## Konzepte von Programmiersprachen FS2013

# **Kurzbericht Projekt 11**

OSPF Simulation in Haskell

Reto Lehnherr, Gabriel Zimmerli

**Abstract**

Unser Programm berechnet die kürzesten Pfade zwischen Routern gemäss dem OSPF-Protokoll und gibt eine Routing Tabelle aus. Das Programm simuliert den Ausgangsrouter in einem Netzwerk. Die Nachbarschaftstabelle, sowie die Link State Updates (LSU) der anderen Router werden aus einem File eingelesen und geparst. Der Kürzeste Pfad wird anhand des erstellen Netzwerk-Graphen mit dem Dijkstra-Algorithmus berechnet. Schwerpunkt der Implementation liegt beim Parsen der LSU und dem Dijkstra-Algorithmus.

**Idee des Projekts**

Wir möchten einen Teil des OSPF-Protokolls implementieren. Die Idee ist, dass unser Programm einen Router im Netz simuliert. Anhand von einer Tabelle sollen die Nachbarschaften zu anderen Routern deklariert sein.

Von diesen Nachbarn werden dann Link-State-Updates empfangen. Da unser Programm lediglich einen Router darstellt, werden dies Updates Files auf dem Dateisystem sein. Diese Files müssen ausgewertet werden um anschliessend eine Topologie Tabelle erstellen zu können. Danach wird anhand dieser Topologie Tabelle mit Hilfe des Dijkstra Algorithmus der «Shortest Path Tree» und damit schlussendlich die Routing-Tabelle erstellt.

Als Input haben wir ein Text-File mit einer Nachbarschaftstabelle, wie sie der Router nach Erhalt der OSPF-Hello Pakete erstellt hat. Als weiterer Input dient ein weiteres Text-File, welches Link State Updates aufführt, welche von jedem Router aus der Nachbarschaft geschickt werden. Mit diesen Informationen wird die Routing Tabelle berechnet, welche in ein File geschrieben wird.

**Hintergrund**

Das Open Shortest Path First (OSPF) Protokoll ist ein Link-State-Routing-Protokoll für IP-Netzwerke und basiert auf den Dijkstra-Algorithmus. Das OSPF Protokoll sammelt Link-State Informationen von den verfügbareren Routern und erstellt so eine Topologie des Netzwerks. Das Routing Protokoll berechnet den kürzesten Pfad anhand der «Link-Kosten» welche zu jedem Knoten im Netzwerk gespeichert wurden.

**Wichtige Begriffe[[1]](#footnote-1)**

**Hello Paket / Hello Protokoll:**

Das Hello-Protokoll ist in OSPF First für den Netzwerkbetrieb ein integraler Bestandteil des gesamten Routingprozesses. Es ist verantwortlich für:

* Senden von Keepalives in bestimmten Intervallen (damit wird bestätigt, ob die Route noch besteht)
* Zur Entdeckung eines Nachbarn
* Aushandlung der Parameter
* Wahl eines Designated Routers (DR) und des Backup-DRs

**Link State Updates / Link State Advertisements:**

OSPF-Router tauschen Informationen über die erreichbaren Netze mit sogenannten LSA-Nachrichten (Link State Advertisements) aus. Es gibt insgesamt über 10 verschiedene LSA Typen. In diesem Projekt konzentrieren wir uns aber nur auf den folgenden:

* Router-LSA (Typ 1): Für jeden aktiven Link des Routers, wird ein Eintrag im Router-LSA erzeugt. In ihm wird neben der IP-Adresse des Links auch die Netzmaske des Links und der Netzwerktyp (Loopback, Point-to-Point, normales Netz) eingetragen.

Ein Link State Update kann mehrere solche Link State Advertisements beinhalten

**Erstellung Nachbarschaftstabelle**

Ein Router der OSPF ausführt erstellt mit Hilfe des Hello Protokolles eine Nachbarschaftstabelle. Diese könnte folgendermassen aussehen:

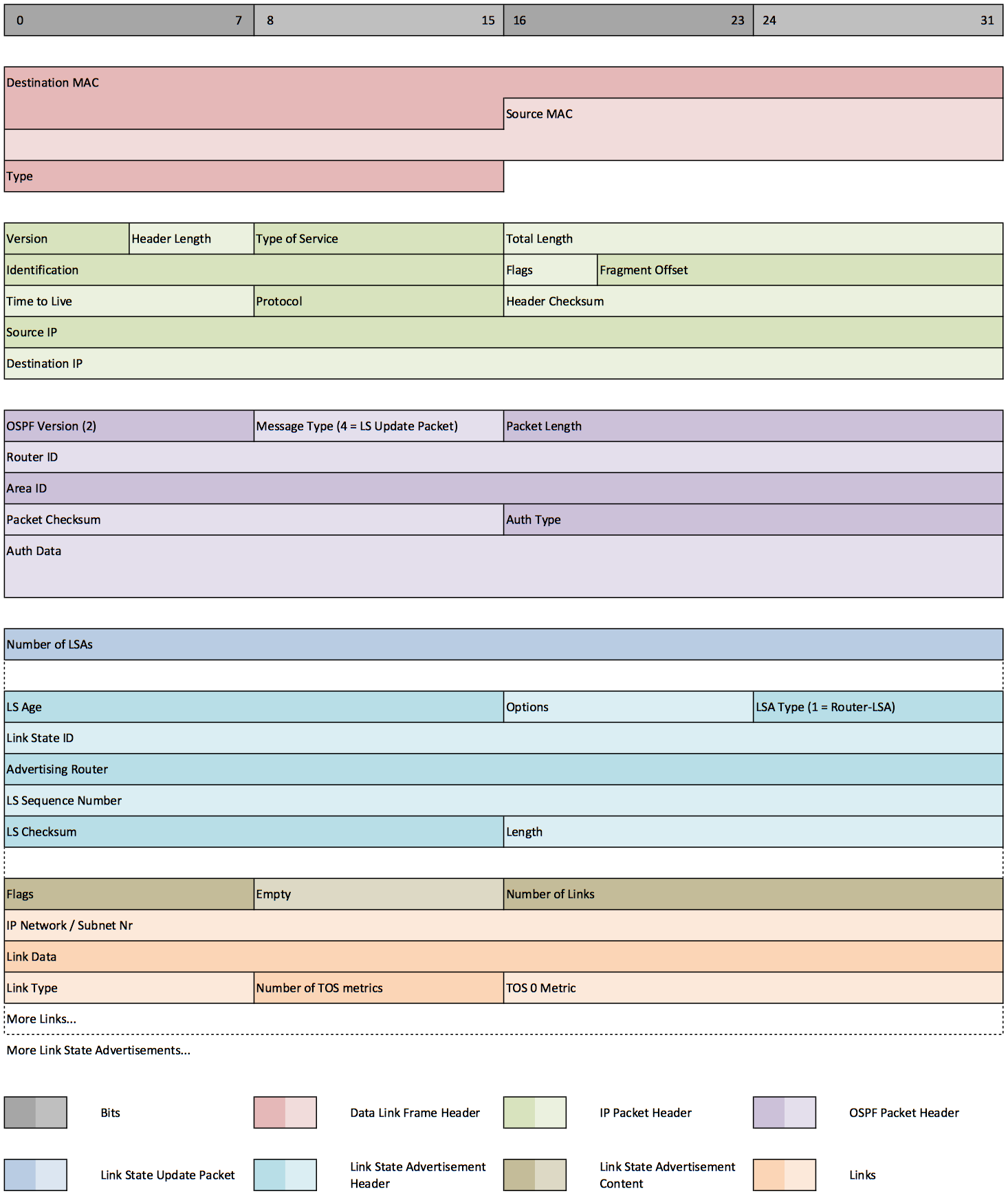
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Address | Interface | State | ID | Priority | Dead |
| 192.168.1.2 | so-0/0/0.0 | Full | R2 | 128 | 36 |
| 192.168.1.3 | so-0/0/1.0 | Full | R3 | 128 | 38 |
| 192.168.1.4 | so-0/0/2.0 | Full | R4 | 128 | 33 |

