

Vorlesung

# **Algorithmen für Kommunikationsnetze**

Prof. Dr. Thomas Erlebach

Eidgenössische Technische Hochschule Zürich

(erstmalig gehalten im Wintersemester 2000/2001)

# Inhaltsverzeichnis

<b>1 Grundlagen</b>	<b>1</b>
1.1 Graphen . . . . .	1
1.2 Algorithmen . . . . .	3
1.2.1 Laufzeit von Algorithmen . . . . .	3
1.2.2 Algorithmen für Optimierungsprobleme . . . . .	4
1.3 Grundlegende Graphenalgorithmen . . . . .	5
1.3.1 Kürzeste Pfade . . . . .	5
1.3.2 Minimale Spannbäume . . . . .	6
1.3.3 Netzwerk-Fluss . . . . .	7
1.4 Lineare Programmierung . . . . .	8
1.5 Literaturhinweise . . . . .	9
<b>2 Minimale Spannbäume und Steiner-Bäume</b>	<b>11</b>
2.1 Minimale Spannbäume . . . . .	11
2.1.1 Algorithmus von Prim . . . . .	13
2.2 Berechnung von Steiner-Bäumen . . . . .	15
2.2.1 Problemdefinition . . . . .	15
2.2.2 Die Distanz-Heuristik . . . . .	16
2.2.3 Bemerkungen . . . . .	20
<b>3 Kürzeste Wege, Spanner und Netzwerkdesign</b>	<b>22</b>
3.1 Berechnung kürzester Wege in Graphen . . . . .	22
3.2 Leichte angenäherte Kürzeste-Wege-Bäume . . . . .	24
3.2.1 Die Präordernummerierung von Bäumen . . . . .	25
3.2.2 Die Berechnung eines LAST . . . . .	27
3.3 Spanner . . . . .	29
3.3.1 Analyse des Stretch-Faktors . . . . .	30
3.3.2 Analyse der Kantenzahl des Spanners . . . . .	30
3.3.3 Analyse des Gewichts des Spanners . . . . .	32
3.4 Modellierung eines Netzwerkdesign-Problems . . . . .	36
3.5 Design von Access-Netzen . . . . .	37
3.5.1 Ein Approximationsalgorithmus für das Design von Access-Netzen . . . . .	38
3.6 Allgemeines Netzwerkdesign . . . . .	39
3.6.1 Ein Approximationsalgorithmus für allgemeines Netzwerkdesign . . . . .	40
3.7 Literaturhinweise . . . . .	41

<b>4</b>	<b>Das Problem des IP Prefix Lookup</b>	<b>43</b>
4.1	IP Prefix Lookup mit einem Trie . . . . .	44
4.1.1	Konstruktion des Trie . . . . .	46
4.1.2	Speicherplatzbedarf des Trie . . . . .	46
4.1.3	Suchen im Trie . . . . .	46
4.1.4	Aktualisieren des Trie . . . . .	47
4.2	IP Prefix Lookup durch Binärsuche nach Präfixlängen . . . . .	47
4.2.1	Suche innerhalb einer Längenkategorie . . . . .	48
4.2.2	Binärsuche . . . . .	48
4.2.3	Speicherplatzbedarf . . . . .	49
4.2.4	Konstruktion . . . . .	50
4.2.5	Aktualisieren der Datenstruktur . . . . .	50
4.2.6	Experimentelle Ergebnisse . . . . .	50
4.3	Präfix-Expansion für schnelleren IP Prefix Lookup . . . . .	51
4.3.1	Expansion von Präfixen . . . . .	51
4.3.2	Dynamische Programmierung zur Wahl der Längensklassen . . . . .	52
4.3.3	Multibit-Tries und Präfix-Expansion . . . . .	54
4.4	Weitere Ansätze zur Realisierung des IP Prefix Lookup . . . . .	59
<b>5</b>	<b>Online-Zugangskontrolle und Routing in ATM-Netzen</b>	<b>61</b>
5.1	Problemmodellierung . . . . .	62
5.2	Der Algorithmus für Zugangskontrolle und Routing . . . . .	63
5.2.1	Analyse des Algorithmus . . . . .	63
5.3	Literaturhinweise . . . . .	66
<b>6</b>	<b>Wellenlängenzuteilung in optischen WDM-Netzen</b>	<b>67</b>
6.1	Problemdefinitionen . . . . .	68
6.2	Überblick über bekannte Ergebnisse . . . . .	69
6.3	Algorithmen für Pfadfärbung . . . . .	71
6.3.1	Pfadfärbung in Ketten . . . . .	71
6.3.2	Pfadfärbung in Bäumen . . . . .	71
6.3.3	Pfadfärbung in Ringen . . . . .	73
6.3.4	Pfadfärbung in Bäumen von Ringen . . . . .	73
6.4	Reduktion von MaxRPC auf MEDP . . . . .	74
6.5	Optimaler Multicast in optischen WDM-Netzen . . . . .	77
6.5.1	Minimierung der maximalen Kantenlast . . . . .	77
6.5.2	Minimierung der Farbanzahl . . . . .	80
6.5.3	Gleichheit von $L^*(R)$ und $W^*(R)$ . . . . .	83
<b>7</b>	<b>Ausfallsicherheit von Netzen</b>	<b>85</b>
7.1	Der Knoten- und Kantenzusammenhang von Graphen . . . . .	85
7.2	Die Blockstruktur von Graphen . . . . .	87
7.2.1	Berechnung der Blöcke mittels Tiefensuche . . . . .	89
7.3	Erhöhung des Knotenzusammenhangs . . . . .	93
7.3.1	Untere Schranke für die Anzahl einzufügender Kanten . . . . .	93
7.3.2	Optimale Auswahl der einzufügenden Kanten . . . . .	94
7.4	Effiziente Berechnung des Kantenzusammenhangs . . . . .	97

7.4.1	Ein einfacher Min-Cut-Algorithmus . . . . .	97
-------	---	----

# Kapitel 1

## Grundlagen

In dieser Vorlesung beschäftigen wir uns mit grundlegenden Techniken für den Entwurf und die Bewertung von Algorithmen im Hinblick auf Anwendungen beim Betrieb oder bei der Optimierung von Kommunikationsnetzen. Für unterschiedliche Arten von Problemstellungen werden jeweils nützliche Techniken für den Entwurf von Algorithmen sowie sinnvolle Bewertungsmethoden behandelt. Bei Kommunikationsnetzen denken wir beispielsweise an Netze wie das Internet, lokale Netze (LANs) oder Telekommunikationsnetze von Telefongesellschaften.

### 1.1 Graphen

Kommunikationsnetze werden in der Regel als Graphen modelliert. Ein Graph  $G = (V, E)$  besteht aus einer Menge  $V$  von Knoten und einer Menge  $E$  von Kanten. Wenn wir einmal mit mehreren Graphen gleichzeitig argumentieren müssen, bezeichnen wir die Knoten- bzw. Kantenmenge eines Graphen  $G$  zur Klarheit mit  $V(G)$  bzw.  $E(G)$ . Wir unterscheiden *ungerichtete Graphen* und *gerichtete Graphen*, siehe Abb. 1.1.

Bei ungerichteten Graphen verbindet jede Kante zwei Knoten, ohne dass eine Richtung der Kante vorgegeben ist. Eine Kante wird daher als zweielementige Teilmenge der Knotenmenge repräsentiert, d.h.  $e = \{u, v\}$  mit  $u, v \in V$ . Wir sagen, dass eine Kante, die zwei Knoten verbindet, zu jedem dieser Knoten *inzident* ist. Wenn zwei Knoten über eine Kante verbunden sind, so sind diese beiden Knoten *adjazent*. Der *Grad* eines Knotens ist die Anzahl der Kanten, die inzident zu dem Knoten sind.

Bei gerichteten Graphen hat jede Kante einen Startknoten und einen Endknoten, sie ist also vom Startknoten zum Endknoten gerichtet. Hier werden Kanten als *geordnete* Paare von Knoten repräsentiert, d.h.  $e = (u, v)$  mit  $u, v \in V$ . Gerichtete Graphen werden auch Digraphen genannt. Die Anzahl der Kanten, die von einem Knoten weg führen, ist der *Ausgangsgrad* des Knotens. Die Anzahl der Kanten, die zu dem Knoten hin führen, ist der *Eingangsgrad*.

Ein Sonderfall gerichteter Graphen sind die *symmetrisch gerichteten* Graphen. In diesen Graphen gilt: wenn der Graph eine Kante  $(u, v)$  mit Startknoten  $u$  und Endknoten  $v$  enthält, dann enthält er auch eine Kante  $(v, u)$  in der anderen Richtung.

In einem Graphen, der ein Kommunikationsnetz repräsentiert, entsprechen die Knoten den Netzwerkknoten (Router, Switches, Hosts) und die Kanten den physikalischen Verbindungen (Links) zwischen diesen Knoten. Symmetrisch gerichtete Graphen modellieren dabei Netze mit Links, auf denen in beide Richtungen unabhängig voneinander Daten übertragen werden können. Wir werden die Begriffe *Kante* und *Link* oft synonym verwenden.

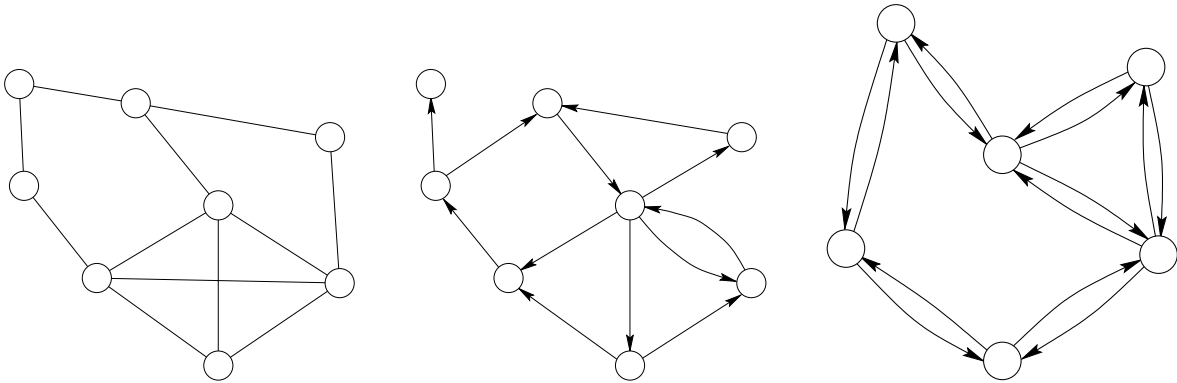


Abbildung 1.1: Beispiele für Graphen: ungerichteter Graph (links), gerichteter Graph (Mitte), symmetrisch gerichteter Graph (rechts).

Wir bezeichnen die Anzahl der Knoten eines Graphen immer mit  $n = |V|$  und die Anzahl der Kanten mit  $m = |E|$ . Die Datenstruktur, die üblicherweise zur Speicherung eines Graphen in einem Programm verwendet wird, besteht grob gesagt aus einer Liste aller Knoten, einer Liste aller Kanten, und zu jedem Knoten einer Liste der zu dem Knoten benachbarten (inzidenten) Kanten. Bei gerichteten Graphen werden bei den Knoten getrennte Listen für eingehende und ausgehende Kanten gespeichert. Weiterhin sind die Knoten und Kanten i.d.R. durchnummeriert und können über ihre Nummer direkt zugegriffen werden. Der Speicherplatzbedarf für diese Datenstruktur ist linear, d.h. proportional zu  $n+m$ . Eine alternative Darstellung verwendet eine Adjazenz-Matrix, d.h. eine  $n \times n$ -Matrix  $A$ , in der der Eintrag  $a_{i,j}$  gleich 1 ist, wenn Knoten  $i$  und  $j$  über eine Kante verbunden sind, und sonst 0. Für eine Adjazenz-Matrix wird aber Speicherplatz proportional zu  $n^2$  benötigt.

Oft sind den Knoten und/oder Kanten eines Graphen Werte zugeordnet. Diese Werte werden als *Gewichte* bezeichnet. Zum Beispiel kann jeder Kante durch eine Funktion  $cap : E \rightarrow \mathbb{R}^+$  eine Kapazität zugeordnet sein, die die Bandbreite des entsprechenden Links angibt.

Ein (einfacher) *Pfad* (oder *Weg*) von  $u$  nach  $v$  in einem gerichteten Graphen  $G = (V, E)$  ist eine Folge  $(x_0, x_1), (x_1, x_2), \dots, (x_{\ell-1}, x_\ell)$  von aufeinanderfolgenden Kanten, wobei  $x_0 = u$ ,  $x_\ell = v$  und jeder Knoten auf dem Pfad nur einmal besucht wird. Werden auf einem Pfad manche Knoten mehrmals besucht, so heisst der Pfad *nicht einfach*. Wenn den Kanten des Graphen keine Gewichte zugeordnet sind, so ist die Länge eines Pfades einfach die Anzahl der Kanten auf dem Pfad. Wenn Kantengewichte vorhanden sind, ist mit der Länge eines Pfades dagegen oft die Summe der Gewichte aller Kanten auf dem Pfad gemeint. Aus dem Zusammenhang wird immer klar, wie der Begriff ‐Länge eines Pfades‐ jeweils zu verstehen ist.

Ein (einfacher) *Kreis* (oder *Zyklus*) in einem gerichteten Graphen  $G = (V, E)$  ist eine Folge  $(x_0, x_1), (x_1, x_2), \dots, (x_{\ell-1}, x_\ell)$  von aufeinanderfolgenden Kanten, wobei  $x_0 = x_\ell$  und  $x_i \neq x_j$  für  $i \neq j$ ,  $0 \leq i, j < \ell$ , gilt.

Pfade und Kreise in ungerichteten Graphen sind analog definiert. Die Richtung der Kanten spielt hier keine Rolle.

Ein ungerichteter Graph heisst *zusammenhängend* (abgekürzt: zshgd.), wenn es von jedem Knoten zu jedem anderen Knoten einen Pfad gibt. Ein gerichteter Graph heisst *stark zusammenhängend*, wenn es von jedem Knoten zu jedem anderen Knoten einen (gerichteten) Pfad gibt. Der gerichtete Graph in Abb. 1.1(Mitte) ist zum Beispiel nicht stark zusammenhängend.

Ein Graph heisst *vollständig*, wenn jede mögliche Kante tatsächlich im Graphen enthalten ist, d.h. wenn alle Paare von Knoten durch Kanten verbunden sind.

Ein Graph  $G' = (V', E')$  ist ein *Teilgraph* von  $G = (V, E)$ , wenn  $V' \subseteq V$  und  $E' \subseteq E$  gilt.  $G'$  heisst *induzierter Teilgraph* von  $G$ , wenn  $E'$  alle Kanten in  $E$  enthält, die zwei Knoten aus  $V'$  miteinander verbinden. Wir sagen dann auch, dass  $G'$  der durch  $V'$  induzierte Teilgraph von  $G$  ist.

Ein ungerichteter Graph ist ein *Baum*, wenn er zusammenhängend ist und keine Kreise enthält. Ein *Spannbaum* eines ungerichteten Graphen  $G = (V, E)$  ist ein Baum mit Knotenmenge  $V$ , der ein Teilgraph von  $G$  ist. Die Knoten eines Baumes, die Grad 1 haben, heissen *Blätter*. Wenn ein Knoten des Baumes als *Wurzel* ausgezeichnet ist, so sprechen wir auch von einem *gewurzelten Baum*. In einem gewurzelten Baum hat jeder Knoten  $v$  ausser der Wurzel genau einen *Vaterknoten*, nämlich den Knoten, der zu  $v$  benachbart ist und auf dem eindeutigen Pfad von  $v$  zur Wurzel liegt. Alle anderen Nachbarn eines Knotens sind seine *Kinder*.

Wenn ein ungerichteter Graph nicht zusammenhängend ist, so heissen die maximal grossen Teilgraphen, die zusammenhängend sind, die *Zusammenhangskomponenten* des Graphen.

## 1.2 Algorithmen

Allgemein gesprochen ist ein Algorithmus eine Angabe von Regeln, mit denen ein Problem in einer endlichen Zahl von Schritten gelöst werden kann. Dabei muss der Algorithmus durch eine präzise endliche Beschreibung gegeben sein und darf nur effektive (tatsächlich ausführbare) Verarbeitungsschritte verwenden. Wir interessieren uns für Algorithmen, die als effiziente Computer-Programme implementiert werden können. Das Wort "Algorithmus" ist vom Namen "al-kawarizmi" eines persischen Mathematikers aus dem neunten Jahrhundert abgeleitet.

Wir wollen vor allem Algorithmen betrachten, die zu Beginn eine *Eingabe* erhalten, dann Berechnungen mit den Eingabedaten ausführen und schliesslich eine *Ausgabe* produzieren. Dabei sagen wir, dass ein Algorithmus für ein Problem  $P$  eine *Instanz* von  $P$  als Eingabe bekommt und eine *Lösung* für diese Instanz berechnet. Als Beispiel können wir uns einen Algorithmus für das Sortieren von Zahlen denken: er bekommt als Eingabe eine Liste von Zahlen und produziert als Ausgabe eine sortierte Liste derselben Zahlen.

Um einen Algorithmus anzugeben, werden wir eine Pseudo-Programmiersprache verwenden, die auch natürlichsprachliche Anweisungen erlaubt. Die Implementierung eines so angegebenen Algorithmus in einer Programmiersprache wie C++ oder Java sollte keine Probleme bereiten.

### 1.2.1 Laufzeit von Algorithmen

Neben der Korrektheit eines Algorithmus ist vor allem seine Laufzeit von Interesse. Die Laufzeit eines Algorithmus auf einer Eingabe  $I$  ist die Anzahl elementarer Rechenschritte, die der Algorithmus bei dieser Eingabe ausführt. Als elementarer Rechenschritt sind dabei im wesentlichen Operationen der Art zugelassen, wie sie von gegenwärtigen Mikroprozessoren mit einer Instruktion ausgeführt werden können.

In der Regel interessiert uns die Worst-Case-Laufzeit in Abhängigkeit von der Grösse der Eingabe. Die Worst-Case-Laufzeit eines Algorithmus ist durch eine Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  beschränkt, wenn der Algorithmus auf allen Eingaben der Grösse  $n$  Laufzeit höchstens  $f(n)$  hat.

Wenn die Eingabe ein Graph ist, so betrachten wir die Laufzeit meist in Abhängigkeit von  $n$  und  $m$  (Knotenzahl und Kantenzahl).

Zur Angabe von Laufzeiten verwenden wir die übliche  $O()$ -Notation. Für zwei Funktionen  $f : \mathbb{N} \rightarrow \mathbb{N}$  und  $g : \mathbb{N} \rightarrow \mathbb{N}$  schreiben wir

$$f(n) = O(g(n)),$$

falls es Konstanten  $c > 0$  und  $n_0 \in \mathbb{N}$  gibt, so dass für alle  $n \geq n_0$  gilt:  $f(n) \leq c \cdot g(n)$ . Man kann “ $f(n) = O(g(n))$ ” also interpretieren als:  $f(n)$  ist kleiner gleich  $g(n)$  bis auf einen konstanten Faktor.

**Beispiel:** Sei  $f(n) \leq 17n^2 + 5n + 10$ . Für alle  $n \geq 1$  gilt  $f(n) \leq 17n^2 + 5n^2 + 10n^2 = 32n^2$ , daher ist  $f(n) = O(n^2)$ . (Wir wählen zum Beispiel  $c = 32$  und  $n_0 = 1$ .)  $\blacklozenge$

Zu beachten ist bei der  $O()$ -Notation, dass “=” hier von links nach rechts gelesen werden muss. Die Rechnung  $O(n^2) = O(n^3)$  ist zum Beispiel richtig, während die Behauptung  $O(n^3) = O(n^2)$  falsch ist.

Ein Algorithmus hat *polynomielle Laufzeit*, falls seine Laufzeit  $O(n^k)$  für eine Konstante  $k$  ist. In der Praxis sind natürlich Algorithmen mit möglichst kleiner Laufzeit interessant, der Exponent  $k$  sollte bei polynomiellen Algorithmen also möglichst nicht grösser als 2 oder 3 sein. Ein Algorithmus mit Laufzeit  $O(n)$  hat *lineare Laufzeit*, ein Algorithmus mit Laufzeit  $O(n^2)$  *quadratische Laufzeit*.

## 1.2.2 Algorithmen für Optimierungsprobleme

Viele der uns interessierenden Probleme sind Optimierungsprobleme. Für eine Instanz  $I$  eines Optimierungsproblems gibt es in der Regel viele verschiedene (*zulässige*) *Lösungen*, und das Ziel ist die Berechnung einer Lösung, die eine gewisse Zielfunktion minimiert oder maximiert. Der optimale Wert der Zielfunktion wird dabei mit *OPT* bezeichnet.

**Beispiel:** Eine Instanz des *Problems des Handlungsreisenden* (Traveling Salesperson Problem, TSP) ist ein vollständiger Graph  $G = (V, E)$  mit Kantengewichten  $c : E \rightarrow \mathbb{R}$ . Eine zulässige Lösung ist ein Kreis in  $G$  (eine *Tour*), der alle Knoten genau einmal besucht. Das Optimierungskriterium ist, das Gesamtgewicht aller Kanten der Tour zu minimieren.  $\blacklozenge$

Wir beschäftigen uns vor allem mit zwei Arten von Algorithmen für Optimierungsprobleme:

- Ein *exakter Algorithmus* für ein Optimierungsproblem  $P$  berechnet für jede Instanz von  $P$  eine Lösung mit optimalem Zielfunktionswert.
- Ein  $\rho$ -*Approximationsalgorithmus* für ein Optimierungsproblem  $P$  hat polynomielle Laufzeit und berechnet für jede Instanz  $I$  von  $P$  eine Lösung, deren Zielfunktionswert  $A$  vom optimalen Wert  $OPT$  höchstens um Faktor  $\rho$  abweicht. Bei Minimierungsproblemen gilt also  $A \leq \rho OPT$ , bei Maximierungsproblemen  $A \geq OPT/\rho$ . Der Faktor  $\rho$  wird auch *Approximationsrate* genannt.

Viele Optimierungsprobleme sind  $\mathcal{NP}$ -schwer und es ist kein exakter Algorithmus mit polynomieller Laufzeit für sie bekannt. Wir wollen hier nicht genauer auf die Komplexitätsklasse  $\mathcal{NP}$  eingehen, sondern nur kurz anmerken, dass ein polynomieller Algorithmus für irgendein  $\mathcal{NP}$ -schweres Problem sofort polynomielle Algorithmen für alle Probleme in  $\mathcal{NP}$  liefern würde

(z.B. für das Problem des Handlungsreisenden). Dies gilt als sehr unwahrscheinlich, es konnte aber bisher niemand beweisen, dass es keine polynomiellen Algorithmen für  $\mathcal{NP}$ -schwere Probleme gibt. Dies ist das berühmte offene Problem “ $\mathcal{P} = \mathcal{NP}$  oder  $\mathcal{P} \neq \mathcal{NP}$ ?”.

Approximationsalgorithmen sind interessant, da sie für  $\mathcal{NP}$ -schwere Optimierungsprobleme in vertretbarer Zeit beweisbar gute Lösungen liefern.

Manche  $\mathcal{NP}$ -schwere Probleme kann man sogar beliebig gut approximieren. Für jede vorgegebene Konstante  $\varepsilon > 0$  erhält man dann einen  $(1 + \varepsilon)$ -Approximationsalgorithmus. Die Laufzeit des Algorithmus wird natürlich länger, je kleiner man  $\varepsilon$  wählt, für jedes feste  $\varepsilon$  ist sie aber durch ein Polynom beschränkt. Ein solcher Algorithmus, der einen zusätzlichen Parameter  $\varepsilon > 0$  als Eingabe bekommt und dann eine  $(1 + \varepsilon)$ -Approximation berechnet, heisst *polynomielles Approximationsschema* (PTAS, polynomial-time approximation scheme). Wenn die Laufzeit des Algorithmus auch in  $\frac{1}{\varepsilon}$  polynomiell ist, dann erhält man ein sogenanntes FPTAS (fully polynomial-time approximation scheme).

In manchen Anwendungen ist die Eingabe nicht vollständig im voraus bekannt, z.B. wenn die Eingabe aus Verbindungsanfragen in einem Kommunikationsnetz besteht. Hier muss ein Algorithmus, der die Zugangskontrolle erledigt, für jede ankommende Anfrage sofort entscheiden, ob und wie die Verbindung eingerichtet werden soll. Dabei hat der Algorithmus keine Kenntnis davon, welche Anfragen in Zukunft noch gestellt werden. Algorithmen dieser Art werden *Online-Algorithmen* genannt. Auch hier vergleicht man den Zielfunktionswert der vom Algorithmus berechneten Lösung mit dem optimalen Zielfunktionswert (den ein Algorithmus ohne Beschränkung der Rechenzeit berechnen hätte können, wenn er alle gestellten Anfragen im Voraus gekannt hätte). Bei Online-Algorithmen wird die Approximationsrate in der Regel *kompetitive Rate* (engl. *competitive ratio*) genannt.

Neben Approximationsalgorithmen und Online-Algorithmen gibt es weitere Ansätze zur Lösung  $\mathcal{NP}$ -schwerer Optimierungsprobleme, mit denen wir uns jedoch in dieser Vorlesung nicht weiter beschäftigen werden. Ein solcher Ansatz ist die Verwendung von *Heuristiken*, d.h. von Verfahren zur Berechnung von Lösungen, bei denen man zwar keine guten Schranken für die erzielten Approximationsraten beweisen kann, die aber in der Praxis oft erstaunlich gut funktionieren. Neben speziell auf ein Problem zugeschnittenen Heuristiken gibt es auch sogenannte *Meta-Heuristiken*, die auf viele verschiedene Optimierungsprobleme angewendet werden können. Dazu zählen Simulated Annealing, Genetische Algorithmen (Evolutionäre Algorithmen), lokale Suche und Tabu Search. Ein anderer Ansatz ist die Berechnung optimaler Lösungen auch für  $\mathcal{NP}$ -schwere Optimierungsprobleme. Hier versucht man, Algorithmen zu entwickeln, die für in der Praxis auftretende Instanzen in vertretbarer Zeit optimale Lösungen berechnen, auch wenn keine guten Schranken für die Worst-Case-Laufzeit angegeben werden können. Schlagworte hier sind zum Beispiel Branch&Bound und Branch&Cut.

## 1.3 Grundlegende Graphenalgorithmien

Hier geben wir einen kurzen Überblick über grundlegende Algorithmen für Graphen, die wir zum Teil in den nachfolgenden Kapiteln als Unterroutinen verwenden werden.

### 1.3.1 Kürzeste Pfade

Sei  $G = (V, E)$  ein gerichteter Graph mit Kantengewichten  $c : E \rightarrow \mathbb{R}$ . Die Länge eines gerichteten Pfades von  $u$  nach  $v$  ist die Summe der Gewichte aller Kanten auf dem Pfad. Das *single-source shortest paths*-Problem besteht darin, für einen gegebenen Startknoten  $s \in V$

**Algorithmus von Kruskal****Eingabe:** zshgd. ungerichteter Graph  $G = (V, E)$  mit Kantengewichten  $c : E \rightarrow \mathbb{R}$ **Ausgabe:** Spannbaum  $T = (V, E')$  minimalen Gewichts $E' := \emptyset;$ **for** alle Kanten  $e = \{u, v\} \in E$  in Reihenfolge aufsteigenden Gewichts **do**    **if**  $u$  und  $v$  sind in verschiedenen Zusammenhangskomponenten von  $G' = (V, E')$  **then**         $E' := E' \cup \{e\};$     **fi;****od;****return**  $T = (V, E');$ 

Abbildung 1.2: Der Algorithmus von Kruskal

die (Länge der) kürzesten Pfade zu allen anderen Knoten zu berechnen. Falls die Kantengewichte nicht-negativ sind, so kann das Problem mit dem Algorithmus von Dijkstra in Zeit  $O(m + n \log n)$  gelöst werden. Falls auch negative Kantengewichte vorkommen, so kann der Algorithmus von Bellman und Ford verwendet werden, der Laufzeit  $O(nm)$  hat. Hier muss aber vorausgesetzt werden, dass der Graph keine Kreise negativer Länge enthält, da sonst das Kürzeste-Pfade-Problem nicht sinnvoll definiert ist.

