Framework for path finding in multi-layer transport networks

Dijkstra, F.

Publication date
2009

Document Version
Final published version

Chapter 7

Path Finding Algorithms

This chapter is based on *Path Selection in Multi-Layer Networks* by F. Kuipers and F. Dijkstra [a12]. The comparison between the two algorithms in this would not have been possible without the kind permission of Fernando Kuipers to include part of his work here.

7.1 Introduction

In the previous two chapters, we created a network model and syntax. The goal in this chapter is to define a path finding algorithm based on this model and syntax. We consider two algorithms: path finding in G_l in section 7.4, developed by us, and an alternative algorithm proposed by Kuipers, path selection in G_s in section 7.5.

Both algorithms operate on a graph, although the graphs (G_l and G_s) are different for each algorithm. In addition, we define a third mapping from the model we presented in chapter 4 to a graph, resulting in the graph G_p.

This chapter is organised as follows. Section 7.2 explains the terminology and notation that is used and section 7.3 presents three graph representations of a multi-layer network. In sections 7.4 and 7.5 we discuss several path selection algorithms for multi-layer graphs consisting of two layers. Section 7.6 extends this work to incompatible labels and an arbitrary number of layers. We end with the conclusions in section 7.8.

7.2 Terminology

7.2.1 Definition of a Network

Section 4.4.5 of chapter 4 defines a network $N = (CP, L, SN, A)$ as a set of connection points CP, physical links L, subnetworks SN, and adaptations A, and its configuration $C = (LB, SC)$ as a set of labels LB, and subnetwork connections SC.

For simplicity, we ignore the details of subnetwork connections in this chapter, and replace connection points and subnetwork connections by the more generic concept of nodes. This simplification allows us to focus on the principles of the algorithms without going in too much detail.

The network technology description in this chapter uses the following sets:

The set $\mathcal{Y}$ of $|\mathcal{Y}|$ layers A layer $y \in \mathcal{Y}$ is set of related encodings, whose only difference may be in the payload, the label (as defined in section 4.4.3), or the encapsulation of the data from a higher layer;

The sets $\mathcal{A}(y)$ of $|\mathcal{A}(y)|$ adaptation functions for each layer y with y the server layer of the adaptation function. Each item $\alpha \in \mathcal{A}(y_s)$ is a tuple (y_c, y_s, b_s) with $y_c, y_s \in \mathcal{Y}$ representing an adaptation from client layer y_c to server layer y_s and b_s the required bandwidth usage at the server layer. $\mathcal{A}(y)$ may contain multiple adaptation functions between the same layer pair;

The sets $\mathcal{LB}(y)$ of $|\mathcal{LB}(y)|$ possible labels for each layer y. The length of each of these sets must be at least 1. If no labels are available, this is the set $\{\epsilon\}$ (ϵ representing the empty label).

The network description in this chapter uses the following sets:

The set N of $|N|$ nodes The definition of a node depends on the granularity, as described below. Roughly, a node can be a domain, device or physical interface;

The set C of $|C|$ channels for each node n and the layers y for that node. Each element $c \in C$ is a tuple (n, y) with $n \in N$ and $y \in \mathcal{Y}$. If a node is a physical interface, this is the same as a connection point;

The set L physical links between two channels c_1, c_2 and as defined in $N = (CP, L, SN, A)$. Each element $l \in L$ is a tuple (c_1, c_2, b) with $c_1, c_2 \in C$ and b the available bandwidth on the link. A network can be either unidirectional or bidirectional;

The set A of adaptations between channel c_1 and c_2 inside a node. Each element $a \in A$ is a tuple (c_1, c_2, α), representing the adaptation function $\alpha \in \mathcal{A}$ from client layer c_1 to server layer c_2. We assume bidirectional adaptations only: an adaptation from c_1 to c_2 implies a de-adaptation from c_2 to c_1. Defined in the network description (CP, L, SN, A);

The set $LB(c)$ of all available labels for each channel $c \in C$. Defined in the network configuration (LB, SC). $LB(c)$ is a subset of $\mathcal{LB}(y)$ with y the layer of channel c.

Note that the set of adaptations A (in $N = (CP, L, SN, A)$) and the set of adaptation functions $\mathcal{A}$ are different sets. $\mathcal{A}$ is the set of adaptation functions, and describes the technology. A is a set of adaptations between two channels in a node, and describes the implementations of this technology in a network. Similarly, the set LB describes the available labels for a specific channel, while $\mathcal{LB}$ describes the labels for a specific layer.

We define $B_a\big((y_c, y_s, b_s)\big) = b_s$, the required bandwidth b_s of the server layer for each adaptation function $(y_c, y_s, b_s) \in \mathcal{A}$.

For convenience, we also define $C_n(n) \subset C$ for all $n \in N$ to be the subset of channels in node n:

$$C_n(n) = \{(n, y) \mid y \in \mathcal{Y} \wedge (n, y) \in C\} \tag{7.1}$$

Furthermore, we define $Y_c(c) \in \mathcal{Y}$ for all $c \in C$ to be the layer of channel c, and $Y_n(n)$ to be the **set** of layers for node n:

$$Y_n(n) = \{Y_c(c) \mid c \in C_n(n)\} \tag{7.2}$$

We restrict the number of channels to one channel per node per layer:

$$C \subset N \times \mathcal{Y} \tag{7.3}$$

This restriction means that each layer can only be present once in $Y_n(n)$:

$$\forall n \in N : Y_n(n) \subset \mathcal{Y} \tag{7.4}$$

Also, this restriction allows us to give an upper limit to the size of C:

$$|C| \leq |N| \times |\mathcal{Y}| \tag{7.5}$$

The downside of this restriction is that the algorithms described in this chapter do not support explicit descriptions of multiplexing and inverse multiplexing. However, the concept of bandwidth yields a surrogate way to define inverse multiplexing (surrogate, since multiplexing is not properly supported, it is always assumed there is only one channel at the client layer). We also allow that an edge is used twice for a path, so implicit multiplexing is still supported.

7.2.2 Granularity

Each of the mappings from model to graph can be done using a different granularity.

Domain granularity corresponds to mapping of domains to vertices, ignoring the intra-domain connections. With this granularity, each node $n \in \mathcal{N}$ is a domain.

Device granularity maps devices to vertices, ignoring the cross connects within a device. With this granularity, each node $n \in \mathcal{N}$ is a physical device.

Interface granularity maps interfaces to vertices, without any abstraction. With this granularity, each node $n \in \mathcal{N}$ is a physical interface.

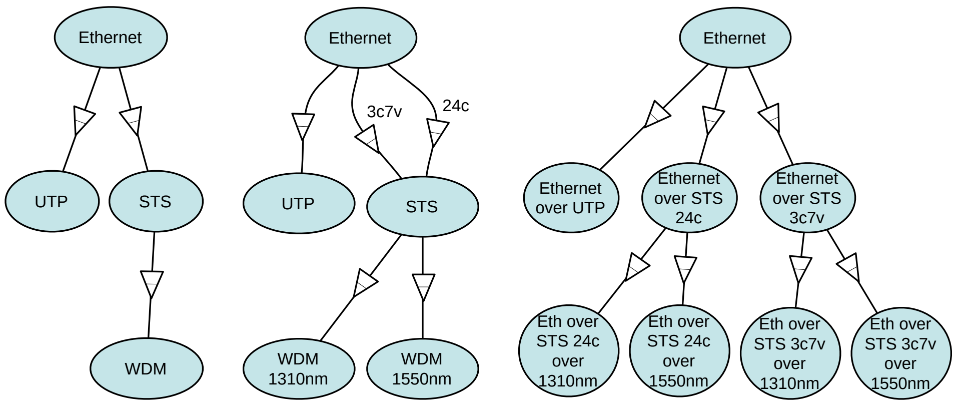
For granularity of devices, $Y_n(n)$ are the layers of the switch matrices SC, as well as the layer of the interfaces in the device. For granularity of interfaces, $C_n(n)$ are the different connection points CP that constitute the interface. For example, the layers $Y_n(n)$ of a node n which represent a Ethernet interface in a SONET switch is $\{\text{Ethernet}, \text{STS}\}$ or perhaps even $\{\text{Ethernet}, \text{STS}, \textit{lambda}, \textit{fibre}\}$

Chapter 8 describes an implementation of the first algorithm with the granularity of individual interfaces (so it can take the intrinsic details of switch matrices into account). In this chapter, we will map domains to vertices, since that corresponds to the example network we have seen in chapter 3. This also reduces the complexity of the examples.

7.2.3 Technology Stacks

An adaptation function describes the technology how data from a client layer can be embedded in the data of a server layer. An adaptation stack describes a sequence of adaptations. A technology stack, or protocol stack, describes a list of layers, where each layer acts as a server layer of the previous layer in the list.

Figure 7.1 shows three representations of a fairly common technology description. It includes four layers: $\mathcal{Y} = \{$Ethernet, STS, UTP, WDM$\}$ (figure 7.1a). Furthermore, there are two ways to encapsulate Ethernet over STS,



(a) *Protocol stack* **(b)** *Technology diagram, with adaptations and labels* **(c)** *Tree of all possible technology stacks*

Figure 7.1: *Three representations of a technology diagram.*

and there are two different labels at the WDM layer: $\mathcal{A}(STS) = \{\text{3c7v, 24c}\}$ and $\mathcal{LB}(WDM) = \{\text{1310nm, 1550nm}\}$ (figure 7.1b). While this diagram uses G.805 graphical convention to describe adaptations, a technology diagram can be represented as a directed graph with layers for vertices and edges from a client layer to a server layer (in the Layer schema of NDL, adaptations are predicates from server layer to client layer, only because that results in smaller RDF/XML files). This technology diagram leads to in total 8 possible adaptation stacks, provided that the top layer is Ethernet (figure 7.1c):

- Ethernet
- Ethernet over UTP
- Ethernet over STS with 24c adaptation
- Ethernet over STS with 3c7v adaptation
- Ethernet over STS with 24c adaptation over 1310nm
- Ethernet over STS with 24c adaptation over 1550nm
- Ethernet over STS with 3c7v adaptation over 1310nm
- Ethernet over STS with 3c7v adaptation over 1550nm

Given a technology description (the layers $\mathcal{Y}$, and for all layers y the adaptations $\mathcal{A}(y)$ and labels $\mathcal{LB}(y)$), and a choice for a ‘root’ layer, we can create a technology diagram, as displayed in figure 7.1b. The choice of root layer is significant. For example, had we chosen STS to be the root, the three possible adaptation stacks would have been:

- STS
- STS over 1310nm
- STS over 1550nm

We define a technology $t \in \mathcal{T}$ as a tuple *(layer, adaptation function, label).*

$$\mathcal{T} = \{(y, \alpha, lb) \mid y \in \mathcal{Y} \wedge \alpha \in \mathcal{A}(y) \wedge lb \in \mathcal{LB}(y)\} \tag{7.6}$$

For convenience, we define $T(y) \subset \mathcal{T}$ as all technologies with layer y:

$$T(y) = \{(y, \alpha, lb) \mid \alpha \in \mathcal{A}(y) \wedge lb \in \mathcal{LB}(y)\} \tag{7.7}$$

The number of technologies per layer is the product of the number of adaptations and the number of different labels for that layer:

$$|T(y)| = |\mathcal{A}(y)| \times |\mathcal{LB}(y)| \tag{7.8}$$

For example, consider the tributary group layer in time division multiplexing. Seven of these groups can be embedded in the underlying layer, each identified by a different label. Thus $|\mathcal{LB}(y)| = 21$. In addition, there are two way to embed tributary groups in the underlying layer: SONET packs the 7 Virtual tributary groups (VTGs) in one STS-1 SPE, while SDH packs the 7 Tributary Unit Group (TUG-2) in one TUG-3. If we model this as incompatible adaptations, then $|\mathcal{A}(y)| = 2$, and thus $|T(y)| = 14$.