Um das ganze ein wenig zu vereinfachen, werden wir diesen Schritt weg lassen und gehen davon aus, dass die Nachbarschaften bereits aufgebaut wurden. Unser Programm (welches einen Router repräsentiert) liest eine solche Nachbarschaftstabelle aus einer Textdatei ein.

**Erstellung Topology Table**

Nachdem die Nachbarschaften ermittelt wurden, erstellt der Router anhand von Link State Updates eine Topology Table (oder auch Topology Graph / Link State Database). Dazu müssen erst mal die Link State Updates ausgewertet werden. In unserem Projekt wird ein solches Update als Datei im Filesystem repräsentiert. Jede solche Datei repräsentiert wiederum ein Link State Update eines benachbarten Routers.

Eine solche Datei beinhaltet eine Kette von Hexadezimalen Werten und weist folgende Struktur auf (eine Zeile sind 4 Byte):

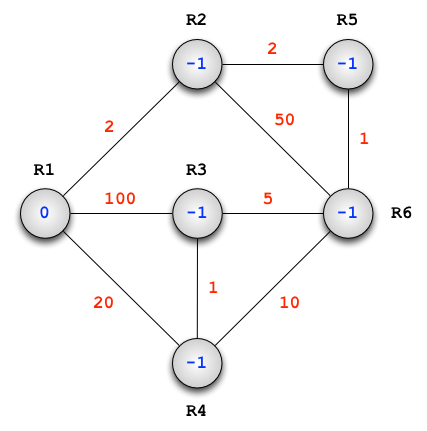


Nachdem die Link State Updates eingelesen und ausgewertet wurden, wird daraus die Topologie Tabelle erstellt. Diese stellt dar, welche Router miteinander verbunden sind und wie hoch die Kosten dieser Verbindungen sind.

**Als Tabelle:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| From Router  (Router ID) | To Router (Router ID) | | | | | |
| **R1** | **R2** | **R3** | **R4** | **R5** | **R6** |
| **R1** |  | 2 | 100 | 20 |  |  |
| **R2** | 2 |  |  |  | 2 | 50 |
| **R3** | 100 |  |  | 1 |  | 5 |
| **R4** | 20 |  | 1 |  |  | 10 |
| **R5** |  | 2 |  |  |  | 1 |
| **R6** |  | 50 | 5 | 10 | 1 |  |

**Als Graph:**



In Haskell wird diese Tabelle bzw. dieser Graph als Liste mit folgendem Typ definiert:

type Graph = [(RouterId, [(RouterId, Distance)])]

Das erste Element im Tupel ist die Router ID, das zweite ist eine Liste von Tupel. Jedes dieser Tupel repräsentiert wiederum einen Nachbar Router. Im „Nachbar Tupel“ ist das erste Element die Router ID und das zweite die Metrik bzw. die Distanz die für den Dijkstra Algorithmus relevant ist.

Die Liste für das oben gezeigte Beispiel würde in Haskell also folgendermassen aussehen:

graphInput = [("R1", [("R2", 2), ("R3", 100), ("R4", 20)]),

("R2", [("R1", 2), ("R5", 2), ("R6", 50)]),

("R3", [("R1", 100), ("R4", 1), ("R6", 5)]),

("R4", [("R1", 20), ("R3", 1), ("R6", 10)]),

("R5", [("R2", 2), ("R6", 1)]),

("R6", [("R2", 50), ("R3", 5), ("R4", 10), ("R5", 1)])]

**Erstellung Routing Table**

Der Router erstellt anhand der Topology Table eine Routing Table. Er berechnet jeweils den kürzesten Pfad zu einer Adresse basierend auf der konfigurierten Metrik, im Falle von OSPF ist dies die Bandbreite einer Verbindung. Um den kürzesten Pfad zu berechnen, erstellen wir aus den Daten einen gewichteten Graphen, wobei das Gewicht einer Kante aus der «cost metric», der Bandbreite einer Verbindung, bestimmt wird. In diesem Graphen berechnen wir für jeden Knoten den kürzesten Pfad, mithilfe des Dijkstra-Algorithmus.

