Framework for path finding in multi-layer transport networks

Dijkstra, F.

Publication date
2009
Document Version
Final published version

Chapter 8

Path Finding Implementation

This chapter brings the results of the previous chapters together: the model in chapter 4, the syntax and technology applicability in chapter 6, and the path finding algorithm in chapter 7. This chapter describes a software implementation of the algorithm, as well as description of the technology and a network, and shows this input and algorithm indeed are capable of producing the shortest paths in a given multi-layer network.

8.1 Modelling the Network

We turn to the example network in figure 3.2 of chapter 3, and model that network in practice. Figure 4.8 already showed how to model this network using functional elements. However, that model did not explicitly take inverse multiplexing into account. A practical implementation should model that, and may also describe the layers below the SDH/SONET layer.

Beside the topology, the technologies must also be described, and we did so in chapter 6. In section 6.2 we saw the implementation of a conceptual layer schema, and in section 6.3 we saw the implementation of technology schemata. Our claim is that we described most technologies in a generic way, using only a few classes (Layer, Label, and Adaptation). This allows the creation of a path finding algorithm that does not have to be changed if new technologies come along, by only relying on these three concepts instead.

Looking at table 6.2, we model the technologies in our example network as the following layers: Ethernet, VC-4, STS-3, OC-192, Lambda, Fibre and UTP, with the following adaptations: Ethernet in 8 VC-4, Ethernet in 7 VC-4, VC-4 in STS-3, 64 STS-3 in OC-192, OC-192 in Lambda, Lambda in Fibre, and Ethernet in UTP.