A technology stack s is an ordered list of technologies $[t_0, t_1, \dots, t_n]$, from highest to lowest layer, with each consecutive layer the server layer of the previous layer (the client layer adaptation of t_{i+1} must have the same layer as technology t_i and its server layer must be the same as the layer of t_{1+1})

For example, Ethernet over STS with 24c adaptation over 1310nm is:

$$s = \left[(\text{Ethernet}, \textsc{nil}, \epsilon), (\text{STS}, \mathit{24c}, \epsilon), (\mathit{WDM}, \mathit{STS\ over\ WDM}, 1310.0)\right]$$

In this example, NIL means there is no higher layer, and thus no adaptation from a client layer. ϵ means the empty label.

The set of all adaptation stacks is $\mathcal{S}$.

For convenience, we define $Y_s(s)$ for every stack $s \in \mathcal{S}$ as the layer of the last technology in the stack s. For example, $Y_s([(\text{Ethernet}, \textsc{nil}, \epsilon), (\text{STS}, \mathit{3c7vc}, \epsilon)])$

= STS. Similarly, we define $\mathcal{A}_s(s)$ to be the adaptation function of the last item of the stack s, and $\mathcal{LB}_s(s)$ to be the label of the last item of the stack s.

A root technology t_1 consists of an given (root) layer y_1, no adaptation function, and a given label lb_1. Given the root technology t_1, we can formally define the set of all technology stacks, $\mathcal{S}$ as a recursion:

$$\mathcal{S}(y_1, lb_1) = \begin{cases} [(y_1, \text{NIL}, lb_1)] \vee & (7.9a) \\ \{s + [(y, \alpha, lb)] \mid t = (y, \alpha, lb) \in \mathcal{T} \wedge \alpha = (y_c, y, b_s) \wedge \\ \qquad s \in \mathcal{S}(y_1, lb_1) \wedge y_c = Y_s(s)\} & (7.9b) \end{cases}$$

Equation 7.9b denotes that any valid extension of a technology stack $s \in \mathcal{S}$ that leads to a new valid technology stack.

While the choice of the root layer is usually obvious (in the above example, Ethernet is the only layer that is not a server layer to another layer), this is not always true. In fact, the technology diagram can contain cycles in case of tunnels: the embedding of one technology in itself. For example, IP over Ethernet over IP for IP tunnels, or –in our model in section 6.3.7– Ethernet over Ethernet over Ethernet using stacked (Q-in-Q) VLAN tags.

Cycles in a technology diagram lead to an infinite size of the adaptation stack tree. In order to avoid this problem, if we work with adaptation stacks, we will require that there are no cycles in the (directed) technology diagram.

7.2.4 Definition of a Graph

Graphs $G(\mathcal{V}, \mathcal{E})$ consist of a set $\mathcal{V}$ of $|\mathcal{V}|$ vertices and a set $\mathcal{E}$ of $|\mathcal{E}|$ edges. A specific edge in the set $\mathcal{E}$ between nodes u and v is denoted by (u, v).

A path P is a sequence of edges, rather than a sequence of nodes.

Each edge $e = (u, v) \in \mathcal{E}$ from node u to node v is characterised by one or more weight parameters $W_e(e)$. The distance of a path $d(p)$ is a function of the weight parameters of each edge, typically the sum of the individual weights $W_e(e)$ for all $e \in P$.

Finally, we define $B_e(e)$ as the bandwidth b of edge e.

Graphs can be directed or undirected. For a undirected graph, the edges $(u, v) \in \mathcal{E}$ and (v, u) are the same edge. Some definitions of undirected graphs require that there can only be one edge between every pair of vertices. We explicitly require that there can be more edges between the same pair of vertices.

7.3 Multi-layer Network Model

In this section we provide three network descriptions. The first is a commonly used model in which each network device or network domain represents one vertex in a graph G_p, and physical network links are represented as edges. The second model represents each network device as multiple vertices in a graph G_l; one for each 'layer'. In this model, links still represent edges, but adaptations also represent edges. Finally, a model where we transform the multi-layer network into a graph G_s consisting of vertices and links on different 'encodings'.

7.3.1 Example Network

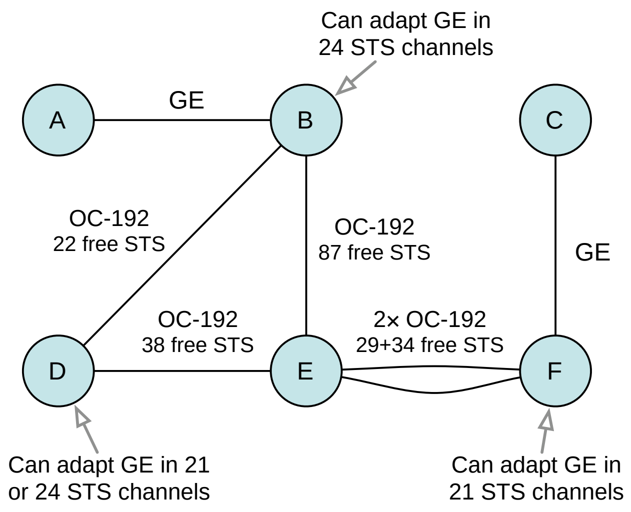


Figure 7.2: *An example of a multi-layer network, equal to the example in chapter 3. GE refers to Gigabit/second Ethernet, OC-192 are SONET-based optical carriers carrying 192 STS channels.*

Figure 7.2 is a representation of the example network we presented in chapter 3. This network consists of 6 domains, the nodes $\mathcal{N} = \{A, B, C, D, E, F\}$, and we only consider two layers $\mathcal{Y} = \{\text{Ethernet}, \text{STS}\}$, ignoring the optical carrier (OC) layer for simplicity. There are two incompatible adaptations: Gigabit Ethernet (GE) can either be adapted in 24 STS channels or in 21 STS channels (7 virtually concatenated groups of 3 concatenated channels). Since the network definition of section 7.2.1 can not explicitly represent (inverse) multiplexing, we use the concept of bandwidth instead. $\mathcal{A}(\text{STS}) = \{\textit{24c}, \textit{3c7v}\} = \{(\text{Ethernet}, \text{STS}, 24), (\text{Ethernet}, \text{STS}, 21)\}$. $\mathcal{A}(\text{Ethernet}) = \varnothing$.

Devices A and C are only aware of the Ethernet layer, and not of the STS layer, while device E only has knowledge about the STS layer, and has no knowledge about Ethernet: $Y_n(A) = Y_n(C) = \{\text{Ethernet}\}$; $Y_n(E) = \{\text{STS}\}$; $Y_n(B) = Y_n(D) = Y_n(F) = \{\text{Ethernet}, \text{STS}\}$. We will denote the channels as A_{Eth}, B_{Eth}, B_{STS}, etc. Not all devices support all adaptations: $A_A = A_C A_E = \varnothing$, $C_B = \{(B_{Eth}, B_{\text{STS}}, \textit{24c})$, $(D_{Eth}, D_{\text{STS}}, \textit{24c})$, $(D_{Eth}, D_{\text{STS}}, \textit{3c7v})$, $(F_{Eth}, F_{\text{STS}}, \textit{3c7v})\}$.

The network has 6 physical links, as shown in figure 7.2 ($L = \{(A_{Eth}, B_{Eth}, 1)$, $(C_{Eth}, F_{Eth}, 1)$, $(B_{\text{STS}}, D_{\text{STS}}, 22)$, $(B_{\text{STS}}, E_{\text{STS}}, 87)$, $(D_{\text{STS}}, E_{\text{STS}}, 38)$, $(E_{\text{STS}}, F_{\text{STS}}, 29)$, $(E_{\text{STS}}, F_{\text{STS}}, 34)\}$).

As we have shown in section 3.3.1, the shortest correct path in this example is $A - B - E - D - B - E - F - C$. This shortest path uses the edge $B_{STS} - E_{STS}$ twice. Consequently, our path finding algorithm will have to take the (de)adaptation functions into account.

As we have stated in chapter 3, path finding in multi-layer networks is a path-constrained problem. One of the consequences is that **a segment of a shortest path does not have to be a shortest path in itself**. For example, a segment of the shortest path between A and C is $D - B - E - F$. However, the shortest path between D and F is $D - E - F$, also if adaptation and channel availability is taken into account.

7.3.2 Device-Based Network Description G_p

We define the graph $G_p = (\mathcal{V}_p, \mathcal{E}_p)$ (p for physical) as follows:

$$\begin{aligned} \mathcal{V}_p &= N \\ \mathcal{E}_p &= \{(n_1, n_2) \mid \big((n_1, y_1), (n_2, y_2), b\big) \in L\} \end{aligned} \tag{7.10}$$

This is a fairly common way to describe the physical properties of a network, with nodes represented as vertices, and (physical) links as edges ($\mathcal{E}_p = L$ for all practical purposes; the formal difference is that L has 3-tuples and is between channels ($c_1 = (n_1, y_1)$ and $c_2 = (n_2, y_2)$), while $\mathcal{E}_p$ has 2-tuples and is between nodes (n_1 and n_2).

If the network is bidirectional, then the graph can be bidirectional. If the network is not fully bidirectional, the graph must be undirected.

Observe that G_p may have multiple edges between the same pair of nodes, like the edge $E - F$ in the example. Some definitions of graphs do not allow this.

The bandwidth $B_e(e)$ of edge e is the bandwidth b of link $(n_1, n_2, b) \in L$

The information on (de)adaptation capabilities is not explicit in the graph defined by G_p, or in another format readable for regular path finding algorithms. Therefore, as we already saw in section 3.3.3, this graph is not very suitable for path finding.

In the next two sections, we present the graphs G_l and G_s which do contain (de)adaptation information.

The graph G_p encodes information on the nodes N and links L. For path finding, we would additionally need information about the channels C, adaptations A and labels LB.

7.3.3 Layer-Based Network Description G_l

Given the set $\mathcal{N}$ of network nodes, and the sets $Y(n)$ of layers for each node n, we construct the graph $G_l = (\mathcal{V}_l, \mathcal{E}_l)$ (l for layer) as follows:

$$\begin{aligned} \mathcal{V}_l &= C \\ \mathcal{E}_l &= \mathcal{E}_{lA} \cup \mathcal{E}_{lD} \cup \mathcal{E}_{lL} \\ \text{with } \mathcal{E}_{lA} &= \{(c_1, c_2) \mid (c_1, c_2, \alpha) \in A\} \text{ (adaptations)} \\ \mathcal{E}_{lD} &= \{(c_2, c_1) \mid (c_1, c_2, \alpha) \in A\} \text{ (de-adaptations)} \\ \text{and } \mathcal{E}_{lL} &= \{(c_1, c_2) \mid (c_1, c_2, b) \in L\} \text{ (links)} \end{aligned} \tag{7.11}$$

The set $\mathcal{V}_l$ consist of all (logical) channels $c = (n, y)$ for all devices $n \in \mathcal{N}$ and for all layers $y \in \mathcal{Y}$ that the node ‘has knowledge of’. The notation we use for vertices $v \in \mathcal{V}$ is n_y. For example, node B in our example network maps to two vertices B_{eth} and B_{STS}

The set $\mathcal{E}_l$ is the union of adaptations $\mathcal{E}_{lA}$, de-adaptations $\mathcal{E}_{lD}$ and (unidirectional) physical links $\mathcal{E}_{lL}$. Only physical links are represented as edges, not derived (logical) links at a higher layer.