**Metric**

Laut Wikipedia definiert Cisco die Metrik wie folgt: 108/Bandbreite. Damit wir in Haskell mit Integern arbeiten können definieren wir die Metrik leicht anders: 1010/Bandbreite.

|  |  |
| --- | --- |
| Bandbreite | Cost Metric |
| 10 Mbit/s | 1000 |
| 100 Mbit/s | 100 |
| 1 Gbit/s (Gigabit-Ethernet) | 10 |
| 10 Gbit/s | 1 |

**Aufbau Routingtabelle**

Gateway of last resort is not set

141.108.0.0/16 is variably subnetted, 8 subnets, 3 masks

O 141.108.1.128/25 [110/65] via 141.108.10.10, 00:15:28, Serial0/0

O 141.108.9.128/25 [110/129] via 141.108.10.10, 00:15:28, Serial0/0

O 141.108.1.0/25 [110/65] via 141.108.10.10, 00:15:28, Serial0/0

C 141.108.10.8/30 is directly connected, Serial0/0

O 141.108.9.0/25 [110/129] via 141.108.10.10, 00:15:28, Serial0/0

O IA 141.108.10.0/30 [110/192] via 141.108.10.10, 00:15:28, Serial0/0

O 141.108.12.0/24 [110/129] via 141.108.10.10, 00:15:28, Serial0/0

O 141.108.10.4/30 [110/128] via 141.108.10.10, 00:15:29, Serial0/0

131.108.0.0/16 is variably subnetted, 9 subnets, 4 masks

C 131.108.4.128/25 is directly connected, Loopback1

O 131.108.5.32/27 [110/1010] via 131.108.1.2, 00:16:04, Ethernet0/0

O 131.108.33.0/24 [110/74] via 141.108.10.10, 00:15:29, Serial0/0

O 131.108.6.1/32 [110/11] via 131.108.1.2, 00:16:04, Ethernet0/0

C 131.108.5.0/27 is directly connected, Loopback2

O 131.108.6.2/32 [110/11] via 131.108.1.2, 00:16:06, Ethernet0/0

C 131.108.4.0/25 is directly connected, Loopback0

C 131.108.1.0/24 is directly connected, Ethernet0/0

O 131.108.26.0/24 [110/138] via 141.108.10.10, 00:15:31, Serial0/0

Listing 1 Beispiel einer Routing Tabelle. Quelle: http://www.ciscopress.com/articles/article.asp?p=26919&seqNum=7

**Eintrag in einer Routing Table**

O 141.108.1.128/25 [110/65] via 141.108.10.10, 00:15:28, Serial0/0

O steht für das Routing-Protokoll, in unserem Fall immer O = OSPF

141.108.1.128/25 Ziel-Adresse

[110/65] [Administrative Distanz/Metrik]  
Die Administrative Distanz des Protokolls ist bei OSPF 110  
Die Metrik wird anhand der link cost berechnet und gibt das Gewicht eines Pfades an

via 141.108.10.10 Gibt an, ob die Route direkt oder über einen anderen Router erreichbar ist.

00:15:28 Dead-Time

Serial0/0 Interface Type

**Einschränkungen in unserem Projekt:**

* Die Routing Table berücksichtigt nur OSPF-Routen
* Die Prefix-Länge einer Adresse wird nicht ausgewertet
* Wir beschränken uns auf Router in einer Area ohne Verbindungen in eine andere
* Wir verzichten auf die Bestimmung eines Designated und Backup Designated Routers
* Bei den Link State Advertisement Typen (LSA) beschränken wir uns auf LSA1 (Router-LSA)

**Source Code**

Der Source Code ist direkt in den Source Files dokumentiert. Jede Funktion hat ihre eigene Beschreibung. Bei solchen mit vielen Parametern werden diese einzeln erläutert. Komplexere Funktionen haben ausserdem viele inline Kommentare die dabei helfen sollen den Überblick nicht zu verlieren und alles möglichst leicht nachvollziehbar zu machen.

In der Digitalen Version sind zudem noch die HTML Seiten des Haddock (äquivalent zu Java Dock) beigelegt, damit die Beschreibungen nicht im Code gesucht werden müssen.

**Dijkstra Algorithmus**

Weil die Wahrscheinlichkeit sehr hoch ist, dass schon viele diesen Algorithmus in Haskell implementiert haben, mussten wir aufpassen, nicht aus versehen im Netz darüber zu stolpern. Also haben wir lediglich eine formale Beschreibung gesucht, und daraus einen auf unser Problem passenden Pseudocode erstellt:

1. Suche "Route" mit geringster Metrik in "Routes" dessen ID noch in "Graph" ist und entferne "Node" mit dieser ID aus "Graph"
2. Besuche alle "ChildNodes" in aufsteigender Reihenfolge
   1. Für jedes "ChildNode": prüfe ob Metrik "ChildNode" > (Metrik "CurrentRoute" + Distanz "ChildNode")  
      falls ja:
      1. Aktualisiere Metrik "ChildRoute"
      2. Aktualisiere "RouteList" von "ChildRoute": "RouteList" von "CurrentNode" ++ [ID von "CurrentNode"]
3. Wiederhohle bis "Graph" leer ist

An dieser Stelle sollte vielleicht noch erwähnt werden, dass wir zwischen „Distanz“ und „Metrik“ folgende Unterscheidung machen:

* Distanz: Die Kosten / Distanz zwischen zwei Knoten (direkte Verbindung).
* Metrik: Die aufsummierten Kosten / Distanz eine Route (z.B. von A über B bis C).

**Quellen:**

* Skript «CCNA2 Routing-Konzepte und -Protokolle», Peter Gysel, IMVS FHNW
* Wireshark Beispiel-Capture
* The Routing Table v1.12 von Aaron Balchunas
* <http://en.wikipedia.org/wiki/Open_Shortest_Path_First>
* <http://de.wikipedia.org/wiki/Open_Shortest_Path_First>
* <http://en.wikipedia.org/wiki/Link-state_advertisement>
* <http://de.wikipedia.org/wiki/Link_State_Advertisement>
* http://de.wikipedia.org/wiki/Dijkstra-Algorithmus
* <http://www.cisco.com/en/US/tech/tk365/technologies_white_paper09186a0080094e9e.shtml>
* <http://www.ciscopress.com/articles/article.asp?p=26919&seqNum=7>
* <http://www.routeralley.com/ra/docs/routing_table.pdf>
* <http://www.routeralley.com/ra/docs/ospf.pdf>
* <http://www.juniper.net/techpubs/en_US/junos9.6/information-products/topic-collections/nog-baseline/ospf-neighbors-introduction.html>
* <http://www.cisco.com/en/US/tech/tk365/technologies_white_paper09186a0080094e9e.shtml>
* <http://www.tcpipguide.com/free/t_OSPFBasicTopologyandtheLinkStateDatabase.htm>
* <http://www.haskell.org/tutorial/modules.html>
* <http://www.haskell.org/cabal/users-guide/>
* <http://stackoverflow.com/questions/5804995/cabal-to-setup-a-new-haskell-project>
* <http://hackage.haskell.org/packages/archive/base/latest/doc/html/>