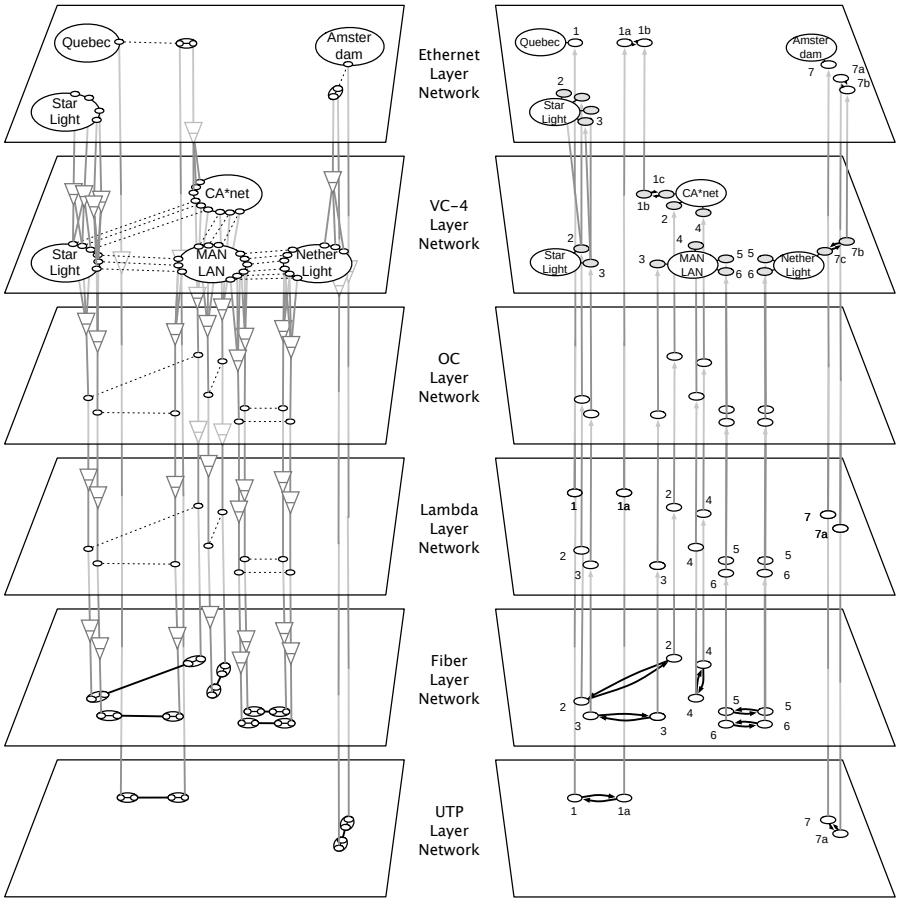

Figure 8.1: *Functional model (left) and graphical representation of the syntax (right) of the network of figure 3.2. The use of* potential interfaces *(the grey interfaces) reduces the number of logical connection points.*

If we also incorporate inverse multiplexing, we model the figure as seen in figure 8.1 on the left.

In order to turn this functional model into the syntax, we use the RDF syntax as defined in chapters 5 and 6. The result is a set of instances with triplets to describe the relation between instances. We can visualise this by using vertices for instances and edges for statements. If we group all instances with the same layer property together, we get the image as seen on the right of figure 8.1. Each circle is an instance of an RDF class; either a *SwitchMatrix* (the named circles), *ConfigurableInterface*, *StaticInterface* (the white circles) or *PotentialInterface* (the grey circles). All lines are RDF properties, either an adaptation properties we defined in the technology schemata, a *switchTo*, *linkTo*, or *hasInterface* property.

While the general structure of the syntax is the same as the functional model on the left, there are a few changes, beside the obvious change from functional elements to RDF classes and properties.

The syntax on the right is more compact than the functional model on the left due to the use of **potential interfaces**. Rather than describing each channel, the VC-4 and STS-3 channels are described as a set of potential interfaces for each physical interface.

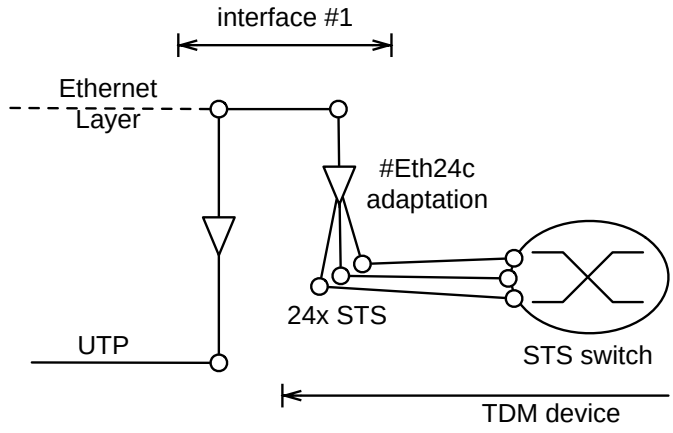

Figure 8.2: *Functional model an inverse multiplexing network interface.*

In section 4.6.2 we discussed that there can be interfaces that have two adaptation stacks. Interface 1 at CAnet (the one connected to Quebec) is such an interface. Figure 8.2 shows an abstract representation of this interface.

Incoming traffic is first de-adapted from the physical layer (in this example, the UTP layer) to Ethernet. The Ethernet packets are subsequently adapted in 24 STS channels, which are connected to the switch matrix. While the connection point after the de-adaptation is logically the same as the connection point just before the adaptation, our network description modelled it as two Interfaces, connected by a cross connect (*switchTo* property). In addition, the connection points after the adaptation are modelled as two Interfaces, connected with a link (*linkTo* statement). While this is not required by the syntax, and arguably is not completely in line with the true meaning of the linkTo and switchTo semantics, it makes our model easier to parse: with this restriction, all de-adaptations lead to a switch matrix (or cross connect), while all adaptations lead to a link. Also, no de-adaptation is immediately followed by an adaptation (there is always a switchTo in between them).

8.2 Software Framework

In addition to the schemata for both the technologies and the network, we created an experimental software framework in Python, called Python NDL toolkit (Pynt). The framework implements classes for layers, adaptations, labels, label sets, as well as devices, logical interfaces, switch matrices, links and domains.