For adaptations $(v_{client}, v_{server}) \in A$, the order is important. If $(v_{client}, v_{server}) \in \mathcal{E}_{lA}$ is an adaptation, then (v_{server}, v_{client}) is a de-adaptation rather than an adaptation. Since we want to be able to distinguish between adaptations and de-adaptations, the resulting graph must be directed, even if the network is fully bidirectional.

The adaptation $\mathcal{A}_e(e)$ of edge $e \in mathcalE_{lA}$ is the adaptation function α of $(c_1, c_2, \alpha) \in A$, which is by definition associated with $e \in mathcalE_{lA}$.

The bandwidth $B_e(e)$ of edge $e \in \mathcal{E}_{lL}$ is the bandwidth b of link $(v_1, v_2, b) \in \mathcal{E}_{lL}$. We assume that the adaptations have no bandwidth restrictions, and set $B_e(e)$ of edge $e \in \mathcal{E}_{lA} \cup \mathcal{E}_{lD}$ to ∞.

Note that the functions $\mathcal{A}_e(e)$ and $B_e(e)$ require that each edge has at least two parameters per edge: the bandwidth and the adaptation.

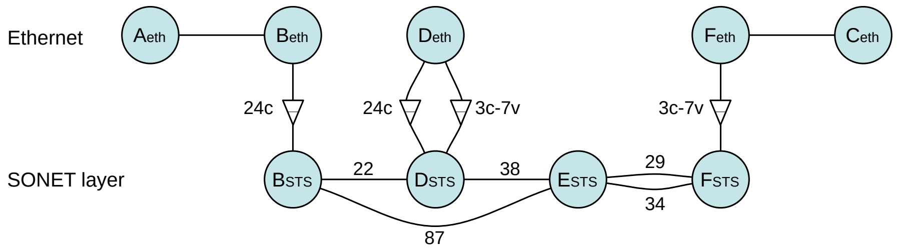


Figure 7.3: *The layer-based representation G_l of graph G in figure 7.2, with adaptation symbols signifying the direction of adaptation edges, rather than using directed edges.*

Figure 7.3 shows the graph G_l for the network described in figure 7.2. The vertices are vertically grouped by layer, and horizontally by node.

For good comparison with the other algorithms, we decided to draw the graph with single edges for each bi-directional network connection, instead of using two directed edges. We signify the direction of the adaptation by the triangles in the edges. This is the standard graphical representation of adaptation in ITU-T G.805 [s42], as we have seen in section 4.3.4.

The number of vertices is (using equation 7.5) $|\mathcal{V}_l| = |C| \leq |N| \times |\mathcal{Y}|$. In our example, $|N| \times |\mathcal{Y}| = 6 \times 2 = 12$, but C is only 9 vertices: nodes A and C are not aware of the STS layer, and node E has no knowledge of Ethernet.

The number of edges $|\mathcal{E}_l|$ is $|A \cup D \cup L'|$. However, since there is no overlap between A, D and L' (an adaptation is never a link or de-adaptation): $|\mathcal{E}_l| = |A| + |D| + |L'|$.

The network in our example is bidirectional, while the graph is directed. So $|L'| = 2|L|$, and $|A| = |D|$, and therefore $|\mathcal{E}_l| = |A| + |D| + |L'| = 2|A| + 2|L|$. For a good comparison with the other graphs G_p and G_s, this compares to $|A| + |L|$ undirected edges, as we can see from figure 7.3.

$$\begin{aligned} |\mathcal{V}_l| &\leq |N| \times |\mathcal{Y}| \\ |\mathcal{E}_l| &= |A| + |L| \qquad \text{for bidirectional edges} \end{aligned} \tag{7.12}$$

In our example network, L consists of 6 physical links, so L' contains 12 directed edges. Furthermore, this network contains four adaptations: nodes B and D support the *24c* adaptation, while nodes D and F support the *3c7v* adaptation. Since each adaptation results in 2 directed edges, this results in another 8 directed edges in $\mathcal{E}_l$.

The graph G_l encodes information on the nodes N, channels C, links L and adaptations A. For path finding, we would additionally need information about the labels LB.

7.3.4 Stack-based network description G_s

The last model, proposed by Kuipers and developed in collaboration with us, explicitly models the possible technology stacks. Our goal is to come to a, in the algorithmic sense, simple network description, which only consists of vertices and edges.

In our example network we can identify three different technology stacks: Ethernet, Ethernet over 24 STS channels, and Ethernet over 21 STS channels:

$$\begin{aligned}\mathcal{S} = \{&[(\text{Ethernet}, \textsc{nil}, \epsilon)],\\ &[(\text{Ethernet}, \textsc{nil}, \epsilon), (\text{STS}, \mathit{24c}, \epsilon)],\\ &[(\text{Ethernet}, \textsc{nil}, \epsilon), (\text{STS}, \mathit{3c7vc}, \epsilon)]\}\end{aligned}$$

Again, NIL means there is no higher layer, and ϵ means the empty label.

We first define the graph $G_s = (\mathcal{V}_s, \mathcal{E}_s)$ (s for stack). The set of vertices $\mathcal{V}_s$ of graph G_s is defined as follows:

$$\begin{aligned}\mathcal{V}_s =&\{(n,s) \mid n \in N \wedge s \in \mathcal{S} \wedge \\ &\big(\exists c \in C_n(n) : Y_s(s) = Y_c(c) \wedge \mathcal{LB}_s(s) \in LB(c)\big)\}\end{aligned} \tag{7.13}$$

Note that:

$$\mathcal{V}_s \subset N \times \mathcal{S} \tag{7.14}$$

Due to the definition of $\mathcal{V}_s$, we can associate one of more channels $C_v(v) \subset C$ with each vertex $v = (n, s) \in \mathcal{V}_s$:

$$C_v(v) = C_v\big((n,s)\big) = \{c \mid c \in C_n(n) \wedge Y_s(s) = Y_c(c) \wedge \mathcal{LB}_s(s) \in LB(c)\} \tag{7.15}$$

The definition in equation 7.13 states that each vertex $v \in V_s$ is defined as a tuple (n, s) with $n \in N$ and $s \in S$. And thus, each vertex $v \in V_s$ has exactly one tuple (n, y) with $n \in N$ and $y = Y_s(s) \in \mathcal{Y}$. If we apply the restriction of equation 7.3, there may be only one channel per (node,layer) tuple, which means that the size of $C_s(v)$ in equation 7.15 is at most one.

Since the definition of vertices $v \in V_s$ requires that there is at least one such channel, we must conclude that, with this restriction, the length $|C_v(v)| = 1$ for every $v \in V_s$.

Using the definition of $C_v(v)$ we can now define the edges $\mathcal{E}_s$ of graph G_s:

$$\begin{aligned}
\mathcal{E}_s =& \mathcal{E}_{sL} \cup \mathcal{E}_{sA} \\
\text{with } \mathcal{E}_{sL} =& \{\big((n_1, s_1), (n_2, s_2)\big) \mid v_1, v_2 = (n_1, s_1), (n_2, s_2) \in \mathcal{V}_s \wedge \\
& \big(\exists (c_1, c_2, b) \in L : c_1 \in C_v(v_1) \wedge c_2 \in C_v(v_2) \wedge \\
& B_a(\mathcal{A}_s(s_1)) \geq b\big)\} \qquad (7.16) \\
\text{and } \mathcal{E}_{sA} =& \{\big((n_1, s_1), (n_2, s_2)\big) \mid v_1, v_2 = (n_1, s_1), (n_2, s_2) \in \mathcal{V}_s \wedge \\
& \big(\exists (c_1, c_2, \alpha) \in A : c_1 \in C_v(v_1) \wedge c_2 \in C_v(v_2) \wedge \\
& \mathcal{A}_s(s_2) = \alpha \wedge \mathcal{LB}_s(s_2) \in LB(c_2)\big)\}
\end{aligned}$$

$\mathcal{E}_{sL}$ is the set of edges representing physical links and $\mathcal{E}_{sA}$ is the set of edges representing adaptations. Both definitions state that an edge exists if a corresponding link or adaptation function exists. In addition, only physical links with enough bandwidth are present, and only adaptations with the correct adaptations function and matching label are present in the graph.

The bandwidth $B_e(e)$ of edge $e \in \mathcal{E}_{sL}$ is the sum of the bandwidths b of the associated links $(c_1, c_2, b) \in L$ (plural, since multiple links may be mapped onto the same edge e). The bandwidth $B_e(e)$ of each edge $e \in \mathcal{E}_{sA}$ is ∞.

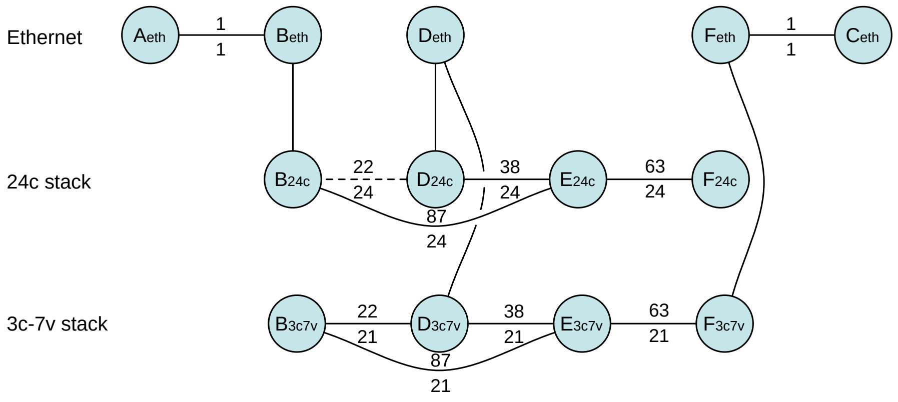


Figure 7.4: *Representation of the network in figure 7.2 as a multi-layered graph.*

Figure 7.4 shows the graph G_s for the network in figure 7.2. The vertices are grouped according in a matrix with each node n in a column, and a row for each of the three technology stacks (Ethernet, Ethernet over 24 STS channels, and Ethernet over 21 STS channels). Such grouping has also been deployed in the context of wavelength routing in WDM networks [p8].

Node B can only adapt Ethernet in 24 channels (and de-adapt back), while node F can only adapt Ethernet into 21 channels. Thus, there is an edge between B_{eth} and B_{24c}, but not between B_{eth} and B_{3c7v}. This is a direct result of the condition $\mathcal{A}_s(s_2) = \alpha$ in the definition of $\mathcal{E}_{sA}$.

