mathcal{S}_{\textsc{StackSize}}$ does not satisfy the well-order requirement of Definition 4.1. It does however satisfy the requirements for an idempotent semiring [20]:

Theorem 4.7. $\mathcal{S}_{\textsc{StackSize}} = (D_{\textsc{StackSize}}, \sqcap, \otimes, \{(\infty,\infty)\}, \{(0,0)\})$ is an idempotent semiring, i.e. it satisfies the following:

- $(D_{\textsc{StackSize}}, \sqcap)$ is a commutative monoid with the identity element $\{(\infty,\infty)\}$, where $\sqcap$ is idempotent: $\forall a \in D_{\textsc{StackSize}}.\ a \sqcap a = a$.
- $(D_{\textsc{StackSize}}, \otimes)$ is a monoid with the identity element $\{(0,0)\}$.
- $\otimes$ distributes over $\sqcap$: $\forall a,b,c \in D_{\textsc{StackSize}}$ we have $a \otimes (b \sqcap c) = (a \otimes b) \sqcap (a \otimes c)$ and $(a \sqcap b) \otimes c = (a \otimes c) \sqcap (b \otimes c)$.
- $\top$ is an annihilator for $\otimes$: $\forall a \in D_{\textsc{StackSize}}, a \otimes \top = \top = \top \otimes a$.

Proof sketch. These properties can be proved by expanding definitions, simplifying and applying structural induction over $D_{\textsc{StackSize}}$. □

An idempotent semiring has a partial order $\sqsubseteq$ defined by $a \sqsubseteq b$ iff $a \sqcap b = a$ called the *natural order* [20].

Using the following weight function $f_{\textsc{StackSize}} : \Delta \to D_{\textsc{StackSize}}$ we can define a weighted pushdown system $\mathcal{W}_{\textsc{StackSize}} = (\mathcal{P}, \mathcal{S}_{\textsc{StackSize}}, f_{\textsc{StackSize}})$, where Δ are the rules in $\mathcal{P}$:

$$f_{\textsc{StackSize}}(\langle p,\gamma\rangle \hookrightarrow \langle p',w\rangle) = \begin{cases} \{(1,1)\} & \text{if } |w| = 2 \\ \{(0,0)\} & \text{if } |w| = 1 \\ \{(0,-1)\} & \text{if } |w| = 0 \end{cases}$$

The *pre** and *post** algorithms in [19] takes as input a weighted pushdown systems, where the weight domain are required to be an idempotent semiring, where the natural order $\sqsubseteq$ contains no infinite descending chains. Unfortunately $\mathcal{S}_{\textsc{StackSize}}$ has the infinite descending chain $\{(0,0)\}, \{(0,-1)\}, \{(0,-2)\}, \ldots$, where each weight is obtainable since it corresponds to a sequence of `pop` rules. However, the 'no infinite descending chains' property is only used to prove termination of the algorithms in [19], so if we prove termination in another way, then $\mathcal{S}_{\textsc{StackSize}}$ being an idempotent semiring is still enough to ensure correctness.

We do not proceed to prove termination, but we conjecture that the *post** algorithm in [19] terminates on the input $\mathcal{W}_{\textsc{StackSize}} = (\mathcal{P}, \mathcal{S}_{\textsc{StackSize}}, f_{\textsc{StackSize}})$ and $\mathcal{A}$, if $\mathcal{A}$ is a $\mathcal{P}$-automaton with no cycles, i.e. the language it recognizes is finite.

The complexity of the *pre** and *post** algorithms in [19] are increased from their unweighted counterparts, which have polynomial complexity, by a factor no larger than the length of the maximal length descending chain to any value that appear in the annotated automaton [19]. This is assuming that the $\sqcap$ and $\otimes$ operations can be performed efficiently.

The result of these algorithms is a $\mathcal{W}$-automaton with transition weights in the domain $D_{\textsc{StackSize}}$. In this $\mathcal{W}$-automaton we need to find the shortest accepting distance for a configuration $\langle p, w\rangle$. Without a total order, we cannot use the classical Dijkstra algorithm. Instead we use a workset, to which nodes may be added multiple times. The tentative shortest distance of a node $\rho'(q)$ is updated as $\rho'(q) := \rho'(q) \sqcap d_{new}$, where d_{new} is calculated using $\otimes$, $\rho'(q')$ for some other state q' and a transition from q' to q. If this changes the value of $\rho'(q)$, then q is added back to the workset. We cannot stop as soon as a final node is reached, instead the process ends, when the workset is empty, i.e. when the weights stop changing. Now the shortest distance is $\rho(\langle p, w\rangle) = \bigsqcap\{\rho'(q) \mid q \in F\}$.

While the minimal maximum stack size for a sub-trace cannot be represented by a total order, we have for a full trace the total order: $(a_1, b_1) \leq (a_2, b_2)$ iff $a_1 \leq a_2$. So we find the value a such that $(a, b) \in \rho(\langle p, w\rangle) \wedge \forall (a', b') \in \rho(\langle p, w\rangle).\ a \leq a'$. This is the minimal maximum stack size of any trace reaching the configuration $\langle p, w\rangle$.

When weights are updated with $\rho'(q) := \rho'(q) \sqcap d_{new}$, they may combine multiple traces, so it is not in general guaranteed that there is a single trace with that weight. However in our case, each value $(a, b) \in W$ for a weight $W \in D_{\textsc{StackSize}}$ corresponds to a specific trace. So when finding the minimal $(a, b) \in \rho(\langle p, w\rangle)$ for a full trace, this will also correspond to a specific trace. To reconstruct the trace, we need to annotate the values $(a, b) \in W$ with information about which trace it corresponds to and make sure that the operators $\sqcap$ and $\otimes$ preserve this information. The details of these annotations are left for future work.

Experimental Evaluation 5

We perform a series of experiments to evaluate the behavior and performance of AalWiNes with our contributions. The experiments are separated in two, where the first aim to clarify the best configuration for the tool, while testing the performance of our contributions. The latter is a case-study on the MPLS network of NORDUNET, from which we evaluate and compare the behavior of the best configuration on a real network. Finally, will we use the tool to analyse some queries provided by NORDUNET and evaluate the applicability of the tool.

5.1 Performance Tests

This experiment uses various size of MPLS networks from Topology Zoo [11], that all are manipulated to larger network topologies, through the network topology manipulation described in Section 2.3. Each network is also annotated with routing tables generated by Algorithm 1. For each of the networks we create queries based on 5 generic query types. To produce consistent results, and to allow better comparisons, we execute each individual test 10 times on the cluster provided by Aalborg University, and use the median values of the 10 executions for analysis.