**1.3.2 Minimale Spannbäume**

Sei  $G = (V, E)$  ein ungerichteter Graph mit Kantengewichten  $c : E \rightarrow \mathbb{R}$ . Ein *minimaler Spannbaum* in  $G$  ist ein Spannbaum  $T = (V, E')$ , so dass  $\sum_{e \in E'} c(e)$  minimal ist. Der Algorithmus von Prim berechnet einen minimalen Spannbaum in Zeit  $O(m + n \log n)$ , der Algorithmus von Kruskal benötigt dazu Zeit  $O(m \log m) = O(m \log n)$  (da  $m \leq n^2$ , gilt  $\log m \leq 2 \log n$  und damit  $O(\log m) = O(\log n)$ ).

Der Algorithmus von Kruskal ist in Abb. 1.2 dargestellt. Dabei sind zwei Implementierungsdetails nicht näher spezifiziert: Erstens ist nicht angegeben, wie die Bearbeitung der Kanten in der Reihenfolge aufsteigenden Gewichts erfolgen soll; dies kann z.B. realisiert werden, indem die Kanten zuerst mit einem  $O(m \log m)$ -Sortieralgorithmus sortiert werden. Zweitens ist nicht angegeben, wie geprüft werden soll, ob  $u$  und  $v$  in derselben Zusammenhangskomponente des Graphen mit Kantenmenge  $E'$  sind; hier kann z.B. eine sogenannte Union-Find-Datenstruktur eingesetzt werden. Wichtig ist, dass die angegebene Gesamtlaufzeit  $O(m \log m)$  nur dann erreicht wird, wenn diese in Abb. 1.2 nicht spezifizierten Details effizient implementiert werden.

**Anwendungsbeispiel:** *Ein Problem, das durch die Berechnung eines minimalen Spannbauams gelöst werden kann, ist das folgende: Es ist ein Kommunikationsnetz aufzubauen, das eine gegebene Menge von Knotenpunkten zusammenhängend verbindet. Dabei kann zwischen bestimmten Paaren von Knotenpunkten eine Leitung eingerichtet werden, deren Kosten von den Endpunkten der Leitung abhängen. Das Problem, eine möglichst billige Menge von einzurichtenden Leitungen auszuwählen, die aber ausreicht, das Netzwerk zusammenhängend zu machen, entspricht genau dem Problem, einen minimalen Spannbaum zu berechnen. Dabei repräsentiert  $V$  die Menge der Knotenpunkte,  $E$  die Menge der möglichen Verbindungsleitungen und  $c(\{v, w\})$  die Kosten für das Einrichten einer Verbindung von  $v$  nach  $w$ .*

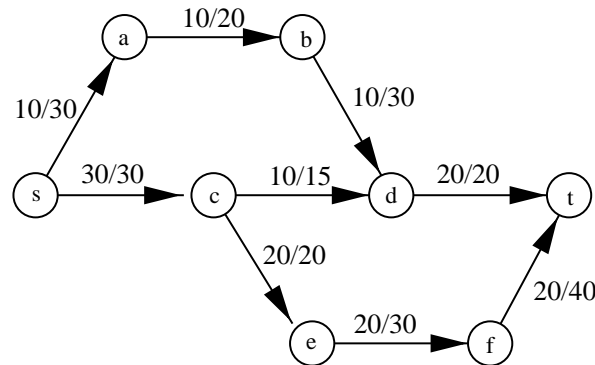


Abbildung 1.3: Beispiel für Netzwerk-Fluss-Problem

Ein anderer Algorithmus zur Berechnung minimaler Spannbäume ist der Algorithmus von Prim. Wir werden ihn in Abschnitt 2.1 kennen lernen.

### 1.3.3 Netzwerk-Fluss

Ein grundlegendes Problem in der kombinatorischen Optimierung, das sehr viele Anwendungen besitzt, ist das Netzwerk-Fluss-Problem. Gegeben ist ein gerichteter Graph  $G = (V, E)$  mit zwei ausgezeichneten Knoten  $s \in V$  (*source*, Quelle) und  $t \in V$  (*target*, Senke) sowie Kantenkapazitäten  $cap : E \rightarrow \mathbb{R}^+$ . Ein *Fluss* in  $G$  ist eine Funktion  $f : E \rightarrow \mathbb{R}$ , die die folgenden beiden Bedingungen erfüllt:

- $0 \leq f(e) \leq cap(e)$  für alle  $e \in E$  (Einhaltung der Kapazitäten)
- $\sum_{e=(u,v) \in E} f(e) = \sum_{e=(v,w) \in E} f(e)$  für alle  $v \in V \setminus \{s, t\}$  (Flusserhaltung)

Man kann sich die Kanten zum Beispiel als Öl-Pipelines vorstellen. Die Kapazität einer Kante gibt dann an, wie viel Öl pro Zeiteinheit durch die Kante maximal fließen kann. Die Flusserhaltungsregel bestimmt, dass bei jedem Knoten (ausser der Quelle und der Senke) genau so viel Öl ankommen muss wie von dem Knoten wegfließt.

Der Wert eines Flusses  $f$  ist definiert als

$$\bar{f} = \sum_{e=(s,w) \in E} f(e) - \sum_{e=(u,s) \in E} f(e),$$

also als der Gesamtfluss, der die Quelle verlässt (und der aufgrund der Flusserhaltung auch bei der Senke ankommt). Das Ziel beim Netzwerk-Fluss-Problem ist, einen Fluss maximalen Wertes zu berechnen.

Ein Beispiel für eine Instanz des Netzwerk-Fluss-Problems und einen zugehörigen Fluss ist in Abb. 1.3 dargestellt. Jede Kante  $e$  hat als Label die Werte  $f(e)/cap(e)$ . Der dargestellte Fluss  $f$  hat den Wert  $\bar{f} = 40$  und ist optimal für dieses Beispiel.

Jede Menge  $S \subset V$  mit  $s \in S$  und  $t \in V \setminus S$  bestimmt einen *Schnitt*  $(S, V \setminus S)$ . Die *Kapazität* eines Schnittes  $(S, V \setminus S)$  ist die Summe der Kapazitäten aller Kanten, die von einem Knoten in  $S$  zu einem Knoten in  $V \setminus S$  laufen. Klar ist, dass der maximale Wert eines Flusses nicht grösser als die minimale Kapazität eines Schnittes sein kann (denn der ganze

Fluss muss ja irgendwie von  $S$  nach  $V \setminus S$  gelangen). Ein klassisches Ergebnis von Ford und Fulkerson ist das sogenannte Max-Flow-Min-Cut-Theorem.

**Satz 1.1 (Ford und Fulkerson, 1956)** *Der maximale Wert eines Flusses in  $G$  ist gleich der minimalen Kapazität eines Schnittes in  $G$ .*

In dem Graphen aus Abb. 1.3 ist der Schnitt  $(\{s, a, b, c, d\}, \{e, f, t\})$  ein Schnitt minimaler Kapazität. Seine Kapazität ist 40. Das zeigt, dass der angegebene Fluss tatsächlich optimal ist.

Es gibt eine ganze Reihe von Algorithmen, die in polynomieller Zeit einen Fluss maximalen Wertes berechnen. Klassische Algorithmen sind zum Beispiel der  $O(n^2m)$ -Algorithmus von Dinits (1970) und der  $O(n^3)$ -Algorithmus von Karzanov (1974). Der bis jetzt theoretisch schnellste Algorithmus (zumindest bei nicht zu grossen Kantenkapazitäten) wurde von Goldberg und Rao (1998) entwickelt und hat Laufzeit  $O(\min\{m^{3/2} \log(n^2/m) \log U, n^{2/3} m \log(n^2/m) \log U\})$ , wobei  $U$  die maximale Kantenkapazität ist.

Eine interessante und oft nützliche Tatsache ist, dass es immer einen maximalen Fluss mit ganzzahligen Werten  $f(e)$  für alle Kanten  $e \in E$  gibt, vorausgesetzt, dass die Kantenkapazitäten  $cap(e)$  alle ganzzahlig sind. Die oben erwähnten Algorithmen liefern in diesem Fall auch tatsächlich nur ganzzahlige Werte  $f(e)$ .

## 1.4 Lineare Programmierung

Ein *lineares Programm* (LP) ist durch eine Menge linearer Ungleichungen und eine lineare Zielfunktion gegeben. Üblicherweise wird es in der folgenden Form geschrieben (das “s.t.” steht dabei für “such that”):

$$\min \quad c^T x \tag{1.1}$$

$$\text{s.t.} \quad Ax \leq b \tag{1.2}$$

$$x \geq 0 \tag{1.3}$$

$x$  ist dabei ein  $n$ -dimensionaler Vektor aus Variablen  $x_1, x_2, \dots, x_n$ . Die Matrix  $A$  ist eine  $m \times n$ -Matrix,  $b$  ein Vektor mit  $m$  Komponenten und  $c$  ein Vektor mit  $n$  Komponenten. Es gilt also  $A \in \mathbb{R}^{m,n}$ ,  $b \in \mathbb{R}^m$ ,  $c \in \mathbb{R}^n$ .

Jeder Vektor  $x \in \mathbb{R}^n$ , der (1.2) und (1.3) erfüllt, ist eine *zulässige Lösung* des linearen Programms. Eine *optimale Lösung* ist eine zulässige Lösung  $x$ , die den Wert  $c^T x$  unter allen zulässigen Lösungen minimiert. Es gibt Algorithmen, die für lineare Programme in polynomieller Zeit eine Optimallösung berechnen: den vor allem aus theoretischer Sicht interessanten Ellipsoid-Algorithmus von Khachiyan (1979) und die auch in der Praxis gut funktionierende Innere-Punkte-Methode von Karmarkar (1984). Ferner gibt es den Simplex-Algorithmus von Dantzig (1947), der zwar im Worst-Case exponentielle Laufzeit hat, der aber bei vielen praktischen Problemen sehr schnell läuft. Es existieren Programmbibliotheken, z.B. das kommerzielle Tool CPLEX, die auch sehr grosse lineare Programme in verblüffend kurzer Zeit lösen können.

Lineare Programmierung erlaubt die Optimierung (Maximierung oder Minimierung) einer linearen Zielfunktion über einem Bereich zulässiger Lösungsvektoren, der durch beliebige lineare Gleichungen und Ungleichungen gegeben ist. Die Komponenten des unbekanntes Vektors sind dabei rationale Zahlen.

Wird in einem linearen Programm zusätzlich gefordert, dass (manche der) Variablen nur ganzzahlige Werte annehmen dürfen, so erhält man ein *ganzzahliges* oder *gemischt-ganzzahliges* lineares Programm (engl. *mixed-integer linear program*, *MILP* oder *ILP*). Das Bestimmen der Optimallösung eines (gemischt)-ganzzahligen Programms ist jedoch im allgemeinen  $\mathcal{NP}$ -schwer.

Wir werden häufig mit Problemen zu tun haben, die sich leicht als ILP formulieren lassen. Um eine Approximationslösung zu bestimmen, ist es dann oft hilfreich, die Ganzzahligkeitsbedingung zu ignorieren und die sogenannte *LP-Relaxierung* optimal zu lösen. Aus der so erhaltenen fraktionalen Lösung (d.h. einem Lösungsvektor, dessen Komponenten rationale Zahlen sind) versucht man dann, mittels gewisser Rundungstechniken eine ganzzahlige Lösung zu konstruieren, ohne dabei den Zielfunktionswert viel schlechter zu machen.

## 1.5 Literaturhinweise

Algorithmen und Worst-Case-Laufzeit wurden hier recht informell eingeführt, insbesondere wurde das Maschinenmodell nicht im Detail definiert. Auch der Begriff  $\mathcal{NP}$ -schwer wurde nicht weiter erklärt. Mehr zu diesen Themen findet man in Büchern zur Komplexitätstheorie, z.B. [Pap94]. Das Thema  $\mathcal{NP}$ -Vollständigkeit wird umfassend in [GJ79] behandelt.

Ein umfassendes und sehr gut lesbares Standardwerk zum Thema effizienter Algorithmen ist [CLR90]. Auch das in deutscher Sprache erschienene Buch [OW90] ist empfehlenswert. Mit dem Thema Netzwerk-Fluss beschäftigt sich das Buch [AMO93]. Für eine Einführung in kombinatorische Optimierung und lineare Programmierung (Simplex-Algorithmus) eignet sich [PS82]. Eine erschöpfende Behandlung des Themas der linearen Programmierung und der ganzzahligen linearen Programmierung findet man in [Sch86]. Ein aktuelles Buch über kombinatorische Optimierung ist [CCPS98].

Einen guten Überblick über das weite Feld der Approximationsalgorithmen bieten die Bücher [ACG<sup>+</sup>99] und [Hoc97]. Online-Algorithmen werden in [FW98] und [BEY98] behandelt.

### Referenzen

- [ACG<sup>+</sup>99] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation. Combinatorial Optimization Problems and their Approximability Properties*. Springer, Berlin, 1999.
- [AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, New Jersey, 1993.
- [BEY98] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
- [CCPS98] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. John Wiley and Sons, New York, 1998.
- [CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, 1990.
- [FW98] Amos Fiat and Gerhard J. Woeginger, editors. *Online Algorithms: The State of the Art*. LNCS 1442. Springer-Verlag, Berlin, 1998.

- [GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York-San Francisco, 1979.
- [Hoc97] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, MA, 1997.
- [OW90] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen*. B.I. Wissenschaftsverlag, Mannheim, 1990.
- [Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, 1994.
- [PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.
- [Sch86] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Chichester, 1986.

## Kapitel 2

# Minimale Spann­bäume und Steiner-Bäume

### 2.1 Minimale Spann­bäume

Es sei ein ungerichteter, zusammenhängender Graph  $G = (V, E)$  mit  $|V| = n$  Knoten,  $|E| = m$  Kanten und Kantengewichten  $c : E \rightarrow \mathbb{R}^+$  gegeben. Ein Spannbaum von  $G$  ist ein Teilgraph  $T = (V, E')$  von  $G$  mit  $E' \subseteq E$ , der zusammenhängend ist und keine Kreise enthält. Das Gewicht eines Spannbaums ist gleich der Summe der Gewichte aller Kanten in dem Spannbaum, also  $\sum_{e \in E'} c(e)$ . Ein Spannbaum  $T$  ist ein *minimaler Spannbaum*, falls kein anderer Spannbaum von  $G$  ein kleineres Gewicht hat als  $T$ . Ein Graph kann mehrere unterschiedliche minimale Spann­bäume haben.

Ein Problem, das durch die Berechnung eines minimalen Spannbaums gelöst werden kann, ist das folgende: Es ist ein Kommunikationsnetzwerk aufzubauen, das eine gegebene Menge von Knotenpunkten zusammenhängend verbindet. Dabei kann zwischen bestimmten Paaren von Knotenpunkten eine Leitung eingerichtet werden, deren Kosten von den Endpunkten der Leitung abhängen. Das Problem, eine möglichst billige Menge von einzurichtenden Leitungen auszuwählen, die aber ausreicht, das Netzwerk zusammenhängend zu machen, entspricht genau dem Problem, einen minimalen Spannbaum zu berechnen. Dabei repräsentiert  $V$  die Menge der Knotenpunkte,  $E$  die Menge der möglichen Verbindungsleitungen und  $c(\{v, w\})$  die Kosten für das Einrichten einer Verbindung von  $v$  nach  $w$ .

Viele Algorithmen zur Berechnung eines minimalen Spannbaums, beginnen mit einer leeren Kantenmenge  $E' = \emptyset$  und fügen nacheinander Kanten in  $E'$  ein, bis schliesslich  $E'$  genau die Kantenmenge eines minimalen Spannbaums ist. Dabei werden zu jedem Zeitpunkt die Knoten des Graphen  $G$  durch die Kanten, die schon in  $E'$  eingefügt wurden, in eine Menge von Teilbäumen partitioniert. Anfangs, wenn  $E' = \emptyset$  gilt, besteht jeder dieser Teilbäume aus einem einzigen Knoten. Durch jedes Einfügen einer Kante  $e$  in  $E'$  werden die zwei Teilbäume, die die Endknoten von  $e$  enthalten, zu einem einzigen Teilbaum vereinigt. Damit keine Kreise entstehen, dürfen nur Kanten in  $E'$  eingefügt werden, die Knoten aus verschiedenen Teilbäumen verbinden. Am Ende, d.h. nach  $n - 1$  Einfügungen, bleibt genau ein Baum übrig. Die Regel, nach der entschieden wird, welche Kanten jeweils in  $E'$  eingefügt werden sollen, muss so gewählt sein, dass der resultierende Spannbaum tatsächlich minimales Gewicht hat.

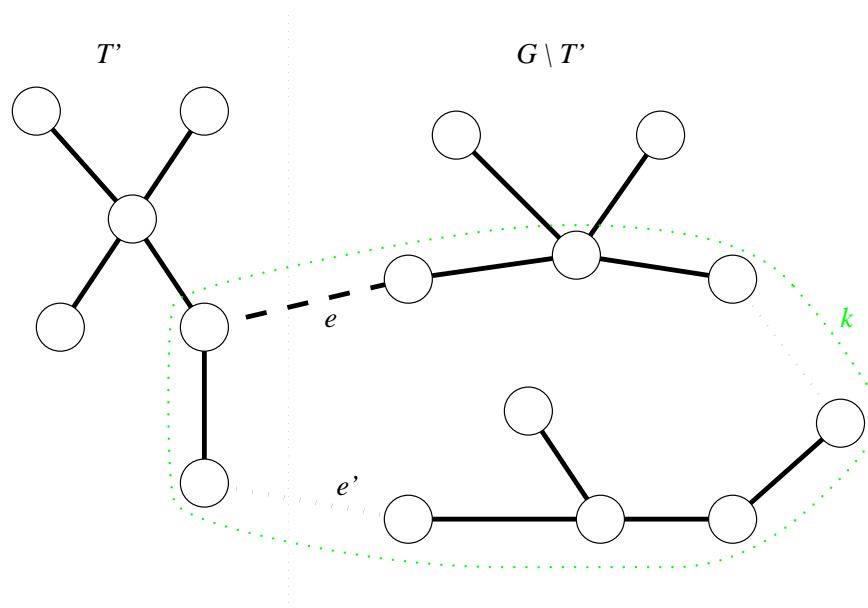


Abbildung 2.1: Skizze zum Beweis von Satz 2.1

**Satz 2.1** Sei  $E' \subset E$  so, dass es einen minimalen Spannbaum  $T$  von  $G$  gibt, der alle Kanten in  $E'$  enthält. Sei  $T'$  einer der Teilbäume, die durch  $E'$  gegeben sind. Sei  $e$  eine Kante mit minimalem Gewicht unter allen Kanten, die einen Knoten aus  $T'$  mit einem Knoten aus  $G \setminus T'$  verbinden. Dann gibt es auch einen minimalen Spannbaum, der alle Kanten aus  $E' \cup \{e\}$  enthält.

**Beweis.** Wie im Satz bezeichne  $e$  die aktuell eingefügte Kante,  $E'$  die aktuelle Kantenmenge vor Einfügen von  $e$ ,  $T'$  den aktuellen Teilbaum, aus dem die Kante  $e$  herausführt, und  $T$  einen minimalen Spannbaum von  $G$ , der  $E'$  enthält. Es ist zu zeigen, dass es auch einen minimalen Spannbaum von  $G$  gibt, der  $E' \cup \{e\}$  enthält.

Falls  $e$  in  $T$  enthalten ist, so ist  $T$  der gesuchte minimale Spannbaum und wir sind fertig. Nehmen wir also an, dass  $e$  nicht in  $T$  enthalten ist. Dann entsteht durch Einfügen von  $e$  in  $T$  ein Kreis  $k$ , der sowohl Knoten in  $T'$  als auch Knoten in  $G \setminus T'$  enthält. Folglich muss  $k$  ausser  $e$  noch mindestens eine weitere Kante  $e'$  enthalten, die einen Knoten in  $T'$  mit einem Knoten in  $G \setminus T'$  verbindet, also aus  $T'$  herausführt, siehe Abbildung 2.1. Da  $e$  minimales Gewicht unter allen solchen Kanten hatte, muss  $c(e') \geq c(e)$  gelten. Wenn wir nun aus  $T$  die Kante  $e'$  entfernen und die Kante  $e$  einfügen, so erhalten wir einen Spannbaum  $T''$ , dessen Gewicht nicht grösser als das von  $T$  ist. Da  $T$  ein minimaler Spannbaum ist, ist das Gewicht von  $T''$  gleich dem Gewicht von  $T$ . Da  $T''$  ausserdem  $E' \cup \{e\}$  enthält, ist  $T''$  der gesuchte minimale Spannbaum.  $\square$

Dieser Satz sagt uns, dass wir einen minimalen Spannbaum berechnen können, indem wir mit einer leeren Kantenmenge  $E'$  beginnen (diese ist auf jeden Fall Teilmenge der Kantenmenge eines minimalen Spannbaums) und in jedem Schritt eine Kante  $e$  auswählen und in  $E'$  aufnehmen, die unter allen Kanten, die aus einem der aktuellen Teilbäume herausführen, minimales Gewicht hat.

```

Algorithmus von Prim
Eingabe: zshgd. ungerichteter Graph  $G = (V, E)$  mit Kantengewichten  $c : E \rightarrow \mathbb{R}$ 
Ausgabe: minimale Spannbaum  $T = (V, E')$ 

 $s :=$  beliebiger Startknoten in  $V$ ;
 $ready[s] := true$ ;
 $ready[v] := false$  für alle  $v \in V \setminus \{s\}$ ;
p-queue  $Q$ ; { Priority-Queue für Knoten }
for alle inzidenten Kanten  $e = \{s, v\}$  von  $s$  do
     $from[v] := e$ ;
     $Q.Insert(v, c(e))$ ;
od;
 $E' := \emptyset$ ;
while  $|E'| < |V| - 1$  do
     $v := Q.DeleteMin()$ ;
     $E' := E' \cup \{from[v]\}$ ;
     $ready[v] := true$ ;
    for alle inzidenten Kanten  $e = \{v, w\}$  von  $v$  do
        if  $w \in Q$  und  $c(e) < c(from[w])$  then
             $from[w] := e$ ;
             $Q.DecreasePriority(w, c(e))$ ;
        else if  $w \notin Q$  und nicht  $ready[w]$  then
             $from[w] := e$ ;
             $Q.Insert(w, c(e))$ ;
        fi;
    od;
od;
return  $T = (V, E')$ ;

```

Abbildung 2.2: Der Algorithmus von Prim für minimale Spannbäume

### 2.1.1 Algorithmus von Prim

Der Algorithmus von Prim wählt einen beliebigen Knoten des Graphen  $G$  als Startknoten. Dann wird immer derjenige Teilbaum betrachtet, der den Startknoten enthält. Unter allen Kanten, die aus diesem Teilbaum herausführen, wird eine mit minimalem Gewicht ausgewählt und in  $E'$  eingefügt. Dadurch wird der Teilbaum, der den Startknoten enthält, um eine Kante und einen Knoten grösser. Zu jedem Zeitpunkt gibt es nur einen Teilbaum, der mehr als einen Knoten enthält. Dieser Teilbaum wächst in  $n - 1$  Schritten zu einem minimalen Spannbaum heran.

Offensichtlich erfüllt diese Auswahlregel zum Einfügen von Kanten in  $E'$  die Bedingung von Satz 1. Folglich liefert der Algorithmus von Prim einen minimalen Spannbaum.

Bei der Implementierung des Algorithmus von Prim (siehe Abb. 2.2) ist zu überlegen, wie in jedem Schritt die in  $E'$  einzufügende Kante effizient berechnet werden kann. Diese Kante muss minimales Gewicht haben unter allen Kanten, die einen Knoten aus dem aktuellen Teilbaum  $T$  mit einem Knoten aus  $G \setminus T$  verbinden. Hier erweist sich eine Priority-Queue (Prioritätswarteschlange) als Datenstruktur sehr nützlich. Eine Priority-Queue ist eine Da-

tenstruktur  $Q$ , die die folgenden Operationen effizient realisiert:

- $Q.\text{Insert}(e,p)$ : füge Element  $e$  mit Priorität  $p$  in  $Q$  ein
- $Q.\text{DecreasePriority}(e,p)$ : erniedrige die Priorität des bereits in  $Q$  befindlichen Elements  $e$  auf  $p$
- $Q.\text{DeleteMin}()$ : liefere das Element aus  $Q$  mit niedrigster Priorität zurück und lösche es aus  $Q$

Werden Priority-Queues mit Fibonacci-Heaps implementiert, so ist der *amortisierte* Zeitaufwand für Insert und DecreasePriority  $O(1)$  und für DeleteMin  $O(\log n)$ , wenn  $Q$  maximal  $n$  Elemente enthält. Hier bedeutet “amortisiert”, dass der Zeitaufwand im Durchschnitt (über alle Operationen gemittelt, die ein Algorithmus ausführt, der die Priority-Queue verwendet) die angegebenen Schranken erfüllt. Es kann zwar z.B. passieren, dass ein einzelnes DeleteMin() länger als  $O(\log n)$  dauert, dafür sind dann andere DeleteMin()-Aufrufe entsprechend schneller, so dass sich insgesamt ein durchschnittlicher Aufwand von  $O(\log n)$  ergibt. Werden zum Beispiel  $k$  DeleteMin()-Operationen ausgeführt, so ist die Gesamtlaufzeit immer durch  $O(k \log n)$  begrenzt.

Zur Implementierung des Algorithmus von Prim speichern wir in der Priority-Queue  $Q$  alle Knoten aus  $G \setminus T$ , die über eine Kante aus  $T$  heraus erreichbar sind. Sei  $v$  ein solcher Knoten aus  $G \setminus T$  und sei  $e_v$  eine Kante mit minimalem Gewicht unter allen Kanten, die einen Knoten aus  $T$  mit  $v$  verbinden. Dann soll die Priorität von  $v$  in  $Q$  gleich  $c(e_v)$  sein. Zusätzlich merken wir uns bei Knoten  $v$  in der Variablen  $from[v]$ , dass  $e_v$  seine billigste Kante zum bisher konstruierten Baum ist. Am Anfang initialisieren wir  $Q$ , indem wir alle Nachbarknoten des Startknotens in  $Q$  einfügen und ihnen als Priorität das Gewicht der betreffenden zum Startknoten inzidenten Kante geben. Ausserdem merken wir uns bei jedem solchen Knoten eben diese Kante als vorläufig günstigste Kante, um den Knoten vom Startknoten aus zu erreichen.