When comparing this graph with figure 7.3, it is clear that the row for each stack s in figure 7.4 corresponds to the row for the associated layer $Y_s(s)$ in figure 7.3. The only difference is that in G_s, there is only one edge per pair of vertices. This is because the definition for $\mathcal{E}_{sL}$, which states that an edge exists if a corresponding link exists, but not that there must be an edge for *every* corresponding link. This is an optimisation to reduce the size of G_s for path finding. While a shortest path may contain two links between two nodes can be used twice, it will never be with the same encoding ('loops' in a shortest path are only present if a remote host performs an unavoidable conversion between two encodings).

The dotted line between B_{24c} and D_{24c} states that in theory these nodes should be able to communicate with each other, but in this case not enough $(22 < 24)$ channels are available, and due to the link constraint $B_a(\mathcal{A}_s(s_1)) \geq b$ in the definition of $\mathcal{E}_{sL}$, the actual edge is disregarded in the graph G_s.

The number above edges in represent the available bandwidth for the associated links. The number beneath each edge represent the required bandwidth as determined by the adaptation function. In this case of multiple links between two nodes, the edge weight is the sum of all link capacities.

The set of technology stack $\mathcal{S}$ is *not a local property* of the nodes, but depends on the choice of the (root technology of the) end-nodes. For example, node E has only knowledge of the STS layer, and for path finding between D and F, the only encoding would be $[(\text{STS}, \text{NIL}, \epsilon)]$. However, for path finding between A and C, the possible encodings are $[(\text{Ethernet}, \text{NIL}, \epsilon), (\text{STS}, \mathit{24c}, \epsilon)]$ and $[(\text{Ethernet}, \text{NIL}, \epsilon), (\text{STS}, \mathit{3c7vc}, \epsilon)]$. Because $\mathcal{S}$ depends on the choice of the root technology, the graph G_s is different if the layer of the end-nodes is different.

In order to give an estimate of the number of vertices and edges in G_s, we first prove that there is only one channel associated with each vertex $v \in V_s$, and then proceed to give an estimate of the size $|\mathcal{S}|$ of $\mathcal{S}$. We need these numbers to estimate the running time of the algorithms.

The number of possible stacks $|\mathcal{S}|$ highly depends on how the technology diagram looks like. We define $\mathcal{S}_y \subset \mathcal{S}$ to be the set of all stacks with lowest layer y:

$$\mathcal{S}_y = \{s \mid s \in \mathcal{S} \wedge Y_s(s) = y\} \tag{7.17}$$

Furthermore, we define the set of client layers for a given (server) layer y_s as:

$$Y_{client}(y_s) = \{y_c \mid y_c \in \mathcal{Y} \wedge \exists (y_c, y_s) \in \mathcal{A}(y_s)\} \tag{7.18}$$

Given the recursive definition of stacks in equation 7.9b, we can see that a stack s_s with lowest layer $Y_s(s)$ can only be formed by appending an adaptation and label to an existing stack s_c, with $Y_s(s_c) \in Y_{client}(Y_s(s_c))$ (for all non-root stacks):

$$\begin{aligned}
\mathcal{S}_y &= \{s \mid s \in \mathcal{S} \wedge Y_s(s) = y\} \\
&= \{s_c + [(y, (y_c, y), lb)] \mid (y, (y_c, y), lb) \in \mathcal{T} \wedge s_c \in \mathcal{S} \wedge y_c = Y_s(s)\} \\
&= \{s_c + [(y, (y_c, y), lb)] \mid (y, (y_c, y), lb) \in \mathcal{T} \wedge s_c \in \mathcal{S}_{y_c}\} \\
&= \{s_c + [(y, (y_c, y), lb)] \mid (y, (y_c, y), lb) \in \mathcal{T}(y) \wedge s_c \in \mathcal{S}_{y_c}\} \\
&= \{s_c + [(y, (y_c, y), lb)] \mid lb \in \mathcal{LB}(y) \wedge (y_c, y) \in \mathcal{A}(y) \wedge s_c \in \mathcal{S}_{y_c}\} \\
&= \{s_c + [(y, \alpha, lb)] \mid lb \in \mathcal{LB}(y) \wedge \alpha \in \mathcal{A}(y) \wedge y_c \in Y_{client}(y) \wedge s_c \in \mathcal{S}_{y_c}\}
\end{aligned} \tag{7.19}$$

Using equations 7.19 and 7.8, we can express the size of $\mathcal{S}_y$, in the size of its client stacks:

$$|\mathcal{S}_y| = \sum_{y_c \in Y_{client}(y)} |\mathcal{S}_{y_c}| \times |\mathcal{LB}(y)| \times |\mathcal{A}(y)| = \sum_{y_c \in Y_{client}(y)} |\mathcal{S}_{y_c}| \times |\mathcal{T}(y)| \tag{7.20}$$

In order to make further estimates of $|\mathcal{S}|$, we will now assume that the protocol stack is an ordered set of layers, ordered from 'highest' to 'lowest' layer, without any branches. There must be at least one adaptation between consecutive layers in the list, and there may be only one root technology (thus the root layer has only one label). We denote this list of layers $\mathcal{Y} = [y_1, y_2, \ldots, y_Y]$ (with $Y = |\mathcal{Y}|$), and denote $\mathcal{Y}_i$ the list of only the first $1 \geq i \geq |\mathcal{Y}|$ layers: $[y_1, y_2, \ldots, y_j]$. With this constraint, each (non-root) layer has only one client layer: $Y_{client}(y_i) = \{y_{i-1}\}$, and thus equation 7.20 reduces to

$$|\mathcal{S}_{y_i}| = |\mathcal{S}_{y_{i-1}}| \times |\mathcal{T}(y_i)| \tag{7.21}$$

Now we can easily calculate the number of different technology stack for layer y_j:

$$\begin{aligned}|\mathcal{S}_{y_i}| &= |\mathcal{S}_{y_{i-1}}| \times |T(y_i)| \\ &= |\mathcal{S}_{y_{i-2}}| \times |T(y_{i-1})| \times |T(y_i)| = \dots \\ &= |\mathcal{S}_{y_1}| \times \prod_{1\leq j\leq i} |T(y_j)| \\ &= \prod_{1\leq j\leq i} |T(y_j)|\end{aligned} \tag{7.22}$$

The last step uses the fact that $|\mathcal{S}_{y_1}| = 1$, since there is only one technology at the highest layer.

Since both $|\mathcal{A}(y_i)| \geq 1$ and $|\mathcal{LB}(y_i)| \geq 1$, also $|T(y_i)| \geq 1$:

$$|\mathcal{S}_{y_i}| = |\mathcal{S}_{y_{i-1}}| \times |T(y_i)| \geq |\mathcal{S}_{y_{i-1}}| \tag{7.23}$$

The number of all technology stacks for an ordered list of layers is:

$$|\mathcal{S}| = \sum_{y_i\in\mathcal{Y}} |\mathcal{S}_{y_i}| = \sum_{y_i\in\mathcal{Y}} \prod_{1\leq j\leq i} |T(y_j)| \tag{7.24}$$

Equation 7.23 gives an upper limit for each $|\mathcal{S}_{y_i}|$, and thus an upper limit on $|\mathcal{S}|$

$$|\mathcal{S}| = \sum_{y_i\in\mathcal{Y}} \prod_{1\leq j\leq i} |T(y_j)| \leq |\mathcal{Y}| \times \prod_{y\in\mathcal{Y}} |T(y)| \tag{7.25}$$

The estimate for $|\mathcal{S}|$ is considerable lower, as $|\mathcal{S}| \approx |\mathcal{S}_{y_Y}|$, provided that $|T(y_Y)| >> 1$, with $y_Y = y_{|\mathcal{Y}|}$ the ‘lowest’ network layer.

$$|\mathcal{S}_{y_i}| \leq |\mathcal{S}_{y_Y}| \text{ for all } y_i \in \mathcal{Y} \tag{7.26}$$

$$|\mathcal{S}| \gtrapprox |\mathcal{S}_{y_Y}| = \prod_{1\leq j\leq |\mathcal{Y}|} |T(y_j)| = \prod_{y\in\mathcal{Y}} |T(y)| \tag{7.27}$$

The upper limit of the number of vertices $|\mathcal{V}_s|$ (using equation 7.14) is:

$$|\mathcal{V}_s| \leq |N| \times |\mathcal{S}| \tag{7.28}$$

If most of the nodes can carry all technology stacks (i.e. $\forall n \in N : |C_n(n)| \approx |\mathcal{Y}|$), this upper limited becomes an estimate: $|\mathcal{V}_s| \lessapprox |N| \times |\mathcal{S}|$. The lower estimate of equation 7.27 and upper estimate of equation 7.28 roughly cancel each other out:

$$|\mathcal{V}_s| \approx |N| \times |\mathcal{S}| \approx |N| \times \prod_{y\in\mathcal{Y}} |T(y)| \tag{7.29}$$

Equation 7.29 results in an estimate of $|\mathcal{V}_s| \approx |N| \times \prod_{y\in\mathcal{Y}} |T(y)| = |N| \times |T(\text{Ethernet})| \times |T(\text{STS})| = 6 = 12$ vertices, while G_s has in fact 13 vertices.

The number of edges in G_s is $|\mathcal{E}_s| = |\mathcal{E}_{sL}| + |\mathcal{E}_{sA}|$. Physical links $l = (c_1, c_2, b) \in L$ map to zero or more edges $(v_1, v_2) \in \mathcal{E}_{sL}$. We define the layer of link $(c_1, c_2, b) \in L$ as $Y_l\big((c_1, c_2, b)\big) = Y_c(c_1) = Y_c(c_2)$ (links must have terminating channels at the same layer).

Given a channel $(n, y) \in C$, there are exactly $|\mathcal{S}_y|$ stacks s with $Y_s(s) = y$, by definition of equation 7.17. By definition of $C_n(n)$ (equation 7.1), the channel $(n, y) \in C_n(n)$. Also, for every tuple (n, s) with $s \in \mathcal{S}_y$, $Y_s(s) = y$. Therefore, by definition of $\mathcal{V}_s$ (equation 7.13), every tuple (n, s) with $s \in \mathcal{S}_y$ is a valid vertex with associated channel $(n, y) \in C_n(n)$ if the label of stack s is a valid label of channel c. Thus, for every channel $c = (n, y) \in C$, there are at most $|\mathcal{S}_y|$ vertices in $\mathcal{V}_s$:

$$\forall c \in C: \ |\{v \mid v \in \mathcal{V}_s \wedge c \in C_v(v)\}| \leq |\mathcal{S}_{Y_c(c)}| \tag{7.30}$$

Consequently, for every link $(c_1, c_2, b) \in L$ there are also at most $|\mathcal{S}_y|$ edges in $\mathcal{E}_{sL}$, with $y = Y_l\big((c_1, c_2)\big)$. Given that two physical links between the same pair of nodes result in the same set of edges, we must only count unique links:

$$|\mathcal{E}_{sL}| \leq \sum_{\text{unique link } l\in L} |\mathcal{S}_{Y_l(l)}| \leq \sum_{l\in L} |\mathcal{S}_{Y_l(l)}| \tag{7.31}$$

Given that we require that an adaptation exists between all subsequent layers, the set $\mathcal{S}_y$ contains at least one stack for every layer. This gives us the lower limit $|\mathcal{E}_{sL}| \geq |\text{unique links in } L|$.