In the analysis of the results we compare the different levels of reduction of the PDA, we compare the engines to each other, and we compare the weighted analysis to the unweighted version.

5.1.1 Generation of test cases

Starting with six different network topologies from Topology Zoo [11], we generate for each five different versions. For $N \in \{1, \ldots, 5\}$ we create N copies of the network topology t_1 to t_N. Using CONCAT we connect t_i to t_{i+1} by connecting every third router to its copy. For the next i we switch the offset for routers, so it is different routers that connect to the copy before and after. Intuitively think of N layers, where some routers are connected vertically to the next.

For each of these 30 test networks, we create a number of queries. We use five different general types of queries that are instantiated to concrete queries for each test network:

1. $<$ `smpls ip` $>$ $[\cdot$ `#` $v_1]$ $\cdot^*$ $[v_2$ `#` $\cdot]$ $<$ `mpls`$^+$ `smpls ip` $>$ k
2. $<$ (`mpls`* `smpls`)? `ip` $>$ $[\cdot$ `#` $v_1]$ $[\hat{}\ \cdot$ `#` $v_2]^*$ $[v_3$ `#` $\cdot]$ $<$ (`mpls` $*$ `smpls`)? `ip` $>$ k
3. $<$ (`mpls`* `smpls`)? `ip` $>$ $[\cdot$ `#` $v_1]$ $\cdot^*$ $[v_2$ `#` $\cdot]$ $<$ (`mpls`* `smpls`)? `ip` $>$ k
4. $<$ (`mpls`* `smpls`)? `ip` $>$ $[\cdot$ `#` $v_1]$ $\cdot^*$ $[v_2$ `#` $\cdot]$ $<$ `smpls ip` $>$ k
5. $<$ (`mpls`* `smpls`)? `ip` $>$ $[\cdot$ `#` $v_1]$ $\cdot^+$ $[v_1$ `#` $\cdot]$ $<$ (`mpls`* `smpls`)? `ip` $>$ k

Query type 1 checks for transparency, i.e. whether it is possible for MPLS labels to leak out on the stack when leaving router v_2. Type 2 is a waypointing query. It checks whether it is possible to avoid router v_2 on a path from v_1 to v_3. If this is not the case, then all paths from v_1 to v_3 goes through the waypoint v_2. Type 3 simply checks for reachability between v_1 and v_2. Type 4 checks if it is possible to end up with exactly one mpls label on the stack. Type 5 checks for loops, i.e. if the router v_1 can occur multiple times in a trace.

For each of these query types we make a concrete query for each $k \in \{0, \ldots, 3\}$ failed links and $Q_i \in \{1, \ldots, 10\}$ random instantiations of the routers v_1, v_2 and v_3. This gives in total 6.000 test cases.

5.1.2 Running the tests

Each test case is executed for combination of the engines $E \in \{Mpost*, Mpre*, post^*, pre^*\}$ and reduction level $R \in \{0, \ldots, 4\}$ each with ten equal executions. This gives 960.000 test executions. The four engine types represent Moped $post^*$, Moped pre^*, and our $post^*$ and pre^* implementations respectively. The reduction levels show the top-of-stack (tos) reductions of the PDA before applying the verification algorithm. The reduction techniques it uses are called: simple, dual-stack and target-tos [1]. They are configured through different reduction levels: R0 = no reduction, R1 = simple, R2 = R1 + dual-stack, R3 = R2 + target-tos, R4 = R1 + target-tos. It turns out that the dual-stack technique can be very time-consuming in the worst-case, so R4 is a good compromise as the following results will show.

For each test execution we record the time for compilation (building the PDA), reduction (of the PDA using level R) and verification (of the PDA using engine E). We compute the total time as the sum of these three times i.e. $t_{total} = (t_{verification} + t_{reduction} + t_{compilation})$. We also record the number of routers, interfaces and labels in each test network and the number of states and rules in the generated PDAs before and after reduction.

For the external Moped engine, the verification time is measured only for its call to its verification algorithm, so all extra parsing is not included in this time.

The test data is organized into three projections of the data that will help us draw conclusions. The first projection show an overall comparison of the reduction levels, the second projection show an overall comparison of verification-time on the four engines alongside a more in depth analysis, lastly we show the performance of the weighted extension while comparing it with the amount of conclusive queries.

5.1.3 Comparing reductions

Figure 5.1 compare the total time on *post** for each test case across the reduction levels. For each execution in the reduction test we set an environmental limit, a timeout of 20 minutes, and memory usage limit of 30 GB. If the test execution exceeds either of these, it is considered failed. From the 6000 test cases 5627 succeeded and 373 failed within the environmental limitations.