**Anhang: Source Code OSPFkpsp11.hs**

-------------------------------------------------------------------------------------------------

--- |

-- Module : OSPFkpsp11

-- Authors : Gruppe 11: Gabriel Zimmerli, Reto Lehnherr

-- Source : https://github.com/Gabriel-Z/kpsp

--

-- OSPFkpsp - Simulation des OSPF Protokoll in Haskell

--

-- Projektarbeit im Modul «Konzepte von Programmiersprachen» bei Edgar Lederer, FHNW

-------------------------------------------------------------------------------------------------

module OSPFkpsp11 where

import Prelude

import System.IO

import Data.Char

-- Typendeklaration zur besseren Lesbarkeit der Funktionsdeklarationen

-------------------------------------------------------------------------------------------------

--- | IP-Adresse eines Routers

type Address = String

--- | Das Interface des Routers

type Interface = String

--- | Priorität des Protokoll

type Priority = Int

--- | Timeout bis Route erneuert werden muss

type Dead = Int

--- | ID des Routers gemäss dem OSPF-Protokoll

type RouterId = String

--- | Die Distanz zwischen zwei Routern

type Distance = Int

--- | Das Gewicht des nach OSPF gewichteten Pfades (Aufsummierung der Distanzen auf Pfad)

type Metric = Int

--- | Liste mit den Hops, welche auf dem Pfad gemacht werden müssen

type Route = [RouterId]

--- | Beschreibt einen Kind Router bezogen zu seinem Eltern Router

type ChildNode = (RouterId, Distance)

--- | Beschreibung eines kürzesten Pfad zu einer Route

type ShortestPath = (RouterId, Metric, Route)

--- | Beschreibt einen Eintrag in der Nachbarschaftstabelle

type NeighboursTableEntry = (Address, Interface, State, RouterId, Priority, Dead)

--- | Beschreibt einen Eintrag im Topologie Graphen

type GraphEntry = (RouterId, [ChildNode])

--- | Beschreibt die Nachbarschaftstabelle der Routers, dessen Rolle diese Applikation eingenommen hat

type NeighboursTable = [NeighboursTableEntry]

--- | Der Graph beschreibt die Netzwerktopologie

type Graph = [GraphEntry]

-- Daten Deklarationen

--- | Status der Verbindung gemäss OSPF

data State = Down | Attempt | Init | TwoWay | ExStart | Exchange | Loading | Full deriving(Read, Show, Eq)

-------------------------------------------------------------------------------------------------

-- KONSTANTENDEKLARATION

-- Längen und Offset Konstanten für die LinkStateUpdates in Anzahl Bytes

--- | Header Länge (in Byte) des Data Link Frames

dataLinkFrameHeaderLength = 14

--- | Header Länge (in Byte) des IP Packet

ipPacketHeaderLength = 20

--- | Header Länge (in Byte) des OSPF Packet

ospfPacketHeaderLength = 24

--- | Offset (in Byte) vom Anfang des OSPF Packet bis zum Message Type

messageTypeOffset = 1

--- | Länge (in Byte) des Message Types

messageTypeLength = 1

--- | Offset (in Byte) vom Anfang des OSPF Packet bis zur Router ID

routerIdOffset = 4

--- | Länge (in Byte) der Router ID

routerIdLength = 4

--- | Header Länge (in Byte) des Link State Update Packet

lsuHeaderLength = 4

--- | Offset (in Byte) vom Anfang des Link State Update Headers bis zur Anzahl Link State Advertisements

numberOfLsaOffset = 0

--- | Länge (in Byte) der Anzahl Link State Advertisements

numberOfLsaLength = 4

--- | Header Länge (in Byte) des Link State Advertisement Packet

lsaHeaderLength = 20

--- | Offset (in Byte) vom Anfang des Link State Advertisement Headers bis zum LSA-Type

lsaTypeOffset = 3

--- | Länge (in Byte) des LSA-Type

lsaTypeLength = 1

--- | Offset (in Byte) vom Anfang des Link State Advertisement Headers bis zur Link State Advertisement Länge

lsaLengthOffset = 18

--- | Länge (in Byte) der Link State Advertisement Länge

lsaLengthLength = 2

--- | Content Länge (in Byte) des Link State Advertisement Packet (ohne die einzelnen Links)

lsaContentLength = 4

--- | Offset (in Byte) vom Anfang des Link State Advertisement Contents bis zu den Anzahl Links

numberOfLinksOffset = 2

--- | Länge (in Byte) der Anzahl Links

numberOfLinksLength = 2

--- | Länge (in Bytes) eines Links

linkLength = 12

--- | Offset (in Byte) vom Anfang des Links bis zur ID des Child Routers

childRouterIdOffset = 0

--- | Länge (in Byte) der Child Router ID

childRouterIdLength = 4

--- | Offset (in Byte) vom Anfang des Links bis zur Metrik

metricOffset = 10

--- | Länge (in Byte) der Metrik

metricLength = 2

-------------------------------------------------------------------------------------------------

-- Funktionen für das Parsen und Auswerten von Link State Update Files

--- | Nimt einen Hex-String entgegen und parst diesen in eine IP-Adresse

--- | Parameter 1: Hex-String (IP Adresse)

hexToIp :: String -> String

hexToIp (x:y:[]) = show ((digitToInt x) \* 16 + digitToInt y)

hexToIp (x:y:zs) = show ((digitToInt x) \* 16 + digitToInt y) ++ "." ++ hexToIp zs