Wenn nun die nächste Kante zum Einfügen in  $E'$  ausgewählt werden soll, führen wir einfach die DeleteMin-Operation der Priority-Queue aus; dadurch bekommen wir den Knoten  $v$  aus  $G \setminus T$ , der von  $T$  aus über eine Kante mit minimalem Gewicht, nämlich die Kante  $e_v$ , erreichbar ist. Wir fügen  $e_v = from[v]$  in  $E'$  ein und müssen zusätzlich einige Daten aktualisieren; da  $v$  nämlich jetzt in  $T$  ist, kann es Nachbarn  $w$  von  $v$  geben, die noch nicht in  $T$  enthalten sind und die entweder vorher überhaupt nicht von  $T$  aus über eine Kante erreichbar waren oder nur über eine Kante mit höherem Gewicht als  $c(\{v, w\})$ . In ersterem Fall fügen wir  $w$  mit Priorität  $c(\{v, w\})$  in  $Q$  ein; in letzterem Fall erniedrigen wir die Priorität von  $w$  auf  $c(\{v, w\})$  mittels einer DecreasePriority-Operation. In beiden Fällen merken wir uns bei Knoten  $w$  in der Variablen  $from[w]$ , dass  $w$  nun von  $T$  aus am günstigsten über die Kante  $e_w = \{v, w\}$  erreicht werden kann.

Der Algorithmus von Prim berechnet den minimalen Spannbaum in  $n - 1$  Schritten, wobei in jedem Schritt durch eine DeleteMin-Operation ein neuer Spannbaumknoten  $v$  gewählt wird und dann alle Nachbarn von  $v$  ggf. durch eine Insert- oder DecreasePriority-Operation in die Queue eingefügt oder bzgl. ihrer Priorität aktualisiert werden müssen. Ohne Berücksichtigung der Operationen auf der Priority-Queue ergibt sich damit ein Zeitaufwand von  $O(n + m) = O(m)$ . Da die Insert- und DecreasePriority-Operationen (amortisiert) Zeitbedarf  $O(1)$  und die DeleteMin-Operation Zeitbedarf  $O(\log n)$  haben und  $n - 1$  Insert-Operationen,  $n - 1$  DeleteMin-Operationen und  $O(m)$  DecreasePriority-Operationen ausgeführt werden, ergibt sich insgesamt eine Laufzeit von  $O(m + n \log n)$ .

## 2.2 Berechnung von Steiner-Bäumen

Bei einer Multicast-Verbindung werden in einem Kommunikationsnetz Daten von einem Sender gleichzeitig an viele verschiedene Empfänger übertragen. Dies ist zum Beispiel dann sinnvoll, wenn bei einem Video-On-Demand-Service viele Kunden zur selben Zeit denselben Film sehen wollen. Um das Netz nicht zu überlasten ist es wichtig, die Wege zu optimieren, entlang derer die Daten die Empfänger erreichen. Diese Wege ergeben zusammen einen Baum, dessen Wurzel der Sender ist und in dem alle Empfänger als Blätter oder innere Knoten vorkommen. Die Optimierung dieses Baumes führt auf das sogenannte *Steiner-Problem*, benannt nach dem Schweizer Mathematiker Jakob Steiner (1796–1863).

Weitere Anwendungen von Steiner-Bäumen finden sich beim VLSI-Routing und bei der Rekonstruktion phylogenetischer Bäume in der Biologie.

### 2.2.1 Problemdefinition

Wir betrachten das Steiner-Problem für Graphen, das folgendermassen definiert ist.

Problem STEINERTREE

**Instanz:** zshgd. ungerichteter Graph  $G = (V, E)$ , Kantengewichte  $c : E \rightarrow \mathbb{R}^+$ ,  
Terminals  $N \subseteq V$

**Lösung:** Teilbaum  $T = (V', E')$  von  $G$ , so dass  $N \subseteq V'$

**Ziel:** minimiere die Kosten  $c(T) = \sum_{e \in E'} c(e)$  des Baums  $T$

Der Graph  $G = (V, E)$  repräsentiert hier das Kommunikationsnetz und das Gewicht  $c(e)$  einer Kante  $e \in E$  bestimmt die Kosten, die entstehen, wenn der Link  $e$  zur Übertragung verwendet wird.  $N$  ist eine Menge von Knoten, die genau den Sender und alle Empfänger enthält. Da wir annehmen, dass die Übertragungskosten für eine Kante unabhängig von der Richtung sind, in der die Daten übertragen werden, müssen wir bei der Menge  $N$  nicht zwischen dem Sender und den Empfängern unterscheiden.

Eine andere Anwendung des Steiner-Problems ergibt sich auch beim Entwurf von Kommunikationsnetzen. Wenn man die Knoten in  $N$  möglichst kostengünstig miteinander verbinden will, dabei aber beliebige Knoten aus  $V \setminus N$  zu Hilfe nehmen kann, stösst man wieder genau auf das Steiner-Problem.

Für eine Instanz des Problems STEINERTREE nennen wir jeden Teilbaum, der alle Knoten aus  $N$  enthält, einen *Steiner-Baum*. Die Knoten aus  $V \setminus N$ , die der Baum enthält, heissen *Steiner-Knoten*. Eine optimale Lösung nennen wir einen *minimalen Steiner-Baum*, und seine Kosten werden mit *OPT* bezeichnet.

In Abb. 2.3 ist ein Beispiel für eine Instanz des Problems STEINERTREE dargestellt. Die Menge der Terminals ist  $N = \{a, b, g\}$ . Rechts ist durch die fett gezeichneten Kanten ein minimaler Steiner-Baum gegeben, der die Knoten  $d, e, h$  als Steiner-Knoten verwendet und Kosten 8 hat.

Ein Spezialfall des Problems STEINERTREE ergibt sich, wenn  $N = V$  ist. In diesem Fall ist der Steiner-Baum ein Spannbaum des Graphen. Der minimale Steiner-Baum kann also effizient mit einem Algorithmus für minimale Spannbäume berechnet werden. Ein anderer Spezialfall ergibt sich, wenn  $N$  genau 2 Knoten enthält. Dann ist der Steiner-Baum gerade ein Pfad zwischen diesen beiden Knoten. Der minimale Steiner-Baum kann dann mit einem Algorithmus für kürzeste Pfade (z.B. Dijkstra) berechnet werden.

Im allgemeinen ist das Problem STEINERTREE aber  $\mathcal{NP}$ -schwer (sogar wenn alle Kanten Gewicht 1 haben) und es ist daher kein polynomieller Algorithmus bekannt, der einen

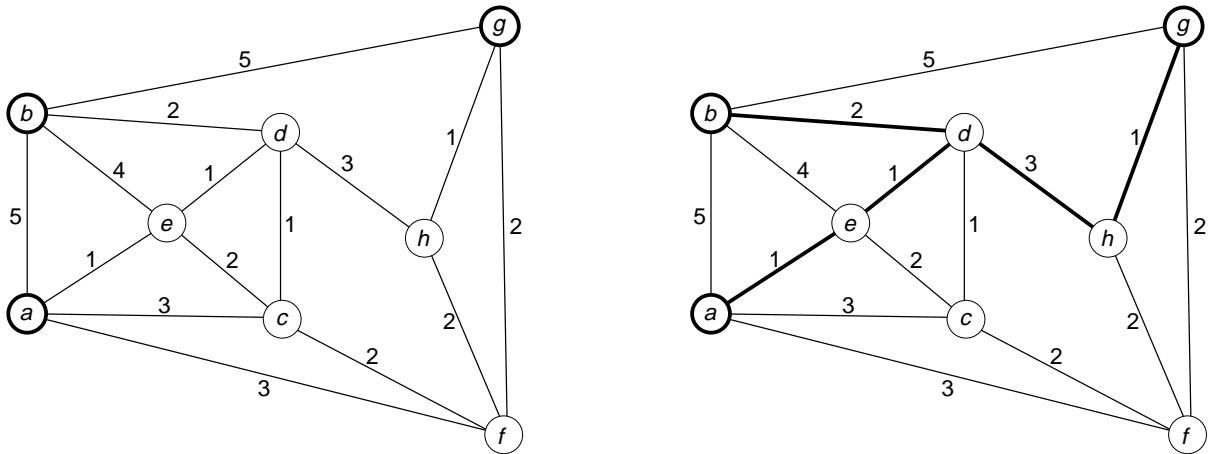


Abbildung 2.3: Beispiel einer Instanz von STEINERTREE und eines minimalen Steiner-Baums

**Algorithmus Distanz-Heuristik**

**Eingabe:** zshgd. ungerichteter Graph  $G = (V, E)$  mit Kantengewichten  $c : E \rightarrow \mathbb{R}$ ,  
Menge  $N \subseteq V$  von Terminals

**Ausgabe:** Steinerbaum  $T = (V', E')$

1. konstruiere Distanzgraph  $G' = (N, E')$ ;
2. berechne minimalen Spannbaum  $M'$  in  $G'$ ;
3. bestimme Teilgraph  $T'$  von  $G$ , indem jede Kante  $\{u, v\}$  des Spannbaums  $M'$  durch den zugehörigen kürzesten Pfad von  $u$  nach  $v$  in  $G$  ersetzt wird;
4. berechne minimalen Spannbaum  $M$  im von  $V(T')$  induzierten Teilgraphen von  $G$ ;
5. entferne wiederholt Steiner-Knoten mit Grad 1 aus  $M$ , so lange möglich;
6. **return**  $M$

Abbildung 2.4: Die Distanz-Heuristik zur Berechnung von Steiner-Bäumen

minimalen Steiner-Baum berechnet. Ferner ist bekannt, dass es kein Approximationsschema für STEINERTREE geben kann, es sei denn  $\mathcal{P} = \mathcal{NP}$ . (Ein Approximationsschema ist ein Algorithmus, der für jedes  $\varepsilon > 0$  eine  $(1 + \varepsilon)$ -Approximation berechnet, wobei die Laufzeit für jedes konstante  $\varepsilon > 0$  polynomiell sein muss.)

Wir werden im Folgenden einen 2-Approximationsalgorithmus für das Problem STEINERTREE kennen lernen.

### 2.2.2 Die Distanz-Heuristik

Die Grundidee des Algorithmus ist, das Steiner-Problem auf das Problem der Berechnung eines minimalen Spannbaums zurückzuführen. Dazu transformieren wir den gegebenen Graphen  $G = (V, E)$  in einen neuen Graphen  $G' = (N, F)$ , der als Knoten nur die Terminals enthält und in dem je zwei Knoten durch eine Kante verbunden sind, deren Kostenwert der Länge eines kürzesten Pfades zwischen den Knoten in  $G$  entspricht. Ein minimaler Spannbaum in  $G'$  kann dann in einen Steiner-Baum in  $G$  transformiert werden. Der Algorithmus ist in Abb. 2.4 skizziert. Im Folgenden besprechen wir die einzelnen Schritte im Detail.

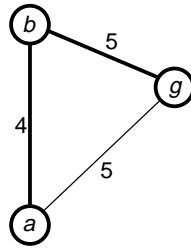
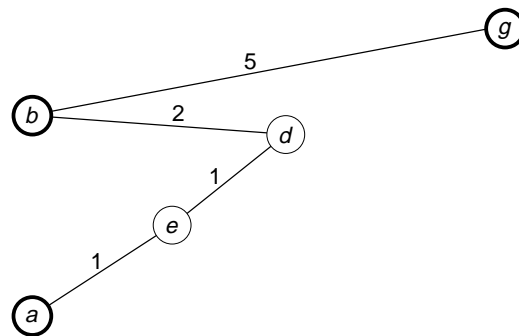


Abbildung 2.5: Der Distanz-Graph für das Steiner-Problem aus Abb. 2.3

Abbildung 2.6: Der Graph  $T'$ .

### Beschreibung des Algorithmus

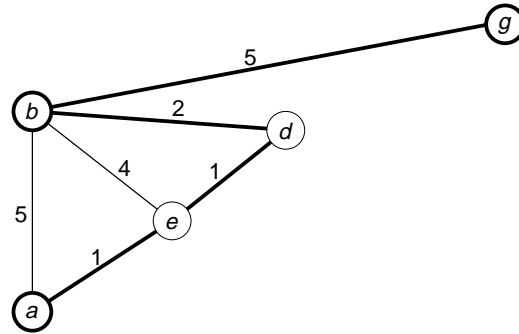
In Schritt 1 wird der sogenannte Distanz-Graph von  $G$  erzeugt. Für zwei Knoten  $u$  und  $v$  des Graphen  $G$  bezeichnen wir mit  $d(u, v)$  die Länge eines kürzesten Pfades von  $u$  nach  $v$  in  $G$ . Die Länge eines Pfades von  $u$  nach  $v$  ist hierbei gleich der Summe der Kosten  $c(e)$  für alle Kanten des Pfades. Der Distanz-Graph  $G' = (N, F)$  von  $G$  enthält dann alle Kanten  $\{u, v\}$  mit  $u, v \in N, u \neq v$ . Das heisst,  $G'$  ist ein *vollständiger Graph*. Das Gewicht der Kante  $\{u, v\}$  in  $G'$  wird auf  $c'(u, v) = d(u, v)$  gesetzt. Für das Beispiel aus Abb. 2.3 ist der Graph  $G'$  in Abb. 2.5 abgebildet.

In Schritt 2 wird ein minimaler Spannbaum  $M'$  in  $G'$  berechnet. Ein minimaler Spannbaum für unser Beispiel ist in Abb. 2.5 durch die fett gezeichneten Kanten markiert.

In Schritt 3 wird ein Teilgraph  $T'$  von  $G$  erzeugt, indem man mit dem leeren Graphen beginnt und für jede Kante von  $M'$  den entsprechenden kürzesten Pfad in  $G$  hinzufügt. In unserem Beispiel können wir für die Kante  $\{a, b\}$  von  $M'$  den Pfad  $a-e-d-b$  wählen und für die Kante  $\{b, g\}$  den Pfad  $b-g$ . Der erhaltene Graph  $T'$  ist in Abb. 2.6 zu sehen.

In Schritt 4 betrachten wir den von  $V(T')$  induzierten Teilgraphen von  $G$  und berechnen darin einen minimalen Spannbaum  $M$ . (Man könnte alternativ auch einen minimalen Spannbaum  $M$  direkt in  $T'$  berechnen, ohne die Approximationsrate zu verschlechtern, aber durch die Betrachtung zusätzlicher Kanten kann in manchen Fällen ein besseres Ergebnis erzielt werden.) Da  $M$  alle Knoten aus  $N$  enthält, ist  $M$  ein Steiner-Baum. Der von  $V(T')$  induzierte Teilgraph von  $G$  und ein minimaler Spannbaum darin sind für unser Beispiel in Abb. 2.7 dargestellt.

Schliesslich werden in Schritt 5 Steinerknoten (also Knoten aus  $V(M) \setminus N$ ) aus  $M$  entfernt, die Grad 1 haben. Solche Knoten sind Blätter des Baumes  $M$ , und  $M$  bleibt daher ein Steiner-Baum, wenn sie entfernt werden. Durch diesen Schritt können die Kosten des Steiner-Baums

Abbildung 2.7: Der von  $V(T')$  induzierte Teilgraph von  $G$ .

nur kleiner werden. In unserem Beispiel (Abb. 2.7) gibt es keine Steiner-Knoten mit Grad 1, der in Abb. 2.7 markierte Baum ist also der Steiner-Baum, den der Algorithmus für dieses Beispiel ausgibt. Wir stellen fest, dass dieser Baum Kosten 9 hat und daher nicht optimal ist. Der minimale Steiner-Baum war ja in Abb. 2.3 dargestellt und hatte Kosten 8.

### Laufzeit des Algorithmus

Sei  $n = |V|$  und  $m = |E|$ . Der Algorithmus kann mit Laufzeit  $O(|N|(n \log n + m))$  implementiert werden. In Schritt 1 muss von jedem der  $|N|$  Terminals aus der kürzeste Pfad zu allen anderen Terminals berechnet werden. Dies kann man erledigen, indem man den Algorithmus von Dijkstra nacheinander mit jedem Terminal als Startknoten aufruft. Das sind insgesamt  $|N|$  Aufrufe des Dijkstra-Algorithmus, und jeder Aufruf benötigt bei einer effizienten Implementierung des Dijkstra-Algorithmus Zeit  $O(n \log n + m)$ .

In Schritt 2 wird ein minimaler Spannbaum  $M'$  in  $G'$  berechnet.  $G'$  hat  $|N|$  Knoten und  $\binom{|N|}{2} = O(n^2)$  Kanten. Der Prim-Algorithmus für minimale Spann bäume, der für Graphen mit  $n$  Knoten und  $m$  Kanten Zeit  $O(n \log n + m)$  braucht, läuft daher für  $G'$  in Zeit  $O(|N|^2)$ .

In Schritt 3 muss jede der  $|N| - 1$  Kanten in  $M'$  durch einen Pfad in  $G$ , der Länge höchstens  $n - 1$  hat, ersetzt werden. Dies braucht höchstens  $O(|N|n)$  Zeit.

In Schritt 4 wird der von  $V(T')$  induzierte Teilgraph bestimmt und darin ein minimaler Spannbaum  $M$  berechnet. Das geht in Zeit  $O(n \log n + m)$ .

Zum Schluss werden in Schritt 5 noch Steiner-Knoten mit Grad 1 aus  $M$  entfernt. Wenn wir bei jedem Knoten seinen Grad im Steinerbaum speichern und wenn wir alle Steiner-Knoten mit Grad 1 in einer Liste verwalten, kann dieser Schritt in Zeit  $O(n)$  implementiert werden.

Die Gesamtlaufzeit des Algorithmus ergibt sich damit zu  $O(|N|(n \log n + m))$ .

### Analyse der Approximationsrate

Bezeichne mit  $OPT$  die Kosten des minimalen Steiner-Baums. Wir wollen zeigen, dass der vorgestellte Algorithmus Approximationsrate 2 hat. Dazu beweisen wir die folgenden beiden Behauptungen:

- (a) Sei  $W'$  das Gesamtgewicht des minimalen Spannbaums  $M'$  von  $G'$ . Dann gilt  $W' \leq \left(2 - \frac{2}{|N|}\right) OPT$ .

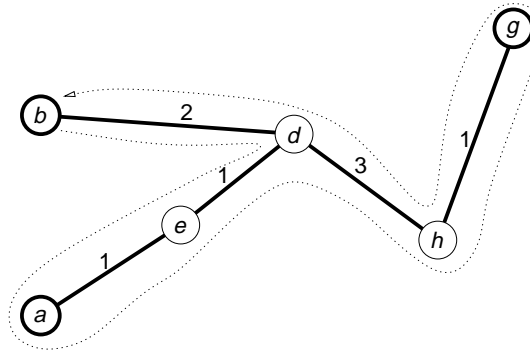


Abbildung 2.8: Der gepunktete Pfad läuft einmal um den Steiner-Baum herum.

- (b) Das Gesamtgewicht (die Kosten) des vom Algorithmus berechneten Steiner-Baumes ist höchstens  $W'$ .

Zuerst beweisen wir (a). Dazu nehmen wir an, dass wir einen minimalen Steiner-Baum  $T$  mit Gewicht  $OPT$  gegeben haben. Nun stellen wir uns  $T$  als Baum in der Ebene gezeichnet vor und betrachten den Pfad, der bei einem beliebigen Knoten in  $N$  startet und einmal “um den Baum  $T$  herum” läuft, d.h. jede Kante von  $T$  genau zweimal verwendet. Dieser Pfad ist nicht einfach, da er Knoten und Kanten mehrfach verwendet. Für den minimalen Steiner-Baum unseres Beispiels ist ein solcher Pfad “um den Baum herum” in Abb. 2.8 gepunktet eingezeichnet. Der Pfad beginnt und endet bei dem Knoten  $b$ .

Nennen wir diesen Pfad  $p$ . Die Summe der Kantengewichte aller Kanten auf  $p$  ist  $2OPT$ , da jede Kante von  $T$  genau zweimal enthalten ist. Nun zerschneiden wir den Pfad  $p$  an all den Stellen, wo ein Knoten aus  $N$  zum ersten Mal erreicht wird. Im Beispiel erhalten wir damit die Pfadstücke  $b-d-e-a$ ,  $a-e-d-h-g$  und  $g-h-d-b$ . Allgemein erhalten wir immer genau  $|N|$  Pfadstücke. Nun streichen wir dasjenige Pfadstück mit dem grössten Gewicht. Da sich die Gewichte der  $|N|$  Pfadstücke zu  $2OPT$  aufsummieren, hat das teuerste Pfadstück Gewicht mindestens  $\frac{2}{|N|}OPT$ . Das Gesamtgewicht der verbleibenden  $|N| - 1$  Pfadstücke ist somit höchstens  $\left(2 - \frac{2}{|N|}\right)OPT$ . Im Beispiel haben zwei der drei Pfadstücke Gewicht 6, wir können dann ein beliebiges dieser beiden Pfadstücke entfernen.

Die  $|N| - 1$  verbleibenden Pfadstücke laufen jeweils von einem Knoten in  $N$  zu einem anderen Knoten in  $N$ , wobei an jedem Knoten höchstens ein Pfadstück beginnt und höchstens eines endet. Wenn wir zu jedem Pfadstück die entsprechende Kante im Distanz-Graphen  $G'$  betrachten (deren Gewicht ausserdem höchstens so gross ist wie das Gewicht des Pfadstücks), dann bilden diese  $|N| - 1$  Kanten einen einfachen Pfad, der alle Knoten in  $N$  genau einmal besucht. Dieser Pfad ist insbesondere ein Spannbaum von  $G'$  mit Gewicht höchstens  $\left(2 - \frac{2}{|N|}\right)OPT$ . Somit hat der minimale Spannbaum von  $G'$  ebenfalls Gewicht höchstens  $\left(2 - \frac{2}{|N|}\right)OPT$ . Damit haben wir (a) bewiesen.

Es bleibt noch (b) zu zeigen. In Schritt 3 ersetzt der Algorithmus jede Kante des minimalen Spannbaums  $M'$  von  $G'$  durch den zugehörigen kürzesten Pfad in  $G$ , um einen Teilgraphen  $T'$  von  $G$  zu erhalten. Wären die kürzesten Pfade alle kantendisjunkt, so wäre das Gewicht von  $T'$  gleich dem Gewicht von  $M'$ . Wenn manche der kürzesten Pfade Kanten gemeinsam haben, so wird das Gewicht von  $T'$  höchstens kleiner. Es folgt also, dass das Gewicht von  $T'$  höchstens gleich dem Gewicht von  $M'$  ist. Das Gewicht eines minimalen Spannbaums  $M$  in

$T'$  ist natürlich höchstens so gross wie das Gewicht von  $T'$ . Wenn wir für die Berechnung des minimalen Spannbaums  $M$  noch weitere Kanten zulassen (wir nehmen ja den von  $V(T')$  induzierten Teilgraphen) und am Ende auch noch eventuell Steiner-Knoten mit Grad 1 löschen, so kann ebenfalls das Gewicht höchstens kleiner werden. Damit ist auch (b) gezeigt.

**Satz 2.2** *Der Algorithmus aus Abb. 2.4 berechnet in Zeit  $O(|N|(n \log n + m))$  einen Steiner-Baum mit Kosten höchstens  $\left(2 - \frac{2}{|N|}\right) OPT$ . Die Approximationsrate des Algorithmus ist  $\left(2 - \frac{2}{|N|}\right) < 2$ .*

Dieser Algorithmus geht auf Takahashi und Matsuyama zurück [TM80]. Kurt Mehlhorn hat eine Modifikation des Algorithmus gefunden, die dieselbe Approximationsrate erzielt, aber in Zeit  $O(n \log n + m)$  implementiert werden kann [Meh88].

### 2.2.3 Bemerkungen

In den letzten 10 Jahren wurden neue Approximationsalgorithmen gefunden, deren Approximationsraten immer besser wurden: 1.834, 1.734, 1.694, 1.667, 1.644, 1.598. Der beste bekannte Approximationsalgorithmus hat Approximationsrate  $1 + \frac{1}{2} \ln 3 \approx 1.55$  und stammt von G. Robins und A. Zelikovsky [RZ00]. Eine wichtige Grundidee bei diesen verbesserten Algorithmen ist, Steinerbäume für Teilmengen der Terminals zu betrachten, die jeweils höchstens  $k$  Terminals enthalten. Dabei treten die Terminals in diesen Teil-Steinerbäumen nur als Blätter auf.

Ein anderer Algorithmus, der in der Praxis oft sehr gut funktioniert, ist die *iterierte 1-Steiner-Heuristik* (IIS) [KR95]. Sie startet mit einer leeren Menge  $S = \emptyset$  von erzwungenen Steiner-Knoten und berechnet dann zuerst mittels der Distanz-Heuristik einen Steinerbaum  $T$  für die Menge  $N \cup S = N$  von Terminals. Dann wird für jeden Knoten  $v \in V \setminus (N \cup S)$  getestet, ob der von der Distanz-Heuristik berechnete Steinerbaum  $T_{N \cup S \cup \{v\}}$  für die Menge  $N \cup S \cup \{v\}$  von Terminals günstiger ist als  $T$ . Ist das der Fall, so wird derjenige Knoten  $v$ , der zu dem günstigsten Steinerbaum geführt hat, in die Menge  $S$  aufgenommen, d.h. man setzt  $S := S \cup \{v\}$ . Der korrespondierende Steinerbaum  $T_{N \cup S}$  (für die neue Menge  $S$ ) wird dann zum aktuellen Steinerbaum. Dies wiederholt sich, bis es keinen Knoten  $v$  mehr gibt, so dass die Distanz-Heuristik für die Menge  $N \cup S \cup \{v\}$  einen günstigeren Steinerbaum berechnet als den aktuellen Steinerbaum. Nach jeder Iteration werden ausserdem Steiner-Knoten in  $S$ , die im aktuellen Steinerbaum Grad 1 oder 2 haben, aus  $S$  entfernt. Am Ende wird der aktuelle Steinerbaum ausgegeben.

### Referenzen

- [KR95] Andrew B. Kahng and Gabriel Robins. *On Optimal Interconnections for VLSI*. Kluwer Publishers, 1995.
- [Meh88] Kurt Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters*, 27(3):125–128, 1988.
- [RZ00] Gabriel Robins and Alexander Zelikovsky. Improved Steiner tree approximation in graphs. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms SODA 2000*, pages 770–779, 2000.

- [TM80] H. Takahashi and A. Matsuyama. An approximate solution for the Steiner problem in graphs. *Math. Japonica*, 24:573–577, 1980.

## Kapitel 3

# Kürzeste Wege, Spanner und Netzwerkdesign

Der Begriff Netzwerkdesign steht für eine Reihe von Problemstellungen, die sich beim Entwurf von Kommunikationsnetzen ergeben. Allgemein ist es dabei meistens das Ziel, ein kostengünstiges Netz zu entwerfen (d.h. unter anderem die Topologie und das Routing festzulegen), das die (oft geschätzten) Verkehrsanforderungen gut erfüllen kann. Je nach der konkreten Anwendung (abhängig von der gewählten Netzwerk-Architektur, den zu verwendenden Protokollen, Kundenwünschen, etc.) können die sich ergebenden Problemstellungen im Detail stark voneinander abweichen. Es gibt daher kein einzelnes Problem, das man “das Netzwerkdesign-Problem” nennen könnte, sondern eine Vielzahl verwandter Probleme, die alle unter dem Begriff Netzwerkdesign zusammengefasst werden. In diesem Kapitel wollen wir uns mit zwei konkreten Problemen aus dem Netzwerkdesign beschäftigen. Dazu benötigen wir Algorithmen zur Berechnung von Kürzeste-Wege-Bäumen, von angenäherten Kürzeste-Wege-Bäumen und von Spanners als Unterroutrinen. Deshalb werden wir zuerst Algorithmen für diese Probleme kennen lernen.

### 3.1 Berechnung kürzester Wege in Graphen

Es sei ein gerichteter Graph  $G = (V, E)$  mit  $|V| = n$  Knoten,  $|E| = m$  Kanten und positiven Kantengewichten  $c : E \rightarrow \mathbb{R}^+$  gegeben. Wir interessieren uns für das Problem, für einen Startknoten  $s \in V$  die kürzesten Wege (Pfade) zu allen anderen Knoten in  $V$  zu berechnen. Wir werden sehen, dass dieses Problem effizient mit dem Algorithmus von Dijkstra gelöst werden kann. Anwendungen für dieses Problem finden sich beispielsweise beim Routing in Kommunikationsnetzen oder bei der Wegewahl in Navigationssystemen.