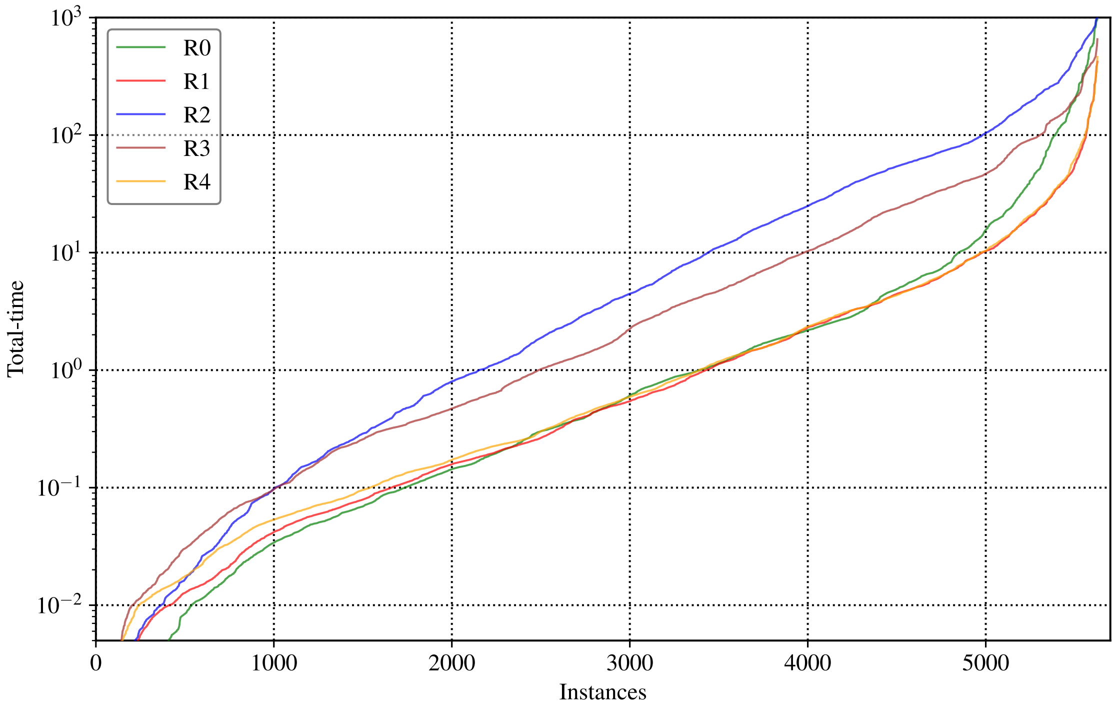


Figure 5.1. Cactus plot over the reduction-levels executed on *post** measuring the t_{total} in logarithmic scale. The plot only include the test-cases where the same result is reached.

The results show that reduction level 4 makes a significant speedup when compared to R2 and R3. From Figure 5.1 using no reduction shows to be fast in the small test-cases, and in the larger test-cases reduction 1 and reduction 4 are very similar. We are especially interested in optimizing the time for the large test cases, where we see the most significant speedup, thus, Table 5.1 show a more thorough comparison of reduction 1 and reduction 4 on *post**.

The table list the size of the network as well the size of the label-set L and rules Δ in the PDA. Since the reductions remove rules, Δ is listed both for reduction 1 and reduction 4. The ratio is calculated on the median of the total time over 10 executions, where $Ratio = median(t^{R1}_{total})/median(t^{R4}_{total})$, and if this is below 1, we change $Ratio = -1/Ratio$, so the absolute values are comparable. To ensure that the data are comparable we filter out the data that do not return the same result, where $True$ and $False$ are conclusive and $None$ is an inconclusive result.

Nodes	Rules	$\lvert L\rvert$	$\lvert\Delta\rvert$R1	$\lvert\Delta\rvert$R4	Query	Result	t^{R1}_{total}	t^{R4}_{total}	Ratio
196	176060	44889	144151	53581	F:3 I:9 T:4	None	44.3078	70.1125	-1.5824
76	36183	10214	41495	10315	F:0 I:7 T:2	True	1.5228	2.2735	-1.493
196	176060	44889	89944	44911	F:0 I:6 T:2	True	6.2791	9.2753	-1.4772
76	36183	10214	52504	10826	F:0 I:1 T:4	True	7.0366	10.2379	-1.4549
118	61872	17885	93110	21320	F:2 I:2 T:5	True	1.2941	1.8285	-1.413
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
67	28440	8275	97473	46222	F:2 I:5 T:3	None	2.0157	2.043	-1.0136
58	24611	7181	70619	31861	F:3 I:6 T:2	None	1.1274	1.1427	-1.0135
51	18359	6727	49389	20439	F:1 I:8 T:5	None	0.7712	0.7816	-1.0135
196	176060	44889	572210	287688	F:3 I:0 T:4	None	442.8557	448.7878	-1.0134
46	12418	4136	35247	17632	F:2 I:2 T:2	None	0.5143	0.5211	-1.0132
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
126	100255	30331	1	1	F:0 I:1 T:4	False	3.2484	1.5386	2.1113
97	124776	36673	1	1	F:0 I:4 T:4	False	4.2683	1.9017	2.2445
145	221829	61305	1	1	F:0 I:2 T:4	False	8.9179	3.3582	2.6555
145	221829	61305	1	1	F:0 I:7 T:4	False	9.1311	3.3669	2.7121
193	356243	95144	1	1	F:0 I:4 T:4	False	30.3483	7.8104	3.8856

Table 5.1. Comparing median $t_{total} = (t_{verification} + t_{reduction} + t_{compilation})$ of 10 executions with *post** on R1 and R4. The query is represented as: F,I,T = Failover, Instance, Type.

From Table 5.1 can we see how the t_{total} of the two reduction levels are very similar in the middle cases. For most of the cases can reduction 4 reduce the system more than reduction 1, and in the middle cases reduction 4 are a margin faster, and thus, for the remainder of this experiment we use on reduction 4.

5.1.4 Comparing the engines

For each execution, in this test, we set an environmental limit, timeout of 10 minutes, and memory usage limit of 16 GB. If the test execution exceeds either of these, it is considered failed. 5250 out of the 6000 test cases succeeded, the failures were due to the environmental limitations of both timeout and out-of-memory failure.

Since the test cases are very different, there is little sense to compute average verification times. Instead we compare the engines' verification times for each test case on reduction 4. To compute an aggregate we count the wins for each engine. To avoid the observed variation affecting the results, we use a cutoff margin for both the relative and absolute difference in verification time.

Engine	Wins
*post**	2504
*pre**	860
Even	1886

Engine	Wins
*Mpost**	2712
*Mpre**	359
Even	2179

Engine	Wins
*Mpost**	1448
*post**	2238
Even	1564

Engine	Wins
*Mpre**	1116
*pre**	2990
Even	1144

Table 5.2. Number of wins for each engine on reduction 4 where M is the Moped implementation. It is considered a win of the faster engine if the verification time is above 30% of the slower engine, and the absolute difference i above 1 millisecond. Otherwise they are considered even.

Table 5.2 depict a similar pattern between the *post** and *pre** implementations in the first two tables from the left. *post** seemingly has an advantage in the general case, but still *pre** is shown to be important. The latter two tables, show some promising results when directly comparing Mopeds implementation to ours. To get a more qualitative view of which tests cases favors which engine and how big the differences are, we sort the test cases by the ratio between the verification times of engine A and engine B, and create a table with the top, middle and bottom test cases. For each pair of engines (A, B) we filter the cases and compute Ratio between the median verification times, similar to Table 5.1.