--- | Nimt einen Hex-String entgegen und parst diesen in einen Int

--- | Parameter 1: Hex-String

hexToInt :: String -> Int

hexToInt [] = 0

hexToInt (x:xs) = (digitToInt x) \* 16 ^ (length xs) + hexToInt xs

--- | Extrahiert aus einem Link State Update (Hex-String) den Message Type (4 für Link State Update)

--- | Parameter 1: Hex-String (Link State Update)

extractMessageType :: String -> Int

extractMessageType lsu = hexToInt (take (messageTypeLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + messageTypeOffset\*2) lsu))

--- | Extrahiert aus einem Link State Update (Hex-String) die Router Id des Senders

--- | Parameter 1: Hex-String (Link State Update)

extractRouterId :: String -> RouterId

extractRouterId lsu = hexToIp (take (routerIdLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + routerIdOffset\*2) lsu))

--- | Extrahiert aus einem Link State Update (Hex-String) die Anzahl Link State Advertisements, die in diesem Update enthalten sind

--- | Parameter 1: Hex-String (Link State Update)

extractNumberOfLsa :: String -> Int

extractNumberOfLsa lsu = hexToInt (take (numberOfLsaLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + ospfPacketHeaderLength\*2 + numberOfLsaOffset\*2) lsu))

--- | Extrahiert aus einem Link State Update (Hex-String) die Längen der einzelnen Link State Advertisements

--- | Parameter 1: Hex-String (Link State Update)

--- | Parameter 2: Anzahl Link State Advertisements in diesem Link State Update

--- | Parameter 3: Aktuelles Link State Advertisement (am Anfang 1)

--- | Parameter 4: Akkumulator Liste für LSA-Lengths (Am Anfang leer)

extractLsaLengths :: String -> Int -> Int -> [Int] -> [Int]

extractLsaLengths lsu numOfLsa nthLsa lengths =

if (numOfLsa == nthLsa)

-- man ist beim letzten LSA angelangt

-- letzten LSA noch analysieren und "LsaLength" hinten an die Liste anhängen

-- Rekursion wird beendet

then

lengths ++ [hexToInt (take (lsaLengthLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + ospfPacketHeaderLength\*2 + lsuHeaderLength\*2 +

(sum lengths)\*2 + lsaLengthOffset\*2) lsu))]

-- man ist noch nicht beim letzten LSA angelangt

-- LSA analysieren und "LsaLength" hinten an die Liste anhängen

-- rekursiver Aufruf mit neuer Liste und nächstem LSA als Parameter

else

extractLsaLengths lsu numOfLsa (nthLsa+1)

(lengths ++ [hexToInt (take (lsaLengthLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + ospfPacketHeaderLength\*2 + lsuHeaderLength\*2 +

(sum lengths)\*2 + lsaLengthOffset\*2) lsu))])

--- | Extrahiert aus einem Link State Update (Hex-String) die LSA Types der einzelnen Link State Advertisements (1 für Router-LSA)

--- | Parameter 1: Hex-String (Link State Update)

--- | Parameter 2: Anzahl Link State Advertisements in diesem Link State Update

--- | Parameter 3: Aktuelles Link State Advertisement (am Anfang 1)

--- | Parameter 4: Liste mit allen Längen der einzelnen Link State Advertisements

--- | Parameter 5: Akkumulator Liste für LSA-Types (Am Anfang leer)

extractLsaTypes :: String -> Int -> Int -> [Int] -> [Int] -> [Int]

extractLsaTypes lsu numOfLsa nthLsa lengths types =

if (numOfLsa == nthLsa)

-- man ist beim letzten LSA angelangt

-- letzten LSA noch analysieren und "LsaType" hinten an die Liste anhängen

-- Rekursion wird beendet

then

types ++ [hexToInt (take (lsaTypeLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + ospfPacketHeaderLength\*2 + lsuHeaderLength\*2 +

(sum (take (nthLsa-1) lengths))\*2 + lsaTypeOffset\*2) lsu))]

-- man ist noch nicht beim letzten LSA angelangt

-- LSA analysieren und "LsaType" hinten an die Liste anhängen

-- rekursiver Aufruf mit neuer Liste und nächstem LSA als Parameter

else

extractLsaTypes lsu numOfLsa (nthLsa+1) lengths

(types ++ [hexToInt (take (lsaTypeLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + ospfPacketHeaderLength\*2 + lsuHeaderLength\*2 +

(sum (take (nthLsa-1) lengths))\*2 + lsaTypeOffset\*2) lsu))])

--- | Extrahiert aus einem Link State Update (Hex-String) die Anzahl Links, die in diesem Link State Advertisement enthalten sind

--- | Parameter 1: Hex-String (Link State Update)

--- | Parameter 2: Anzahl Link State Advertisements in diesem Link State Update

--- | Parameter 3: Aktuelles Link State Advertisement (am Anfang 1)

--- | Parameter 4: Liste mit allen Längen der einzelnen Link State Advertisements

--- | Parameter 5: Liste mit allen LSA-Typen der einzelnen Link State Advertisements

--- | Parameter 6: Akkumulator Liste für NumberOfLinks (Am Anfang leer)

extractNumbersOfLinks :: String -> Int -> Int -> [Int] -> [Int] -> [Int] -> [Int]

extractNumbersOfLinks lsu numOfLsa nthLsa lengths types numOfLinks =

if (types!!(nthLsa-1) == 1)

-- Nur Router-LSAs können interpretiert werden (1 = Router-LSA)

then

if (numOfLsa == nthLsa)

-- man ist beim letzten LSA angelangt

-- letzten LSA noch analysieren und "NumberOfLinks" hinten an die Liste anhängen

-- Rekursion wird beendet

then

numOfLinks ++ [hexToInt (take (numberOfLinksLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + ospfPacketHeaderLength\*2 + lsuHeaderLength\*2 +

(sum (take (nthLsa-1) lengths))\*2 + lsaHeaderLength\*2 + numberOfLinksOffset\*2) lsu))]

-- man ist noch nicht beim letzten LSA angelangt

-- LSA analysieren und "NumberOfLinks" hinten an die Liste anhängen

-- rekursiver Aufruf mit neuer Liste und nächstem LSA als Parameter

else

extractNumbersOfLinks lsu numOfLsa (nthLsa+1) lengths types

(numOfLinks ++ [hexToInt (take (numberOfLinksLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + ospfPacketHeaderLength\*2 + lsuHeaderLength\*2 +