Wir nehmen an, dass alle anderen Knoten von  $s$  aus erreichbar sind (zu Knoten, die nicht über einen Pfad erreichbar sind, kann es auch keinen kürzesten Pfad geben). Die Länge (Summe der Kantengewichte) eines kürzesten Pfades vom Startknoten  $s$  zu einem Knoten  $w$  nennt man auch den *Abstand* von  $s$  zu  $w$ .

Weiterhin bezeichne  $from[v]$  für alle Knoten  $v \neq s$  den letzten Knoten vor  $v$  auf einem kürzesten Pfad von  $s$  zu  $v$ . Die Kanten  $(from[v], v)$  für alle Knoten  $v \in V \setminus \{s\}$  ergeben zusammen einen *Kürzeste-Wege-Baum*, d.h. einen Baum mit Wurzel  $s$ , in dem kürzeste Pfade von  $s$  zu allen anderen Knoten enthalten sind. Ein Beispielgraph und ein zugehöriger *Kürzeste-Wege-Baum* sind in Abb. 3.1 zu sehen.

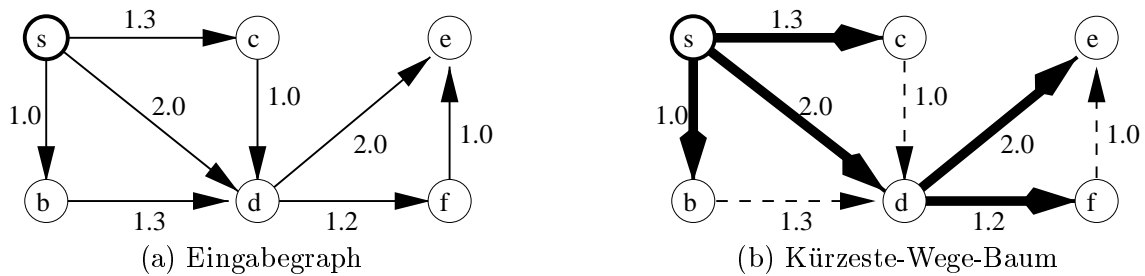


Abbildung 3.1: Beispiel eines Kürzeste-Wege-Baums

Die Hauptidee beim Algorithmus von Dijkstra (siehe Abb. 3.2) ist, die kürzesten Pfade in Reihenfolge aufsteigender Länge zu berechnen. Der Algorithmus berechnet dabei für jeden Knoten  $w$  einen Wert  $dist[w]$ , der die Länge eines kürzesten Pfades vom Startknoten  $s$  zu  $w$  angibt, und einen Knoten  $from[w]$  (ausser für  $w = s$ ), der den letzten Knoten vor  $w$  auf einem kürzesten Pfad von  $s$  nach  $w$  angibt. Aus den  $from$ -Werten lassen sich dann die kürzesten Pfade leicht konstruieren.

Der Algorithmus markiert einen Knoten  $v$  mittels  $ready[v] = true$  als *erledigt*, falls bereits ein kürzester Pfad zu  $v$  bestimmt wurde, d.h. falls die  $dist$ - und  $from$ -Werte für  $v$  bereits korrekt sind und nicht mehr verändert werden. Bei der Initialisierung wird  $ready[s]$  auf true gesetzt und  $dist[s]$  auf 0. Alle anderen Knoten erhalten  $ready[v] = false$  und  $dist[v] = \infty$ . Alle Knoten, die von einem erledigten Knoten aus über eine Kante erreichbar sind, werden in einer Prioritätswarteschlange  $Q$  gespeichert (siehe Kapitel 2.1), wobei für jeden Knoten  $v \in Q$  gilt:

- $dist[v]$  ist die Länge eines kürzesten Pfades von  $s$  nach  $v$  unter allen Pfaden, die ausser  $v$  nur erledigte Knoten (d.h. Knoten  $w$  mit  $ready[w] = true$ ) besuchen.
- $from[v]$  ist der letzte Knoten vor  $v$  auf einem solchen Pfad.
- Die Priorität des Knotens  $v$  ist gleich  $dist[v]$ .

Zu Anfang sind nur die Nachbarn von  $s$  über eine Kante von einem erledigten Knoten erreichbar. Daher werden alle Nachbarn (Knoten am anderen Ende von ausgehenden Kanten) von  $s$  in die Prioritätswarteschlange eingefügt und ihre Werte entsprechend initialisiert.

Dann wird in jedem Schritt ein zusätzlicher Knoten  $v$  erledigt. Dabei ist  $v$  immer der Knoten mit der kleinsten Priorität in  $Q$ , der mittels einer DeleteMin-Operation gefunden wird. Es ist leicht zu sehen, dass der Wert  $dist[v]$  zu diesem Zeitpunkt bereits den richtigen Wert hat. Nachdem  $v$  mit  $ready[v] = true$  als erledigt markiert wurde, müssen die Nachbarn von  $v$  ggf. aktualisiert werden: Ein Nachbar  $w$  von  $v$ , der bisher noch überhaupt nicht erreicht wurde, muss mit Priorität  $dist[v] + c(v, w)$  in  $Q$  eingefügt werden. Für einen Nachbarn  $w$  von  $v$ , der mit einer Priorität grösser als  $dist[v] + c(v, w)$  in  $Q$  enthalten ist, muss die Priorität auf  $dist[v] + c(v, w)$  erniedrigt werden, da jetzt über  $v$  ein kürzerer Pfad nach  $w$  gefunden wurde. Nach  $n - 1$  Schritten ist der Algorithmus beendet.

Die Laufzeitanalyse des Algorithmus ist völlig analog zum Algorithmus von Prim für minimale Spannbäume, da sich die beiden Algorithmen lediglich durch die Prioritätszuweisung für Knoten in  $Q$  unterscheiden. Bei Verwendung einer mit Fibonacci-Heaps implementierten Prioritätswarteschlange ergibt sich daher wieder eine Gesamtlaufzeit von  $O(n \log n + m)$ .

Die Korrektheit des Algorithmus von Dijkstra lässt sich leicht durch Induktion zeigen, indem man beweist, dass nach  $k < n$  Schritten des Algorithmus die  $k$  Knoten mit kleinsten Abständen von  $s$  richtig berechnet wurden.

```

Algorithmus von Dijkstra
Eingabe: gerichteter Graph  $G = (V, E)$  mit Kantengewichten  $c : E \rightarrow \mathbb{R}^+$  und
Startknoten  $s \in V$ 
Ausgabe: kürzeste Pfade von  $s$  zu allen von  $s$  erreichbaren Knoten
(gegeben durch  $dist$  und  $from$ )

 $ready[s] := true;$ 
 $ready[v] := false$  für alle  $v \in V \setminus \{s\};$ 
 $dist[s] := 0;$ 
 $dist[v] := \infty$  für alle  $v \in V \setminus \{s\};$ 
p_queue  $Q; \{ Priority-Queue \text{ für Knoten } \}$ 
for alle ausgehenden Kanten  $e = (s, v)$  von  $s$  do
     $from[v] := s;$ 
     $dist[v] := c(e);$ 
     $Q.Insert(v, dist[v]);$ 
od;
while  $Q$  ist nicht leer do
     $v := Q.DeleteMin();$ 
     $ready[v] := true;$ 
    for alle ausgehenden Kanten  $e = (v, w)$  von  $v$  do
        if  $w \in Q$  und  $dist[v] + c(e) < dist[w]$  then
             $from[w] := v;$ 
             $dist[w] := dist[v] + c(e);$ 
             $Q.DecreasePriority(w, dist[w]);$ 
        else if  $w \notin Q$  und  $ready[w] = false$  then
             $from[w] := v;$ 
             $dist[w] := dist[v] + c(e);$ 
             $Q.Insert(w, dist[w]);$ 
        fi;
    od;
od;

```

Abbildung 3.2: Der Algorithmus von Dijkstra für kürzeste Wege

Der Algorithmus von Dijkstra ist auch auf ungerichtete Graphen anwendbar. Man betrachtet dazu bei jedem Knoten statt der ausgehenden Kanten einfach alle inzidenten Kanten.

### 3.2 Leichte angenäherte Kürzeste-Wege-Bäume

In diesem Abschnitt beschäftigen wir uns mit der Berechnung von Spannbäumen eines Graphen, die die Eigenschaften von minimalen Spannbäumen und von Kürzeste-Wege-Bäumen in gewissem Sinne vereinen.

Sei ein ungerichteter Graph  $G = (V, E)$  mit Kantengewichten (Kosten)  $c : E \rightarrow \mathbb{R}^+$  gegeben. Im Hinblick auf Anwendungen beim Netzwerkdesign stellen wir uns dabei vor, dass  $V$  die Menge der Knoten ist, die wir vernetzen wollen, und dass die Kanten in  $E$  alle Links sind, die wir potentiell verwenden könnten. Das Gewicht  $c(e)$  einer Kante  $e$  gibt dabei die

```

Algorithmus: Präordernummerierung
Eingabe: Baum  $T = (V, E)$  mit Wurzel  $s \in V$ 
Ausgabe: Präordernummern  $num[v]$  für alle  $v \in V$ 

procedure preorder(node  $v$ )
begin
     $num[v] := i; i++;$ 
    for alle Kinder  $w$  von  $v$  do
        preorder( $w$ );
    od
end

 $i := 1;$ 
preorder( $s$ );

```

Abbildung 3.3: Berechnung der Präordernummerierung eines Baumes

Kosten an, die die Verwendung des Links  $e$  mit sich bringen würde.

Sei ferner ein Knoten  $s \in V$  ausgezeichnet. Bei der Anwendung auf Netzwerkdesign-Probleme sollen mit diesem Knoten  $s \in V$  alle anderen Knoten verbunden werden.

Wir erinnern uns, dass ein minimaler Spannbaum von  $G$  ein Spannbaum von  $G$  ist, für den die Summe der Kantengewichte minimal ist. Bezeichne mit  $T_M(G)$  einen minimalen Spannbaum von  $G$  und mit  $c(T_M(G))$  sein Gewicht. Ein Kürzester-Wege-Baum für  $G$  und  $s$  ist ein Spannbaum von  $G$ , so dass für jeden Knoten  $v \in V$  die Länge des Pfades von  $s$  nach  $v$  in dem Baum gleich der Länge eines kürzesten Pfades von  $s$  nach  $v$  in  $G$  ist. Hierbei ist die Länge eines Pfades als die Summe der Gewichte aller Kanten auf dem Pfad definiert. Wir bezeichnen die Länge des kürzesten Pfades von  $s$  nach  $v$  in  $G$  mit  $d_G(s, v)$ . Sowohl minimale Spannbäume als auch Kürzeste-Wege-Bäume können effizient (z.B. in Zeit  $O(n \log n + m)$ ) berechnet werden, wie wir in Kapitel 2.1 und 3.1 gesehen haben.

Nun interessieren wir uns für einen Spannbaum, der die Qualitäten eines minimalen Spannbaums und eines Kürzeste-Wege-Baums näherungsweise miteinander vereint. Für gegebene Parameter  $\alpha \geq 1$  und  $\beta \geq 1$  nennen wir einen Spannbaum  $T$  von  $G$  einen  $(\alpha, \beta)$ -*leichten angenäherten Kürzeste-Wege-Baum* (kurz  $(\alpha, \beta)$ -LAST, wegen engl. *light approximate shortest-paths tree*), wenn

1. für jeden Knoten  $v \in V$  die Länge des Pfades von  $s$  nach  $v$  in  $T$ , bezeichnet mit  $d_T(s, v)$ , höchstens  $\alpha d_G(s, v)$  ist, und
2. das Gesamtgewicht  $c(T)$  aller Kanten von  $T$  höchstens  $\beta c(T_M(G))$  ist, wobei  $T_M(G)$  ein minimaler Spannbaum von  $G$  ist.

Unser Ziel ist, einen Algorithmus zu finden, der für möglichst kleine Werte von  $\alpha$  und  $\beta$  effizient einen  $(\alpha, \beta)$ -LAST berechnet.

### 3.2.1 Die Präordernummerierung von Bäumen

Der Algorithmus macht starken Gebrauch von einer Präordernummerierung des minimalen Spannbaums. Die Präordernummerierung eines Baumes ist wie folgt definiert. Sei ein Baum mit  $n$  Knoten gegeben und sei ein Knoten als Wurzel des Baumes ausgezeichnet. (Dadurch hat

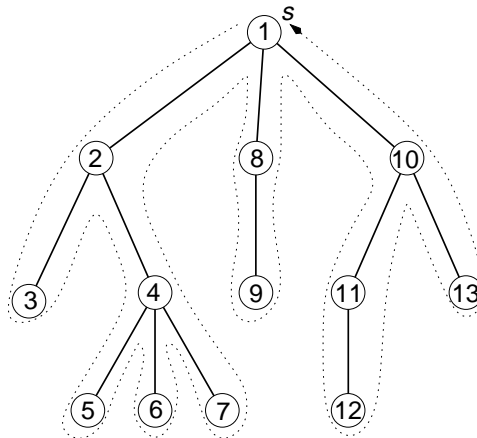


Abbildung 3.4: Beispiel für Präordernummerierung eines Baumes

jeder Knoten ungleich der Wurzel einen eindeutigen Vaterknoten, und seine anderen adjazenten Knoten sind seine Kinder.) Eine Präordernummerierung des Baumes ist eine bijektive Zuordnung der Zahlen  $1, 2, \dots, n$  zu den Knoten des Baumes in der Reihenfolge, wie die Knoten in einem bei der Wurzel beginnenden Präorderdurchlauf besucht werden. Ein Präorderdurchlauf ist ein Durchlauf des Baumes, bei dem nach dem Besuch eines Knotens nacheinander Präorderdurchläufe in den an seinen Kindern gewurzelten Teilbäumen durchgeführt werden. Ein einfacher rekursiver Algorithmus zur Präordernummerierung eines Baumes ist in Abb. 3.3 dargestellt. Der Algorithmus verwendet die rekursive Prozedur “preorder”, die im Hauptprogramm für die Wurzel aufgerufen wird. Die globale Variable  $i$  speichert jeweils die nächste zu vergebende Nummer. Für ein Beispiel ist in Abb. 3.4 eine mögliche Präordernummerierung dargestellt.

Zunächst benötigen wir das folgende Lemma über Spannbäume von  $G$ .

**Lemma 3.1** *Sei  $T$  ein Spannbaum von  $G$ . Betrachte  $s$  als die Wurzel von  $T$ . Wenn  $z_0, z_1, \dots, z_k$  beliebige  $k+1$  verschiedene Knoten von  $T$  in Reihenfolge aufsteigender Präordernummern sind, dann gilt*

$$\sum_{i=1}^k d_T(z_{i-1}, z_i) \leq 2c(T).$$

**Beweis.** Betrachte den (nicht einfachen) Weg  $w$ , den ein Präorderdurchlauf durch  $T$  zurücklegt. Dieser Weg beginnt und endet bei der Wurzel  $s$  und durchläuft jede Kante des Baums genau zweimal, einmal abwärts und einmal aufwärts. In Abb. 3.4 ist dieser Weg durch die gepunktete Linie angedeutet. Die Summe der Kantengewichte auf dem Weg  $w$  ist offensichtlich  $2c(T)$ .

Für jedes  $i$ ,  $1 \leq i \leq k$ , enthält der Weg  $w$  einen Teilpfad vom ersten Vorkommen von  $z_{i-1}$  bis zum ersten Vorkommen von  $z_i$ . Die Länge dieses Teilpfades von  $w$  ist mindestens  $d_T(z_{i-1}, z_i)$ . Ausserdem sind alle  $k$  solchen Teilpfade disjunkte Teile von  $w$ , da die Knoten  $z_i$  in Reihenfolge aufsteigender Präordernummern gegeben sind. Damit folgt die behauptete Ungleichung.  $\square$

**Algorithmus:  $(\alpha, \beta)$ -LAST**

**Eingabe:** zshgd. ungerichteter Graph  $G = (V, E)$ , ausgezeichneter Knoten  $s \in V$ ,  
Kantengewichte  $c : E \rightarrow \mathbb{R}^+$ , Werte  $\alpha, \beta$  mit  $\alpha > 1$  und  $\beta = 1 + \frac{2}{\alpha-1}$

**Ausgabe:**  $(\alpha, \beta)$ -LAST  $T$

1. berechne minimalen Spannbaum  $T_M$  von  $G$ ;
2. berechne kürzeste Pfade von  $s$  zu allen anderen Knoten in  $G$ ;
3. berechne Präordernummerierung von  $T_M$  mit  $s$  als Wurzel;
4.  $H := T_M$ ;  
    **for** alle Knoten  $v \in V$  in Reihenfolge aufsteigender Präordernummern **do**  
       berechne kürzesten Pfad  $p$  von  $s$  nach  $v$  in  $H$ ;  
       **if**  $c(p) > \alpha d_G(s, v)$  **then**  
           füge einen kürzesten Pfad von  $s$  nach  $v$  in  $G$  in den Graphen  $H$  ein;  
       **fi**  
    **od**
5. berechne in  $H$  einen Kürzeste-Wege-Baum  $T$  mit Startknoten  $s$ ;
6. **return**  $T$ ;

Abbildung 3.5: Algorithmus für die Berechnung eines  $(\alpha, \beta)$ -LAST

### 3.2.2 Die Berechnung eines LAST

Ein Algorithmus zur Berechnung eines  $(\alpha, \beta)$ -LAST ist in Abb. 3.5 angegeben. Der Algorithmus berechnet zuerst einen minimalen Spannbaum  $T_M$  von  $G$  und einen Kürzeste-Wege-Baum mit Startknoten  $s$ . Ausgehend von  $T_M$  erzeugt der Algorithmus einen Teilgraphen  $H$  von  $G$ , in dem jeder Knoten  $v$  Abstand höchstens  $\alpha d_G(s, v)$  von  $s$  hat. Dazu prüft er für jeden Knoten  $v$  nach, ob diese Bedingung bereits erfüllt ist, und fügt, falls nicht, einfach alle Kanten eines kürzesten Pfades von  $s$  nach  $v$  in  $G$  in den Graphen  $H$  ein. Am Ende wird ein Kürzeste-Wege-Baum  $T$  in  $H$  berechnet und zurückgeliefert. Der Parameter  $\alpha > 1$  kann dem Algorithmus vorgegeben werden, der Parameter  $\beta$  ergibt sich dann zu  $1 + \frac{2}{\alpha-1}$ . Je näher  $\alpha$  also an 1 liegt, desto grösser wird der Wert von  $\beta$ .

Es ist klar, dass der Algorithmus einen Baum  $T$  liefert, in dem für jeden Knoten  $v$  die Bedingung  $d_T(s, v) \leq \alpha d_G(s, v)$  gilt: der Graph  $H$  enthält ja nach Konstruktion einen Pfad von  $s$  nach  $v$  mit Länge höchstens  $\alpha d_G(s, v)$ , und wir berechnen am Ende in  $H$  einen Kürzeste-Wege-Baum mit Startknoten  $s$ .

Somit müssen wir nur noch beweisen, dass  $c(T) \leq \beta c(T_M)$  gilt. Wir zeigen, dass sogar  $c(H) \leq \beta c(T_M)$  gilt. Da  $T$  ein Teilgraph von  $H$  ist, folgt daraus die Behauptung.

Seien dazu  $z_1, z_2, \dots, z_k$  diejenigen Knoten, für die bei Ausführung des Algorithmus in Abb. 3.5 in Schritt 4 die if-Bedingung erfüllt war, für die also alle Kanten eines kürzesten Pfades zu  $H$  hinzugefügt wurden. Ferner wählen wir  $z_0 = s$ . Da die Knoten in Schritt 4 in Reihenfolge aufsteigender Präordernummern bearbeitet werden, ist Lemma 3.1 auf die so definierten Knoten  $z_0, z_1, \dots, z_k$  anwendbar. Daher erhalten wir:

$$\sum_{i=1}^k d_{T_M}(z_{i-1}, z_i) \leq 2c(T_M) \quad (3.1)$$

Als der Algorithmus in Schritt 4 den Knoten  $z_i$  bearbeitete ( $i = 1, 2, \dots, k$ ), enthielt  $H$  auf jeden Fall den Pfad von  $s$  nach  $z_i$ , der aus dem kürzesten Pfad von  $s$  nach  $z_{i-1}$  in  $G$  und dem

Pfad von  $z_{i-1}$  nach  $z_i$  in  $T_M$  besteht. Dieser Pfad hat Länge  $d_G(s, z_{i-1}) + d_{T_M}(z_{i-1}, z_i)$ . Da der kürzeste Pfad von  $s$  nach  $z_i$  in  $H$  zu diesem Zeitpunkt länger als  $\alpha d_G(s, z_i)$  war (sonst wäre die if-Bedingung nicht erfüllt gewesen), muss also gelten:

$$d_G(s, z_{i-1}) + d_{T_M}(z_{i-1}, z_i) > \alpha d_G(s, z_i)$$

Dies können wir zu  $\alpha d_G(s, z_i) - d_G(s, z_{i-1}) < d_{T_M}(z_{i-1}, z_i)$  umformen. Wenn wir diese Ungleichungen für  $i = 1, 2, \dots, k$  aufsummieren, erhalten wir (unter Berücksichtigung von  $d_G(s, z_0) = 0$ ):

$$\begin{array}{rcl} \alpha d_G(s, z_1) - d_G(s, z_0) & < & d_{T_M}(z_0, z_1) \\ \alpha d_G(s, z_2) - d_G(s, z_1) & < & d_{T_M}(z_1, z_2) \\ & & \vdots \\ \alpha d_G(s, z_k) - d_G(s, z_{k-1}) & < & d_{T_M}(z_{k-1}, z_k) \\ \hline \alpha d_G(s, z_k) + \sum_{i=1}^{k-1} (\alpha - 1) d_G(s, z_i) & < & \sum_{i=1}^k d_{T_M}(z_{i-1}, z_i) \end{array}$$

Mit der Ungleichung (3.1) und der Abschätzung  $(\alpha - 1)d_G(s, z_k) \leq \alpha d_G(s, z_k)$  erhalten wir daraus  $(\alpha - 1) \sum_{i=1}^k d_G(s, z_i) < 2c(T_M)$  bzw. nach Umformung

$$\sum_{i=1}^k d_G(s, z_i) < \frac{2}{\alpha - 1} c(T_M).$$

Da nach Konstruktion des Algorithmus  $c(H) \leq c(T_M) + \sum_{i=1}^k d_G(s, z_i)$  gilt, folgt  $c(H) \leq (1 + \frac{2}{\alpha - 1})c(T_M)$ . Damit ist der Beweis abgeschlossen und wir erhalten den folgenden Satz.

**Satz 3.2** Sei  $\alpha > 1$  und  $\beta = 1 + \frac{2}{\alpha - 1}$ . Der Algorithmus aus Abb. 3.5 berechnet in polynomieller Zeit einen  $(\alpha, \beta)$ -LAST, d.h. einen Spannb Baum  $T$  mit  $d_T(s, v) \leq \alpha d_G(s, v)$  für alle  $v \in V$  und  $c(T) \leq \beta c(T_M(G))$ .

**Algorithmus: Greedy-Spanner**  
**Eingabe:** zshgd. Graph  $G = (V, E)$  mit Kantengewichten  $c : E \rightarrow \mathbb{R}^+$ , Parameter  $t > 1$   
**Ausgabe:** zshgd. Teilgraph  $H = (V, E')$  von  $G$

sortiere die Kanten in  $E = \{e_1, e_2, \dots, e_m\}$  so, dass  $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$  gilt;  
 $E' := \emptyset$ ;  
 $H := (V, E')$ ;  
**for**  $i = 1$  to  $m$  **do**  
    // sei  $e_i = \{u, v\}$   
    **if**  $d_H(u, v) > t \cdot c(\{u, v\})$  **then**  
         $E' := E' \cup \{e_i\}$ ;  
         $H := (V, E')$ ;  
    **fi**  
**od**;  
**return**  $H$ ;

Abbildung 3.6: Algorithmus zur Konstruktion eines  $t$ -Spanners

### 3.3 Spanner

Ein  $(\alpha, \beta)$ -LAST in einem Graphen enthält für einen bestimmten Knoten Wege zu allen anderen Knoten, die nicht viel länger als die kürzesten Wege im Ursprungsgraphen sind. Im Unterschied dazu wollen wir nun einen Teilgraphen finden, in dem die Wege für *alle* Paare von Knoten nicht viel länger als im Ursprungsgraphen sind.

Sei  $G = (V, E)$  ein ungerichteter zusammenhängender Graph mit Kantengewichten  $c : E \rightarrow \mathbb{R}^+$ . Bezeichne mit  $d_G(u, v)$  die Länge eines kürzesten Pfades (bezüglich der Kantengewichte  $c(e)$ ) von  $u$  nach  $v$  in  $G$ . Sei  $H = (V, E')$  ein zusammenhängender Teilgraph von  $G$ , der alle Knoten von  $G$  enthält (solche Teilgraphen nennt man auch *spannende Teilgraphen*). Bezeichne mit  $d_H(u, v)$  die Länge eines kürzesten Pfades (bezüglich der Kantengewichte  $c(e)$ ) von  $u$  nach  $v$  in  $H$ . Definiere

$$\text{Stretch}(H) = \max_{u, v \in V} \frac{d_H(u, v)}{d_G(u, v)}.$$

Somit gibt  $\text{Stretch}(H)$  an, um welchen Faktor der kürzeste Pfad zwischen zwei Knoten in  $H$  höchstens länger sein kann als in  $G$ . Wenn  $\text{Stretch}(H) \leq t$  für ein  $t \geq 1$  gilt, so nennen wir  $H$  einen  $t$ -Spanner von  $G$ .

In Abb. 3.6 ist der Algorithmus *Greedy-Spanner* angegeben, der einen  $t$ -Spanner von  $G$  berechnet. Der Algorithmus beginnt mit dem leeren Graph  $H$  und betrachtet dann alle Kanten von  $G$  in der Reihenfolge aufsteigenden Gewichts, wobei die aktuelle Kante  $\{u, v\}$  immer dann in  $H$  eingefügt wird, wenn der kürzeste Pfad von  $u$  nach  $v$  in  $H$  länger als  $t \cdot c(\{u, v\})$  ist. Falls in  $H$  noch gar kein Pfad von  $u$  nach  $v$  enthalten ist, so ist  $d_H(u, v) = \infty$  und die Kante  $\{u, v\}$  wird auf jeden Fall in  $H$  eingefügt.

Wir bezeichnen die Summe der Gewichte aller Kanten in  $H$  wie üblich mit  $c(H)$  und das Gewicht eines minimalen Spannbaums in  $G$  mit  $MST(G)$ . Mit  $\lceil x \rceil$  bezeichnen wir die kleinste ganze Zahl, die grösser oder gleich  $x$  ist, und mit  $\lfloor x \rfloor$  die grösste ganze Zahl, die kleiner oder gleich  $x$  ist. Die Klammern  $\lceil \cdot \rceil$  bzw.  $\lfloor \cdot \rfloor$  werden *obere* bzw. *untere Gaussklammern* genannt.