Nodes	Rules	$\|L\|$	$\|P\|$	$\|\Delta\|$	Query	Result	$Mpost^*$	$post^*$	Ratio
76	36183	10214	43633	56722	F:2 I:7 T:4	True	0.5494	8.9492	-16.2899
96	63897	17460	660	18714	F:0 I:8 T:4	True	3.2037	44.5725	-13.9128
77	43917	12232	534	13288	F:0 I:1 T:4	True	0.8147	10.9656	-13.4596
96	63897	17460	658	18670	F:0 I:9 T:4	True	2.2556	29.7362	-13.1835
111	77709	20365	99950	129484	F:2 I:8 T:4	None	8.3884	108.1571	-12.8937
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
76	38277	12706	472	14758	F:0 I:1 T:2	True	0.9846	0.5854	1.6819
96	63897	17460	82802	105202	F:3 I:8 T:5	None	2.4758	1.469	1.6854
96	63897	17460	659	19237	F:0 I:2 T:3	True	2.202	1.3042	1.6884
58	24611	7181	28773	36457	F:2 I:8 T:4	True	2.2841	1.3526	1.6887
76	38277	12706	23574	45422	F:1 I:8 T:3	None	1.2846	0.7569	1.6972
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
193	356243	95144	448753	592875	F:3 I:6 T:4	False	213.9265	11.584	18.4674
196	176060	44889	7070	53633	F:3 I:0 T:5	False	32.008	1.6819	19.0304
196	176060	44889	245467	286074	F:3 I:8 T:4	False	39.224	1.7262	22.7222
145	221829	61305	152340	281422	F:1 I:6 T:5	False	98.2886	4.0534	24.2485
145	221829	61305	252159	354473	F:2 I:2 T:4	False	96.1797	3.4423	27.9406

Table 5.3. Median verification time of 10 runs with reduction 4 and engines: $Mpost^*$ and $post^*$, the query is specified through F,I,T = Failover amount, Instance number, Query Type

The data shown in Table 5.3 show a minimal speedup for $post^*$ in the general case. With both engines showing potential speedup on all types of queries, with results being *True*, *False*, and *None*, we expect the efficiency to be very similar.

Nodes	Rules	$\|L\|$	$\|P\|$	$\|\Delta\|$	Query	Result	$Mpre^*$	pre^*	Ratio
40	13682	4802	73	4921	F:1 I:9 T:3	True	0.4881	2.5999	-5.3268
26	8841	3476	171	3524	F:0 I:5 T:3	True	0.2268	0.6584	-2.9027
51	18359	6727	10297	19970	F:1 I:1 T:3	True	1.1164	3.1699	-2.8394
79	34368	10567	491	11069	F:0 I:3 T:3	True	2.4715	6.7204	-2.7191
26	8841	3476	8015	11556	F:3 I:3 T:3	True	0.327	0.8881	-2.7162
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
26	8841	3476	7994	12468	F:2 I:9 T:5	True	6.3778	0.6957	9.1676
89	54767	14841	614	16690	F:0 I:9 T:5	True	108.9158	11.8711	9.1749
45	13492	4249	327	5546	F:0 I:6 T:5	True	11.3175	1.2325	9.1826
45	13492	4249	15532	22215	F:2 I:4 T:5	True	13.8767	1.5096	9.1922
77	43917	12232	27893	52099	F:1 I:5 T:3	None	69.3562	7.5433	9.1944
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
97	124776	36673	143931	201614	F:3 I:2 T:3	False	449.717	0.7225	622.4254
126	100255	30331	111972	155692	F:2 I:7 T:2	False	261.7125	0.3947	663.0148
145	221829	61305	152902	281139	F:1 I:0 T:4	False	127.1445	0.1525	833.9258
145	221829	61305	152340	281422	F:1 I:6 T:5	False	381.8485	0.4577	834.2422
118	61872	17885	2460	20801	F:2 I:9 T:4	False	2.7644	0.0031	878.6163

Table 5.4. Median verification time of 10 runs with reduction 4 and engines: $Mpre^*$ and pre^*, the query is specified through F,I,T = Failover amount, Instance number, Query Type

The comparison of $Mpre^*$ and pre^* in Table 5.4, show a major speedup for pre^* on the test cases with *False* output result. The verification times are more equal for the output result *True* and *None*.

The data in Table 5.3 and Table 5.4 both support our expectation of the implementations to be very similar, and with our data structures seemingly being more efficient in the test cases resulting in *False*. These test results show that the pre^* and $post^*$ engines perform well enough to replace the use of Moped in AalWiNes, which will also remove the overhead of writing and parsing Moped files.

5.1.5 Which configuration to choose?

Now the question emerges, whether the tool should use the pre^* or $post^*$ engine. Table 5.5 show a comparison of $post^*$ and pre^*, similar to those in Tables 5.3 and 5.4.

Nodes	**Rules**	$\lvert L\rvert$	$\lvert P\rvert$	$\lvert\Delta\rvert$	**Query**	**Result**	$post^*$	pre^*	**Ratio**
67	28440	8275	33972	47523	F:2 I:5 T:5	None	0.2482	10.8794	-43.8364
26	8841	3476	8953	14892	F:2 I:9 T:3	True	0.0518	2.0219	-39.0535
51	18359	6727	10889	23091	F:1 I:6 T:5	None	0.2037	7.5296	-36.9615
79	34368	10567	45016	54962	F:3 I:7 T:5	True	0.3654	13.4887	-36.9186
61	22656	6779	13827	25927	F:1 I:6 T:5	None	0.1541	5.6748	-36.8253
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
79	34368	10567	38479	48741	F:2 I:7 T:3	None	0.4274	3.1081	-7.2727
118	61872	17885	34687	63079	F:1 I:3 T:5	None	0.8091	5.8758	-7.2619
111	77709	20365	100219	130592	F:2 I:8 T:3	None	1.7066	12.3871	-7.2583
39	10864	3855	269	4971	F:0 I:4 T:3	True	0.1014	0.7354	-7.2542
89	54767	14841	36644	68678	F:1 I:5 T:5	None	1.032	7.4767	-7.2447
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
111	77709	20365	768	21893	F:0 I:4 T:4	True	64.138	0.107	599.2665
96	63897	17460	660	18680	F:0 I:5 T:4	True	43.4486	0.0703	618.0319
49	58824	19033	344	19394	F:0 I:6 T:4	True	15.6583	0.0225	697.3417
157	109478	29415	63509	114619	F:1 I:1 T:4	False	57.919	0.0817	708.9604
157	109478	29415	63502	114061	F:1 I:4 T:4	False	64.4546	0.0794	812.0776

Table 5.5. Median verification time of 10 runs with reduction 4 and engines: $post^*$ and pre^*, the query is specified through F,I,T = Failover amount, Instance number, Query Type

The data in the overall comparison of Table 5.2, and the thorough comparison of Table 5.5 show $post^*$ to be the faster engine in most of the test-cases, hence performing well compared to the state-of-the-art tool Moped. However, pre^* is very fast in the test cases of query type 4. Through more testing of general heuristics it is very interesting to explore the performance increase with an automatic engine choice in the tool, such that $post^*$ is the default engine, and pre^* is chosen for the cases proved to be more efficient. Alternatively both engines can run in parallel, and the result from fastest is used.