The Pynt software is freely available under a BSD-style license [u1].

This framework is intended for use in various tools:

- Description of the current configuration of our network, and trace network connections;
- Generation of sample networks;
- Path finding of multi-layer connections through the network;
- Fault Isolation of errors in multi-layer network connections.

Using this framework, we were able to make multi-layer network descriptions of the example network given in chapter 3.

In addition, we also modelled an Ethernet network with tagged and untagged links, and our own network containing optical cross connects, Ethernet switches and end hosts. The configuration of our network is automatically generated from the state of the devices using a cron job. The other sample networks were created by hand.

The framework could be used to create a random network. However, it is not obvious how a realistic random network generator would look like. While algorithms such as Barabási-Albert preferential attachment model [p1] or Holme-Kim algorithm [p17] create realistic graphs with power-law degree distribution, this only helps in a realistic network of devices or of domains, thus a single-layer network. It does not help to generate a realistic layer distribution– most multi-layer networks are far from random, since network engineers consciously try to avoid incompatibilities. It is unclear how the distribution of incompatibilities is in practice.

8.3 Path Finding Software

The path finding algorithm is one of the modules of the whole software framework.

8.3.1 Path Finding in G_l

Using the software framework described in the previous section, we build a path finding algorithm. We used the algorithm for path finding in G_l, as described in section 7.4. The reason to choose for path finding in G_l as opposed to G_s is that it is is easy to map the network topology to the model, as explained in section 8.3.5.

We did modify the algorithm as described in chapter 7 slightly: instead using domains for nodes, we used interfaces for nodes. The advantage was twofold: first this is closer to the model we defined (which does define physical devices and logical interfaces, but not logical devices), and secondly it allows us to take the switching and swapping capabilities of switch matrices into account.

8.3.2 Software Logic

Input:

- URI of source and destination interface; and
- URL of network description that includes the source interface.

Output:

- Sequence of connection points, connection type between the connection points (link, adaptation, de-adaptation or cross connect), and available labels for each connection point; or

- Error, in case no shortest path can be found.

In the initialisation phase, the software reads the network description and determines the technologies that are used for that network description. It then downloads and parses those network descriptions before proceeding. Figure 8.3 shows the output of this initialisation phase for an example network.

```
Fetching technology descriptions
Parsing RDF input http://www.science.uva.nl/research/sne/schema/ethernet.rdf
Parsing RDF input http://www.science.uva.nl/research/sne/schema/tdm.rdf
Parsing RDF input http://www.science.uva.nl/research/sne/schema/wdm.rdf
Parsing RDF input http://www.science.uva.nl/research/sne/schema/copper.rdf
Building description of given network in memory
```

Figure 8.3: *Output of the initialisation phase of the path finding software.*

The main routine of the software keeps a list of possible paths, which is initialised with one path consisting only of the source interface.

After initialisation, the software enters a loop where it continuously picks the shortest path from the list and checks if it is a valid path from source to destination. If so, it returns it as the shortest valid path. If not, it extends the path with a hop to a neighbouring interface, and checks if that could lead to a valid path, eliminating hops with incompatible labels or incompatible de-adaptation functions. It adds these extended options to the list of possible paths and continuous the loop. This is the Multi-Layer-Breadth-First algorithm described in chapter 7.

8.3.3 Path Walk

The routine in the path finding algorithm that checks if an extension of a path can lead to a valid path or is unfeasible contains the logic of the algorithm. This is the Extend-Path subroutine that is part of the Path selection in G_l algorithm, as described in chapter 7.

If we modify this subroutine, we can create other algorithms. For example, we also created a path walk algorithm. This path walk algorithm does not extend the path with *possible connections*, but extends it with *currently configured connections*. The result is that it simply follows existing connections (including point-to-multi-point connection in multicast or broadcast technologies such as Ethernet), rather than viable connections.