If all physical links are at the lowest layer y_Y, and there are no links between the same channels (since that would result in an equal set of edges), then the number of edges $|\mathcal{E}_{sL}|$ is equal to the upper limit. This is a reasonable estimate:

$$|\mathcal{E}_{sL}| \lessapprox |L| \times |\mathcal{S}_{y_Y}| = |L| \times \prod_{y\in\mathcal{Y}} |T(y)| \tag{7.32}$$

The estimate for $|\mathcal{E}_{sA}|$ is similar to that of $|\mathcal{E}_{sL}|$, although the exact quantification is more complex due to the added label restriction (not every vertex pair as defined in equation 7.30 should have an adaptation). Nevertheless, each adaptation $a \in A$ results in 1 to $|\mathcal{S}_{y_Y}|$ edges in $\mathcal{E}_{sA}$:

$$|A| \leq |\mathcal{E}_{sA}| \leq |A| \times |\mathcal{S}_{y_Y|}| = |A| \times \prod_{y\in\mathcal{Y}} |T(y)| \tag{7.33}$$

This gives us an estimate of $|\mathcal{E}_s|$:

$$|A| + |\text{unique links in } L| \leq |\mathcal{E}_s| \leq (|A| + |L|) \times \prod_{y \in \mathcal{Y}} |T(y)| \tag{7.34}$$

Equation 7.34 results in an lower limit of $|\mathcal{E}_s| \geq |A| + |\text{unique links in } L| = 4 + 5 = 9$ edges, and a high estimate of $|\mathcal{E}_s| \leq (|A| + |L|) \times \prod_{y \in \mathcal{Y}} |T(y)| = (|A| + |L|) \times |T(\text{Ethernet})| \times |T(\text{STS})| = (4+6) \times 1 \times 2 = 20$ edges. In reality, G_s has 13 edges after removal of one edge due to bandwidth constraints (4 edges representing adaptations, and 9 representing links).

The graph G_s encodes information on the nodes N, channels C, links L, adaptations A and labels LB. In addition, edge labels can be used for available and required bandwidths. For multi-layer path finding, this is all the information we need.

7.4 Path Selection in G_l

Given a layer-based graph $G_l = (\mathcal{V}_l, \mathcal{E}_l)$ and a source vertex v_{src} and a destination vertex v_{dst} in $\mathcal{V}_l$, our objective is to find the path P^* for which $W(P^*) \leq W(P)$ for all feasible paths P between v_{src} and v_{dst}. As discussed in the previous section 7.3.3, we may pass through vertices and even edges multiple times, and since G_l maps links 1:1 to edges, we can not simply use a link-constrained algorithm (see our claim in section 3.3.2).

Algorithm 7.1 presents MULTI-LAYER-BREADTH-FIRST(G_l, v_{src}, v_{dst}) to compute the shortest path P^* from v_{src} to v_{dst} in G_l. We first explain the general properties of the algorithm, than explain it line by line.

This algorithm has two main features: (1) we keep track of the stack of (de)adaptations along the path and (2) multiple paths may be stored at a single vertex (similarly to a k-shortest paths algorithm). So instead of working with a queue of vertices, as in the breadth-first-search algorithm and Dijkstra's algorithm [p9, p11], the algorithm keeps a queue of paths. Basically, this algorithm simply tries all possible paths P ordered by length, even those with loops or already visited vertices.

The base of the algorithm is a queue Q of path objects. The algorithm stores the following properties for each path object $p \in Q$:

1. Sequence of edges $\mathcal{E}[p]$;
2. The last visited vertex $V_p[p]$
3. Current technology stack $S_p[p]$;

Algorithm 7.1 MULTI-LAYER-BREADTH-FIRST(G_l, v_{src}, v_{dst})

```
1: $p \leftarrow$ new path with $\mathcal{E}[p] = \varnothing; V_p[p] = v_{src}; S_p[p] = \varnothing; B[p] = 1; R[p] = \varnothing; B[p,e] = \varnothing; d[p] = 0$
2: ENQUEUE($Q, p$) {Queue $Q$ consists of paths}
3: while $Q \neq \varnothing$ do
4:   $p \leftarrow$ DEQUEUE($Q$) {Replace with EXTRACT-MIN for the shortest path}
5:   $v \leftarrow V_p[p]$ {The last vertex in the path}
6:   if $v = v_{dst}$ and $\mathcal{A}_s(S_p[p]) = \varnothing$ then
7:     return $p$ {Reached destination}
8:   else
9:     for all $v \in \text{adj}[u]$ do
10:      $p_{new} \leftarrow$ EXTEND-PATH($p, (u,v), v$)
11:      if $p_{new} \neq$ unfeasible then
12:        ENQUEUE($Q, p_{new}$)
```

4. The available labels $LB(S_p[p])$ (this extension is discussed in section 7.6);
5. The list of used bandwidths $B[p,e]$ for every edge $e \in \mathcal{E}[p]$;
6. The set of visited (vertex, stack) tuples $R[p]$;
7. Distance $d[p]$.

Only the $\mathcal{E}[p]$ property is required, and all other properties can be deduced from $\mathcal{E}[p]$ (except $V_p[p]$ if $\mathcal{E}[p] = \varnothing$), but they are present for speed.

Lines 1-2 initialise the algorithm with a starting path starting at vertex v_{src}, and without an adaptation in the technology stack. Lines 3-12 form the main loop. It takes the path with the shortest length from the queue, and extends it by one hop in all directions. Lines 6-7 checks if the shortest path ends at the destination, and if so, returns that path as the result of the algorithm. Not all extensions of a path with one hop will result in a feasible path. Line 11 checks for this condition, and only considers feasible paths.

The actual extension of the path and the feasibility check is done in the Extend-Path routine in algorithm 7.2. This algorithm takes the existing path p and extends it via edge e to vertex v. It sets the six properties of the path accordingly. In case of an adaptation (lines 5-6), the new adaptation is added to the stack. In case of de-adaptation, the last adaptation is removed from the stack (lines 7-10), but only if the de-adaptation is equal to the last adaptation on the stack. If it is not, it is an unfeasible path. Lines 14-16 are an extension to the base algorithm to check if two adjacent vertices have a common channel label available for transmission of data.Lines 21-24 check if the vertex has been

Algorithm 7.2 Extend-Path(p, e, v)

```
Require: $p$ is a path, $n$ an adjacent vertex, $e$ the connecting edge
Require: The sets of edges $\mathcal{E}_{lL}$, $\mathcal{E}_{lA}$, and $\mathcal{E}_{lD}$
Require: Adaptation $\mathcal{A}_e$ for each edge $e \in \mathcal{E}_{lA} \cup \mathcal{E}_{lD}$
Require: The weight of all edges $W_e(e)$
Require: Set of possible labels per vertex $LB(v)$
 1: $p_{new} \leftarrow p$
 2: $\mathcal{E}[p_{new}] \leftarrow \mathcal{E}[p] + e$ {Extend $p_{new}$ with edge $e$}
 3: $V_p[p_{new}] \leftarrow v$
 4: $d[p_{new}] \leftarrow d[p] + W_e(e)$ {Length of path increases. If $W_e$ is always 1, the
    queue $Q$ remains sorted}
 5: if $e \in \mathcal{E}_{lA}$ then {Adaptation}
 6:    $S_p(p_{new}) \leftarrow S_p[p] + t$ with $t = (y, \alpha, Lb)$; $y = Y_c(v)$; $\alpha = \mathcal{A}_e(e)$; $Lb \in$
       $LB(v)$ {The choice of $Lb$ is ambiguous if $|LB(v)| > 1$}
 7: else if $e \in \mathcal{E}_{lD}$ then {De-adaptation}
 8:    if $\mathcal{A}_s(S_p(p_{new})) \neq \mathcal{A}_e(e)$ then {Wrong de-adaptation}
 9:       return unfeasible
10:    else
11:       Pop-Element$(S_p(p_{new}))$ {Remove last adaptation from the stack}
12: else {$e \in \mathcal{E}_{lL}$, Link}
13:    $S_p(p_{new}) \leftarrow S_p[p]$
14:    $LB(S_p(p_{new})) \leftarrow LB(S_p(p_{new})) \cap LB(v)$
15:    if $LB(S_p(p_{new})) = \varnothing$ then {No compatible labels left}
16:       return unfeasible
17: $b \leftarrow$ Bandwidth-Required$(S_p(p_{new}))$ {Determined by adaptation}
18: $B[p_{new}, e] \leftarrow B[p, e] - b$
19: if $B[p_{new}, e] < 0$ then {No bandwidth available}
20:    return unfeasible
21: if $(\mathcal{V}_p(p_{new}), S_p(_{new})) \in R[p]$ then {Node $\mathcal{V}_p(p_{new})$ has been visited before
    with the same stack}
22:    return unfeasible
23: else
24:    $R[p_{new}] \leftarrow R[p] \bigcup(\mathcal{V}_p(p_{new}), S_p(p_{new}))$
25: return $p_{new}$
```

Algorithm 7.3 Bandwidth-Required(s)

```
Require: $s$ is an adaptation stack
1: if $\mathcal{A}_s(s) \neq$ NIL then
2:     $(y_c, y_s, b_s) \leftarrow \mathcal{A}_s(s)$ {$b_s$ determined by the adaptation function}
3:     $b \leftarrow b_s$
4: else
5:     $b \leftarrow 1$
6: return $b$
```

covered earlier in the current path with the same stack, and if so, skip this extension. This reduces the flooding nature of this algorithm.

The running time of a breadth first search algorithm is $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$ [p9]. However, this assumes that each vertex is processed only once. The running time for this algorithm is considerably longer since each vertex can be covered multiple times. In fact, in the worst-case scenario, the length of the queue Q can grow exponentially with the average out-degree of the vertices.

Kuipers showed that the multi-layer path finding problem is NP-complete [a12]. In contrast, path finding in a single layer network can be solved in polynomial time (e.g. using Dijkstra's algorithm or a breadth first search algorithm). This is in accordance with an exponential running time.

The running time of algorithm 7.1 is (see appendix A.4):

$$\begin{aligned}\mathcal{O}(\text{Algorithm 7.1}) &= \mathcal{O}(|Q|) \times \mathcal{O}(\text{loop}) = \\ &= \mathcal{O}(|Q|) \times (\mathcal{O}(\text{DEQUEUE}) + \mathcal{O}(|adj|) \times \mathcal{O}(\text{EXTEND-PATH}))\end{aligned} \tag{7.35}$$

With $\mathcal{O}$(DEQUEUE) caused by line 4, $\mathcal{O}(|adj|)$ by line 9, and $\mathcal{O}$(Extend-Path) by line 10.

As we have seen in section 7.3.1, a segment of a shortest path may not be a shortest path in itself. In order to limit the flooding nature of this algorithm even further, it is possible to make this assumption. This makes the algorithm faster at the cost that it sometimes will return a false negative (no path is found, even though it exists). A proper replacement of this check would verify if the vertex with that stack is already present in any path, instead of in path p only. This check be accomplished by making $R[p]$ a global variable, rather than tied to a particular path p.