5.1.6 Comparing weighted *post**

So far have we only been testing the unweighted implementation of *post**. Here we test the implementation of the weighted extension formally described in Chapter 4. The tests show verification time overhead of introducing weight notation on each rule in the PDA. We expect the overhead to be due to the different data structure of the implementation as the weight has to be kept in memory and use of a priority queue rather than the stack for unweighted.

Figure 5.2 depicts a cactus plot [22], listing the test results for each individual weight properties, a weight annotation of zero to all rules, a more complex weight expression, the unweighted over- and under-approximation as well as the unweighted dual-mode. It also shows the tests which were solved by the reduction, i.e. where the PDA had no rules after reduction. The complex weight expression used in the test is $(5 * \textsc{LocalFailures} + \textsc{Hops}, (3 * \textsc{Tunnels} + 2 * \textsc{LocalFailures}, (\textsc{Distance})))$

The x-axis shows the number of test cases verified with the time on the logarithmic y-axis. The test-cases included are those that conclude either a reachable trace or unreachable, where the solid lines represent weighted and dashed lines the unweighted.

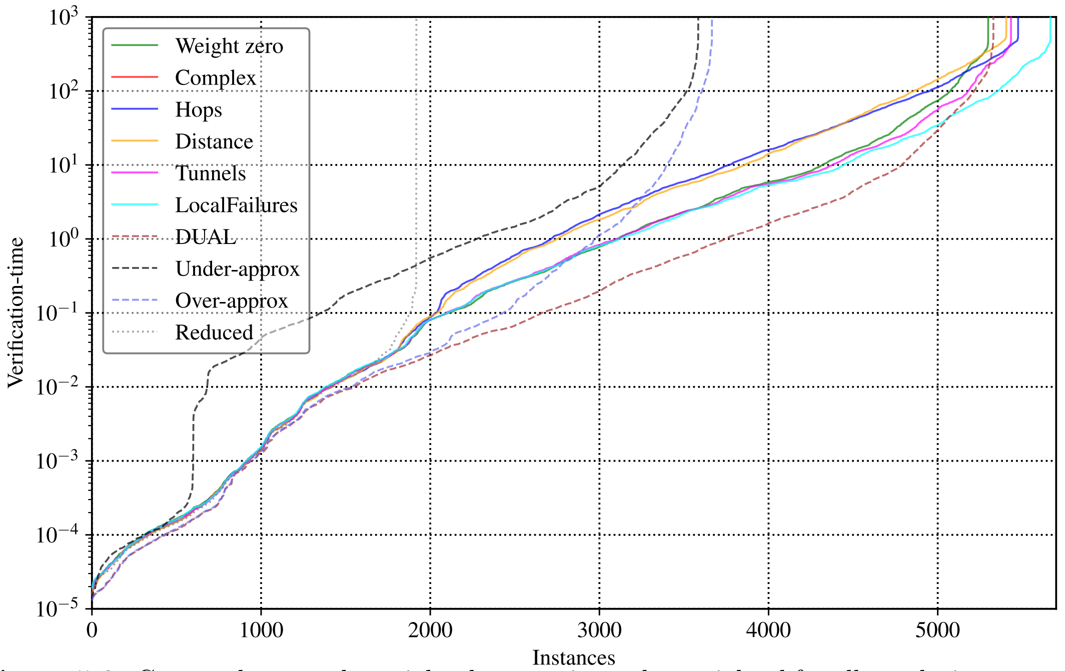


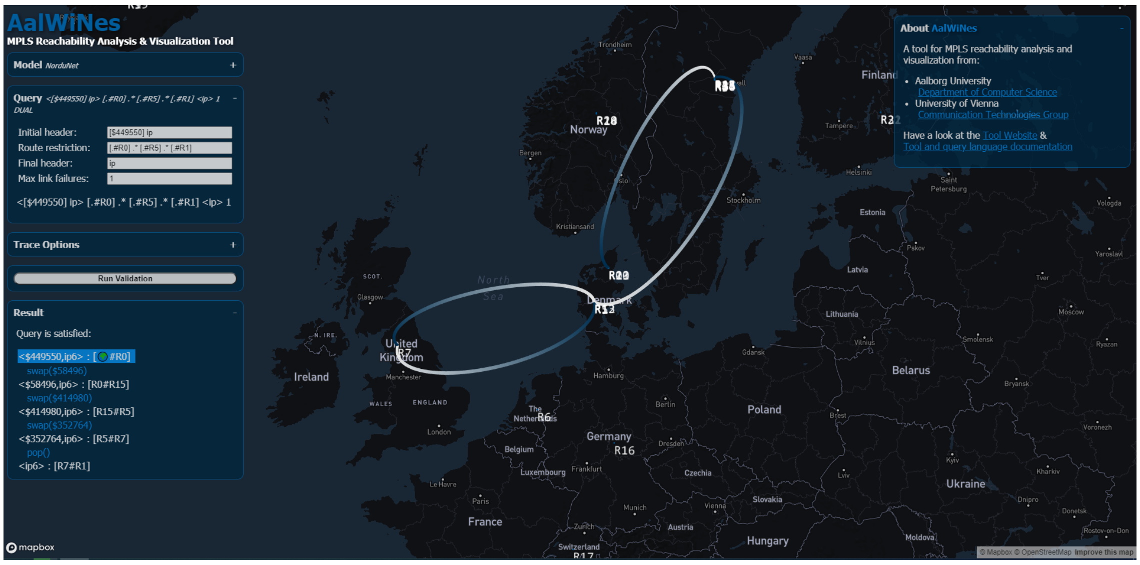
Figure 5.2. Cactus plot over the weighted properties and unweighted for all conclusive test cases, executed on *post** with reduction 4.

The cactus plot in Figure 5.2 shows how the weight annotation increase the likelihood to find a conclusive answer over the unweighted approaches. Remember that dual is a combination of the unweighted over- and under-approximation, such that if over-approximation is inconclusive then the under-approximation is used. From these unweighted graphs can we see the importance of both over- and under-approximation as they in collaboration solve many test-cases. Dual are however conclusive in fewer test-cases than for example than for example using the weight LocalFailures, but dual has in most test-cases a slightly faster verification-time.