**Satz 3.3** Der Algorithmus Greedy-Spanner liefert für einen Graphen  $G = (V, E)$  mit  $n$  Knoten und Kantengewichten  $c: E \rightarrow \mathbb{R}^+$  einen Teilgraphen  $H$ , für den gilt:

- (a)  $H$  ist ein  $t$ -Spanner von  $G$ .
- (b)  $H$  enthält höchstens  $n \cdot \left\lceil n^{\frac{2}{t-1}} \right\rceil$  Kanten.
- (c) Für jedes  $\delta$  mit  $0 < \delta < \min\{t-1, 1\}$  gilt  $c(H) \leq \left(5 + \frac{32t}{\delta^2}\right) \cdot n^{\frac{2+\delta}{t-1-\delta}} \cdot \text{MST}(G)$ .

Der Satz liefert obere Schranken sowohl für die Anzahl der Kanten von  $H$  als auch für das Gesamtgewicht aller Kanten von  $H$ . Den Beweis dieses Satzes teilen wir in eine Reihe von Lemmas auf.

### 3.3.1 Analyse des Stretch-Faktors

**Lemma 3.4** Der vom Algorithmus Greedy-Spanner berechnete Teilgraph  $H = (V, E')$  ist ein  $t$ -Spanner von  $G = (V, E)$ .

**Beweis.** Betrachte eine Kante  $e = \{u, v\}$ , die in  $G$  enthalten ist, aber nicht in  $H$ . Als der Algorithmus die Kante  $e$  bearbeitet hat, muss  $H$  bereits einen Pfad von  $u$  nach  $v$  der Länge höchstens  $t \cdot c(e)$  enthalten haben. Also enthält  $H$  für jede Kante  $e = \{u, v\} \in E$  einen Pfad von  $u$  nach  $v$  der Länge höchstens  $t \cdot c(e)$ .

Seien  $x$  und  $y$  zwei beliebige Knoten in  $V$  und sei  $p$  ein kürzester Pfad von  $x$  nach  $y$  in  $G$ . Wir können jede Kante  $e$  des Pfades  $p$  durch einen Pfad der Länge höchstens  $t \cdot c(e)$  in  $H$  ersetzen und erhalten so einen Pfad von  $x$  nach  $y$  der Länge höchstens  $t \cdot d_G(x, y)$  in  $H$ . Also gilt  $\text{Stretch}(H) \leq t$ .  $\square$

Damit haben wir Teil (a) von Satz 3.3 gezeigt.

### 3.3.2 Analyse der Kantenzahl des Spanners

Wir benötigen zunächst die folgende Aussage aus der Graphentheorie.

**Lemma 3.5** Sei  $H$  ein ungerichteter Graph mit  $n$  Knoten und  $m$  Kanten, in dem jeder Kreis aus mindestens  $g$  Kanten besteht. Dann gilt  $m \leq n \cdot \left\lceil n^{\frac{2}{g-2}} \right\rceil \leq 2 \cdot n^{1+\frac{2}{g-2}}$ .

**Beweis.** Falls  $m \leq 2n$  gilt, ist die Behauptung offensichtlich richtig. Daher nehmen wir nun an, dass  $m > 2n$ . Sei  $k = \lfloor \frac{m}{n} \rfloor + 1$ . Es gilt  $k \geq 3$ . Solange  $H$  einen Knoten mit Grad kleiner als  $k$  enthält, entfernen wir diesen Knoten und alle seine inzidenten Kanten aus  $H$ . Dadurch können nicht alle Knoten entfernt werden, weil wir beim Entfernen der ersten  $n-1$  Knoten höchstens  $(n-1)(k-1) < m$  Kanten entfernen und dann ja keine Kante mehr übrig sein kann, ein Widerspruch. Also bleibt ein Teilgraph  $H'$  von  $H$  übrig, in dem alle Knoten Grad mindestens  $k$  haben.

Betrachten wir zuerst den Fall, dass  $g$  ungerade ist,  $g = 2d+1$ . Sei  $x$  ein beliebiger Knoten in  $H'$  und betrachten wir alle Knoten, die von  $x$  aus über höchstens  $d$  Kanten erreicht werden können. Diese Knoten müssen einen Baum bilden, da alle Kreise in  $H$  und damit auch in  $H'$  aus mindestens  $2d+1$  Kanten bestehen (siehe Abb. 3.7, links). (Keine zwei der in diesem

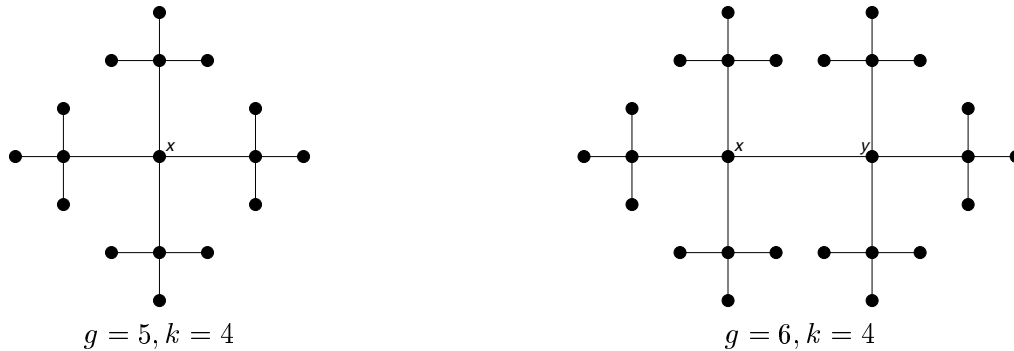


Abbildung 3.7: Bäume im Beweis von Lemma 3.5

Baum enthaltenen Knoten können identisch sein, da  $H'$  sonst einen Kreis der Länge höchstens  $2d$  enthalten würde.) Da alle Knoten Grad mindestens  $k$  haben, muss dieser Baum mindestens

$$1 + k \sum_{i=1}^d (k-1)^{i-1} = 1 + k \frac{(k-1)^d - 1}{k-2} = 1 + k \frac{(k-1)^{\frac{g-1}{2}} - 1}{k-2}$$

Knoten enthalten.

Im Fall, dass  $g$  gerade ist,  $g = 2d$ , betrachten wir eine beliebige Kante  $\{x, y\}$  in  $H'$  und alle Knoten, die von  $x$  oder  $y$  aus über höchstens  $d-1$  Kanten erreichbar sind. Wieder müssen diese Knoten einen Baum bilden (siehe Abb. 3.7, rechts). Der Baum enthält mindestens

$$2 \sum_{i=1}^d (k-1)^{i-1} = 2 \frac{(k-1)^d - 1}{k-2} = 2 \frac{(k-1)^{\frac{g}{2}} - 1}{k-2}$$

Knoten.

In beiden Fällen können wir folgern, dass die Anzahl Knoten in  $H'$  und damit auch die Anzahl Knoten in  $H$  grösser als  $(k-1)^{\frac{g}{2}-1}$  ist. Also gilt:

$$n > \left( \left\lfloor \frac{m}{n} \right\rfloor \right)^{\frac{g}{2}-1}$$

Das können wir umformen zu:

$$n^{\frac{2}{g-2}} > \left\lfloor \frac{m}{n} \right\rfloor$$

Dann gilt aber auch

$$\left\lceil n^{\frac{2}{g-2}} \right\rceil \geq \frac{m}{n}$$

und daraus folgt durch Multiplikation mit  $n$  die Behauptung des Lemmas.  $\square$

Um das Lemma verwenden zu können, müssen wir eine untere Schranke für die Länge von Kreisen in dem vom Algorithmus Greedy-Spanner berechneten  $t$ -Spanner  $H$  herleiten.

**Lemma 3.6** *Jeder Kreis in  $H$  besteht aus mehr als  $t+1$  Kanten.*

**Beweis.** Beweis durch Widerspruch. Nehmen wir an, es gäbe in  $H$  einen Kreis  $C$  aus höchstens  $t+1$  Kanten. Sei  $\{u, v\}$  die letzte Kante des Kreises, die der Algorithmus in  $H$  eingefügt hat.

**Algorithmus von Kruskal**  
**Eingabe:** zshgd. Graph  $G = (V, E)$  mit Kantengewichten  $c : E \rightarrow \mathbb{R}^+$   
**Ausgabe:** Spannbaum  $T = (V, E')$  von  $G$  mit minimalem Gewicht

```

sortiere die Kanten in  $E = \{e_1, e_2, \dots, e_m\}$  so, dass  $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$  gilt;
 $E' := \emptyset$ ;
 $T := (V, E')$ ;
for  $i = 1$  to  $m$  do
    // sei  $e_i = \{u, v\}$ 
    if  $u$  und  $v$  sind in verschiedenen Zusammenhangskomponenten von  $T$  then
         $E' := E' \cup \{e_i\}$ ;
         $T := (V, E')$ ;
    fi
od;
return  $T$ ;

```

Abbildung 3.8: Algorithmus von Kruskal

Der Rest des Kreises besteht aus höchstens  $t$  Kanten, deren Gewicht nicht grösser als das von  $\{u, v\}$  ist. Also enthielt  $H$  vor dem Einfügen der Kante  $\{u, v\}$  bereits einen Pfad von  $u$  nach  $v$  mit Gewicht höchstens  $t \cdot c(\{u, v\})$ , ein Widerspruch zur Definition des Algorithmus.  $\square$

Aus Lemma 3.5 und Lemma 3.6 folgt, dass  $H$  höchstens  $n \cdot \left\lceil n^{\frac{2}{t-1}} \right\rceil$  Kanten enthält. Damit haben wir auch Teil (b) von Satz 3.3 gezeigt.

### 3.3.3 Analyse des Gewichts des Spanners

Nun kommen wir zum Teil (c) von Satz 3.3, dem Teil, für den der Beweis am aufwendigsten ist.

Sei  $T$  der minimale Spannbaum von  $G$ , der mit dem Algorithmus von Kruskal (siehe Abb. 3.8) berechnet wird, wenn die Kanten in derselben Reihenfolge bearbeitet werden wie vom Algorithmus Greedy-Spanner. (Die Korrektheit des Algorithmus von Kruskal folgt aus Satz 2.1 aus Kapitel 2.1.) Mit  $E(T)$  bezeichnen wir die Menge der Kanten von  $T$ , mit  $MST$  das Gewicht von  $T$ . Die Kanten von  $H$ , dem vom Algorithmus berechneten  $t$ -Spanner, bezeichnen wir weiterhin mit  $E'$ .

**Lemma 3.7**  $E'$  enthält alle Kanten des minimalen Spannbaums  $T$ , d.h. es gilt  $E(T) \subseteq E'$ .

Den Beweis dieses Lemmas überlassen wir dem Leser als Übungsaufgabe.

Sei  $P$  die Tour, die einmal "um  $T$  herum" läuft (vgl. Abb. 3.9, links). Anders ausgedrückt ist  $P$  der Weg, den ein Präorderdurchlauf durch  $T$  zurücklegt (vgl. Kapitel 3.2.1). Sei  $L$  die Summe der Kantengewichte von  $P$ . Da jede Kante von  $T$  genau zweimal in  $P$  enthalten ist, gilt  $L = 2 \cdot c(T) = 2 \cdot MST$ . Wir ordnen jeden Knoten von  $T$  der Stelle auf der Tour zu, wo der Knoten zum ersten Mal besucht wird. Damit können wir die Tour als einen Kreis auffassen, der alle Knoten in der Reihenfolge ihrer Präordernummerierung enthält und in dem der Abstand zweier aufeinanderfolgender Knoten  $u$  und  $v$  gleich  $d_T(u, v)$  ist (siehe Abb. 3.9, rechts). Offensichtlich gilt immer  $d_P(u, v) \geq d_T(u, v)$ . Im Beispiel aus Abb. 3.9 gilt etwa  $d_P(9, 13) = 23 > 9 = d_T(9, 13)$ .

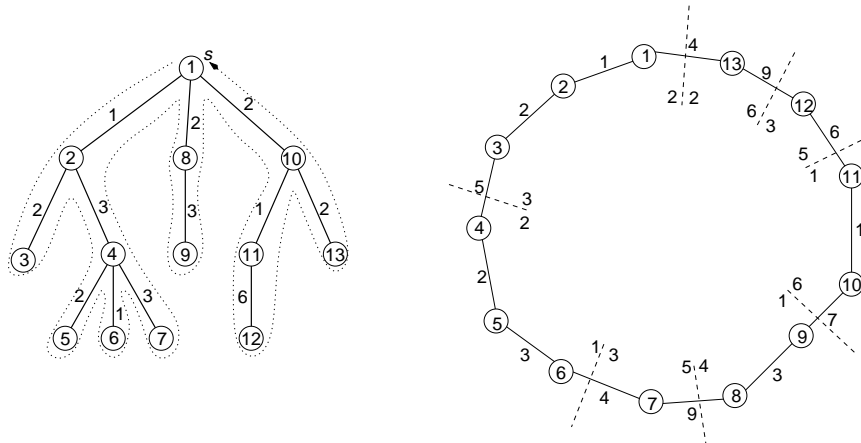


Abbildung 3.9: Die Tour  $P$  um den minimalen Spannbaum

Wir teilen die Ausführung des Algorithmus Greedy-Spanner in  $\log n + 1$  Phasen ein. Hierbei steht  $\log n$  für  $\log_2 n$  und wir nehmen ausserdem an, dass  $n$  eine Zweierpotenz ist. (Falls  $n$  keine Zweierpotenz ist, werden nur die Rechnungen etwas umständlicher, es ändert sich aber nichts Wesentliches.)

Es ist leicht zu sehen, dass das Gewicht jeder Kante in  $T$  oder  $H$  im Intervall  $[0, L]$  liegen muss. Wir partitionieren das Intervall  $[0, L]$  in  $\log n + 1$  Teilintervalle:

$$I_0 = \left[0, \frac{L}{n}\right]$$

$$I_j = \left(2^{j-1} \cdot \frac{L}{n}, 2^j \cdot \frac{L}{n}\right] \text{ für } j = 1, 2, \dots, \log n$$

Innerhalb einer Phase bearbeitet der Algorithmus Kanten, deren Gewichte im selben Intervall  $I_j$  liegen. Für  $0 \leq j \leq \log n$  definieren wir:

$$E_j = \{e \in E' \setminus E(T) \mid c(e) \in I_j\}$$

$E_j$  ist also die Menge derjenigen Kanten, die der Algorithmus in Phase  $j$  in  $E'$  einfügt und die *nicht* im minimalen Spannbaum  $T$  enthalten sind. Nun schätzen wir das Gewicht der Kanten in den Mengen  $E_j$  ab.

**Lemma 3.8**  $c(E_0) \leq 2 \cdot \left\lceil n^{\frac{2}{t-1}} \right\rceil \cdot MST \leq 4 \cdot n^{\frac{2}{t-1}} \cdot MST$ .

**Beweis.** Aus Teil (b) von Satz 3.3 wissen wir, dass  $E'$  und damit auch  $E_0$  aus höchstens  $n \cdot \left\lceil n^{\frac{2}{t-1}} \right\rceil$  Kanten besteht. Jede Kante in  $E_0$  hat Gewicht höchstens  $\frac{L}{n} = \frac{2}{n} \cdot MST$ . Also gilt  $c(E_0) \leq n \cdot \left\lceil n^{\frac{2}{t-1}} \right\rceil \cdot \frac{2}{n} \cdot MST = 2 \cdot \left\lceil n^{\frac{2}{t-1}} \right\rceil \cdot MST$ .  $\square$

Betrachten wir nun  $E_j$  für  $j \geq 1$ . Sei  $a = 2^{j-1} \cdot \frac{L}{n}$ , so dass  $I_j = (a, 2a]$ . Alle in Phase  $j$  bearbeiteten Kanten haben ein Gewicht, das in  $I_j$  enthalten ist.

Sei ein  $\delta$  mit  $0 < \delta < \min\{t-1, 1\}$  vorgegeben. Wir teilen den Kreis  $P$  in  $\frac{2L}{\delta a}$  Intervalle der Grösse  $\frac{\delta a}{2}$  ein, indem wir an einer beliebigen Stelle einen Schnitt machen und von dort aus um den Kreis herumlaufen und nach jeweils  $\frac{\delta a}{2}$  zurückgelegten Längeneinheiten wieder einen

Schnitt machen. Die resultierenden Intervalle nennen wir *Cluster*. In Abb. 3.9 ist eine Einteilung des Kreises in Intervalle der Grösse 8 durch die gestrichelten Schnitte angedeutet, wobei bei jedem Schnitt angegeben ist, in welche beiden Teile die betreffende Kante zerschnitten wird.

Sei  $n_j$  die Anzahl der Cluster, die mindestens einen Knoten aus  $V$  enthalten. Offensichtlich gilt  $n_j \leq \frac{2L}{\delta a} = \frac{2n}{\delta 2^{j-1}}$ . Eine Kante  $\{u, v\}$  nennen wir *Intercluster-Kante*, wenn  $u$  und  $v$  in verschiedenen Clustern liegen. In dem Beispiel aus Abb 3.9 enthalten alle 7 Cluster Knoten aus  $V$ . Die Cluster sind hier  $\{1, 2, 3\}$ ,  $\{4, 5, 6\}$ ,  $\{7\}$ ,  $\{8, 9\}$ ,  $\{10, 11\}$ ,  $\{12\}$  und  $\{13\}$ .

**Lemma 3.9** *Jede Kante  $\{u, v\} \in E_j$  ist eine Intercluster-Kante.*

**Beweis.** Nach Definition von  $E_j$  gilt  $\{u, v\} \notin E(T)$ . Betrachte den Pfad  $Q$ , der  $u$  und  $v$  in  $T$  verbindet. Alle Kanten auf  $Q$  haben Gewicht höchstens  $c(\{u, v\})$  (sonst wäre  $T$  kein minimaler Spannbaum) und wurden von Kruskal's Algorithmus und damit auch vom Algorithmus Greedy-Spanner vor der Kante  $\{u, v\}$  bearbeitet. Da nach Lemma 3.7  $E(T) \subseteq E'$  gilt, muss  $Q$  bereits in  $H$  enthalten gewesen sein, als Greedy-Spanner die Kante  $\{u, v\}$  bearbeitet hat. Da  $\{u, v\} \in E'$ , muss  $c(Q) > t \cdot c(\{u, v\})$  gelten. Aus  $\delta < t - 1$  folgt  $t > 1 + \delta$ . Damit erhalten wir:

$$d_P(u, v) \geq d_T(u, v) = c(Q) > t \cdot c(\{u, v\}) > (1 + \delta)c(\{u, v\}) > \delta a$$

Der Abstand zwischen zwei Knoten innerhalb eines Clusters ist höchstens  $\frac{\delta a}{2}$ . Also liegen  $u$  und  $v$  in verschiedenen Clustern.  $\square$

**Lemma 3.10**  $|E_j| \leq 2 \cdot n_j^{\min\{2, 1 + \frac{2+\delta}{t-1-\delta}\}}$ .

**Beweis.** Da  $E_j$  nur Intercluster-Kanten enthält, gilt offensichtlich  $|E_j| \leq n_j^2$ . Wir müssen daher nur noch  $|E_j| \leq 2 \cdot n_j^{1 + \frac{2+\delta}{t-1-\delta}}$  zeigen. Sei  $M$  der Graph, der aus  $(V, E_j)$  entsteht, indem man die Knoten jedes Clusters zu einem einzigen Knoten verschmilzt. Sei  $C$  ein beliebiger Kreis in  $M$ , der aus  $g$  Kanten besteht. Wir wollen zeigen, dass  $g \geq \frac{t+1}{1+\delta}$  gilt.

Seien  $w_1, w_2, \dots, w_g$  die Gewichte der  $g$  Kanten von  $C$  in aufsteigender Reihenfolge. Als die letzte Kante (mit Gewicht  $w_g$ ) in  $H$  eingefügt wurde, gab es in  $H$  bereits einen Pfad der Länge höchstens  $g \cdot \frac{\delta a}{2} + \sum_{i=1}^{g-1} w_i$  zwischen den Endknoten der Kante: der Term  $g \cdot \frac{\delta a}{2}$  schätzt die höchstens  $g$  Teilpfade innerhalb eines Clusters ab, der Term  $\sum_{i=1}^{g-1} w_i$  die Intercluster-Kanten. Da die Kante mit Gewicht  $w_g$  in  $H$  eingefügt wurde, muss  $g \cdot \frac{\delta a}{2} + \sum_{i=1}^{g-1} w_i > t \cdot w_g$  gelten. Wegen  $a \leq w_g$  und  $w_i \leq w_g$  für  $1 \leq i \leq g - 1$  können wir folgern:

$$t \cdot w_g < g \cdot \frac{\delta w_g}{2} + (g - 1)w_g$$

Das ergibt  $t < g(\frac{\delta}{2} + 1) - 1$  und damit  $g > \frac{t+1}{1+\frac{\delta}{2}}$ . Also enthält jeder Kreis in  $M$  mehr als  $\frac{t+1}{1+\frac{\delta}{2}}$  Kanten. Mit Lemma 3.5 erhalten wir  $|E_j| \leq 2 \cdot n_j^{1 + \frac{2+\delta}{t-1-\delta}}$ .  $\square$

**Lemma 3.11**  $c(E_j) \leq \frac{32}{\delta^2} \cdot n^{\frac{2+\delta}{t-1-\delta}} \cdot MST \cdot \left(\frac{t-1}{t}\right)^{j-1}$ .

**Beweis.** Jede Kante in  $E_j$  hat Gewicht höchstens  $2a$ , daher gilt mit Lemma 3.10:

$$\begin{aligned}
c(E_j) &\leq 2a \cdot 2 \cdot n_j^{\min\{2, 1 + \frac{2+\delta}{t-1-\delta}\}} \\
&\leq 2 \cdot 2^{j-1} \frac{L}{n} \cdot 2 \cdot \left( \frac{2n}{\delta 2^{j-1}} \right)^{\min\{2, 1 + \frac{2+\delta}{t-1-\delta}\}} \\
&\leq 2 \cdot 2^{j-1} \frac{L}{n} \cdot 2 \cdot \left( \frac{2}{\delta} \right)^2 \cdot \left( \frac{n}{2^{j-1}} \right)^{1 + \frac{2+\delta}{t-1-\delta}} \\
&\leq 2 \cdot 2^{j-1} \frac{L}{n} \cdot 2 \cdot \left( \frac{2}{\delta} \right)^2 \cdot \frac{n}{2^{j-1}} \cdot \left( \frac{n}{2^{j-1}} \right)^{\frac{2+\delta}{t-1-\delta}} \\
&= \frac{16L}{\delta^2} \cdot \left( \frac{n}{2^{j-1}} \right)^{\frac{2+\delta}{t-1-\delta}} \\
&= \frac{16L}{\delta^2} \cdot n^{\frac{2+\delta}{t-1-\delta}} \cdot \left( \frac{1}{2^{j-1}} \right)^{\frac{2+\delta}{t-1-\delta}} \\
&\leq \frac{32}{\delta^2} \cdot MST \cdot n^{\frac{2+\delta}{t-1-\delta}} \cdot \left( \frac{1}{4^{j-1}} \right)^{j-1} \\
&\leq \frac{32}{\delta^2} \cdot MST \cdot n^{\frac{2+\delta}{t-1-\delta}} \cdot \left( \frac{t-1}{t} \right)^{j-1}
\end{aligned}$$

Die letzte Ungleichung in der Kette gilt, weil  $4^{\frac{1}{t-1}} \geq e^{\frac{1}{t-1}} \geq 1 + \frac{1}{t-1}$  für alle  $t > 1$  richtig ist.  $\square$

Mit Lemma 3.8 und Lemma 3.11 können wir wegen  $E' = E(T) \cup E_0 \cup E_1 \cup \dots \cup E_{\log n}$  das Gewicht  $c(H)$  wie folgt abschätzen:

$$\begin{aligned}
c(H) &\leq MST + 4 \cdot n^{\frac{2}{t-1}} \cdot MST + \\
&\quad \frac{32}{\delta^2} \cdot n^{\frac{2+\delta}{t-1-\delta}} \cdot MST \cdot \sum_{j=1}^{\log n} \left( \frac{t-1}{t} \right)^{j-1} \\
&\leq MST + 4 \cdot n^{\frac{2}{t-1}} \cdot MST + \frac{32}{\delta^2} \cdot n^{\frac{2+\delta}{t-1-\delta}} \cdot MST \cdot t \\
&\leq 5 \cdot n^{\frac{2}{t-1}} \cdot MST + \frac{32t}{\delta^2} \cdot n^{\frac{2+\delta}{t-1-\delta}} \cdot MST \\
&\leq \left( 5 + \frac{32t}{\delta^2} \right) \cdot n^{\frac{2+\delta}{t-1-\delta}} \cdot MST
\end{aligned}$$

Damit haben wir auch Teil (c) von Satz 3.3 bewiesen. Durch Festsetzen des Parameters  $t$  zu  $\log_2 n$  und unter Verwendung von  $\delta = \frac{1}{2}$  (zum Beispiel) bekommen wir aus Satz 3.3 das folgende Korollar.

**Korollar 3.12** *Wenn der Algorithmus Greedy-Spanner mit Parameter  $t = \log_2 n$  aufgerufen wird, so liefert er einen  $t$ -Spanner  $H$  mit  $O(n)$  Kanten und  $c(H) = O(\log n) \cdot MST$ .*

Mit  $t = \log_2 n$  liefert der Algorithmus Greedy-Spanner also einen  $\log n$ -Spanner, der nicht viele Kanten enthält (nur  $O(n)$  Stück) und bei dem das Gesamtgewicht aller Kanten höchstens um einen Faktor  $O(\log n)$  grösser ist als das Gewicht des minimalen Spannbaums. Genau diesen Spanner werden wir in Kapitel 3.6 zur Lösung eines Netzwerkdesign-Problems benutzen.

### 3.4 Modellierung eines Netzwerkdesign-Problems

Gegeben ist eine Menge  $V$  von  $n$  Knoten, die vernetzt werden sollen. Die Knoten sind von 1 bis  $n$  durchnummeriert. Für jedes Knotenpaar  $(i, j)$  ist eine Verkehrsanforderung  $r_{i,j}$  gegeben, die bestimmt, wie viel Verkehr pro Zeiteinheit von  $i$  nach  $j$  geschickt werden soll. Wie nehmen an, dass die  $r_{i,j}$  als Vielfache einer gewissen Basis-Einheit (z.B. 2 Mbps) gegeben sind. Nicht-ganzzahlige Vielfache sind zugelassen.

Eine Lösung des Problems gibt an, welche Links mit welcher Kapazität zwischen den Knoten installiert werden und über welchen Pfad jede einzelne Anforderung geroutet wird. Die Kapazität jedes Links muss dabei mindestens so gross sein wie die Summe aller Anforderungen, die durch den Link geroutet werden.

Die Kosten einer Lösung ergeben sich als Summe der Kosten für jeden einzelnen Link. Die Kosten eines Links bestehen aus zwei Komponenten: erstens fallen kapazitätsunabhängige Grundkosten  $F$  an, die für jeden Link gleich sind; zweitens fallen kapazitätsabhängige Kosten an, die von der Länge des Links und der Kapazität abhängen. Die Kapazität kann nur als ganzzahliges Vielfaches der Basis-Einheit gewählt werden. Die Länge des Links von  $i$  nach  $j$  ist durch einen Wert  $p_{i,j}$  vorgegeben. Wir nehmen an, dass für die Einrichtung des Links zwischen  $i$  und  $j$  mit Kapazität  $u_{i,j}$  die kapazitätsabhängigen Kosten  $p_{i,j}u_{i,j}$  betragen. Die Gesamtkosten für den Link sind also  $F + p_{i,j}u_{i,j}$ .