If we would restrict the search space of the algorithm using the above modification, the running time would reduce to (see equation A.15)

$$\begin{aligned}\mathcal{O}(\text{Algorithm 7.1}) &= \mathcal{O}(|N|) \times |\mathcal{Y}| \times T^{|\mathcal{Y}|}) \times \mathcal{O}(\text{DEQUEUE})+\\ &\quad \mathcal{O}((|N| \times |\mathcal{Y}| + |L|) \times T^{|\mathcal{Y}|}) \times \mathcal{O}(\text{EXTEND-PATH})\\ &\approx \mathcal{O}((|N| \times |\mathcal{Y}| + |L|) \times T^{|\mathcal{Y}|})\times\\ &\quad \mathcal{O}\big(\big(\log(|N|) + \log(|\mathcal{Y}|) + |\mathcal{Y}| \cdot \log(T)\big) + \log(\langle lb\rangle)\big)\end{aligned} \tag{7.36}$$

The estimated average running time is (see equation A.16):

$$\mathcal{O}(\text{Algorithm 7.1}) \approx \mathcal{O}((|N| + \frac{|L|}{|\mathcal{Y}|}) \times T^{|\mathcal{Y}|} \times \big(\log(|N| \times T^{|\mathcal{Y}|}) + \log(\langle lb\rangle)\big)) \tag{7.37}$$

7.5 Path Selection in G_s

If we would apply the Dijkstra algorithm [p11] to the graph G_s in figure 7.4, we would find the path $A_{Eth} - B_{Eth} - B_{24c} - E_{24c} - D_{24c} - D_{Eth} - D_{3c7v} - E_{3c7v} - F_{3c7v} - F_{Eth} - C_{Eth}$, which relates to path $A - B - E - D - E - F - C$. If the link between D and E would have enough capacity, this would result in a feasible path, which is not the case here. The correct path is $A - B - E - D - B - E - F - C$. Consequently (contrary to the approach in [p8]), we have to modify our algorithm to account for capacity on links traversed multiple times.

Our goal is to come up with a simple path finding algorithm. The graph G_s seemed a prime candidate for this purpose, given that a shortest path in the graph G_s will never contain the same vertex twice. This can easily be proven. Assume a path $P_{src-dest}$ from *src* to *dest* contains the vertex v twice. Then we can split path $P_{src-dest}$ in three segments, P_{src-v} from *src* to v, P_{v-v} from v to v, and P_{v-dest} from v to *dest*. Given that the path length d is additive, then $d(P_{src-dest}) = d(P_{src-v}) + d(P_{v-v}) + d(P_{v-dest})$. The path $P'_{src-dest} = P_{src-v} + P_{v-dest}$ has then length $d(P'_{src-dest}) = d(P_{src-v}) + d(P_{v-dest}) \leq d(P_{src-dest})$ (provided that $d(P_{v-v}) \geq 0$). $P'_{src-dest}$ is a valid path, according to the definition in section 4.4.5 if the technology vertex v is the same each encounter. This is true for a vertex $v = (n, s) \in G_s$, as the technology is uniquely defined by the adaptations stack s. This may not be true for vertices in G_p or in G_l as the adaptation stack is not uniquely defined for each vertex.

Given that a shortest path in the graph G_s will never contain the same vertex twice, our first approach was to create a variant of the Dijkstra algorithm (see appendix A). However, it turned out that such variant implicitly assumes that a segment of a shortest path is also a shortest path, while we saw in section 7.3.1 that this assumption is not true for multilayer networks.

Algorithm 7.4 MULTI-LAYER-K-SHORTEST-PATH(G_s, v_{src}, v_{dst})

```
1:  $r[v] \leftarrow$ DIJKSTRA($G_s, v_{dst}$) {Lower bounds for all nodes}
2:  for all vertices $v \in \mathcal{V}_s$ do
3:      $counter[v] \leftarrow 0$
4:  $maxlength \leftarrow \infty$
5:  $counter[v_{src}] \leftarrow counter[v_{src}] + 1$
6:  $\pi[v, counter[v_{src}]] \leftarrow$ NIL, NIL {Source $v_{src}$ has no predecessor vertex and index}
7:  ENQUEUE($Q, v_{src}, counter[v_{src}], 0, r[v_{src}]$) {Queue $Q$ consists of vertex, path index, path length so far, and lower bound of the total path length}
8:  while $Q \neq \varnothing$ do
9:      $u, i, d \leftarrow$ EXTRACT-MIN($Q$){Extract path with lowest lower bound of the total path length}
10:     if $u = v_{dst}$ then
11:         return path {Created by backtracing $\pi[v, i]$, starting with $v_{dst}, i$}
12:     else
13:         for all $v \in \text{adj}[u]$ do {for each neighbour of $u$}
14:             if $d + w(u,v) + r[v] < maxlength$ then {Skip paths that exceed known maximum length}
15:                 if FEASIBILE($G_s, u, i, v$) then {Backtracking}
16:                     $d' \leftarrow d + w(u,v)$ {Distance is sum of weights}
17:                     $counter[v] \leftarrow counter[v] + 1$
18:                     $\pi[v, counter[v]] \leftarrow u, i$
19:                     ENQUEUE($Q, v, counter[v], d', d' + r[v]$)
20:                     if $v = v_{dst}$ and $d' + r[v] < maxlength$ then
21:                         $maxlength \leftarrow d' + r[v]$
```

The algorithm MULTI-LAYER-K-SHORTEST-PATH(G_s, v_{src}, v_{dst}), which is presented in listing 7.4 is a k-shortest path algorithm, and does not make the assumption that a segment of a shortest path is also a shortest path. It employs brute force to always return an exact answer, just as the algorithm for path finding in G_l which is discussed in the previous section.

Our objective is to find a shortest path from v_{src} to v_{dst} in G_s. The MULTI-

Algorithm 7.5 Feasible(G_s, u, i, v)

```
Require: Available bandwidth $B_e(e)$ for the edge $e = (u, v)$
Require: Required bandwidth for the vertex $v = (n, s)$, based on its stack $s$
Ensure: {Backtracking to check whether the path is loop-free and has enough
    capacity available. $u, i$ is our current node, $v$ is the node under considera-
    tion for extending our path with.}
 1: $t \leftarrow u$
 2: $j \leftarrow i$
 3: $B' \leftarrow B_e\big((u, v)\big)$
 4: while $\pi[t, j] \neq$ NIL,NIL do
 5:   $t', j' \leftarrow \pi[t, j]$ {Previous hop}
 6:   if $C_v(t') = C_v(u) \wedge C_v(t) = C_v(v)$ then {$(u, v)$ and $(t', t)$ are the same
      link}
 7:     $(n, s) \leftarrow t$ {Stack $s$ determined by the vertex $t$}
 8:     $B' \leftarrow B' -$ Bandwidth-Required($s$) {See algorithm 7.3}
 9:   $t, j \leftarrow \pi[t, j]$ {Trace back}
10:   if $t = v$ then
11:     return FALSE {loop}
12: $(n, s) \leftarrow v$ {Stack $s$ determined by the vertex $v$}
13: if $B' <$ Bandwidth-Required($s$) then
14:   return FALSE {no bandwidth available}
15: else
16:   return TRUE
```

Layer-k-Shortest-Path algorithm is based on SAMCRA [p19, p26], and the Multi-Layer-Breadth-First algorithm in listing 7.1. The two main variables are a queue Q of paths, and a list of previous hops, stored in matrix π. Multiple paths can be stored at a vertex u, u, i is used to denote the i-th path at node u. For example $\pi[E_{3c7v}, 1]$ may refer to $D_{3c7v}, 1$ while $\pi[E_{3c7v}, 2]$ may refer to $B_{3c7v}, 1$. By backtracking the predecessor list $\vec{\pi}$, an entire path can be reconstructed, starting at the last hop. *counter*$[v]$ refers to the number of paths stored at node v.

The queue Q keeps track of a path using the tuple (u, i), and in addition keeps track of the length of the path. A feature taken from SAMCRA is to use two search-space reduction techniques. First, the *maxlength* variable keeps track of the maximum length of the end-to-end path. Second, the Dijkstra algorithm is used to give a lower bound of the path length. While both techniques increase the running time in the worst-case scenario, they significantly optimise the order in which all possibilities are searched, eliminating possibilities that will never lead to a shortest path.

Lines 1-7 of the meta-code initialises all nodes. Line 1 computes the shortest paths from v_{dst} to all other nodes in the graph by one execution of the Dijkstra shortest paths algorithm that disregards any bandwidth constraints. The weights of these paths serve as lower bound estimates, referred to as $r[v]$ for all nodes $v \in \mathcal{V}_s$. Note that if the shortest path from v_{src} to v_{dst} is feasible, we can stop the algorithm and return this solution. Else the algorithm should proceed. Line 7 inserts the source node with predicted length $d[v_{src}] + r[v_{src}] = r[v_{src}]$ and predecessor NIL in the queue Q, which was initially empty. The main algorithm starts at line 8. Line 9 extracts the node u and index i from the queue that has the predicted length. Hop u, i can be regarded as the new scanning node towards destination v_{dst}. If $u = t$ we have found the shortest feasible path, else we continue. Lines 13 to 19 perform the relaxation procedure [p9] for each adjacent node v of u. Line 14 checks whether the length of the path extended from u to v does not exceed *maxlength*, otherwise it is discarded because we already have a better candidate. If this first test is passed, the module Feasible is called to check for loops (each vertex in G_s should be only used once in a shortest path) and for enough available bandwidth on the link (u, v). If the result is a feasible path, we insert v into the queue.

The Feasible algorithm is described in listing 7.5. $B_e(u, v)$ contains the available bandwidth on the edge (u, v). $b(u, v)$ gives the remaining available bandwidth on the edge (u, v), which is initialised as $B_e(e)$ in line 3 and decreases in the course of the algorithm (lines 6-8). Lines 4-11 form loop through all edges in the path, by backtracing $\vec{\pi}$ (line 9). If the path contains the same vertex twice (line 10-11) or not enough bandwidth is left on edge (u, v) (line 13-14), the path is deemed unfeasible.

7.6 Extension to Multiple Labels

So far, we mostly looked at two layers, with the adaptations between the two layers as the only technological incompatibility. All algorithms can easily be extended to support multiple layers, and for the most part we already put those extensions in the definitions. In this section we will illustrate this with the example network in figure 7.5.