Since dual does not include the time of potentially both over- and under-approximation, we expect the trade-off between dual and local failures to be less significant, since local failure only execute on over-approximation. From the local failures graph can we see the importance of weight annotation as it is conclusive for the most test-cases, in fact we notice that it provides conclusive answers for all test-cases within the environmental limitations of this test. The results show that local failures can solve 535 test-cases that dual cannot solve within the same time limitations, however dual solves 139 test-cases where local failures times out. In the following case study of NORDUNET, we will explore this space more thoroughly by measuring the full time of executing a query on a realistic sized network without having a timeout limit.

5.2 Case Study: NORDUNET

NORDUNET is a collaboration between the National Research and Education Networks (NRENs) of the five Nordic countries, connecting more than 400 research and education institutions with more than 1.2 million users [23]. The MPLS network of NORDUNET operate on a global scale, to interconnect the NRENs of the five Nordic countries, and to provide international connectivity. This connectivity is achieved through 31 routers distributed across three continents: North America, Asia and Europe.



AalWiNes
MPLS Reachability Analysis & Visualization Tool

Model *NorduNet* +

Query *<[$449550] ip> [.#R0] .* [.#R5] .* [.#R1] <ip> 1 DUAL* -

Initial header: [$449550] ip
Route restriction: [.#R0] .* [.#R5] .* [.#R1]
Final header: ip
Max link failures: 1

<[$449550] ip> [.#R0] .* [.#R5] .* [.#R1] <ip> 1

Trace Options +

Run Validation

Result -

Query is satisfied:

```
<$449550,ip6> : [ #R0]
  swap($58496)
<$58496,ip6> : [R0#R15]
  swap($414980)
<$414980,ip6> : [R15#R5]
  swap($352764)
<$352764,ip6> : [R5#R7]
  pop()
<ip6> : [R7#R1]
```

About AalWiNes -

A tool for MPLS reachability analysis and visualization from:

- Aalborg University
  Department of Computer Science
- University of Vienna
  Communication Technologies Group

Have a look at the Tool Website &
Tool and query language documentation

Figure 5.3. NORDUNET MPLS network loaded into the AalWiNes GUI with a validated trace, from router 0 through router 5 to router 1 with maximum 1 link failure.

Figure 5.3 shows the result of verifying a query on the NORDUNET network loaded into the graphical user-interface (GUI), which is developed in collaboration with Bernhard Schrenk at the University of Vienna [14]. This work improves the experience of the AalWiNes tool, providing visualization for our contributions to the verification part of the tool.

The tool includes a large set of example network topologies extracted from Topology Zoo [11], with a routing table generated using our algorithm from Section 2.4. The user can experiment with these networks either by applying the example queries or by specifying a query in the tool. For convenience this is split into four fields for specifying respectively the initial header, path restrictions, final header and maximum number of failures. Trace options in the GUI allow the user to navigate our main contributions by specifying which engine to use; *post**, *pre** or Moped, and to create a weight expression of the quantitative trace properties to use in the verification.

The example experiment shown in Figure 5.3, contains the query:

$$< [\texttt{\$449550}]\ \texttt{ip} > [\cdot\ \texttt{\#}R0]\ \ \cdot^*\ \ [\cdot\ \texttt{\#}R5]\ \ \cdot^*\ \ [\cdot\ \texttt{\#}R1] < \texttt{ip} > 1$$

and the weight (LocalFailures) verified on *post**. The witness trace starts on an edge from router 0, towards an edge ending at router 1 passing through router 5 while minimizing the number of failures in the trace. The result prove that such a trace exists, with 1 or 0 failures encountered.

In this section we conduct a more thorough experiment on NORDUnet's MPLS network to test the behavior and the applicability of AalWiNes as a tool. The queries we explore are similar to the previous section with $k \in \{0, \dots 3\}$ and in total 6000 test cases.

Note the smaller size of NORDUnet compared to one of the largest generated networks where NORDUnet (Routers: 31, Labels: 20.962, Rules: 562.249) and Bellcanada-5 (Routers: 240, Labels: 138.288, Rules: 537.812).

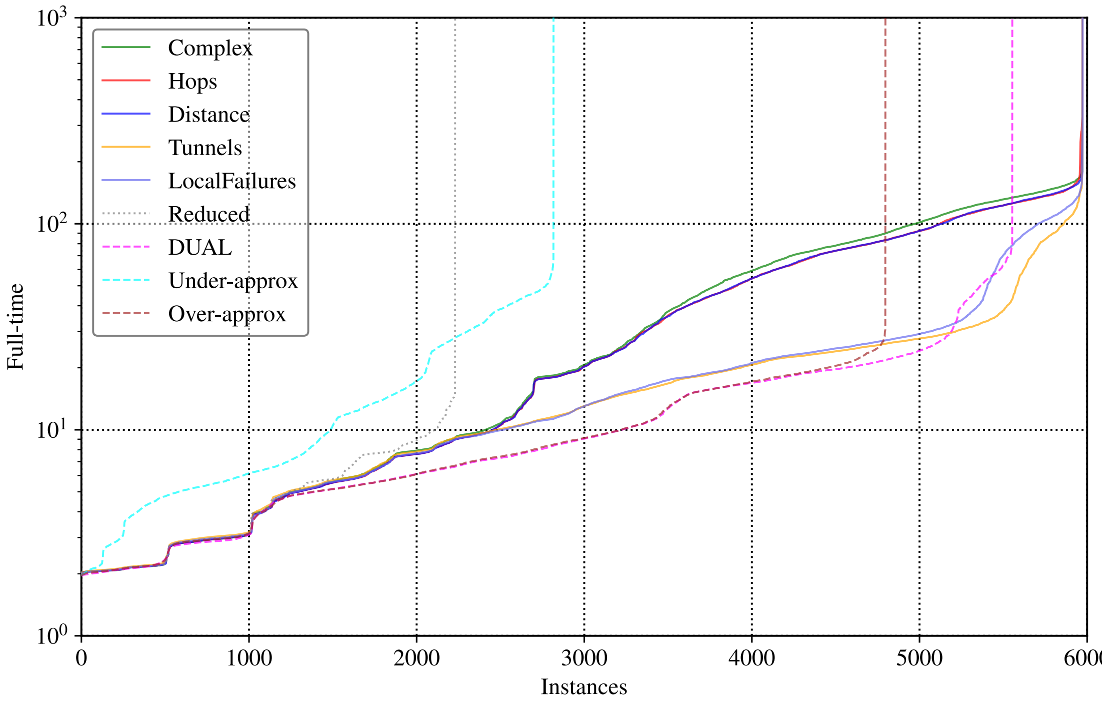


Figure 5.4. Cactus plots of all queries on NORDUnet MPLS network, executed on *post** with reduction 4.