Wir beobachten, dass sich die einzurichtenden Links und ihre Kapazität direkt aus dem Routing der Anforderungen ableiten lassen. Nehmen wir an, dass jede Anforderung  $r_{i,j}$  entlang einem Pfad  $P_{i,j}$  geroutet wurde. Für jede Kante  $e$  bezeichnet  $q(e)$  die Summe aller Anforderungswerte  $r_{i,j}$ , deren Pfad  $P_{i,j}$  die Kante  $e$  enthält. Dann ist klar, dass wir genau diejenigen Links  $e$  mit  $q(e) > 0$  einrichten müssen und dass wir die Kapazität eines eingerichteten Links  $e$  am günstigsten als  $\lceil q(e) \rceil$  wählen. Mit  $E'$  bezeichnen wir für ein Routing die Menge der Kanten  $e$  mit  $q(e) > 0$ .

Wir nennen das so modellierte Problem “Netzwerkdesign mit einem Kabeltyp”, da wir die Einrichtung eines Links mit Kapazität  $u_{i,j}$  so auffassen können, dass  $u_{i,j}$  Kabel desselben Typs zwischen  $i$  und  $j$  installiert werden. In allgemeineren Modellen würde man annehmen, dass man verschiedene Kabeltypen mit verschiedenen Kapazitäten und Preisen zur Auswahl hat. Zusammengefasst ist unser Problem also wie folgt definiert.

Problem Netzwerkdesign mit einem Kabeltyp (ND1K)

**Instanz:** Menge  $V$  von  $n$  Knoten, Anforderungsmatrix  $R = (r_{i,j})$ , Fixkosten  $F$ , Preismatrix  $P = (p_{i,j})$

**Lösung:** Pfad  $P_{i,j}$  für jede Anforderung  $r_{i,j} > 0$ ,  $1 \leq i < j \leq n$

**Ziel:** minimiere  $\sum_{e=\{i,j\} \in E'} (F + \lceil q(e) \rceil \cdot p_{i,j})$

Wir nehmen an, dass das entworfene Netz auf jeden Fall zusammenhängend sein muss, so dass auch die Optimallösung mindestens  $n - 1$  Kanten enthalten muss. Ausserdem nehmen wir an, dass die Matrizen  $R$  und  $P$  symmetrisch sind. Wir betrachten das geplante Netz als ungerichteten Graphen und die Verkehrsströme als ungerichtete Pfade. Schliesslich nehmen wir noch an, dass für die Preise  $p_{i,j}$  die *Dreiecksungleichung* gilt, d.h. es gilt immer  $p_{i,j} \leq p_{i,k} + p_{k,j}$  für alle  $k$ . Diese Annahme ist sinnvoll, da man ja notfalls die Verbindung von  $i$  nach  $j$  über den Knoten  $k$  einrichten könnte, wenn dieser Weg tatsächlich billiger wäre als die direkte Verbindung.

Man beachte, dass diese Problemformulierung eine Vereinfachung von in der Praxis auftretenden Netzwerkdesign-Problemen darstellt. In der Praxis hängen zum Beispiel die Kosten

eines Links oft nicht linear von der Kapazität des Links ab. Ausserdem müssen oft zusätzliche Einschränkungen berücksichtigt werden, etwa Begrenzung des Netzwerkdurchmessers und Anforderungen bzgl. der erlaubten Verzögerungen und der Ausfallsicherheit des Netzes.

Um Approximationsalgorithmen für ND1K analysieren zu können, benötigen wir untere Schranken für die Kosten  $OPT$  einer optimalen Lösung.

**Lemma 3.13** *Für die Kosten  $OPT$  einer optimalen Lösung von ND1K gelten die folgenden beiden unteren Schranken:*

(a)  $OPT \geq F \cdot (n - 1) + MST$ , wobei  $MST$  das Gewicht eines minimalen Spannbaums im vollständigen Graphen auf  $n$  Knoten mit Kantengewichten  $p_{i,j}$  ist.

(b)  $OPT \geq F \cdot (n - 1) + \sum_{1 \leq i < j \leq n} r_{i,j} \cdot p_{i,j}$

**Beweis.** Die Schranke (a) folgt daraus, dass das Netz nach Annahme zusammenhängend sein muss und damit einen Spannbaum enthalten muss, in dem jede Kante Kapazität mindestens 1 hat. Die kapazitätsunabhängigen Kosten für die  $n - 1$  Kanten in diesem Spannbaum sind  $F \cdot (n - 1)$  und die kapazitätsabhängigen Kosten sind mindestens  $MST$ , da kein Spannbaum mit Kantenkapazität 1 weniger kapazitätsabhängige Kosten haben kann als der minimale Spannbaum.

Die Schranke (b) erhält man folgendermassen. Erstens muss das Netz  $n - 1$  Kanten enthalten (sonst wäre es nicht zusammenhängend), daher betragen die kapazitätsunabhängigen Kosten mindestens  $F \cdot (n - 1)$ . Zweitens verursacht jede Anforderung  $r_{i,j}$  mindestens kapazitätsabhängige Kosten  $r_{i,j} \cdot p_{i,j}$ , da kein Pfad von  $i$  nach  $j$  eine Länge kleiner als  $p_{i,j}$  haben kann (für dieses Argument benötigen wir, dass die Dreiecksungleichung für die Preismatrix  $P = (p_{i,j})$  gilt).  $\square$

Im Folgenden wollen wir das Problem ND1K zuerst für den Spezialfall betrachten, dass die Anforderungsmatrix eine besondere Struktur hat, dass nämlich jede Anforderung einen ausgezeichneten Knoten  $s$  mit einem anderen Knoten verbindet. Dieses Problem tritt beim Design von Access-Netzen auf, wo eine Reihe von Endkunden mit einem speziellen Vermittlungsknoten verbunden werden müssen.

### 3.5 Design von Access-Netzen

Als Access-Netze werden allgemein diejenigen Teile eines Kommunikationsnetzes bezeichnet, mit denen Endkunden bzw. Endgeräte (z.B. Telefonanschlüsse) an das Netz angebunden werden. Hier denkt man sich die Vernetzung auf zwei Hierarchie-Ebenen: das Backbone-Netz (Rückgrat) ist üblicherweise sehr gut vernetzt und hat eine grosse Kapazität. Endkunden sind über ein Access-Netz an einen oder mehrere Backbone-Knoten angeschlossen. Eine Verbindung zwischen zwei Endkunden  $A$  und  $B$  geht dann von  $A$  über das Access-Netz zum nächsten Backbone-Knoten, von diesem über das Backbone-Netz zum für  $B$  zuständigen Backbone-Knoten, und schliesslich über ein (anderes) Access-Netz von dort aus zu  $B$ .

Auch beim Entwurf von Netzen wird oft diese Zweiteilung in Backbone und Access gemacht. Zuerst wird entschieden, wo überall Backbone-Knoten installiert werden sollen. Dann wird das Backbone-Netz, das alle Backbone-Knoten miteinander verbindet, entworfen. Die Endkunden werden jeweils einem Backbone-Knoten zugeordnet. Das Access-Netz, das einen Backbone-Knoten mit allen ihm zugeordneten Endkunden (Access-Knoten) verbindet, kann

**Algorithmus: Netzwerkdesign mit einer Quelle**  
**Eingabe:** Menge  $V$  mit  $n$  Knoten (Knoten 1 ist die Quelle), Anforderungen  $r_{1,i}$  für  $2 \leq i \leq n$ , Fixkosten  $F$ , Preismatrix  $P = (p_{i,j})$   
**Ausgabe:** Pfade  $P_{1,i}$  für  $2 \leq i \leq n$

1. berechne  $(1 + \sqrt{2}, 1 + \sqrt{2})$ -LAST  $T$  mit Startknoten 1 in dem vollständigen Graphen mit Knotenmenge  $V$  und Kantengewichten  $p_{i,j}$ ;
2.  $P_{1,i} :=$  der Pfad von 1 nach  $i$  in  $T$  (für alle  $i = 2, \dots, n$ );

Abbildung 3.10: Algorithmus für ND1K mit einer Quelle

dann getrennt vom Rest des Netzes optimiert werden. In der Praxis ist das Design von Netzen natürlich ein iterativer Prozess, und Änderungen an einem Teil des Netzes haben wieder Auswirkungen auf andere Teile, die dann neu optimiert werden müssen. Trotzdem ist diese konzeptuelle Aufteilung in relativ unabhängig voneinander zu bearbeitende Teilprobleme oft sehr nützlich.

Im Folgenden werden wir sehen, wie wir mit Hilfe eines Algorithmus zur Berechnung eines  $(\alpha, \beta)$ -LAST auch die Variante von ND1K für Access-Netze approximieren können.

### 3.5.1 Ein Approximationsalgorithmus für das Design von Access-Netzen

Wir betrachten die Variante des Problems ND1K, wo alle Anforderungen einen ausgezeichneten Knoten, sagen wir den Knoten 1, mit einem anderen Knoten verbinden. Wir nennen den Knoten 1 *Quelle* und das sich ergebende Problem ND1K *mit einer Quelle*. Es ist wie folgt definiert.

Problem ND1K mit einer Quelle  
**Instanz:** Menge  $V$  von  $n$  Knoten, Anforderungen  $r_{1,i} > 0$  für  $2 \leq i \leq n$ , Fixkosten  $F$ , Preismatrix  $P = (p_{i,j})$   
**Lösung:** Pfad  $P_{1,i}$  für jede Anforderung  $r_{1,i}$ ,  $2 \leq i \leq n$   
**Ziel:** minimiere  $\sum_{e=\{i,j\} \in E'} (F + \lceil q(e) \rceil \cdot p_{i,j})$

Dabei ist  $q(e)$  wieder die Summe aller Anforderungen, deren Pfade den Link  $e$  enthalten, und  $E'$  die Menge aller Kanten mit  $q(e) > 0$ . Man beachte, dass wir  $r_{1,i} > 0$  für alle  $2 \leq i \leq n$  voraussetzen, deshalb muss die Quelle wirklich mit allen anderen Knoten verbunden werden.

In Abb. 3.10 ist ein Algorithmus für das Problem ND1K mit einer Quelle angegeben. Der Algorithmus berechnet zuerst in dem vollständigen Graphen  $G$  mit Knotenmenge  $V$  und Kantengewichten  $p_{i,j}$  einen  $(1 + \sqrt{2}, 1 + \sqrt{2})$ -LAST  $T$  mit Startknoten 1 (unter Verwendung des Algorithmus aus Abb. 3.5). Dann wird für  $2 \leq i \leq n$  als Pfad  $P_{1,i}$  für Anforderung  $r_{1,i}$  der eindeutige Pfad von 1 nach  $i$  in  $T$  verwendet. Offensichtlich läuft der Algorithmus in polynomieller Zeit.

Nun wollen wir die Approximationsrate bestimmen, die der Algorithmus erzielt.

**Satz 3.14** *Der Algorithmus aus Abb. 3.10 liefert ein Routing, bei dem die Kosten für das Netz höchstens  $(2+2\sqrt{2}) \cdot OPT$  sind. Der Algorithmus ist somit ein  $(2+2\sqrt{2})$ -Approximationsalgorithmus für ND1K mit einer Quelle.*

**Beweis.** Bezeichne mit  $E'$  die Menge der Kanten des  $(\alpha, \beta)$ -LAST  $T$ , die der Algorithmus in Schritt 1 berechnet. Jede dieser Kanten ist am Ende auch in mindestens einem der berechneten

Pfade  $P_{1,i}$  enthalten. Wir erinnern uns, dass für  $e = \{i, j\} \in E'$  der Wert  $q(e)$  die Summe der durch  $e$  gerouteten Anforderungen angibt. Wir definieren  $\Delta(e) = \lceil q(e) \rceil - q(e)$ . Damit ist  $\Delta(e)$  der Teil der Kapazität von  $e$ , den wir zwar bezahlen müssen, den wir aber eigentlich nicht benötigen. Die Kosten des Netzes, das sich aus dem Routing des Algorithmus ergibt, sind  $C = F \cdot (n - 1) + \sum_{e=\{i,j\} \in E'} \lceil q(e) \rceil \cdot p_{i,j}$ . Diese Kosten können wir in zwei Teile  $A$  und  $B$  aufspalten (d.h.  $C = A + B$ ):

$$\begin{aligned} A &= \sum_{e=\{i,j\} \in E'} q(e) p_{i,j} \\ B &= F \cdot (n - 1) + \sum_{e=\{i,j\} \in E'} \Delta(e) p_{i,j} \end{aligned}$$

Der Beitrag einer Anforderung  $r_{1,i}$  zu der obigen Summe für  $A$  ist offensichtlich  $r_{1,i} d_T(1, i)$ , wobei  $d_T(1, i)$  die Länge des Pfades von 1 nach  $i$  in  $T$  bezüglich der Kantengewichte  $p_{j,k}$  ist. Wir können  $A$  daher umformulieren zu

$$A = \sum_{i=2}^n r_{1,i} d_T(1, i).$$

Da  $T$  ein  $(\alpha, \beta)$ -LAST ist, gilt  $d_T(1, i) \leq \alpha d_G(1, i) = \alpha p_{1,i}$ . (Da die Kantengewichte  $p_{i,j}$  die Dreiecksungleichung erfüllen, gilt  $d_G(i, j) = p_{i,j}$  für alle  $i, j$ .) Wir erhalten somit  $A \leq \alpha \sum_{i=2}^n r_{1,i} \cdot p_{1,i}$ . Mit Ungleichung (b) aus Lemma 3.13 erhalten wir daher  $A \leq \alpha OPT$ .

Versuchen wir nun,  $B$  abzuschätzen. Da  $\Delta(e) < 1$  gilt, folgt  $B \leq F \cdot (n - 1) + \sum_{e=\{i,j\} \in E'} p_{i,j}$ . Da  $T$  ein  $(\alpha, \beta)$ -LAST ist, gilt  $\sum_{e=\{i,j\} \in E'} p_{i,j} \leq \beta \cdot MST$ . Mit Ungleichung (a) aus Lemma 3.13 folgt daher  $B \leq \beta OPT$ .

Somit erhalten wir  $C = A + B \leq (\alpha + \beta) OPT$ . Da nach Satz 3.2  $\beta = 1 + \frac{2}{\alpha - 1}$  gewählt werden muss, wird  $\alpha + \beta$  minimal, wenn wir  $\alpha = \beta = 1 + \sqrt{2}$  setzen. (Um das zu sehen, muss man nur das Minimum der Funktion  $f(x) = x + 1 + \frac{2}{x-1}$  bestimmen.) Das ist der Grund für die Wahl von  $\alpha = 1 + \sqrt{2}$  und  $\beta = 1 + \sqrt{2}$  bei der Berechnung des  $(\alpha, \beta)$ -LAST in Schritt 1 des Algorithmus. Damit erhalten wir Approximationsrate  $2 + 2\sqrt{2} \approx 4.828$ .  $\square$

### 3.6 Allgemeines Netzwerkdesign

Nun beschäftigen wir uns mit der allgemeinen Version des Problems ND1K. Für die Entwicklung eines Approximationsalgorithmus gehen wir sehr ähnlich vor wie beim Spezialfall des Problems mit einer Quelle. Wir bestimmen zuerst einen Teilgraphen, der nicht viel teurer als ein minimaler Spannbaum ist, der nicht viele Kanten enthält und in dem die kürzesten Wege nicht viel länger sind als im vollständigen Graphen mit Kantengewichten  $p_{i,j}$ . Im Gegensatz zum Design von Access-Netzen, wo wir als Teilgraph den  $(\alpha, \beta)$ -LAST gewählt haben, muss jetzt der gewählte Teilgraph nicht nur kurze Pfade zu einer ausgezeichneten Quelle enthalten, sondern für beliebige Paare von Knoten. Daher verwenden wir jetzt die Spanner aus Abschnitt 3.3. Am Ende werden dann alle Anforderungen entlang kürzester Pfade in diesem Spanner geroutet.

**Algorithmus: Netzwerkdesign**  
**Eingabe:** Menge  $V$  von  $n$  Knoten, Anforderungsmatrix  $R = (r_{i,j})$ ,  
 Fixkosten  $F$ , Preismatrix  $P = (p_{i,j})$   
**Ausgabe:** Pfade  $P_{i,j}$  für alle  $1 \leq i < j \leq n$  mit  $r_{i,j} > 0$

1. berechne mit Algorithmus Greedy-Spanner einen  $\log n$ -Spanner  $H$  in dem vollständigen Graphen mit Knotenmenge  $V$  und Kantengewichten  $p_{i,j}$ ;
2.  $P_{i,j} :=$  ein kürzester Pfad von  $i$  nach  $j$  in  $H$  (für alle  $1 \leq i < j \leq n$  mit  $r_{i,j} > 0$ )

Abbildung 3.11: Algorithmus für ND1K

### 3.6.1 Ein Approximationsalgorithmus für allgemeines Netzwerkdesign

Der Approximationsalgorithmus für ND1K, der in Abb. 3.11 angegeben ist, ist analog zu dem, den wir für ND1K mit einer Quelle verwendet haben (Abb. 3.10). Die Analyse des Algorithmus ist ebenfalls analog zum Fall mit einer Quelle.

**Satz 3.15** *Der Algorithmus aus Abb. 3.11 liefert für Probleme mit  $n$  Knoten ein Routing, bei dem die Kosten für das Netz höchstens  $O(\log n) \cdot OPT$  sind. Der Algorithmus ist somit ein  $O(\log n)$ -Approximationsalgorithmus für ND1K.*

**Beweis.** Bezeichne mit  $\hat{E}$  die Menge der Kanten des  $\log n$ -Spanners  $H$ , den der Algorithmus in Schritt 1 (mit Hilfe des Algorithmus Greedy-Spanner aus Abb. 3.6) berechnet. Sei  $E' \subseteq \hat{E}$  die Menge der Kanten, die am Ende in mindestens einem der berechneten Pfade  $P_{i,j}$  enthalten sind. Wir erinnern uns, dass für  $e = \{i, j\} \in E'$  der Wert  $q(e)$  die Summe der durch  $e$  gerouteten Anforderungen angibt. Wir definieren wieder  $\Delta(e) = \lceil q(e) \rceil - q(e)$ . Die Kosten des Netzes, das sich aus dem Routing des Algorithmus ergibt, sind  $C = F \cdot |E'| + \sum_{e=\{i,j\} \in E'} \lceil q(e) \rceil \cdot p_{i,j}$ . Diese Kosten können wir in zwei Teile  $A$  und  $B$  aufspalten (d.h.  $C = A + B$ ):

$$A = \sum_{e=\{i,j\} \in E'} q(e) p_{i,j}$$

$$B = F \cdot |E'| + \sum_{e=\{i,j\} \in E'} \Delta(e) p_{i,j}$$

Wir können  $A$  umformulieren zu

$$A = \sum_{1 \leq i < j \leq n} r_{i,j} \cdot d_H(i, j),$$

wobei  $d_H(i, j)$  die Länge eines kürzesten Pfades von  $i$  nach  $j$  in  $H$  bezüglich der Kantengewichte  $p_{i,j}$  ist. Da  $H$  ein  $\log n$ -Spanner von  $G$  ist, gilt  $d_H(i, j) \leq (\log n) \cdot d_G(i, j) = (\log n) \cdot p_{i,j}$ . (Da die Kantengewichte  $p_{i,j}$  die Dreiecksungleichung erfüllen, gilt  $d_G(i, j) = p_{i,j}$  für alle  $i, j$ .) Wir erhalten somit  $A \leq (\log n) \cdot \sum_{1 \leq i < j \leq n} r_{i,j} \cdot p_{i,j}$ . Mit Ungleichung (b) aus Lemma 3.13 erhalten wir daher  $A \leq \log n \cdot OPT$ .

Versuchen wir nun,  $B$  abzuschätzen. Da  $\Delta(e) < 1$  gilt, folgt  $B \leq F \cdot |E'| + \sum_{e=\{i,j\} \in E'} p_{i,j}$ . Nach Korollar 3.12 gilt  $|E'| \leq |\hat{E}| = O(n)$  und damit  $F \cdot |E'| \leq c \cdot F \cdot (n-1)$  für eine Konstante  $c$ . Ausserdem gilt nach Korollar 3.12 auch  $\sum_{e=\{i,j\} \in E'} p_{i,j} \leq \sum_{e=\{i,j\} \in \hat{E}} p_{i,j} = O(\log n) \cdot MST$ . Mit Ungleichung (a) aus Lemma 3.13 folgt daher  $B \leq O(\log n) \cdot OPT$ .

Somit erhalten wir  $C = A + B \leq O(\log n) \cdot OPT$ . □

### 3.7 Literaturhinweise

Der Algorithmus von Dijkstra zur Berechnung kürzester Wege in Graphen mit positiven Kantengewichten ist in allen Lehrbüchern zum Thema Algorithmen beschrieben, z.B. in [Tur96]. Der Algorithmus für leichte angenäherte Kürzeste-Wege-Bäume aus Abschnitt 3.2 stammt von Khuller, Raghavachari und Young [KRY95]. Der Algorithmus zur Berechnung eines  $t$ -Spanners aus Abschnitt 3.3 geht auf Althöfer, Das, Dobkin, Joseph und Soares zurück [ADD<sup>+</sup>93], seine Analyse stammt von Chandra, Das, Narasimhan und Soares [CDNS95]. Das graphentheoretische Lemma 3.5 lässt sich leicht aus Theorem 3.7 in [Bol78, Chapter III] ableiten. Die Modellierung des Netzwerkdesign-Problems ND1K aus Abschnitt 3.4 und der Approximationsalgorithmus für die allgemeine Variante aus Abschnitt 3.6.1 wurden von Mansour und Peleg entwickelt [MP94]. Der Approximationsalgorithmus für die Variante mit einer Quelle aus Abschnitt 3.5.1 wurde in [SCRS97] erwähnt. Die Darstellung hier orientiert sich teilweise an [CR98].

#### Referenzen

- [ADD<sup>+</sup>93] Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete and Computational Geometry*, 9:81–100, 1993.
- [Bol78] B. Bollobàs. *Extremal Graph Theory*. Academic Press, London, 1978.
- [CDNS95] Barun Chandra, Gautam Das, Giri Narasimhan, and José Soares. New sparseness results on graph spanners. *International Journal of Computational Geometry and Applications*, 5(1–2):125–144, 1995.
- [CR98] J. Cheriyan and R. Ravi. Lecture notes on approximation algorithms for network problems, 1998. Available at <http://www.math.uwaterloo.ca/~jcheriya/lecnotes.html>.
- [KRY95] S. Khuller, B. Raghavachari, and N. E. Young. Balancing minimum spanning and shortest path trees. *Algorithmica*, 14:305–322, 1995.
- [MP94] Yishay Mansour and David Peleg. An approximation algorithm for minimum-cost network design. Technical Report CS94-22, The Weizman Institute of Science, 1994.
- [SCRS97] F. S. Salman, J. Cheriyan, R. Ravi, and S. Subramanian. Buy-at-bulk network design: Approximating the single-sink edge installation problem. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms SODA'97*, pages 619–628, 1997.
- [Tur96] Volker Turau. *Algorithmische Graphentheorie*. Addison-Wesley, 1996.



## Kapitel 4

# Das Problem des IP Prefix Lookup

Im Internet werden alle Daten in Form von IP-Paketen (IP steht für *Internet Protocol*) vom Sender zum Empfänger übertragen. Dabei kann ein Paket über viele Knoten weitergeleitet werden, bevor es an seinem Ziel ankommt. Auf jedem Knoten (Router) wird anhand der Zieladresse des Pakets entschieden, zu welchem Nachbarknoten das Paket weitergeleitet wird (*destination-based routing*). Die dazu nötigen Informationen sind auf jedem Knoten in einer Routing-Tabelle bzw. Forwarding-Tabelle gespeichert. Da in einem Router sehr viele IP-Pakete innerhalb kürzester Zeit (evtl. mehrere Millionen Pakete pro Sekunde) bearbeitet werden müssen, ist es wichtig, dass für jedes einzelne Paket sehr effizient der entsprechende Eintrag in der Forwarding-Tabelle gefunden werden kann. In diesem Kapitel wollen wir uns daher mit Datenstrukturen und Algorithmen befassen, mit denen die Suche in der Forwarding-Tabelle eines IP-Routers sehr effizient realisiert werden kann.

Jeder Rechner im Internet hat gegenwärtig eine Adresse, die ein Binärstring der Länge 32 ist. Mit der Weiterentwicklung von IPv4 zu IPv6 erhöht sich diese Länge dann auf 128. Wir wollen allgemein annehmen, dass die Adressen Binärstrings der Länge  $W$  sind. Die Routing-Tabelle  $R$  eines Routers enthält Einträge der Form  $(x, y)$ , wobei  $x$  ein Binärstring der Länge höchstens  $W$  ist (den wir *Präfix* nennen) und  $y$  einen ausgehenden Link des Routers bezeichnet. Dabei ist erlaubt, dass  $x$  der leere String ist. (Der leere Binärstring wird mit  $\epsilon$  bezeichnet.)

Die Anzahl der Einträge in der Routing-Tabelle bezeichnen wir mit  $N$ . Zum Beispiel hatten Mitte 1998 grosse Internet-Router Routing-Tabellen mit über 40000 Einträgen.

Kommt nun ein Paket mit Zieladresse  $k$  bei dem Router an, so wird ein Eintrag  $(x, y)$  in  $R$  gesucht, bei dem  $x$  ein Präfix von  $k$  ist und bei dem  $x$  maximale Länge hat. Diesen Eintrag nennen wir das *längste passende Präfix* (engl. *best matching prefix*, kurz BMP). Das Paket wird dann über denjenigen ausgehenden Link weitergeleitet, der durch das  $y$  im BMP  $(x, y)$  bestimmt ist. Mit *IP Prefix Lookup* bezeichnen wir das Problem, in einer Routing-Tabelle für eine gegebene Zieladresse das längste passende Präfix zu finden. Wir nehmen an, dass eine Routing-Tabelle für jeden Binärstring  $x$  höchstens einen Eintrag  $(x, y)$  enthalten kann.

**Beispiel:** Sei  $W = 5$ . Betrachte eine Routing-Tabelle mit den folgenden Einträgen:

$$(0, y_1), (001, y_2), (10, y_1), (110, y_3), (111, y_2)$$

Für ein Paket mit Zieladresse  $k = 00100$  ist  $(001, y_2)$  das längste passende Präfix und das Paket wird über den Link  $y_2$  weitergeleitet. Für  $k = 01000$  ist  $(0, y_1)$  das längste passende Präfix, für  $k = 11010$  ist es  $(110, y_3)$ .  $\blacklozenge$

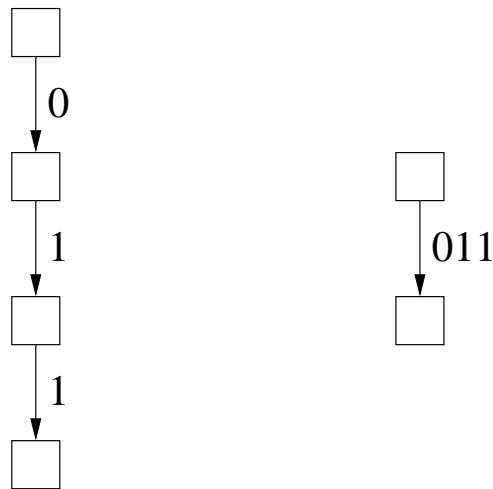


Abbildung 4.1: Pfadkomprimierung.

Eine Datenstruktur zur Speicherung der Routing-Tabelle sollte die folgenden Eigenschaften haben:

- Für jede Zieladresse  $k$  sollte man sehr schnell das längste passende Präfix (BMP) bestimmen können (Lookup). In der Praxis wäre es ideal, wenn nicht mehr als drei oder vier Speicherzugriffe nötig wären, um das BMP zu finden. Wir messen im Folgenden die Zeit für das Auffinden des BMP in Abhängigkeit von  $W$  und streben an, unabhängig von der Anzahl  $N$  der Einträge in der Routing-Tabelle Laufzeiten wie  $O(\log W)$  zu erreichen.
- Der Speicherplatz für die Datenstruktur sollte möglichst klein sein, damit die Datenstruktur oder zumindest ein grosser Teil davon im Cache gehalten werden kann. Wenn die Routing-Tabelle  $N$  Einträge enthält, sollte der Speicherplatz am besten  $O(N)$  sein, also linear.
- Eventuell sollte die Datenstruktur auch Aktualisierungen der Routing-Tabelle (Einfügen, Löschen und Ändern von Einträgen) effizient unterstützen.