The example network of figure 7.5 consists of 7 nodes, $N = \{A, B, C, D, E, F, G\}$, 3 layers $\mathcal{Y} = \{\text{Ethernet, SONET, WDM}\}$, with two incompatibilities at the SONET layer, the previously mentioned STS-24c and STS-3c-7v adaptations, $\mathcal{A}(STS) = \{\text{3c7v, 24c}\}$ and two incompatible wavelengths at the Wavelength

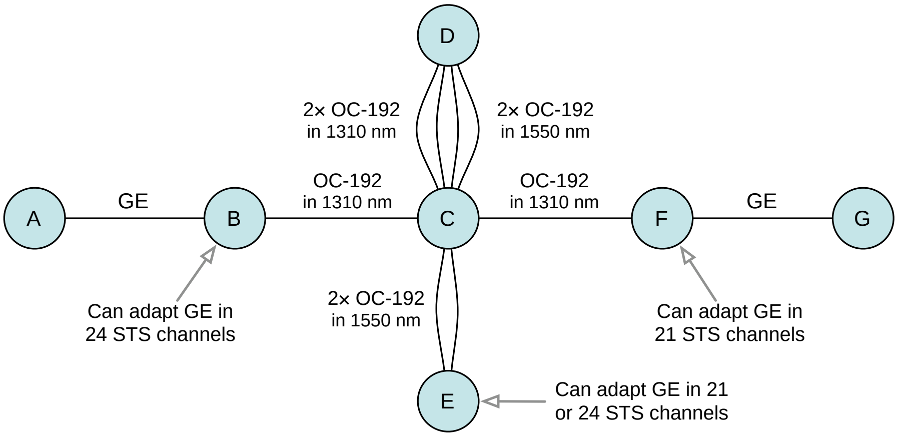


Figure 7.5: *Example of a three-layer network..*

Division Multiplexing (WDM) layer, $\mathcal{LB}(WDM) = \{1310\text{nm}, 1550\text{nm}\}$. Nodes A and G are pure Ethernet devices, nodes B, D, E and F are SONET devices, with B, D and F capable of (de)adapting Ethernet in SONET, and C is an optical cross connect. Nodes B and F are equipped with 1310 nm lasers, node E is equipped with 1550 nm lasers, and node D has tuneable lasers.

The shortest working path from A to G is $A - B - C - D - C - E - C - D - C - F - G$. B and F can not directly communicate due to the different adaptation of Ethernet in SONET, and B and E and also E and F can not communicate because of the different wavelengths.

7.6.1 Extension to Graph G_l

Graph G_l of this network is given in figure 7.6. The shortest path through this graph is $A_{Eth} - B_{Eth} - B_{STS}(24c) - B_{WDM}(1310) - C_{WDM}(1310) - D_{WDM} - DSTS - D_{WDM}(1550) - C_{WDM}(1550) - E_{WDM} - E_{STS}(24c^{-1}) - E_{Eth}(3c - 7v) - E_{STS} - E_{WDM}(1550) - C_{WDM}(1550) - D_{WDM} - D_{STS} - D_{WDM}(1310) - C_{WDM}(1310) - F_{WDM} - F_{STS}(3c - 7v^{-1}) - D_{Eth} - G_{Eth}$, with the label as used on the following edge denoted between brackets. A few new characteristics of G_l emerge. There is only a single adaptation between D_{STS} and D_{WDM}, even though that edge is used four times in the shortest path. In G_l edges can be traversed multiple times, as long as the available bandwidth is not exceeded. On the other hand, there are 4 edges between D_{WDM} and C_{WDM}, since the actual network has four different physical links.

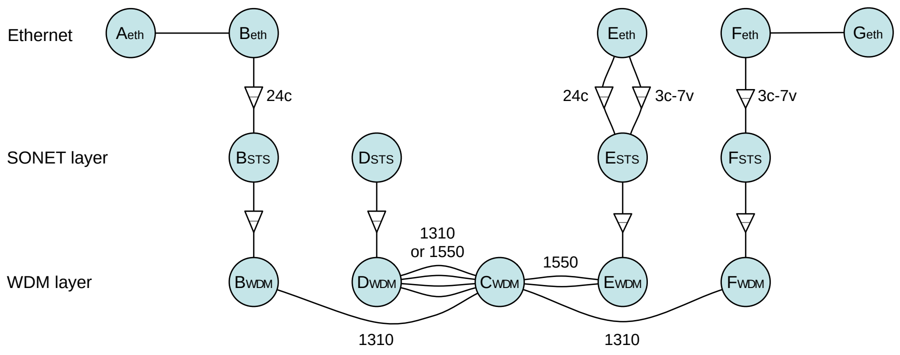


Figure 7.6: *Graph G_l of the network of figure 7.5.*

G_l treats the various incompatibilities differently. In section 6.3.1 we distinguish between four distinct incompatibilities: (1) in layer, (2) in adaptation, (3) in label (channel identifier), and (4) in other encoding characteristics (e.g., different MTU size for Ethernet). The graph G_l represents different adaptations as different edges, while different labels are represented as different labels or label sets for the edges. See the algorithm 7.2 in section 7.4: the incompatibility check for adaptation (line 15) is different from the incompatibility check for labels (line 22). This is a conscious choice: usually there are only a few (if any) incompatible adaptation functions, but many incompatible labels. In addition, G_l allows for *sometimes* incompatible labels. For example, one node may not be able to convert wavelengths, while another node may convert between wavelengths without de-adapting and adapting the wavelength. This is also referred to as *label swapping*. To support this, line 18 in the Extend-Path algorithm needs to be changed to an if statement:

```
**if** vertex $v$ supports label swapping **then**
    $LB(S_p(p_{new})) \leftarrow LB(v)$
**else**
    $LB(S_p(p_{new})) \leftarrow LB(S_p(p_{new})) \cap LB(v)$
```

Furthermore, the list of visited channels, $R[p]$, needs to be updated to include the labels, not just the node and layer.

The estimate of the running time algorithm as we presented in section 7.4 is not correct if we distinguish between labels and adaptations. In particular line 13 of algorithm takes *one* label $LB \in LB(v)$. This is ambiguous if $|LB(v)| > 1$. One solution to solve this is to simply return a set of path extensions, one for each label, rather than one label. In that case, the number of different stacks is approximately T^Y, and the estimate for the running time is correct. However, it is more efficiently to instead keep a set of allowed labels rather then a single label in the technology stacks. Thus each entry LB in $(y, \alpha, LB) \in \mathcal{S}$ would no longer be a single label, but a set of labels. In fact, the extension on lines 21-23 already does this: it reduces the set of allowed labels (or increases for devices with swapping capability), instead of terminating paths with incompatible label.

We define $|\langle Lb \rangle|$ as the number of different labels per layer, and $|\langle A \rangle|$ as the number of different adaptations per layer. T, the average number of different technologies per layer is:

$$T \approx |\langle Lb \rangle| \times |\langle A \rangle| \tag{7.38}$$

The maximum number of adaptation stacks no longer depends on the labels, and reduces from $T^{|\mathcal{Y}|}$ to $|\langle A \rangle|^{|\mathcal{Y}|}$. The running time for algorithm 7.2 increases from $\mathcal{O}(1)$ to $\mathcal{O}(|\langle Lb \rangle|)$. Thus, the estimate of the average running time of algorithm 7.1 in equation 7.37 changes to:

$$\mathcal{O}(\text{Algorithm 7.1}) \approx \mathcal{O}((|N| + \frac{|L|}{|\mathcal{Y}|}) \times |\langle A \rangle|^{|\mathcal{Y}|} \times \big(\log(|N| \times T^{|\mathcal{Y}|}) + \log(|\langle Lb \rangle|)\big)) \tag{7.39}$$

7.6.2 Extension to Graph G_s

Graph G_s of the network of figure 7.5 is given in figure 7.7. The first notable characteristic is the many different adaptation stacks. The five different technologies (one for Ethernet, two for SONET and two for WDM) lead to seven possible adaptations stacks: Ethernet, Ethernet in STS-24c, Ethernet in STS-3c-7v, Ethernet in STS-24c in 1310 nm, Ethernet in STS-24c in 1550 nm, Ethernet in STS-3c-7v in 1310 nm, and Ethernet in STS-3c-7v in 1550 nm. ($|\mathcal{S}| = 7$, and $\prod_{y \in \mathcal{Y}} |T(y)| = 4$)

The shortest path through the graph G_s is $A_{Eth} - B_{Eth} - B_{24c} - B_{24c;1310} - C_{24c;1310} - D_{24c;1310} - D24c - D_{24c;1550} - C_{24c;1550} - E_{24c;1550} - E_{24c} - E_{Eth} - E_{3c7v} - E_{3c7v;1550} - C_{3c7v;1550} - D_{3c7v;1550} - D_{3c7v} - D_{3c7v;1310} - C_{3c7v;1310} - F_{3c7v;1310} - F_{3c7v} - D_{Eth} - G_{Eth}$.

Many of the vertices in this graph only have two neighbours. One can filter the topology by removing one-degree vertices and by removing two-degree vertices and connecting their neighbours via a direct edge. However, in the process of filtering, we would have to keep track of the adaptation. The graph G_s can be considered as the memory needed to find a path in a layered network, as it explicitly keeps track of adaptation stacks along the path.

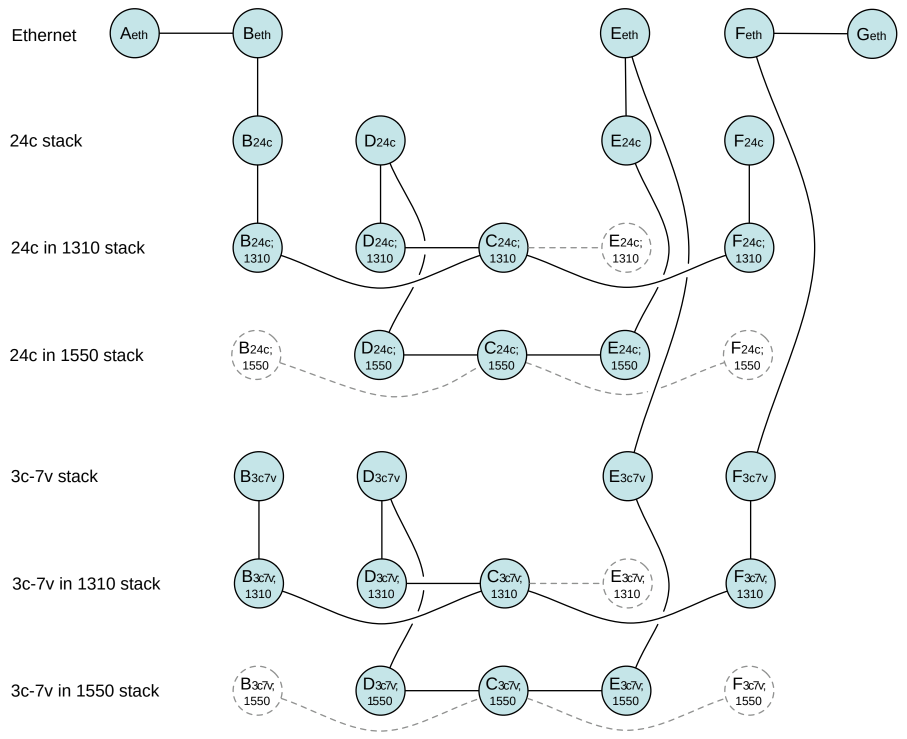


Figure 7.7: *Graph G_s of the network of figure 7.5.*

A few of the vertices in figure 7.7 are greyed, and are not part of $\mathcal{V}_s$. This is due to condition $\mathcal{LB}_s(s) \in LB(c)$ in the definition of $\mathcal{V}_s$ (see equation 7.13). These greyed vertices represent labels, which are not supported by that node. Vertex E for example does not support 1310 nm lasers.

There are only few extensions required to support labels in G_s. In fact, both labels and adaptations are simply instantiations of different technologies, which are reflected in the different technology stacks. The only difference between labels and adaptations is that in $\mathcal{V}_s$, the unsupported labels correspond to missing vertices in the graph G_s, while unsupported adaptations correspond to missing edges.

The changes to include label support do not change the time complexity as mentioned in equation A.18. It only changes the value of T, which is already present in this estimate.