In this test we measure the full time used when querying the tool, such that $t_{full} = t_{total} + t_{parsing}$. For dual mode this includes the time of applying both over- and under-approximation in sequence in the case over-approximation is inconclusive. Figure 5.4 plots the results of the different weight properties similarly to the experiment conducted in Subsection 5.1.6, and confirms how minimizing any of the weight properties will increase the likelihood of finding a valid witness trace.

We observe for e.g. LocalFailures that it is inconclusive for eight test-cases, and that dual is inconclusive for 427 test-cases. Thus, LocalFailures are conclusive in 419 test cases for which dual is inconclusive, and dual cannot solve any of the queries where the weighted approaches are inconclusive. For Tunnels and LocalFailures in Figure 5.4 we note a small performance trade-off when comparing to the unweighted dual approach. However, we note a larger performance trade-off for Complex, Distance, and Hops. We expect this to be due to the different order of transition affecting when the early termination can be used. Similar to Figure 5.2, we notice that this performance trade-off occurs when reduction 4 is not able to reduce the constructed PDA to zero rules, shown as the reduced graph (grey dotted line).

In this last part, we explore a small set of queries provided by NORDUnet, such that, we test the applicability of AalWiNes for a set of queries that are relevant for NORDUnet to answer on their real MPLS network. Each of the queries are executed on the cluster for both $Mpost^*$ and $post^*$ with reduction 0 for a direct comparison. All queries are conclusive for both engines, however the verification times vary. Table 5.6 lists the provided queries and the verification time for the reachability analysis on $Mpost^*$ and $post^*$. It shows the performance increase gained from our contributions to unweighted AalWiNes.

Query	$t^{Mpost^*}_{verification}$	$t^{post^*}_{verification}$
$< \texttt{smpls ip} > [\cdot\ \#R6]\ \cdot^*\ [\cdot\ \#R4] < \texttt{smpls ip} > 1$	10.6855	1.0935
$< \texttt{smpls ip} > [\cdot\ \#R2]\ \cdot^*\ [\cdot\ \#R18] < (\texttt{mpls}^*\ \texttt{smpls})?\ \texttt{ip} > 1$	11.7013	0.8959
$< \texttt{ip} > [\cdot\ \#R0]\ \cdot^*\ [\cdot\ \#R4] < \texttt{ip} > 0$	0.9722	0.0099
$< [\$\texttt{449550}]\ \texttt{ip} > [\cdot\ \#R0]\ \cdot^*\ [\cdot\ \#R5]\ \cdot^*\ [\cdot\ \#R1] < \texttt{ip} > 0$	1.7478	0.0169
$< [\$\texttt{449550}]\ \texttt{ip} > [\cdot\ \#R0]\ \cdot^*\ [\cdot\ \#R5]\ \cdot^*\ [\cdot\ \#R1] < \texttt{ip} > 1$	6.9725	0.0453
$< \texttt{smpls ip} > \cdots\cdots\ \cdot^* < \cdot^* > 1$	112.457	6.5489
$< \texttt{smpls? ip} > \cdot^* < \cdot\ \texttt{smpls ip} > 0$	108.357	15.8453

Table 5.6. Seven queries provided by NORDUnet solved by two different AalWiNes engines.

Each of the seven queries in Table 5.6 indicate a significant performance increase on verification time, when using the PDAAAL reachability library. The first five queries show acceptable verification times on $Mpost^*$, however, the performance of the remaining two queries are very slow. The same queries are executed much faster on our $post^*$ implementation and show the importance of the performance improvements from our contributions. We expect that a full analysis of NORDUnet's MPLS network will require at least several thousand queries, so the performance of verification will become a large factor in the usefulness of the tool.

The combination of the GUI, the weighted extension and the performance increase show how AalWiNes is improved, making it a more applicable tool for MPLS network analysis.

Conclusion 6

We extended the Multiprotocol Label Switching (MPLS) network analysis tool AalWiNes [4, 1] with quantitative analysis, we presented an algorithm for generating realistic MPLS networks, and we improved the performance of reachability analysis of pushdown systems in the library PDAAAL used by AalWiNes.

For quantitative analysis we defined an expression language for properties of traces in MPLS networks. The weighted trace problem was defined as finding a query-satisfying trace that minimizes a given weight expression of these network trace properties. We solved the weighted trace problem by decomposing the weight expression into weights on each rule of a weighted pushdown system. We defined the reachability problem for weighted pushdown systems and presented a concrete algorithm for shortest trace reachability analysis.

The property of maximum stack size in a trace is a global property in the sense that it cannot be decomposed. For this property we presented a solution that keeps all relevant information in the weights, so a trace that minimizes this property can be reconstructed in the end.

For generating MPLS networks, we presented the functions Concat and Inject that combines network topologies to larger topologies. We also presented an algorithm that for any topology generates a routing table with label switched paths for data transfer and fast-reroute paths to overcome failures based on a shortest path algorithm. To reduce the size of these routing tables, we showed how sequence of operations can be reduced to smaller equivalent sequences.

We implemented the *pre** and *post** algorithms for reachability analysis in the library PDAAAL. We improved the performance by implementing an early termination check that allows the algorithms to terminate as soon as the target is found to be reachable. We also improved the performance by using the right data structures in various parts of the program.

We tested the configurations of the tool to find the best overall performance increase. The experiments were conducted on automatically generated networks of various sizes, and on a case-study of a real network from NORDUnet. The performance tests showed that the *pre** and *post** algorithms in PDAAAL on average outperform the state-of-the-art tool Moped, which was previously used as a pushdown system reachability back-end for AalWiNes. The extension with weighted analysis did incur a little time overhead compared to the unweighted algorithms.

On the other side, the weighted reachability algorithm was better at succesfully reconstructing traces from the over-approximation that AalWiNes uses to model MPLS networks as pushdown systems. In fact the weight LocalFailures completely eliminates the use of the under-approximation in AalWiNes, which gives a performance increase, when the result of over-approximation is inconclusive. The extension with weights also improved the tool in the sense that fewer queries give inconclusive results.

6.1 Future Work

We suggest some ideas for future work to extend to network manipulation operations, further improve verification performance, find an exact solution to query satisfiability and extend the query language to support more types of questions.

6.1.1 Manipulation of networks with routing tables

We defined the functions Concat and Inject to manipulate network topologies, and we defined an algorithm to create routing tables for topologies. However, this cannot be used to combine networks containing routing tables. There are non-trivial decisions on how to extend Concat and Inject to work on full networks. It is interesting for future work to find a general solution with intuitive semantics for extending the manipulation operators to full MPLS networks.