Datenstrukturen, die das Einfügen und Löschen von Einträgen unterstützen, nennt man *dynamisch*.

## 4.1 IP Prefix Lookup mit einem Trie

Eine einfache Möglichkeit, die Routing-Tabelle zu speichern, ist die Verwendung eines *Trie*. Ein Trie ist ein Suchbaum, bei dem die Kanten beschriftet sind und für jeden Knoten gilt, dass die Beschriftungen der Kanten, die den Knoten mit seinen Kindern verbinden, mit verschiedenen Zeichen beginnen. Jeder Knoten repräsentiert den String, der sich durch Konkatenation der Kantenbeschriftungen auf dem Pfad von der Wurzel zu dem Knoten ergibt.

Oft werden Tries betrachtet, bei denen die Beschriftung jeder Kante aus einem einzigen Zeichen besteht. Wir lassen dagegen zu, dass Kantenbeschriftungen Strings beliebiger Länge (aber mindestens der Länge 1) sind. Wenn ein Trie eine Kette von Knoten enthält, die jeweils

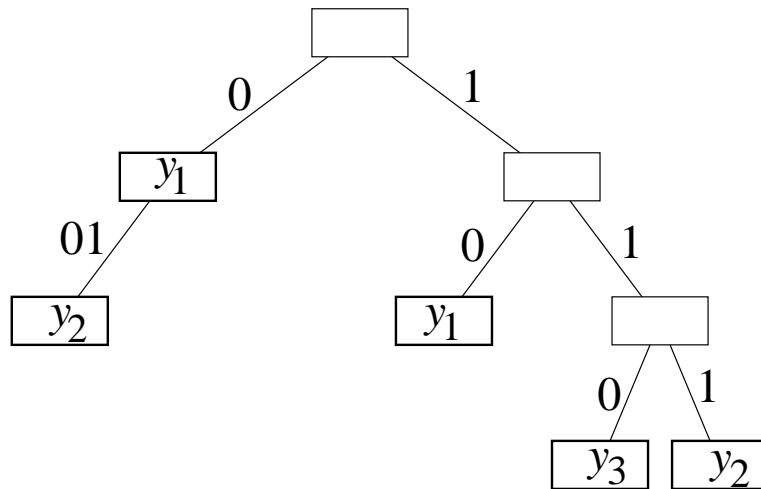


Abbildung 4.2: Speicherung einer Routing-Tabelle als Trie.

nur ein einziges Kind haben, so lässt sich diese Kette dann durch einen einzigen Knoten und eine einzige Kante ersetzen, deren Beschriftung die Konkatenation der Kantenbeschriftungen der ursprünglichen Kanten ist. Diese Operation nennt man *Pfadkomprimierung* (siehe Abb. 4.1). Ein Trie, bei dem alle möglichen Pfadkomprimierungen durchgeführt sind, heisst auch *pfadkomprimierter Trie*. Der Vorteil pfadkomprimierter Tries ist die geringere Anzahl von Trie-Knoten und damit der geringere Speicherplatzbedarf.

Wird ein Trie zur Speicherung einer Routing-Tabelle verwendet, so sind alle Kantenbeschriftungen Binärstrings. Jeder Knoten hat dann höchstens zwei Kinder: die Beschriftung der Kante zum ersten Kind beginnt mit 0, die zum zweiten Kind mit 1. Für jeden Eintrag  $(x, y)$  der Routing-Tabelle muss es im Trie einen Knoten geben, der den String  $x$  repräsentiert. In diesem Knoten des Trie ist dann der Link  $y$  abgespeichert. Für jeden Binärstring  $x$  der Länge höchstens  $W$  enthält der Trie genau dann einen Knoten, der  $x$  repräsentiert, falls mindestens eine der folgenden Bedingungen gilt:

- $x$  ist der leere String (dann entspricht  $x$  die Wurzel des Trie)
- Die Routing-Tabelle enthält einen Eintrag  $(x, y)$ .
- Die Routing-Tabelle enthält zwei Einträge  $(x_0, y_0)$  und  $(x_1, y_1)$ , so dass  $x_0$  ein Präfix von  $x_1$  ist und  $x_1$  ein Präfix von  $x_1$ .

Dadurch ist bereits eindeutig festgelegt, wie der Trie zu einer gegebenen Routing-Tabelle aussieht. Für das Beispiel der Routing-Tabelle mit den Einträgen

$$(0, y_1), (001, y_2), (10, y_1), (110, y_3), (111, y_2)$$

ist der sich ergebende Trie in Abb. 4.2 dargestellt. Die Trie-Knoten, die einem Eintrag in der Routing-Tabelle entsprechen, sind durch dickere Umrandungen gekennzeichnet. Wir bezeichnen diese Knoten auch als *markiert*.

Nun wollen wir uns überlegen, wie der Trie aufgebaut werden kann, wie viel Speicherplatz er benötigt und wie lange das Suchen und Aktualisieren des Trie dauert. Dabei nehmen wir an, dass ein Binärstring der Länge  $W$  in konstant vielen Maschinenwörtern gespeichert werden kann.

### 4.1.1 Konstruktion des Trie

Um den Trie für eine Routing-Tabelle zu konstruieren, beginnt man mit dem leeren Trie, der nur aus der Wurzel besteht. Dann fügt man die  $N$  Einträge der Routing-Tabelle in beliebiger Reihenfolge in den Trie ein. Für einen Eintrag  $(x, y)$  beginnen wir an der Wurzel des Trie und folgen dem Pfad abwärts, der durch  $x$  festgelegt wird. Das heisst, wir gehen von jedem Trie-Knoten mit derjenigen Kante weiter abwärts, deren Beschriftung mit demselben Zeichen beginnt wie der verbleibende Teil von  $x$ . Nun gibt es mehrere Möglichkeiten:

- Der Pfad endet bei einem Trie-Knoten  $v$ , wenn  $x$  ganz abgearbeitet ist. In diesem Fall markieren wir  $v$  als einen Knoten, der einem Eintrag der Routing-Tabelle entspricht, und speichern den Link  $y$  in dem Knoten  $v$  ab.
- Der Pfad endet auf einer Kante  $(v, w)$  mit Beschriftung  $\alpha\beta$  zwischen  $\alpha$  und  $\beta$ , wenn  $x$  ganz abgearbeitet ist. In diesem Fall erzeugen wir einen neuen Knoten  $u$  in der Mitte der Kante  $(v, w)$  und weisen den neuen Kanten  $(v, u)$  und  $(u, w)$  die Beschriftungen  $\alpha$  und  $\beta$  zu. Der Link  $y$  wird in dem neuen Knoten  $u$  abgespeichert und  $u$  wird markiert als Knoten, der einem Eintrag in der Routing-Tabelle entspricht.
- Der Pfad endet bei einem Trie-Knoten  $v$ , bevor  $x$  ganz abgearbeitet ist. Sei  $\gamma$  der verbleibende Teil von  $x$ . Wir fügen ein neues Kind  $u$  von  $v$  in den Trie ein und beschriften die Kante  $(v, u)$  mit  $\gamma$ . Den Link  $y$  speichern wir in dem neuen Knoten  $u$ .
- Der Pfad endet auf einer Kante  $(v, w)$  mit Beschriftung  $\alpha\beta$  zwischen  $\alpha$  und  $\beta$ , bevor  $x$  ganz abgearbeitet ist. Sei  $\gamma$  der verbleibende Teil von  $x$ . Wir spalten die Kante  $(v, w)$  zwischen  $\alpha$  und  $\beta$  durch Einfügen eines neuen Knotens  $u$  auf, erzeugen ein neues Kind  $u'$  von  $u$  und beschriften die Kante  $(u, u')$  mit  $\gamma$ .

Da auf diese Weise jeder Eintrag der Routing-Tabelle in Zeit  $O(W)$  in den Trie eingefügt werden kann, ist die Gesamtlaufzeit für die Konstruktion des Trie  $O(N \cdot W)$ .

### 4.1.2 Speicherplatzbedarf des Trie

Jeder Knoten des Trie benötigt konstanten Speicherplatz, ebenso jede Kante (ein Binärstring der Länge höchstens  $W$  kann laut Annahme in konstant vielen Wörtern gespeichert werden). Genau  $N$  der Knoten im Trie repräsentieren einen Eintrag der Routing-Tabelle. Alle anderen Knoten sind innere Knoten des Trie, die genau zwei Kinder haben (andernfalls wären diesen Knoten gemäss der Definition des Tries nicht im Trie enthalten). In einem Baum mit höchstens  $N$  Blättern kann es aber höchstens  $N - 1$  innere Knoten mit zwei Kindern geben. Daher enthält der Trie insgesamt höchstens  $2N - 1$  Knoten und  $2N - 2$  Kanten. Der Speicherplatzbedarf ist daher  $O(N)$ , also linear.

### 4.1.3 Suchen im Trie

Beim Suchen nach dem längsten passenden Präfix zu einer Zieladresse  $k$  geht man wie folgt vor. Man beginnt bei der Wurzel des Trie und folgt dem Pfad, der durch  $k$  bestimmt wird. Dabei merkt man sich immer den letzten markierten Knoten, den man durchlaufen hat. Wenn nun  $k$  komplett abgearbeitet ist oder wenn der Pfad im Trie nicht um das nächste Bit von  $k$  verlängert werden kann, so ist der letzte markierte Knoten, den man durchlaufen hat, genau der Knoten, der das längste passende Präfix repräsentiert. Die Suche nach dem längsten passenden Präfix benötigt also  $O(W)$  Zeit.

#### 4.1.4 Aktualisieren des Trie

Wenn ein neuer Eintrag  $(x, y)$  in den Trie eingefügt werden soll, so geschieht dies wie bei der Konstruktion des Trie in Zeit  $O(W)$ . Soll ein Eintrag  $(x, y)$  gelöscht werden, so sucht man in Zeit  $O(W)$  den entsprechenden Trie-Knoten  $v$  und geht folgendermassen vor:

- Falls  $v$  zwei Kinder hat, so wird nur die Markierung von  $v$  aufgehoben, der Knoten bleibt aber im Trie enthalten.
- Falls  $v$  genau ein Kind hat und nicht die Wurzel ist, so wird der Knoten  $v$  gelöscht und die beiden Kanten, die  $v$  mit seinem Vater und seinem Kind verbunden haben, werden zu einer einzigen Kante kombiniert.
- Falls  $v$  ein Blatt ist, so wird der Knoten  $v$  gelöscht. Falls der Vater  $w$  von  $v$  nicht die Wurzel des Trie ist und nicht markiert ist, so wird mit  $w$  wie im vorigen Fall mit  $v$  verfahren (d.h.  $w$  wird entfernt und die beiden zu  $w$  inzidenten Kanten werden zu einer Kante kombiniert).

Offensichtlich ist also auch der Zeitbedarf für das Löschen eines Eintrags  $O(W)$ .

Um für einen Eintrag  $(x, y)$  nur den Link zu ändern, reicht es, in Zeit  $O(W)$  den entsprechenden Trie-Knoten zu lokalisieren und dann dort den gespeicherten Link zu ersetzen.

Die Ergebnisse unserer Analyse der Trie-Datenstruktur werden in dem folgenden Satz zusammengefasst.

**Satz 4.1** *Mittels eines Trie lässt sich eine Routing-Tabelle in Speicherplatz  $O(N)$  so speichern, dass jede Suche und jede Aktualisierung in Zeit  $O(W)$  durchgeführt werden kann. Der Zeitbedarf für die Konstruktion des Trie ist  $O(N \cdot W)$ .*

Vorteilhaft ist bei Tries also bereits, dass der Speicherplatzbedarf linear in  $N$  ist, dass die Suchzeit nur von  $W$  abhängt und dass alle Aktualisierungen effizient durchgeführt werden können. Für die Suche ist eine Laufzeit von  $O(W)$  (mit etwa  $W$  tatsächlichen Hauptspeicherezugriffen) allerdings noch nicht ausreichend, um in einem IP-Router den IP Prefix Lookup effizient in Software realisieren zu können. Daher werden wir in den folgenden Abschnitten Verbesserungen dieser einfachen Trie-Datenstruktur betrachten.

## 4.2 IP Prefix Lookup durch Binärsuche nach Präfixlängen

Nun wollen wir ein Verfahren für IP Prefix Lookup betrachten, mit dem das längste passende Präfix in  $O(\log W)$  Zeit (mit etwa  $\log W$  Speicherezugriffen) gefunden werden kann. Das Verfahren wurde von Waldvogel, Varghese, Turner und Plattner entwickelt [WVTP97]. Der Ausgangspunkt für die schnellere Suche ist die Überlegung, dass man das Problem des IP Prefix Lookup in konstanter Zeit lösen kann, wenn alle Einträge der Routing-Tabelle dieselbe Länge haben (d.h. wenn der String  $x$  in allen Einträgen  $(x, y)$  der Routing-Tabelle gleich lang ist). Um dies auszunutzen, teilt man Einträge der Routing-Tabelle anhand ihrer Länge in höchstens  $W$  Klassen ein. Die Einträge jeder Klasse speichert man so ab, dass man in konstanter Zeit überprüfen kann, ob ein gegebener Binärstring der entsprechenden Länge in der Klasse vorkommt. Wenn man nun mittels Binärsuche in den Längensklassen diejenige finden könnte, die das längste passende Präfix enthält, so wäre das Problem des IP Prefix Lookup mit einer Suchzeit von  $O(\log W)$  gelöst. Im Folgenden werden wir sehen, wie diese Ideen realisiert werden können.

### 4.2.1 Suche innerhalb einer Längenklasse

Eine Längenklasse  $L_i$  speichert alle Einträge  $(x, y)$  der Routing-Tabelle, bei denen das Präfix  $x$  die Länge  $i$  hat. Sei  $N_i$  die Anzahl solcher Einträge. Es gibt Datenstrukturen, die zur Speicherung von  $L_i$  Speicherplatz  $O(N_i)$  benötigen und in Zeit  $O(1)$  für einen gegebenen Binärstring  $k$  der Länge  $i$  entscheiden können, ob ein Eintrag  $(k, y)$  in  $L_i$  enthalten ist.

Eine Möglichkeit ist, eine Hash-Tabelle zur Speicherung von  $L_i$  zu verwenden. Dadurch wird zwar das Auffinden eines Präfix nur in *durchschnittlich* konstanter Zeit erreicht (im Worst-Case kann die Suche länger dauern), man nimmt das in der Praxis aber gerne in Kauf, da sich Hash-Tabellen leicht und effizient implementieren lassen. Die Anzahl von zur Verfügung stehenden Indizes in der Hash-Tabelle für  $L_i$  muss mindestens gleich  $c \cdot N_i$  für eine Konstante  $c > 1$  gewählt werden. Die Hash-Tabelle basiert auf einer Hash-Funktion  $f$ , die jedem Präfix  $x$  eines Eintrags in  $L_i$  einen Index  $f(x)$  in der Tabelle zuweist. Wir nehmen an, dass die Hash-Funktion in konstanter Zeit berechnet werden kann. Werden mehrere Präfixe auf denselben Index abgebildet, so nennt man dies eine Kollision. Eine Möglichkeit, solche Kollisionen aufzulösen, ist Verkettung, d.h. man speichert bei jedem Index  $j$  der Hash-Tabelle eine verkettete Liste aller Präfixe, die durch die Hash-Funktion auf  $j$  abgebildet werden. Wenn nun in der Hash-Tabelle nach einem Präfix  $k$  gesucht werden soll, so wird  $j = f(k)$  berechnet und an Stelle  $j$  in der Tabelle nachgesehen, um die dort gespeicherte Liste von Einträgen den Präfix  $k$  enthält. Bei Verwendung einer geeigneten Hash-Funktion ist die erwartete Länge der Liste  $O(1)$  und die Suche nach  $k$  benötigt daher im Durchschnitt nur  $O(1)$  Zeit.

Es gibt auch Datenstrukturen, mit denen man  $L_i$  in Speicherplatz  $O(N_i)$  speichern kann und in denen die Suche nach einem Präfix  $k$  auch im Worst-Case nur Zeit  $O(1)$  benötigt. Die Grundidee dieser Datenstrukturen besteht darin, abhängig von den zu speichernden Werten die Hash-Funktion so zu wählen, dass keine Kollisionen auftreten [FKS84].

Wir wollen im folgenden annehmen, dass wir für jedes  $L_i$  eine Tabelle (Hash-Tabelle oder sonstige Datenstruktur) haben, die Speicherplatz  $O(N_i)$  benötigt und in der wir in  $O(1)$  Zeit nach einem Präfix suchen können.

### 4.2.2 Binärsuche

Wenn wir für jede Längenklasse  $L_i$  eine Tabelle haben, die das Auffinden eines Präfix  $k$  in Zeit  $O(1)$  ermöglicht, wie suchen wir dann nach dem längsten passenden Präfix für eine gegebene Zieladresse  $z$ ? Eine erste Idee ist, zuerst in der "mittleren" Längenklasse nach dem betreffenden Präfix von  $z$  zu suchen und dann abhängig vom Ergebnis zu entscheiden, ob wir rekursiv bei den längeren Präfixen (falls die Suche erfolgreich war) oder bei den kürzeren Präfixen (andernfalls) weitersuchen. Da es insgesamt höchstens  $W$  Längenklassen gibt, könnte die Binärsuche in Zeit  $O(\log W)$  ausgeführt werden, da jede Suche in einer Längenklasse nur Zeit  $O(1)$  dauert. Das folgende Beispiel zeigt aber, dass das so noch nicht richtig funktioniert:

Eine Routing-Tabelle habe Einträge für die Präfixe 0, 1, 00 und 111. Damit haben wir die Längenklassen  $L_1 = \{0, 1\}$ ,  $L_2 = \{00\}$  und  $L_3 = \{111\}$ . Wenn wir nun nach Zieladresse 11101 suchen und zuerst in der mittleren Längenklasse  $L_2$  nach dem Präfix 11 von 11101 suchen, so ist die Suche erfolglos und wir würden bei den kürzeren Präfixen weitersuchen, obwohl das längste passende Präfix 111 in  $L_3$  enthalten ist.

Um dieses Problem zu umgehen, fügen wir sogenannte *Marker* in die Längenklassen ein. Sei  $x$  ein Präfix der Länge  $i$  und sei  $x[1..j]$  das Präfix der ersten  $j$  Zeichen von  $x$ ,  $1 \leq j \leq i$ .

Wenn  $x$  in der Längenkategorie  $L_i$  enthalten ist, so speichern wir in allen Längenkategorien  $L_j$  für  $j < i$ , die keinen Eintrag für  $x[1..j]$  enthalten, einen Marker für  $x[1..j]$ . Die Aufgabe des Markers ist, dafür zu sorgen, dass bei der Suche nach  $x[1..j]$  in  $L_j$  erkannt wird, dass es einen Eintrag mit längerem Präfix in der Routing-Tabelle gibt, der mit  $x[1..j]$  beginnt. Damit kann die Binärsuche dann die richtige Entscheidung treffen und bei den längeren Präfixen weitersuchen.

In obigem Beispiel würden wir in  $L_2$  den Marker 11 einfügen, da in  $L_3$  das Präfix 111 enthalten ist. Bei einer Suche nach der Zieladresse  $z = 11101$  würde dann nach der erfolgreichen Suche in  $L_2$  korrekterweise in  $L_3$  weitergesucht.

Es gibt aber noch immer ein Problem, das wir wieder an unserem Beispiel deutlich machen können:

Wenn im Beispiel nach der Zieladresse  $z = 11010$  gesucht wird, so würde der Marker 11 in  $L_2$  dazu führen, dass die Binärsuche als nächstes in  $L_3$  nach 110 sucht, aber der längste passende Präfix 1 ist in  $L_1$  enthalten. Wenn nun in  $L_3$  festgestellt wird, dass dort kein Präfix 110 enthalten ist, wäre das Ergebnis der Binärsuche, dass 11 das längste passende Präfix ist. Da 11 aber nur ein Marker ist, der keinem tatsächlichen Eintrag der Routing-Tabelle entspricht, müsste die Binärsuche zurückgehen und stattdessen in  $L_1$  weitersuchen, um dort das echte längste passende Präfix 1 zu finden.

Durch solches Zurückgehen (*backtracking*) wäre aber nicht mehr die gewünschte Laufzeit  $O(\log W)$  für die Suche zu erreichen. Um auch dieses Problem zu umgehen, speichern wir bei jedem Marker  $x$  einen Verweis  $x.BMP$  ab, der auf denjenigen Eintrag  $(x', y')$  der Routing-Tabelle verweist, bei dem  $x'$  ein Präfix von  $x$  ist und bei dem  $x'$  maximale Länge hat. Wenn die Binärsuche nun einen Marker  $x$  als längstes passendes Präfix findet, so liefert der Verweis  $x.BMP$  das gesuchte echte längste passende Präfix.

Im obigen Beispiel würde 11.BMP auf den Eintrag für das Präfix 1 verweisen. Wenn bei Zieladresse 11010 der Marker 11 das Ergebnis der Binärsuche ist, so liefert 11.BMP also das gewünschte längste passende Präfix 1.

Mit diesen beiden Ergänzungen (Einführung der Marker und Speichern des Verweises  $x.BMP$  bei jedem Marker  $x$ ) funktioniert die Binärsuche dann offensichtlich richtig und die Worst-Case-Laufzeit für die Suche nach dem längsten passenden Präfix zu einer Zieladresse ist  $O(\log W)$ . Falls weniger als  $W$  verschiedene Präfixlängen in der Routing-Tabelle vorkommen, etwa nur  $\ell$  verschiedene Längen, dann ist die Laufzeit sogar  $O(\log \ell)$ , da die Binärsuche nur die  $\ell$  nicht-leeren Längenkategorien  $L_i$  berücksichtigen muss.

### 4.2.3 Speicherplatzbedarf

Ohne die Einführung der Marker wäre der Speicherplatzbedarf  $O(N)$ , da die Tabellen für die  $L_i$  jeweils nur Platz  $O(N_i)$  benötigen. Die Marker bedeuten aber zusätzliche Einträge in den Tabellen. Werden die Marker wie oben beschrieben eingefügt (d.h. für ein Präfix in  $L_i$  werden in allen  $L_j$  mit  $j < i$  Marker eingefügt, falls noch kein entsprechendes Präfix in  $L_j$  enthalten ist), so können bis zu  $O(W)$  Marker pro Präfix nötig sein und der Speicherplatzbedarf für die gesamte Datenstruktur kann nur durch  $O(N \cdot W)$  begrenzt werden. Wenn Marker aber nur in den Längenkategorien gespeichert werden, wo sie tatsächlich nötig sind (Übungsaufgabe!), lässt sich der Speicherplatzbedarf zu  $O(N \cdot \log W)$  vermindern.

#### 4.2.4 Konstruktion

Die Zeit für die Konstruktion der Datenstruktur ist linear in ihrer Grösse (unter der Annahme, dass die Tabellen für die  $L_i$  in konstanter Zeit pro Eintrag aufgebaut werden können, was für Hash-Tabellen erfüllt ist). Man bearbeitet die Einträge der Routing-Tabelle in Reihenfolge aufsteigender Präfixlänge. Ein Präfix  $x$  der Länge  $i$  wird dann in die Tabelle für  $L_i$  eingefügt und die entsprechenden Marker werden in die Tabellen  $L_j$  mit  $j < i$  in absteigender Reihenfolge eingefügt, bis in einer Tabelle das betreffende Präfix bereits enthalten ist.

#### 4.2.5 Aktualisieren der Datenstruktur

Beim Einfügen eines Eintrags  $(x, y)$  muss der Eintrag zuerst in die betreffende Tabelle  $L_i$  (wobei  $i$  die Länge von  $x$  ist) eingefügt werden. In den Tabellen  $L_j$  für  $j < i$  muss ggf. ein Marker eingefügt werden. Ferner muss bei allen Markern  $x'$  in Tabellen  $L_j$  für  $j > i$ , für die  $x$  das längste passende Präfix von  $x'$  ist, der Verweis  $x'.BMP$  auf  $(x, y)$  aktualisiert werden. Diese Aktualisierung kann potentiell sehr teuer sein, da eventuell sehr viele Marker aktualisiert werden müssen. Ähnlich stellt sich die Situation beim Löschen eines Eintrags dar. Man kann für diese Datenstruktur also keine guten Zeitschranken für das Einfügen und Löschen von Einträgen angeben. Es wird daher empfohlen, Änderungen der Einträge der Routing-Tabelle eine Weile zu sammeln und dann die Datenstruktur komplett neu aufzubauen. Diese Lösung ist effizienter als die potentiell sehr teure Realisierung jeder einzelnen Aktualisierung für sich. Nachteil bei dieser Lösung ist aber, dass Aktualisierungen der Routing-Tabelle erst nach einer gewissen Wartezeit aktiv werden.

#### 4.2.6 Experimentelle Ergebnisse

Das Verfahren der Binärsuche nach Präfixlängen wurde experimentell anhand einer realistischen Routing-Tabelle mit 33000 Einträgen für Zieladressen mit  $W = 32$  Bits überprüft. Dabei war der Speicherplatzbedarf für die Datenstruktur 1.4 MB und eine Suche nach dem längsten passenden Präfix benötigte im Durchschnitt nur 2 Suchen in Tabellen  $L_i$  (da in den meisten Fällen ein Präfix mit Länge 16 oder 24 der längste passende Präfix war und diese beiden Längenklassen von der Binärsuche zuerst durchsucht wurden). Auf einem 200 MHz Pentium Pro war die durchschnittliche Zeit für eine Suche nach dem längsten passenden Präfix 180 ns, die längste gemessene Zeit war 850 ns. Diese Werte zeigen, dass das Problem des IP Prefix Lookup bei geeigneter Wahl der Datenstrukturen auch in Software sehr schnell gelöst werden kann.

### 4.3 Präfix-Expansion für schnelleren IP Prefix Lookup

In Abschnitt 4.2.2 haben wir gesehen, dass die Worst-Case-Laufzeit für IP Prefix Lookup mittels Binärsuche nach Präfixlängen  $O(\log \ell)$  beträgt, wobei  $\ell$  die Anzahl nicht-leerer Längenklassen  $L_i$  ist. In der Praxis sind in Routing-Tabellen für IPv4 meistens die Längenklassen  $L_1$  bis  $L_7$  leer, während jede der Längenklassen  $L_8$  bis  $L_{32}$  tatsächlich Routing-Einträge enthält. Es gilt dann also  $\ell = 25$ . Die Längenklassen  $L_{16}$  und  $L_{24}$  enthalten oft weitaus die meisten Einträge (aus historischen Gründen: früher wurden nämlich nur Routing-Einträge mit Präfixlänge 8, 16 oder 24 zugelassen). Es liegt also nahe zu betrachten, ob durch Modifikation einer Routing-Tabelle die Anzahl nicht-leerer Längenklassen reduziert werden kann. Das im Folgenden vorgestellte Verfahren und seine Anwendungen stammen von Srinivasan und Varghese [SV99].