(sum (take (nthLsa-1) lengths))\*2 + lsaHeaderLength\*2 + numberOfLinksOffset\*2) lsu))])

-- LSA ist kein Router-LSA

else

if (numOfLsa == nthLsa)

-- man ist beim letzten LSA angelangt

-- da aktuelles LSA kein Router-LSA ist, muss nichts mehr gemacht werden

-- Rekursion wird beendet

then numOfLinks

-- man ist noch nicht beim letzten LSA angelangt

-- rekursiver Aufruf mit nächstem LSA als Parameter

else extractNumbersOfLinks lsu numOfLsa (nthLsa+1) lengths types numOfLinks

--- | Extrahiert aus einem Link State Update (Hex-String) die Router Id und entsprechenden Distanzen der Child Router

--- | Parameter 1: Hex-String (Link State Update)

--- | Parameter 2: Anzahl Link State Advertisements in diesem Link State Update

--- | Parameter 3: Aktuelles Link State Advertisement (am Anfang 1)

--- | Parameter 4: Liste mit allen Längen der einzelnen Link State Advertisements

--- | Parameter 5: Liste mit allen LSA-Typen der einzelnen Link State Advertisements

--- | Parameter 6: Liste mit den Anzahl Links pro Link State Advertisement (nur für Router-LSA)

--- | Parameter 7: Aktueller Link (am Anfang 1)

--- | Parameter 8: Akkumulator Liste für ChildNodes (Am Anfang leer)

extractChildNodes :: String -> Int -> Int -> [Int] -> [Int] -> [Int] -> Int -> [ChildNode] -> [ChildNode]

extractChildNodes lsu numOfLsa nthLsa lengths types (n:numOfLinks) nthLink childNodes =

if (types!!(nthLsa-1) == 1)

-- Nur Router-LSAs können interpretiert werden (1 = Router-LSA)

then

if (numOfLsa == nthLsa)

-- man ist beim letzten LSA angelangt

then

if (n == nthLink)

-- man ist beim letzten Link angelengt

-- letzten Link noch analysieren und ChildNode hinten an die Liste anhängen

-- Rekursion wird beendet

then

childNodes ++ [(hexToIp (take (childRouterIdLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + ospfPacketHeaderLength\*2 + lsuHeaderLength\*2 +

(sum (take (nthLsa-1) lengths))\*2 + lsaHeaderLength\*2 + lsaContentLength\*2 + (nthLink-1)\*linkLength\*2 + childRouterIdOffset\*2) lsu)),

hexToInt (take (metricLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + ospfPacketHeaderLength\*2 + lsuHeaderLength\*2 +

(sum (take (nthLsa-1) lengths))\*2 + lsaHeaderLength\*2 + lsaContentLength\*2 + (nthLink-1)\*linkLength\*2 + metricOffset\*2) lsu)))]

-- man ist noch nicht beim letzten Link angelangt

-- Link analysieren und ChildNode hinten an die Liste anhängen

-- rekursiver Aufruf mit neuer Liste, dem selben LSA und nächsem Link als Parameter

else

extractChildNodes lsu numOfLsa nthLsa lengths types (n:numOfLinks) (nthLink+1)

(childNodes ++ [(hexToIp (take (childRouterIdLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + ospfPacketHeaderLength\*2 + lsuHeaderLength\*2 +

(sum (take (nthLsa-1) lengths))\*2 + lsaHeaderLength\*2 + lsaContentLength\*2 + (nthLink-1)\*linkLength\*2 + childRouterIdOffset\*2) lsu)),

hexToInt (take (metricLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + ospfPacketHeaderLength\*2 + lsuHeaderLength\*2 +

(sum (take (nthLsa-1) lengths))\*2 + lsaHeaderLength\*2 + lsaContentLength\*2 + (nthLink-1)\*linkLength\*2 + metricOffset\*2) lsu)))])

-- man ist noch nicht beim letzten LSA angelangt

else

if (n == nthLink)

-- man ist beim letzten Link angelengt

-- letzten Link noch analysieren und ChildNode hinten an die Liste anhängen

-- rekursiver Aufruf mit neuer Liste, nächstem LSA und erstem Link als Parameter

then

extractChildNodes lsu numOfLsa (nthLsa+1) lengths types numOfLinks 1

(childNodes ++ [(hexToIp (take (childRouterIdLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + ospfPacketHeaderLength\*2 + lsuHeaderLength\*2 +

(sum (take (nthLsa-1) lengths))\*2 + lsaHeaderLength\*2 + lsaContentLength\*2 + (nthLink-1)\*linkLength\*2 + childRouterIdOffset\*2) lsu)),

hexToInt (take (metricLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + ospfPacketHeaderLength\*2 + lsuHeaderLength\*2 +

(sum (take (nthLsa-1) lengths))\*2 + lsaHeaderLength\*2 + lsaContentLength\*2 + (nthLink-1)\*linkLength\*2 + metricOffset\*2) lsu)))])

-- man ist noch nicht beim letzten Link angelangt

-- Link analysieren und ChildNode hinten an die Liste anhängen

-- rekursiver Aufruf mit neuer Liste, dem selben LSA und nächstem Link als Parameter

else

extractChildNodes lsu numOfLsa nthLsa lengths types (n:numOfLinks) (nthLink+1)

(childNodes ++ [(hexToIp (take (childRouterIdLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + ospfPacketHeaderLength\*2 + lsuHeaderLength\*2 +

(sum (take (nthLsa-1) lengths))\*2 + lsaHeaderLength\*2 + lsaContentLength\*2 + (nthLink-1)\*linkLength\*2 + childRouterIdOffset\*2) lsu)),

hexToInt (take (metricLength\*2)

(drop (dataLinkFrameHeaderLength\*2 + ipPacketHeaderLength\*2 + ospfPacketHeaderLength\*2 + lsuHeaderLength\*2 +

(sum (take (nthLsa-1) lengths))\*2 + lsaHeaderLength\*2 + lsaContentLength\*2 + (nthLink-1)\*linkLength\*2 + metricOffset\*2) lsu)))])