6.1.2 Performance improvement by abstraction

Some real world networks have many labels with the exact same behaviour. To improve the performance of verification in these cases, it is interesting to explore using counterexample-guided abstraction refinement (CEGAR) [24] to find the groupings of these equivalent labels and perform verification on the smaller abstract model. CEGAR works by first creating an over-abstraction of the system, and then refining it until the reachability result can be reconstructed in the original system. This way the system undergoing reachability analysis will only be as complex as needed to verify the query, which may improve performance.

6.1.3 Towards an exact solution for query satisfiability

The LocalFailures weight atom outperforms the under-approximation construction in [4]. However it is still not an exact solution to the query satisfiability problem, since in the case of loops on the trace, the same links may be counted twice and links may be used as both active and failed in different parts of the trace, hence trace validation may still fail. We described the weight atom Failures, which avoids counting links twice. Due to Failures being a global property, we did not provide a solution for it, but it is interesting for future work to try using a construction similar to the global property StackSize to solve the problem of minimizing the Failures property. Combining StackSize and Failures with the local properties in the weight expression language is also left for future work.

Given that the FAILURES property may be solvable, it is also interesting to investigate an exact solution along the same lines. This would require the weight to keep track of both the set of failed links and set of active links, and encode a form of on-the-fly trace validation into the weight operators. Such an approach would have an exponential worst-case complexity, but it may in practice perform much better than constructing a pushdown system of exponential size, which is the more straightforward approach for an exact solution.

6.1.4 Extending the query language

The current query language models queries asking whether there *exists* a set of failed links no larger than k, such that there exists a network trace satisfying $< a > b < c >$. The query language cannot model queries asking whether *for all* sets of failed links no larger than k there exists a network trace satisfying $< a > b < c >$. This type of query is desirable for analysing connectivity under failure, e.g. whether two routers will always have a connection if at most k links fail.

Alternating pushdown systems generalise pushdown systems with existential and universal states, and reachability of alternating pushdown systems is decidable in polynomial time [25]. It is interesting for future research to extend the query language with quantifiers and investigate if the corresponding satisfiability problem can be modelled using alternating pushdown systems.

Another type of question that cannot be answered with the current query language, is e.g. whether there exists an infinite trace that keeps visiting a specific router. This type of query is a liveness property and can be expressed in linear temporal logic (LTL). Model checking LTL properties of pushdown systems is DEXPTIME-complete, but for a fixed LTL formula it is polynomial in the size of the pushdown system [25]. Given this, it would be interesting to extend the query language and verification algorithms to support LTL model checking.

Bibliography

[1] *AalWiNes*. `https://github.com/DEIS-Tools/AalWiNes`. Visited: 11/05/2020.

[2] *Moped - A Model-Checker for Pushdown Systems*. `http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/`. Visited: 03/03/2020.

[3] D. Kozen, "Netkat—a formal system for the verification of networks," in *Asian Symposium on Programming Languages and Systems*, pp. 1–18, Springer, 2014.

[4] J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "P-rex: Fast verification of mpls networks with multiple link failures," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pp. 217–227, 2018.

[5] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pp. 113–126, 2012.

[6] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pp. 15–27, 2013.

[7] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 290–301, 2011.

[8] *Introduction to MPLS*. `https://www.cisco.com/c/dam/global/fr_ca/training-events/pdfs/Intro_to_mpls.pdf`. Visited: 19/05/2020.

[9] Cisco, *What Is a WAN? Wide-Area Network*. `https://www.cisco.com/c/en/us/products/switches/what-is-a-wan-wide-area-network.html`. Visited: 27/05/2020.

[10] Cisco, *Introduction to Cisco MPLS VPN Technology*. `https://www.cisco.com/c/en/us/td/docs/net_mgmt/vpn_solutions_center/2-0/mpls/provisioning/guide/PGmpls1.html`. Visited: 27/05/2020.

[11] *The Internet Topology Zoo*. `http://www.topology-zoo.org/`. Visited: 19/04/2020.

[12] *IP overview*. `https://www.cisco.com/c/en/us/support/docs/ip/routing-information-protocol-rip/13769-5.html#ip`. Visited: 19/05/2020.

[13] G. A. Mazhin, M. Bag-Mohammadi, M. Ghasemi, and S. Feizi, “Multi-layer architecture for realization of network virtualization using mpls technology,” *ICT Express*, vol. 3, no. 1, pp. 43–47, 2017.

[14] B. Schrenk, *VisuAalWiNes - A visualization frontend for the AalWiNes tool.* `https://github.com/DEIS-Tools/VisuAalWiNes`. Visited: 03/06/2020.

[15] S. Schwoon, *Model-checking pushdown systems.* PhD thesis, Technische Universität München, 2002.

[16] J. R. Büchi, “Regular canonical systems,” *Archiv für mathematische Logik und Grundlagenforschung*, vol. 6, no. 3-4, pp. 91–111, 1964.

[17] M. Holzer and M. Kutrib, “Descriptional and computational complexity of finite automata—a survey,” *Information and Computation*, vol. 209, no. 3, pp. 456–470, 2011.

[18] M. Sipser, “Introduction to the theory of computation,” *ACM Sigact News*, vol. 27, no. 1, pp. 27–29, 1996.

[19] T. Reps, S. Schwoon, S. Jha, and D. Melski, “Weighted pushdown systems and their application to interprocedural dataflow analysis,” *Science of Computer Programming*, vol. 58, no. 1-2, pp. 206–263, 2005.

[20] M. Mohri, “Semiring frameworks and algorithms for shortest-distance problems,” *Journal of Automata, Languages and Combinatorics*, vol. 7, no. 3, pp. 321–350, 2002.

[21] M. K. Schou, *Reachability Analysis of Pushdown Systems - Featuring Trace Generation.* Aalborg University, January 2020. 2.5 ECTS assignment.

[22] *Cactus Plot.* `https://github.com/alexeyignatiev/mkplot`. Visited: 17/05/2020.

[23] *NordUnet.* `https://www.nordu.net/`. Visited: 25/03/2020.

[24] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement,” in *International Conference on Computer Aided Verification*, pp. 154–169, Springer, 2000.

[25] A. Bouajjani, J. Esparza, and O. Maler, “Reachability analysis of pushdown automata: Application to model-checking,” in *International Conference on Concurrency Theory*, pp. 135–150, Springer, 1997.