We use this path walk algorithm as the basis for a fault isolation framework, which is able to detect anomalies in the published network configuration, and thus detect and isolate faults across domains.

8.3.4 Switch Matrix Properties

One of the findings whilst implementing the logic to determine if a path is valid or unfeasible is more complex than we anticipated. For example, we first defined that a bi-directional cross connect between $cp1$ and $cp2$ exists if there is an unidirectional cross connection from $cp1$ to $cp2$, and there is an unidirectional cross connection from $cp2$ to $cp1$. Now, for a uni-directional loopback connection holds that $cp1 = cp2$. Thus according to our definition, this is also a bi-directional cross connection. However, this is not intuitive, and we had to redefine our definition of "bi-directional" to explicitly exclude this particular case.

The following section lists the properties of a switch matrix:

- switching capability: two interfaces with the same label can be crossed.
- swapping capability: two interfaces with a different label can be crossed.
- unicast: an interface can be crossed to another interface, provided that the labels match, and no other cross connect exist for the source or destination.
- multicast: an interface can be crossed to another interface, even if other cross connects with the same source exist.
- broadcast: if and only if the labels of two interfaces match, they are crossed. Broadcast is mutually incompatible with unicast.

The labels as defined above are the internal labels of interfaces. These can be different from the external (egress and ingress) labels, even though in practice the same channel identifier is used. An empty or undefined labelset ($\varnothing$) is considered to be a labelset with one element, the empty label ($\{\epsilon\}$).

Given the switching capabilities of a switch matrix, and the connected interfaces, software can enquire about the following cross connects:

Actual cross connects returns currently configured cross connects.

Potential cross connects returns cross connect that can be made by allowing any other cross connect to be broken. Takes label sets into account, but ignores current labels and current cross connects.

Available cross connects return cross connect that can be made without affecting any existing traffic (cross connects). Honours label sets, cross connects, but only labels if they are part of existing cross connects.

By default all checks only verify if a unidirectional cross connect from source to destination can be made. If the bidirectional modifier is given, it checks for the reverse connection as well. This check significantly reduced the number of possible cross connects for switch matrices with multicast capability.

The request for cross connects set can be further modified by the modifiers:

bidirectional Makes sure the reverse cross connect is also in place or is (potentially) available. (Applies to actual, potential and available cross connects)

break own cross connect Do not honour cross connects originating from source and (if bidirectional) destined to the source interface. All other cross connects remain in place (including multicast and broadcast cross connects that are part of the same group). (Applies to available cross connects only)

allow data merger Allows cross connects that create new data mergers (currently only supported for broadcast switch matrices with more than two interfaces with the same label.) A merge may affect the available bandwidth of existing connections. (Applies to available cross connects only)

honour label Honour the current label of all interfaces, not just the labels of interface that are part of cross connects. (Applies to potential and available cross connects)

So a request for available bi-directional cross connect, for a unicast switching matrix will not return cross connects with a connection point as destination if that connection point is already in use by an existing cross connect. If *allow data merger* is set, then it will return such cross connect.

While the exact details of the above logic are interesting, the main point is that it shows that the logic to check for available network connections is

far from trivial, despite our relative simple network model. The logic of a technology dependent model would be even more complex, and it would indeed be unfeasible to deploy such logic reliably in a multi-domain environment.

8.3.5 Multi-Domain scalability

We used the algorithm for path fining in G_l as opposed to the algorithm for path finding in G_s since path finding in G_s would require the construction of a graph incorporating every possible adaptation stack, which requires a-priori knowledge of the whole network.

The algorithm for path finding in G_l only has to fetch information about a domain if there is a path that leads to that domain, but not earlier than that. For that reason, G_l scales better in a multi-domain environment, and that is the reason we implemented that algorithm.