-- LSA ist kein Router-LSA

else

if (numOfLsa == nthLsa)

-- man ist beim letzten LSA angelangt

-- da aktuelles LSA kein Router-LSA ist, muss nichts mehr gemacht werden

-- Rekursion wird beendet

then childNodes

-- man ist noch nicht beim letzten LSA angelangt

-- rekursiver Aufruf mit nächstem LSA und erstem Link als Parameter

else extractChildNodes lsu numOfLsa (nthLsa+1) lengths types (n:numOfLinks) 1 childNodes

-- es sind keine Links mehr vorhanden die analysiert werden könnten

-- Rekursion wird beendet

extractChildNodes lsu numOfLsa nthLsa lengths types [] nthLink childNodes = childNodes

--- | Liest ein LinkStateUpdate aus einem File ein und gibt die Topologie Tabelle (Graph) als Liste zurück

--- | Parameter 1: Datei Pfad zum Link State Update File das eingelesen werden soll

readLinkStateUpdate :: FilePath -> IO GraphEntry

readLinkStateUpdate filePath = do

update <- readFile filePath

let upperUpdate = map toUpper update

let messageType = extractMessageType upperUpdate

let routerId = if (messageType == 4)

then extractRouterId upperUpdate

else "0.0.0.0"

let numberOfLsa = if (messageType == 4)

then extractNumberOfLsa upperUpdate

else 0

let lsaLengths = if (numberOfLsa > 0)

then extractLsaLengths upperUpdate numberOfLsa 1 []

else []

let lsaTypes = if (numberOfLsa > 0)

then extractLsaTypes upperUpdate numberOfLsa 1 lsaLengths []

else []

let numbersOfLinks = if (lsaTypes /= [])

then extractNumbersOfLinks upperUpdate numberOfLsa 1 lsaLengths lsaTypes []

else []

let childNodes = if (numbersOfLinks /= [])

then extractChildNodes upperUpdate numberOfLsa 1 lsaLengths lsaTypes numbersOfLinks 1 []

else []

return (routerId, childNodes)

--- | Liest alle definierten Link State Updates ein und baut daraus den Topology Graph

buildTopoGraph :: IO [GraphEntry]

buildTopoGraph = do

lsu1 <- readLinkStateUpdate "data/lsu1.txt"

lsu2 <- readLinkStateUpdate "data/lsu2.txt"

lsu3 <- readLinkStateUpdate "data/lsu3.txt"

lsu4 <- readLinkStateUpdate "data/lsu4.txt"

lsu5 <- readLinkStateUpdate "data/lsu5.txt"

lsu6 <- readLinkStateUpdate "data/lsu6.txt"

let graph = [lsu1, lsu2, lsu3, lsu4, lsu5, lsu6]

return graph

-------------------------------------------------------------------------------------------------

-- Funktionen für den Shortest Path Algorithmus

--- | Initialisiere die Routes Tabelle anhand des Graphen. Die Metriken für jeden Pfad werden mit -1 initialisiert

setupRoute :: Graph -> [ShortestPath]

setupRoute ((a,\_):graph) = (a,0,[]):[(fst g, -1, []) | g <- graph]

--- | Gibt RouterId zurück, welcher noch nicht abgearbeitet ist und den aktuell kürzesten Pfad hat

nextRouterId :: Graph -> [ShortestPath] -> RouterId

nextRouterId graph shortestpaths = nextRId

where

graphIds = [nodeId | (nodeId,\_) <- graph]

possibleRoutes = [(routeId, metric, route) | (routeId, metric, route) <- shortestpaths, metric >= 0, elem routeId graphIds]

nextRId = fst (foldl sortShortestPath ("255.255.255.255", 10000) possibleRoutes)

--- | Vergleicht ein (RouterId, Metric)-Tupel mit einem ShortestPath und gibt das Element mit kleinerer Metrik zurück, als (RouterId, Metric)

--- | Wird verwendet um den Router mit dem aktuell kürzesten Pfad zu finden

sortShortestPath :: (RouterId, Metric) -> ShortestPath -> (RouterId, Metric)

sortShortestPath (routeIdL, metricL) (routeIdR,metricR,\_) = if metricL < metricR then (routeIdL, metricL) else (routeIdR, metricR)

--- | Aktualisiert die Liste der kürzesten Pfade anhand des aktuell ausgewählten Router.

--- | Das erste Argument beschreibt die Distanz zu den Nachbarn des Routers

--- | Das zweite Argument ist der kürzeste Pfad zum aktuellen Router

--- | Das letzte Argument ist die Liste der aktuell kürzesten Pfade.

--- | Die Rückgabe ist die aktualisierte Shortestpath-Liste

updateShortestPaths :: [ChildNode] -> ShortestPath -> [ShortestPath] -> [ShortestPath]

updateShortestPaths ((neighbourId, neibourDistance):ns) currentRoute routes = updateShortestPaths ns currentRoute updated

where

(currentNodeId, currentNodeMetrik, currentNodeRoute) = currentRoute

updated = [if (metrik == -1 || currentNodeMetrik + neibourDistance < metrik) && neighbourId == nodeId

then (nodeId, currentNodeMetrik + neibourDistance, currentNodeRoute ++ [currentNodeId])

else (nodeId, metrik, route) | (nodeId, metrik, route) <- routes]

updateShortestPaths [] \_ routes = routes

--- | Gibt den aktuellen ShortestPath zu einem Router aus der Liste zurück

currentRoute :: [ShortestPath] -> RouterId -> ShortestPath

currentRoute routes targetNode = head [(nodeId, metric, route) | (nodeId, metric, route) <- routes, nodeId == targetNode]

--- | Finde die Nachbarn im Graph für den aktuellen Router

neighboursOfCurrentRouter :: RouterId -> Graph -> [ChildNode]

neighboursOfCurrentRouter routerId graph = head ([neighbours | (rid, neighbours) <- graph, rid == routerId])

-- neighboursOfCurrentRouter routerId graph = snd (head (filter (\(routerID, neighbours) -> routerID == routerId) graph))