Contrary to G_l, the shortest path through G_s never traverses the same vertex twice. In fact, this is a very useful property, since it means we can limit the number of edges between two vertices to at most one edge, even if there are multiple physical links. The resulting graph G_s is rather efficient. In fact, removing all 'dead ends' in the graph of figure 7.7, ends up with only the links of the shortest path, proving that there is only one shortest path in this example. Such a proof would be very hard for G_l.

In multi-layer networks, the number of incompatibilities may grow quite large. In the given example, the WDM layer only has two wavelengths. However, for Dense Wavelength Division Multiplexing (DWDM), about 100 different wavelengths are not uncommon. For Ethernet, there are 4096 incompatible VLAN tags (incompatible, since it is uncommon for devices to be able to alter the IEEE 801.1Q tag in packets). Tagged Ethernet over DWDM would thus yield about $4096 + 100 \times 4096 = 413696$ rows in G_s. This is contrary to G_l, which would have 2 rows. Of course, the complexity is inherent to the network itself, and emerges as increased memory use for path finding in G_l. For improved scalability, it may be possible to aggregate available channels into groups. For instance, consider an interface with the following VLAN channels available $[1, 50]$, $[53]$, $[89, 93]$, $[106, 123]$, $[400, 530]$ and another interface with the channels $[20, 30]$, $[50, 55]$, $[100, 110]$, $[3000, 4095]$. An intersection of these groups yields: $[1, 19]$, $[20, 30]$, $[31, 50]$, $[51, 52]$, $[53]$, $[54, 55]$, $[89, 93]$, $[100, 105]$, $[106, 110]$, $[111, 123]$, $[400, 530]$, $[3000, 4095]$ which can be represented by 12 (instead of 4096) rows. Both path finding in G_s and G_l benefit from such condensation of the topological constraints.

7.7 Discussion and Comparison

7.7.1 Commonalities

If we consider path finding in G_l and path finding in G_s, there are more similarities than differences between the two algorithms.

- Both algorithm keep track of stacks, and find a path starting at the source, and extending the path hop by hop. G_l does so by explicitly keep track of paths, G_s does so by keeping track of the shortest route to the source for each possible adaptation stack.

- Both algorithms are capable of finding paths with loops (that is, that use the same physical link twice).

- Neither algorithm supports real multiplexing, but only has a concept of bandwidth.
- Both network descriptions assume there is only one channel per layer per node.

7.7.2 Differences

We summarise the differences between the graphs G_l and G_s and the associated algorithms:

- G_s has a simple graph in the algorithmic sense. It only extends vertices, edges, with a bandwidth shared among those edges representing the same physical link. The shortest path through G_s never visits the same vertex twice, so there only has to be one edge between different vertices.
- G_s has multiple edges representing the same physical link. Algorithms needs to be adapted to take this into account.
- The edges in G_l representing an adaptation are directed, even in fully bidirectional networks. Moreover, the shortest path through G_l may traverse edges and vertices multiple times.
- Path finding in G_s assumes that the technology diagram has no cycles. If there are cycles, the number of possible adaptation stacks becomes infinite, and G_s can not be created.
- A local network node can be mapped to one or more vertices in G_l without having further knowledge of the network. To map a local node to one or more vertices in G_s, knowledge about the technologies of all layers above is needed.

The last point requires some explanation. In our second example, to describe node C in G_l, knowledge is required about which layer C is aware of (the WDM layer), as well as all physical links connected to C. To describe C in G_s, in addition knowledge is required about the incompatibilities on the higher layer (e.g. STS-24c versus STS-3c-7v), to know that we need to describe C as four vertices in G_s. This property makes it difficult to update G_s is a small aspect of the network topology changes.

The Multi-Layer-Breadth-First algorithm for G_l does not require full knowledge of the whole network, only of the set $\mathcal{V}$ of vertices v at distance $w(s, v) <$

$w(P^*)$, including all adjacencies $ajd[v]$ of v, with s the start point of the algorithm and P^* the shortest path from s to t. While in practice this set $\mathcal{V}$ can easily span most of the network, it may be interesting for extremely large multi-layer networks, and if s and t are relatively close to each other.

7.7.3 Time Complexity

One of the assumptions that can be made in link-constrained algorithms is that a segment of a shortest path is also a shortest path. As we have seen in section 7.3.1, this assumption is not true. Both algorithms introduced in this chapter can deal with this, but at the cost of severely diminished scalability.

The time estimates given in this chapter made the following assumptions, even though the algorithms are not restricted to these cases:

- It is assumed that it is only necessary to keep track of shortest path per technology stack per node.
- Path finding in G_s assumes that the technology diagram has neither branches nor cycles.

The estimate number of vertices and edges, as well as the actual figures for both example networks (figure 7.2 and figure 7.5) is listed in table 7.1. In these numbers, multiple links between the same pair of nodes are counted once, so we can more easily compare these figures. The numbers between brackets count all individual links.

The number of iterations required for each algorithm is listed in table 7.2. The shortest path in example 1 contains 12 vertices (11 edges). The shortest

		Example 1 (figure 7.2)		Example 2 (figure 7.5)	
		estimate	real	estimate	real
G_p	$\lvert\mathcal{V}_p\rvert = \lvert N\rvert$	6	6	7	7
	$\lvert\mathcal{E}_p\rvert = \lvert L\rvert$	6 (7)	6 (7)	10 (6)	10 (6)
G_l	$\lvert\mathcal{V}_l\rvert \lessapprox \lvert N\rvert \times \lvert\mathcal{Y}\rvert$	12	9	21	14
	$\lvert\mathcal{E}_l\rvert = \lvert A\rvert + \lvert L\rvert$	10 (11)	10 (11)	14 (18)	14 (18)
G_s	$\lvert\mathcal{V}_s\rvert \approx \lvert N\rvert \times T^{\lvert\mathcal{Y}\rvert}$	12	13	28	27 (33)
	$\lvert A\rvert + \lvert L'\rvert \leq \lvert\mathcal{E}_s\rvert \lessapprox (\lvert A\rvert + \lvert L\rvert) \times T^{\lvert\mathcal{Y}\rvert}$	9 - 20	13 (14)	14 - 72	26 (32)

Table 7.1: *Estimates and real sizes of the graphs.*

path in example 2 contains 23 vertices (22 edges).

		Example 1 (figure 7.2)			Example 2 (figure 7.5)		
		queue size	adj. checks	hop iter.	queue size	adj. checks	hop iter.
G_l	Breadth First	17	43	284	836	3113	46321
	Breadth First w/ collapsed links	13	31	189	36	81	945
G_s	k-Shortest Path w/o node removal	13	27	95	28	57	381
	k-Shortest Path	13	25	90	25	48	306
	k-Shortest Path w/ previous adjacency skipping	13	14	79	25	25	283
	k-Shortest Path w/ full distance estimate	13	24	89	24	45	298

Table 7.2: *Number of vertices and edges processed for path finding in two example networks.*

Each algorithm loops through a number of paths or vertices. For each of these iterations, it checks all adjacent neighbours. The following variants are tested:

Collapse Link By default, we model multiple links between the same pair of nodes as one link, with the combined capacity. Alternatively, the links can be individually described. In G_s all links are collapsed by default;

Node Removal The graph G_s contains information about every possible adaptation stack by default. This can be used to remove additional nodes, such as E with 24 STS channels over 1310 nm (see figure 7.4);

Skip previous adjacency By default, all adjacencies are checked. In G_s, a shortest path will never return to the node it was coming from, so that adjacency can be skipped. In G_l this is not true. A valid path may return over the previous edge in G_l if the node is capable of label conversion;

Estimate full distance By default, the Extract-Min procedure in both algorithms extracts the path with the shortest length so far. However, the MULTI-LAYER k-SHORTEST PATH algorithm (listing 7.4) will instead choose the path with the shortest estimate of the total length. The total length is calculated by appending a lower bound (found with Dijkstra's algorithm) to the length of a path so far. Although it is possible to implement such estimate in path finding in G_l, we have not implemented that.

Only exact algorithms are listed. Variants such as the Multi-Layer-Dijkstra (see appendix A.2) are not included in this table. In order to compare the algorithms for G_l and G_s the variants *Multi-Layer Breadth First with collapsed links* for G_l and *Multi-Layer k-Shortest Path without node removal and without Dijkstra's algorithm* for G_s should be examined, as these algorithms are basically the same, albeit using a different graph.

		Barabási-Albert ($n = 30$, single link)			Barabási-Albert ($n = 30$, double link)		
		queue size	adj. checks	hop iter.	queue size	adj. checks	hop iter.
G_l	Breadth First w/ collapsed links	196	245	762	191	237	737
G_s	k-Shortest Path w/o node removal	194	240	656	207	257	711
	k-Shortest Path w/o node removal	21	29	37	21	29	36

Table 7.3: *Average number of vertices and edges processed for path finding in twenty random networks. The twenty networks were generated with* 30 *nodes and up to* 3 *edges per new node.* $n = 30, m = 3$. *Both networks are single layer networks.*

Before we draw conclusions, we first must realise that that two example graphs in this chapter are specifically crafted to contain many incompatibilities. Table 7.3 gives the average results for 20 random networks, generated according to the Barabási-Albert preferential attachment model. Each random network was processed four times: once with a capacity of 1 per link, and once with a capacity of 2 per link. All random graphs are single layer network, and do not contain any incompatibility whatsoever. It remains to be seen how this applies to real life networks. From these 'incompatibility-free' networks we can conclude that indeed, the *Multi-Layer Breadth First with collapsed links* algorithm and the *Multi-Layer k-Shortest Path without node removal and* *without Dijkstra's* algorithm have a similar running time. Also, it is immediate clear that a breadth first search algorithms is very inefficient if it has no hint as the direction of the destination. The trick used to use a lower bound using the Dijkstra algorithm (first presented by Kuipers) certainly helps in this respect.

We expected algorithms in G_s to be marginally faster than algorithms in G_l. While this is the case for single layer networks, it is not the case for the manually crafted networks. In particular, the extremely high results for the Multi-Layer Breadth First search in example network 2 were unexpected. Apparently, the degrees of freedom caused by the abundance of links severely degrades the running time in this particular example.

7.8 Conclusion

In this chapter we have discussed the problem of finding paths in multi-layer networks. We have considered modelling a multi-layer network as a graph based on nodes and layers (G_l), and a graph based on nodes and technology stacks (G_s). For each model we have discussed the problem of finding feasible paths, given an algorithm and discussed the pros and cons of both approaches, including an estimate of the running times of the algorithm.

We found that a shortest path can contain loops, and both algorithms can deal with that situation.

In addition, a segment of a shortest path does not have to be a shortest path in itself. Again, both algorithms can deal with that situation, but the running time becomes exponential.

G_s is a simple graph in the algorithmic sense, and a shortest path can not contain loops. Nevertheless, it is not possible to apply a simple algorithm such as Dijkstra's algorithm or a Breadth search first algorithm to G_s.

Concluding, a graph based on layers and nodes is probably most suitable to describe actual networks in a multi-domain environment, where domains are reluctant to provide details about their own networks and there is no full topology knowledge. However, the simplicity of the algorithm for the graph based on nodes and stacks makes it more suitable for path finding calculations.

Whichever algorithm is used, we have conclusively proven that path finding in multi-layer networks is far more complex than path finding in single layer networks, and that assumptions valid for path finding in single layer networks no longer hold.