Basically, the path find and path walk algorithm find end-to-end paths by following the links, adaptations and cross connects through domains. When reading a network description, the algorithm stores information about the network elements in memory, including information `seeAlso` statements pointing from a given network element to external resource with further information about that network element.

As soon as the path finding algorithm extends the path to a network element that contains a `seeAlso` statements, it then loads that information, but not earlier than that. In this way, the algorithm can follow the path through multiple operational domains.

8.3.6 Result

If we feed the implemented algorithm in our example network, it is indeed able to find the shortest valid path.

Figure 8.4 shows a graphical representation of the classes and properties that make up the network description. Edges in figure 8.4 represent RDF properties between the different instances in the model, which in turn represent either link connections or adaptations in the network.

This vertices and edges in this figure should be **exactly** the same as the right hand side of figure 8.1. The only difference is that the vertices (RDF instances) in figure 8.1 are grouped by layer, while in figure 8.4 they are grouped by domain. The different colours on the vertices represent the different layers.

The figure also shows the end result of path finding algorithm. The lighter green edges are part of the resulting path. The part of the network between CAnet and MAN LAN is used twice, as is represented by a thicker line.

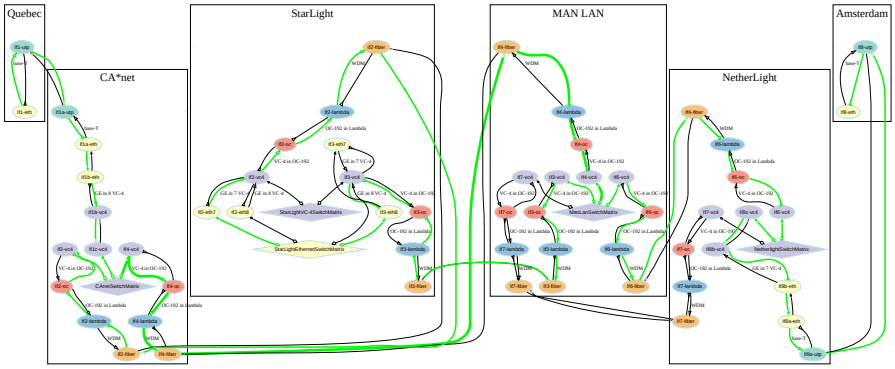

Figure 8.4: *The visual output of an automated path finding algorithm.*

Indeed, this path is the shortest path as we have seen in figure 3.5 in chapter 3.

Given that our algorithm is a path-constrained path finding algorithm, the algorithm and implementation have no dependencies on the actual network or technology, but only on the technology-independent multi-layer network model, and given that we could describe all technologies that are in use in the GLIF community in this model, then we must conclude that **path-constrained algorithms are sufficient for path finding in multi-layer networks**, at least for all technologies as used in the GLIF community.

This claim has implications for the information exchange between domains. In case a path finding is done domain-by-domain (as opposed to having one central orchestrator), each request must contain information about the final source and final destination. If that information is not present, the request may lead to false positives or false negatives.

8.3.7 Ambiguity of Labels

In section 6.3.7, we discussed that the label concept is used for two separate meanings: (1) to distinguish between channels in multiplexing; and (2) to determine allowed cross connects in switch matrices using the switching and swapping capability.

Recall that Ethernet labels are VLANs, and are either an integer in the range $0 - 4095$ or the empty label ϵ. In all circumstances, connection points have a label which is used in the second meaning, to determine valid cross

connects (the labels must be the same: it is not possible to exchange traffic between two different VLANs). For untagged Ethernet, this VLAN label is not present on the wire, and the external egress label is ϵ. For tagged Ethernet, modelled as Ethernet over Ethernet, the VLAN label is present on the wire as the IEEE 802.1q label, and the external egress label must be set after the cross connect.