--- | Berechung der kürzesten Pfade. Nimmt Topologie Tabelle als input Parameter und gibt eine Liste von Trippeln (ID, Metrik, Route) zurück

dijkstra :: Graph -> [ShortestPath] -> [ShortestPath]

dijkstra graph [] = dijkstra graph (setupRoute graph) -- einstieg: erstelle initiale Route tabledijkstra [] routes = routes -- basisfall: Routentabelle erstellt

dijkstra [] routes = routes -- basisfall: Routentabelle erstellt

dijkstra graph routes = dijkstra restGraph shortestPaths --- iteration: Wende Algorithmus für jeden Router im Graphen an

where

nRid = nextRouterId graph routes -- id des nächsten zu bearbeitenden Routers

neighbours = neighboursOfCurrentRouter nRid graph -- Die Nachbarn des nächsten Routers

restGraph = [(routerId, neighbours) | (routerId, neighbours) <- graph, routerId /= nRid] -- Rest des Graph, alle noch nicht "markierten" (bearbeiteten) Router

shortestPaths = updateShortestPaths neighbours (currentRoute routes nRid) routes -- aktualisierter kürzester Pfad

-------------------------------------------------------------------------------------------------

-- Funktionen für die Ausgabe

--- | Generiert die Routing Tabelle und schreibt diese auf die Konsole

printRoutingTable :: [ShortestPath] -> NeighboursTable -> IO()

printRoutingTable [] \_ = putStrLn ""

printRoutingTable (sp:xs) neighboursTable = do

let routerId = getRouterId sp

let metric = getMetric sp

let via = getVia sp routerId

let neighbourEntry = getNeighboursTableEntry via neighboursTable

let dead = getDeadFromNeighboursTable neighbourEntry

let interface = getInterfaceFromNeighboursTable neighbourEntry

if (via == "own")

then

putStr ""

else

if (via == routerId)

then

putStrLn ("C " ++ routerId ++ "/24 is directly connected, " ++ interface)

else

putStrLn ("O " ++ routerId ++ "/24 [110/" ++ (show metric) ++ "] via " ++ via ++ ", 00:00:" ++ (show dead) ++ ", " ++ interface)

printRoutingTable xs neighboursTable

where

getRouterId (routerId,\_,\_) = routerId

getMetric (\_,metric,\_) = metric

getVia (\_,\_,(r0:r1:rs)) \_ = r1

getVia (\_,\_,[]) \_ = "own"

getVia (\_,\_,(r:[])) rid = rid

getNeighboursTableEntry rid (r:rs) = if ((getRouterIdFromNeighboursTable r) == rid) then r else getNeighboursTableEntry rid rs

getInterfaceFromNeighboursTable (\_,i,\_,\_,\_,\_) = i

getRouterIdFromNeighboursTable (\_,\_,\_,r,\_,\_) = r

getDeadFromNeighboursTable (\_,\_,\_,\_,\_,d) = d

--- | Schreibt den Graphen auf die Konsole

printGraph :: Graph -> IO()

printGraph ((rid, children):xs) = do

putStrLn ("Router: " ++ rid ++ " \nhas neighbours:" ++ (formatNeighbours children) ++ "\n" )

printGraph xs

where

formatNeighbours ns = concat ["\n\tRouter: " ++ (show rid) ++ " Distance: " ++ (show dist) | (rid, dist) <- ns ]

printGraph [] = do putStr ""

--- | Schreibt die Shortestpath Trees auf die Konsole

printShortestPaths :: [ShortestPath] -> IO()

printShortestPaths ((rid,metric,[]):xs) = do printShortestPaths xs

printShortestPaths ((rid,metric,route):xs) = do

putStrLn ("Target Router: " ++ rid ++ "\tMetric: " ++ (show metric) ++ "\tPath:" ++ (formatPath route))

printShortestPaths xs

where

formatPath path = concat [" -> " ++ r | r <- path]

printShortestPaths [] = do putStr ""

-------------------------------------------------------------------------------------------------

-- Funktionen für die Nachbarschaftstabelle

--- | Teilt einen String an den Tabulatorenzeichen

splitStringOnTab :: String -> [String]

splitStringOnTab x = splitIt [] [] x

where

splitIt accu curstring (x:xs) | x == '\t' = splitIt (accu ++ [curstring]) [] xs

| otherwise = splitIt accu (curstring ++ [x]) xs

splitIt accu [] [] = accu

splitIt accu curstring [] = splitIt (accu ++ [curstring]) [] []

--- | Parst eine Zeile in der Nachbarschaftstabelle

parseNeighbourTableLine :: [String] -> NeighboursTableEntry

parseNeighbourTableLine (a:i:s:r:p:d:[]) = (a, i, read s, r, read p, read d) :: (Address,Interface,State,RouterId,Priority,Dead)

-- Konvertieren auf den richtigen Typ. Vorsicht bei von [Char] abgeleiteten typen

--- | Liest die Nachbarschaftstabelle (in Tabellenform) aus einem File ein und parst diese in eine Liste mit Nachbarn und deren Eigenschaften

readNeighbourTable :: String -> NeighboursTable

readNeighbourTable fileContents = processNeighbourTable (tail (lines fileContents)) -- ignoriere erste zeile

where

processNeighbourTable xs = [processLine x | x <- xs ]

processLine x = parseNeighbourTableLine (splitStringOnTab x)

--- | Das Programm.

main = do

neighboursTableContents <- readFile "data/neighboursTable.txt"

expectedResultFileContents <- readFile "data/expectedResult.txt"

topoGraphInput <- buildTopoGraph

let neighboursTable = readNeighbourTable neighboursTableContents

let expectedResultInput = read expectedResultFileContents :: [ShortestPath]

let shortestPaths = dijkstra topoGraphInput []

putStrLn "\n\*\*\*\*\* Topology Graph \*\*\*\*\*"

printGraph topoGraphInput

putStrLn "\n\*\*\*\*\* Shortest Paths \*\*\*\*\*"

printShortestPaths shortestPaths

putStrLn "\n\*\*\*\*\* Routing Table \*\*\*\*\*"

printRoutingTable shortestPaths neighboursTable

putStrLn "\*\*\*\*\* Test Result \*\*\*\*\*"

print (shortestPaths == expectedResultInput)

putStrLn ""

1. Quelle: http://de.wikipedia.org/wiki/Open\_Shortest\_Path\_First [↑](#footnote-ref-1)