We came up with the following logic to determine if the external egress label is set:

- If a label is set for an Interface, then it is assumed to be the internal label, and used in the switch matrix to determine if a cross connect (switchTo) is allowed.
- If a switch matrix has swapping capability, the label may change. The next hop uses the label constraints of the interface. If the switch matrix has no swapping capability, the next hop uses the intersection of the label constraints of the interface and the path so far.
- In addition to label swapping in a matrix, labels may change along the path if the label is not carried on the wire. For example, an untagged Ethernet interface in VLAN 42 may be linkTo an untagged Ethernet interface in VLAN 28. The logic to determine if a label is carried on the wire is:
  - there is a `hasexternallabel` property explicitly set to true
  - if the interface is client of a multiplexing adaptation, the label is carried on the wire.
  - if the interface is client of a non-multiplexing adaptation, the label is **not** carried on the wire (and may change)
  - if there is no adaptation, but only a link, the label is carried on the wire.

Since this logic is mostly deterministic, and not 100% accurate, it is now recommended to explicitly set the `hasexternallabel` property, which signifies if a specific interface is tagged or untagged Ethernet.

8.4 Optimization

Section 8.3.3 discussed the routine in the path finding algorithm that checks if an extension of a path can lead to a valid path. This routine should never

return a path that is unfeasible. That is, a section of a path which can never be part of a valid connection.

The software documentation describes this routine as follows:

> Verifies if a path is valid. Only the last hop in the path has to be checked; it can be assumed that the rest of the path has been checked earlier.
>
> Typical checks to perform are:
>
> - is the adaptation stack valid, especially if the connection is De-Adaptation
> - does the current connection point further restrict the number of allowed labels
> - are there more available labels left then the interfacecount
> - does the current connection point restrict the number of layer Properties to None
> - does it not give a loop (i.e. has the interface been used before in the same path, with the same stack). This a non-obvious check in multi-layer networks: data may get transported through two channels, which use the same physical interface.
> - is the configuration not forbidden because an earlier configuration (e.g., we can't use the same VLAN if it was already used earlier in the same path)
> - Are there policy constraints.
> - etc., etc.

However, there are no restrictions if it should only return a subsection of a path which can never be part of the shortest **valid** connection. In fact, since it does not know the shortest valid connection beforehand, it will regularly return paths turn out to be "dead ends", paths that are not part of the shortest valid connection. It is possible to optimise this part.

We developed a few variants to our base algorithm.

Unrestricted flooding No optimisation. Practically impossible loopbacks, such as $A-B-C-B-C-D$ are not filtered out from the intermediate

results, although they never show up as final result since they are clearly not shorter than $A - B - C - D$.

No loopbacks Do not allow a path to return in the direction it just came from. E.g. $A - B - C - B$. Such loopbacks can be part of a shortest path if C is a potential interface in a switch matrix with label switching capability.

Explicit direction Define two directions: inbound and outbound, and keep track if a path is currently going inbound or outbound. A switch matrix or cross connect changes the direction from inbound to outbound, and a link connection changes it from outbound to inbound. Adaptation are always outbound, de-adaptations always inbound. This simple check eliminates the need to check for impossible loopbacks inside a device, such as a cross connect followed by another cross connect.

No repeated stacks in path Broadcast suppression. If we reach an interface with a certain stack, but this path already contains a hop with this interface and the same stack, we can stop the algorithm.

All the above algorithms will come up with the correct shortest path. Crucial to all above algorithms is the loop detection. Each interface may be used multiple times if it contains enough available channels. These channels may not be on the layer of the last connection point in the path, but may be available higher up the adaptation stack. This can be avoided by only checking the available channel count check right after a cross connect of a potential interface, which is the client interface of a multiplexing adaptation function, and not to check at all other interfaces.

Besides these algorithms, we also developed a few heuristic algorithms that contain a less elaborate loop detection mechanism, at the cost of given false positives or false negatives.

Shortest path to stack only Broadcast suppression. If there are two ways to get from A to B, then the second path that traverses along switch matrix B stops there. This yields false negatives if the shortest path uses a link that is required later. This optimisation greatly reduces the flooding mechanism of breadth first search algorithm, but can not be used if the goal is to find multiple (backup) paths.

Interfaces used once Interfaces may only be used once. This yields false negatives if the shortest path contains a loop (that path will not be found).

Adaptation Restriction only Do not check for the number of available channels nor check for compatible labels. This will yield false positives if the number of available channels affects the shortest path, such as in our example network on the link between CAnet and StarLight.

Topology Restriction only Do not check the number of available channels, compatible labels or compatible adaptations. This will yield false positives if the shortest path is restricted by the number of available channels, or is restricted by an incompatible adaptation such as in our example network between CAnet and NetherLight.

Table 8.1 shows the number of iterations that are required to find the shortest path in the example network introduced in chapter 3.

Algorithm variant	Number of iterations
Unrestricted flooding	$1.6 \times 10^{15} \pm 36\%$ iterations
No loopbacks	$18 \times 10^{12} \pm 27\%$ iterations
Explicit direction	587 iterations
No repeated stacks in path	486 iterations
Shortest path to stack only	245 iterations
Interfaces used once	No result found
Adaptation restriction only	False positive after 219 iterations
Topology restriction only	False positive after 134 iterations

Table 8.1: *Number of iterations required by algorithm variants to find the shortest path in our example network.*

The *unrestricted flooding* and *no loopback* variants are extremely inefficient, and it was not possible to finish the algorithm run. Thankfully, the tree built by the algorithm grows nearly exponentially (the correlation parameter R^2 is between 0.9993 and 0.9998), and it is possible to estimate the finishing time after running the algorithm for roughly 50,000 iterations, since we know that the shortest path is 57 hops long (a metric length of 56.0 if adaptations, links and switched all have a metric length of 1.0). The error in the table comes from the sheer number of possible paths of 57 hops long, and it is unknown if the correct path is found in the first or the last iteration, or somewhere in between.

The *explicit direction* variant finds alternative paths. In our example, it finds the path Quebec – CAnet – MAN LAN – NetherLight – MAN LAN – StarLight – CAnet – MAN LAN – NetherLight – Amsterdam (with a pointless

extension MAN LAN – NetherLight – MAN LAN) after 1855 iterations. The *no repeated stacks in path* variant does not finds alternative paths, and stops after 317 iterations.

The *explicit direction* variant, which is the fastest exact variant, does return two solutions, after 486 and 488 iterations. The second solution uses the second link between MAN LAN and NetherLight. The *shortest path to stack only* variant does not find this alternative path.

The *adaptation restriction only* variant finds the false positive Quebec – CAnet – StarLight – MAN LAN – NetherLight – Amsterdam, as we predicted in figure 3.4 in chapter 3. The *topology restriction only* variant finds the false positive Quebec – CAnet – MAN LAN – NetherLight – Amsterdam, as we predicted in figure 3.3.

The *interface used once* variant does not find a path, because the shortest path in our example contains a loop: the same interface *is* used twice. In order to compare this algorithm variant with others, we created an alternative network example with two link between CAnet and MAN LAN, and additional restrictions in available time slot labels. In this case, the *interface used once* variant is able to find the solution after 6710 iterations. This a high number. The *no repeated stacks in path* variant applied to the same networks finds the shortest path after 534 iterations.

8.5 Conclusion

In conclusion, both network example gives us a clear indication that the no repeated stacks in path variant of the algorithm is able to significantly restrict the broadcast nature of our algorithm, and this seems like a very viable multi-domain multi-layer path finding algorithm. Multi-layer, because it can cope with constraints in layers, adaptation and labels. Multi-domain, because it does not need full topology knowledge beforehand.

Further optimisation, including the use of completely different algorithms, may be subject of further study. We have presented a model, syntax, algorithm and software framework that are a good start for such research.