#### 4.3.1 Expansion von Präfixen

Wir nennen zwei Routing-Tabellen  $R_1$  und  $R_2$  *äquivalent*, wenn für jede Zieladresse  $k$  gilt, dass Pakete mit Zieladresse  $k$  von  $R_1$  und  $R_2$  auf denselben ausgehenden Link weitergeleitet werden. Nun wollen wir ein Verfahren kennen lernen, mit dem man die Anzahl verschiedener Präfixlängen in einer Routing-Tabelle kleiner machen kann, ohne das Verhalten der Routing-Tabelle zu verändern (d.h. die erhaltene Routing-Tabelle ist äquivalent zur ursprünglichen Tabelle). Die Grundidee dabei ist: Ein Präfix  $x$  der Länge  $k < W$  kann durch zwei Präfixe  $x0$  und  $x1$  der Länge  $k + 1$  ersetzt werden, ohne das Verhalten der Routing-Tabelle zu verändern. Jede Zieladresse, auf die das Präfix  $x$  passt, muss entweder mit  $x0$  oder  $x1$  beginnen, und somit werden durch die beiden Präfixe  $x0$  und  $x1$  dieselben Adressen abgedeckt wie durch das Präfix  $x$ . Das Ersetzen von  $x$  durch  $x0$  und  $x1$  nennt man *Expansion* des Präfixes  $x$ . Wir betrachten hier der Einfachheit halber nur die Präfixe und nicht die ganzen Routing-Einträge; die Expansion des Präfixes  $x$  bedeutet für die Routing-Einträge, dass der Eintrag  $(x, y)$  durch  $(x0, y)$  und  $(x1, y)$  ersetzt wird. Falls es jedoch bereits einen Eintrag  $(x0, y')$  bzw.  $(x1, y')$  in der Routing-Tabelle gibt, so darf man  $(x0, y)$  bzw.  $(x1, y)$  nicht noch einmal in die Tabelle einfügen: für Zieladressen mit Präfix  $x0$  bzw.  $x1$  ist in diesem Fall ja weiterhin der Eintrag  $(x0, y')$  bzw.  $(x1, y')$  zuständig. Durch  $j$ -fache Expansion eines Präfix  $x$  der Länge  $k$  erhält man damit höchstens  $2^j$  neue Präfixe der Länge  $k + j$ .

Wenn man nun eine Routing-Tabelle mittels Präfix-Expansion so modifizieren will, dass nur noch  $r$  verschiedene Präfixlängen vorkommen, so kann man zuerst die  $r$  Längen  $\ell_1 < \ell_2 < \dots < \ell_r$  derjenigen Längenklassen festlegen, die nicht-leer sein sollen.  $\ell_r$  muss dabei mindestens so gross sein wie die Länge der längsten Präfixe in der gegebenen Routing-Tabelle, da Präfixe durch Expansion nur länger gemacht werden können, aber nicht kürzer. Nun betrachtet man die Einträge der Routing-Tabelle in Reihenfolge aufsteigender Präfixlängen. Bei jeder nicht-leeren Längenkategorie  $L_i$  unterscheidet man die folgenden beiden Fälle:

- Falls  $i$  eine der Längen  $\ell_1, \dots, \ell_r$  ist, so geht man zur nächsten Längenkategorie weiter.
- Andernfalls expandiert man jeden Präfix  $x$  aus  $L_i$  zu zwei Präfixen  $x0$  und  $x1$ , die man jeweils in  $L_{i+1}$  einfügt, falls das Präfix  $x0$  bzw.  $x1$  noch nicht in  $L_{i+1}$  enthalten ist.

Auf diese Weise erhält man eine neue Routing-Tabelle, in der nur noch die Längenkategorien  $L_{\ell_1}, L_{\ell_2}, \dots, L_{\ell_r}$  nicht-leer sind und die zur ursprünglichen Routing-Tabelle äquivalent ist.

**Beispiel:** Wir beginnen mit einer Routing-Tabelle, bei der die nicht-leeren Längenklassen wie folgt aussehen:

$$\begin{aligned} L_1 &= \{(0, y_1), (1, y_2)\} \\ L_2 &= \{(10, y_3)\} \\ L_3 &= \{(111, y_4)\} \\ L_4 &= \{(1000, y_5)\} \\ L_5 &= \{(11001, y_6)\} \\ L_6 &= \{(100000, y_7)\} \\ L_7 &= \{(1000000, y_8)\} \end{aligned}$$

Wir haben also 7 nicht-leere Längenklassen und insgesamt 8 Einträge. Reduzieren wir nun mit dem oben angegebenen Verfahren die vorkommenden Längen auf  $\ell_1 = 2$ ,  $\ell_2 = 5$  und  $\ell_3 = 7$ . Man beachte, dass zum Beispiel bei der Expansion des Präfixes 1 aus  $L_1$  nur der Eintrag  $(11, y_2)$  in  $L_2$  eingefügt wird, da  $L_2$  bereits einen Eintrag mit Präfix 10 enthält. Wir erhalten:

$$\begin{aligned} L_2 &= \{(00, y_1), (01, y_1), (10, y_3), (11, y_2)\} \\ L_5 &= \{(11100, y_4), (11101, y_4), (11110, y_4), (11111, y_4), (10001, y_5), (10000, y_5), (11001, y_6)\} \\ L_7 &= \{(1000000, y_8), (1000001, y_7)\} \end{aligned}$$

Die neue Tabelle hat nur noch drei nicht-leere Längenklassen, aber dafür insgesamt 13 Einträge. Für die Binärsuche nach Präfixlängen benötigen wir mit der neuen Routing-Tabelle im Worst-Case nur noch 2 statt 3 Schritte (im ersten Schritt Suche in  $L_5$  und im zweiten Schritt, falls nötig, Suche entweder in  $L_2$  oder in  $L_7$ ).  $\blacklozenge$

Wir können durch Präfix-Expansion also eine neue, äquivalente Routing-Tabelle konstruieren, die weniger nicht-leere Längenklassen hat, die aber dafür in der Regel mehr Präfixe enthält. Durch die kleinere Anzahl nicht-leerer Längenklassen ergibt sich bei verschiedenen Verfahren zum IP Prefix Lookup eine kürzere Suchzeit, beispielsweise beim Verfahren der Binärsuche nach Präfixlängen (Kapitel 4.2). Im nächsten Abschnitt beschäftigen wir uns mit der Frage, wie man die Längenklassen  $\ell_1, \dots, \ell_r$  günstig wählen kann, wenn man eine gegebene Routing-Tabelle auf  $r$  nicht-leere Längenklassen reduzieren will, wobei  $r$  vorgegeben ist.

### 4.3.2 Dynamische Programmierung zur Wahl der Längenklassen

Da der Speicherplatzbedarf der meisten Datenstrukturen für IP Prefix Lookup von der Anzahl der Präfixe in der Routing-Tabelle abhängt, ist ein erstes sinnvolles Kriterium, nach dem man die  $r$  Längenklassen für die Präfix-Expansion wählen könnte, die Minimierung der Gesamtanzahl von Präfixen in der resultierenden Routing-Tabelle. Wir werden im Folgenden sehen, dass man in diesem Fall eine optimale Wahl der  $r$  Längenklassen leicht mittels *dynamischer Programmierung* (Zusammensetzen einer optimalen Gesamtlösung aus vorher berechneten optimalen Lösungen für Teilprobleme) berechnen kann.

Sei  $m$  die maximale Länge eines Präfix in der gegebenen Routing-Tabelle. Wir definieren für  $\ell \in \{0, 1, \dots, m\}$  und  $k \in \{1, 2, \dots, r\}$  den Wert  $M[\ell][k]$  als die minimale Anzahl von Präfixen, die man erhält, wenn man die ursprünglichen Längenklassen  $L_1, \dots, L_\ell$  (für  $\ell = 0$

**Berechnung der  $M[\ell][k]$  mittels dynamischer Programmierung**  
**Eingabe:** Werte  $E[k_1][k_2]$  für  $1 \leq k_1 \leq k_2 \leq m$ , Anzahl  $r$  von Längenklassen  
**Ausgabe:** Werte  $M[\ell][k]$  und  $P[\ell][k]$  für  $0 \leq \ell \leq m$ ,  $1 \leq k \leq r$

```

for  $\ell = 0$  to  $m$  do
  for  $k = 1$  to  $r$  do
    if  $\ell = 0$  then  $M[\ell][k] := 0$ ;  $P[\ell][k] := 0$ ;
    else if  $k = 1$  then  $M[\ell][k] := E[1][\ell]$ ;  $P[\ell][k] := 0$ ;
    else
       $M[\ell][k] := \min_{0 \leq \ell' < \ell} M[\ell'][k - 1] + E[\ell' + 1][\ell]$ ;
       $P[\ell][k] :=$  das  $\ell'$ , für das sich in der vorigen Zeile das Minimum ergab;
    fi;
  od;
od;

```

Abbildung 4.3: Dynamische Programmierung zur Minimierung der Präfix-Anzahl bei der Expansion in  $r$  nicht-leere Längenklassen.

ist das die leere Menge) mittels Präfix-Expansion so umwandelt, dass höchstens  $k$  nicht-leere Längenklassen verbleiben und  $L_\ell$  die nicht-leere Längenkategorie mit der grössten Länge ist. Der Wert  $M[m][r]$  gibt uns dann die Anzahl von Präfixen an, die wir erhalten, wenn wir durch optimale Präfix-Expansion eine Routing-Tabelle mit  $r$  nicht-leeren Längenklassen konstruieren.

Um die Werte  $M[\ell][k]$  berechnen zu können, benötigen wir noch Hilfsvariablen  $E[k_1][k_2]$  für  $1 \leq k_1 \leq k_2 \leq m$ , die angeben, wie viele Präfixe in der Längenkategorie  $L_{k_2}$  entstehen, wenn wir alle Präfixe aus den Längenkategorien  $L_{k_1}, L_{k_1+1}, \dots, L_{k_2-1}$  auf Länge  $k_2$  expandieren. In obigem Beispiel galt zum Beispiel  $E[3][5] = 7$ . Die Werte  $E[k_1][k_2]$  für alle Paare  $(k_1, k_2)$  können effizient berechnet und in einer Tabelle gespeichert werden (Übungsaufgabe).

Nun können wir die folgenden Gleichungen für  $M[\ell][k]$  angeben:

$$M[\ell][1] = E[1][\ell] \quad (\text{für alle } \ell \geq 1) \quad (4.1)$$

$$M[0][k] = 0 \quad (\text{für alle } k \geq 1) \quad (4.2)$$

$$M[\ell][k] = \min_{0 \leq \ell' < \ell} M[\ell'][k - 1] + E[\ell' + 1][\ell] \quad (\text{für } \ell > 0, k > 1) \quad (4.3)$$

Die Gleichung (4.1) gilt, da  $M[\ell][1]$  und  $E[1][\ell]$  identisch definiert sind. Die Gleichung (4.2) gilt, da aus einer leeren Menge von Längenkategorien auch durch Präfix-Expansion keine neuen Präfixe entstehen. In Gleichung (4.3) ist  $\ell'$  die nächstkleinere Länge (nach  $\ell$ ), für die wir eine nicht-leere Längenkategorie wählen. Offensichtlich sind für  $\ell'$  nur Werte von 0 bis  $\ell - 1$  möglich ( $\ell' = 0$  bedeutet, dass wir gar keine nicht-leeren Längenkategorien mehr wählen; dies ist erlaubt, da wir für  $M[\ell][k]$  ja nur fordern, dass es *höchstens*  $k$  nicht-leere Längenkategorien gibt). Für jedes  $\ell'$  setzt sich eine optimale Präfix-Expansion für die Längenkategorien  $1, 2, \dots, \ell$  in  $k$  nicht-leere Längenkategorien zusammen aus einer optimalen Präfix-Expansion für die Längenkategorien  $1, 2, \dots, \ell'$  in  $k - 1$  Längenkategorien (ergibt  $M[\ell'][k - 1]$  Präfixe) und einer Präfix-Expansion für die Längenkategorien  $\ell' + 1, \dots, \ell$  in eine Längenkategorie  $L_\ell$  (ergibt  $E[\ell' + 1][\ell]$  Präfixe). Da wir bei (4.3) das Minimum über alle möglichen Wahlen von  $\ell'$  bilden, erhalten wir tatsächlich den richtigen Wert für  $M[\ell][k]$ .

Mit Hilfe der obigen Formeln kann man nun leicht in Zeit  $O(rW^2)$  alle Werte  $M[\ell][k]$

berechnen. Ein Algorithmus hierzu ist in Abb. 4.3 angegeben. Hier wird zusätzlich zu  $M[\ell][k]$  noch ein Wert  $P[\ell][k]$  berechnet, der angibt, welches  $\ell'$  in der Formel (4.3) das Minimum ergeben hat. Am Ende der Berechnung gibt der Wert  $M[m][r]$  die optimale Präfix-Anzahl an und die  $r$  zugehörigen Präfix-Längen (in absteigender Reihenfolge) sind gerade:

$$m, P[m][r], P[P[m][r]][r-1], P[P[P[m][r]][r-1]][r-2], \dots$$

Wir können also für jedes gegebene  $r$  mittels dynamischer Programmierung effizient berechnen, wie wir die  $r$  nicht-leeren Längenklassen für die Präfix-Expansion wählen müssen, so dass am Ende die minimale Anzahl von Präfixen in der Routing-Tabelle mit  $r$  nicht-leeren Längenklassen enthalten ist.

Die Gesamtanzahl von Präfixen in der Routing-Tabelle ist nur eines von mehreren möglichen Kriterien, die bei der Wahl der Längenklassen für die Präfix-Expansion optimiert werden können. Beispiele für weitere sinnvolle Optimierungskriterien, für die man ebenfalls mit dynamischer Programmierung die optimale oder eine fast-optimale Wahl der Längenklassen berechnen kann, sind:

- Man betrachtet eine konkrete Datenstruktur für IP Prefix Lookup (z.B. Binärsuche nach Präfixlängen) und wählt die Längenklassen für die Präfix-Expansion so, dass für diese Datenstruktur der Speicherplatzbedarf minimiert wird (für Binärsuche nach Präfixlängen ist der gesamte Speicherplatzbedarf zum Beispiel gegeben durch die Anzahl von Präfixen plus die Anzahl der nötigen Marker).
- Man betrachtet eine konkrete Datenstruktur für IP Prefix Lookup und gibt sich den erlaubten Speicherplatzbedarf vor (z.B. bestimmt durch die Grösse des vorhandenen Second-Level-Caches). Dann wählt man die Längenklassen für die Präfix-Expansion so, dass der erlaubte Speicherplatzbedarf nicht überschritten wird und die Worst-Case-Laufzeit für die Suche nach dem längsten passenden Präfix minimiert wird.

Im nächsten Abschnitt wollen wir eine Variante der Tries aus Abschnitt 4.1 betrachten und sehen, wie diese Datenstruktur von der Präfix-Expansion profitieren kann und wie man mittels dynamischer Programmierung die Längenklassen so wählt, dass tatsächlich der Speicherplatzbedarf der Datenstruktur minimiert wird.

### 4.3.3 Multibit-Tries und Präfix-Expansion

Der Nachteil der Tries aus Abschnitt 4.1 war, dass die Suche nach dem längsten passenden Präfix im Worst-Case  $W$  Schritte (Hauptspeicherzugriffe) erforderte, da in jedem Schritt nur ein Bit der Zieladresse bearbeitet wurde. Ein Ansatz, dieses Verhalten zu verbessern, besteht darin, in jedem Schritt nicht nur ein Bit, sondern gleich mehrere zu betrachten. Wenn man in einem Trie-Knoten zum Beispiel die nächsten 4 Bits der Zieladresse auf einmal betrachten möchte, so muss dieser Trie-Knoten ein Feld mit 16 Elementen (für alle  $2^4$  Kombinationen dieser 4 Bits) enthalten, die ggf. auf einen anderen Trie-Knoten verweisen oder einen ausgehenden Link spezifizieren (oder beides). Die so erhaltenen Multibit-Tries sind also Tries, bei denen die Beschriftungen der Kanten eines Knotens  $v$  zu seinen Kindern alle dieselbe Länge  $k_v$  haben; statt die Beschriftungen an den Kanten zu speichern, verwenden wir bei Knoten  $v$  ein Feld mit  $2^{k_v}$  Elementen, in dem für jede mögliche Kantenbeschriftung der Länge  $k_v$  steht, ob es eine solche Kante gibt (und wohin sie führt).

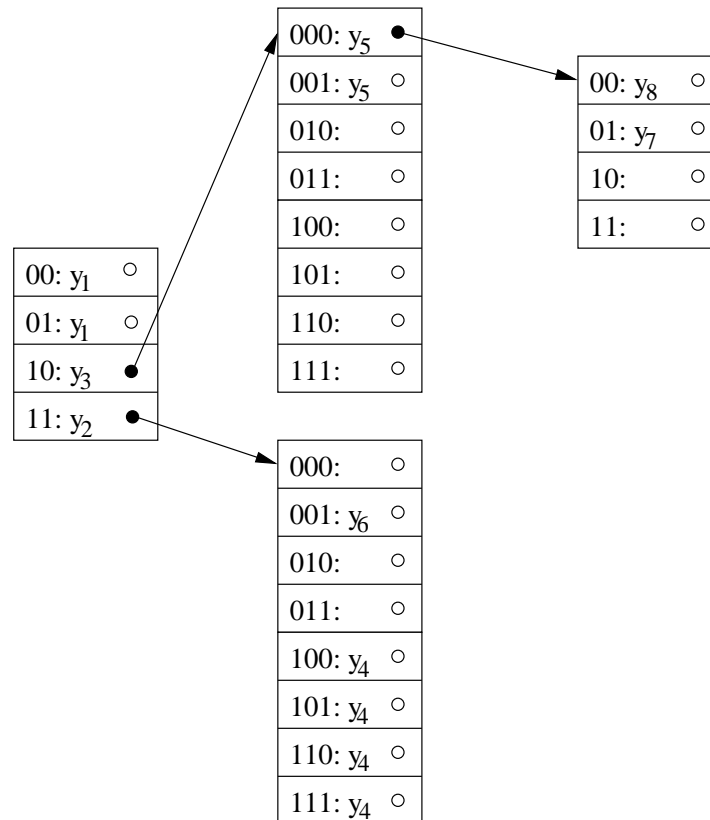


Abbildung 4.4: Multibit-Trie für IP Prefix Lookup.

**Beispiel:** Betrachten wir die Routing-Tabelle mit den folgenden Längenklassen (aus dem Beispiel aus Abschnitt 4.3.1):

$$L_2 = \{(00, y_1), (01, y_1), (10, y_3), (11, y_2)\}$$

$$L_5 = \{(11100, y_4), (11101, y_4), (11110, y_4), (11111, y_4), (10001, y_5), (10000, y_5), (11001, y_6)\}$$

$$L_7 = \{(1000000, y_8), (1000001, y_7)\}$$

Der entsprechende Multibit-Trie ist in Abb. 4.4 dargestellt. Die Anzahl der Bits, die in einem Trie-Knoten berücksichtigt werden, ist gleich der Differenz von einer nicht-leeren Längenkategorie zur nächsten, hier also 2 Bits in der Wurzel,  $5 - 2 = 3$  Bits in den Kindern der Wurzel und  $7 - 5 = 2$  Bits in den Kindeskindern der Wurzel. In jedem Feldelement eines Trie-Knotens ist die zugehörige Bitkombination dargestellt, der zugehörige ausgehende Link  $y_i$  (falls diesem Element ein Präfix aus der Routing-Tabelle entspricht) und der Verweis auf einen anderen Trie-Knoten (falls die Routing-Tabelle längere Präfixe enthält, die mit dem Präfix des aktuellen Elements beginnen). Falls ein Element keinen Verweis auf einen anderen Trie-Knoten enthält, so ist dies in Abb. 4.4 durch einen leeren Kreis angedeutet.

Wenn man in dem Multibit-Trie aus Abb. 4.4 nach der Zieladresse  $1000010\dots$  sucht, so werden die ersten 7 Bits der Adresse in die Teile 10, 000 und 10 aufgespalten und man folgt dem entsprechenden Pfad im Trie (d.h. über das Element '10:  $y_3$ ' im Wurzelknoten zum Knoten in der Mitte oben und von dort über das Element '000:  $y_5$ ' zum rechten Knoten, wo man schliesslich beim Element '10:' terminiert). Der letzte Wert  $y_i$ , den man auf dem

Pfad gesehen hat, ist  $y_5$ . Daher wird ein Paket mit Zieladresse 1000010... über den Link  $y_5$  weitergeleitet.  $\blacklozenge$

Die Worst-Case-Laufzeit für die Suche im Multibit-Trie ist gegeben durch die *Tiefe* des Trie, d.h. durch die maximale Anzahl Knoten auf einem Pfad von der Wurzel zu einem Blatt (im Beispiel aus Abb. 4.4 ist die Tiefe gleich 3). In jedem Knoten muss man nämlich einfach in dem Feldelement nachsehen, dessen Index durch die nächsten Bits der Zieladresse gegeben ist, und dieser Zugriff benötigt nur konstante Zeit (und nur einen Speicherzugriff).

Wir nehmen der Einfachheit halber an, dass in jeder *Schicht* des Trie (d.h. in jeder Menge von Trie-Knoten mit demselben Abstand von der Wurzel) alle Knoten gleich viele Feldelemente enthalten sollen. Das heisst, dass die Anzahl Bits der Zieladresse, die wir beim aktuellen Trie-Knoten als Index verwenden, nur von der aktuellen Schicht abhängt, aber nicht davon, in welchem Knoten der Schicht wir uns gerade befinden.<sup>1</sup> In diesem Fall entspricht jede Schicht des Trie genau einer Längengruppe der Routing-Tabelle (wie das auch in Abb. 4.4 der Fall war). Wenn wir durch Präfix-Expansion eine gegebene Routing-Tabelle so modifizieren, dass es nur noch  $r$  nicht-leere Längengruppen gibt, so ist die Laufzeit für eine Suche im resultierenden Multibit-Trie also  $O(r)$  ( $r$  Speicherzugriffe). Das Einfügen und Löschen von Routing-Einträgen lässt sich in Multibit-Tries ebenfalls verhältnismässig effizient realisieren; dies ist das Thema einer der Übungsaufgaben.

### Minimierung des Speicherplatzes für Multibit-Tries

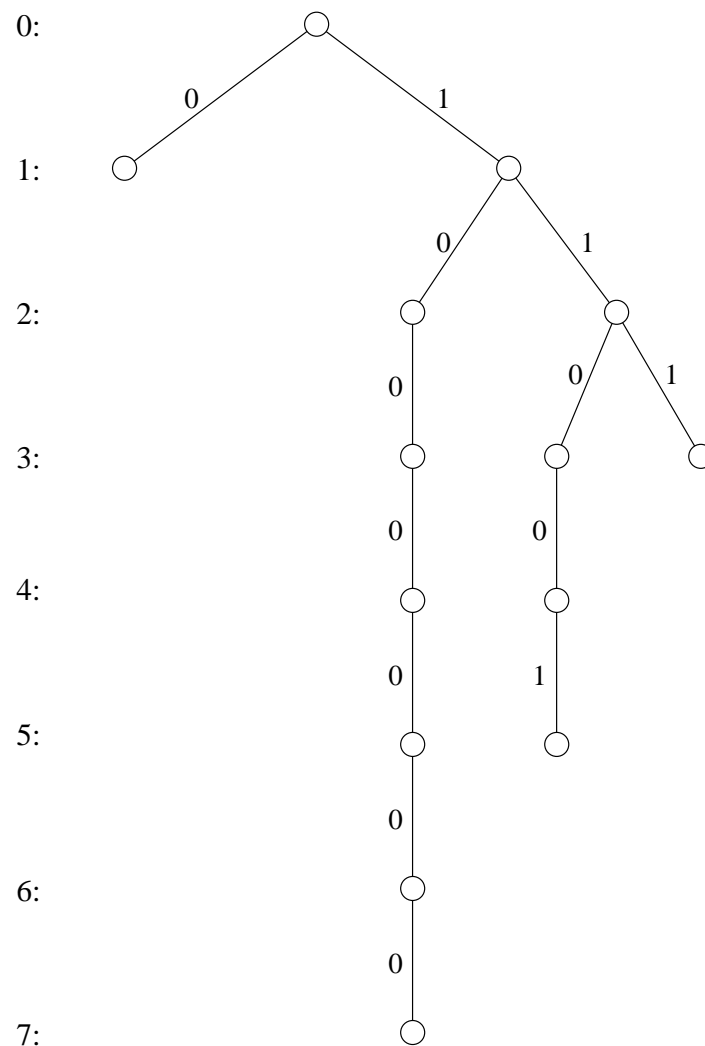
Wenn eine gewünschte Tiefe  $r$  des Multibit-Trie (und damit die Worst-Case-Anzahl von Speicherzugriffen bei der Suche) vorgegeben ist, würde man gerne die  $r$  nicht-leeren Längengruppen der Routing-Tabelle so wählen, dass der Speicherplatzbedarf für den Multibit-Trie minimal wird. Wir messen den Speicherplatzbedarf in Vielfachen der Grösse eines Feldelementes in einem Trie-Knoten. Der Speicherplatzbedarf für jeden Knoten des Trie ist dann gerade die Anzahl seiner Feldelemente, und der für den gesamten Trie benötigte Speicher ergibt sich aus der Summe der Speicheranforderungen aller Knoten. Für den Beispiel-Trie aus Abb. 4.4 ist der Speicherplatzbedarf beispielsweise  $2^2 + 2^3 + 2^3 + 2^2 = 24$ .

Um mittels dynamischer Programmierung die  $r$  nicht-leeren Längengruppen zu bestimmen, die zu einem minimalen Speicherplatzbedarf des Multibit-Trie führen, müssen wir uns zuerst überlegen, wie viel Speicherplatz die Schicht für Längengruppe  $L_\ell$  benötigt, wenn die nächstkleinere nicht-leere Längengruppe  $L_{\ell'}$  mit  $\ell' < \ell$  ist. Wir bezeichnen diesen Speicherplatzbedarf mit  $A[\ell'][\ell]$  (für  $0 \leq \ell' < \ell$ ). Um  $A[\ell'][\ell]$  bestimmen zu können, bauen wir zuerst einen 1-Bit-Trie  $T_1$  (wie in Abschnitt 4.1, aber ohne Pfadkomprimierung) für die ursprünglichen Präfixe auf. Sei  $n(i)$  die Anzahl von Knoten in  $T_1$ , die Abstand  $i$  von der Wurzel des Trie haben und die mindestens ein Kind haben. Man sieht nun leicht, dass

$$A[\ell'][\ell] = 2^{\ell-\ell'} \cdot n(\ell')$$

gilt: in den Knoten der Schicht für  $L_{\ell'}$  des Multibit-Trie verweist jedes Feldelement, das einem Knoten im 1-Bit-Trie mit mindestens einem Kind entspricht, auf einen Knoten der Schicht für  $L_\ell$ ; also muss die Schicht für  $L_\ell$  genau  $n(\ell')$  Knoten enthalten, die jeweils  $2^{\ell-\ell'}$  Feldelemente speichern. Mit Hilfe von  $T_1$  können wir also die Werte  $A[\ell'][\ell]$  für alle  $\ell' < \ell$  leicht berechnen.

<sup>1</sup>Man kann auch Tries betrachten, in denen die Anzahl der Feldelemente für jeden Knoten anders sein darf. Mehr zu solchen Tries mit *variable stride* kann man in [SV99] nachlesen.

Abbildung 4.5: 1-Bit-Trie zur Berechnung der  $A[l'][l]$ 

**Beispiel:** Betrachten wir wieder die Routing-Tabelle mit den folgenden nicht-leeren Längerklassen:

$$L_1 = \{(0, y_1), (1, y_2)\}$$

$$L_2 = \{(10, y_3)\}$$

$$L_3 = \{(111, y_4)\}$$

$$L_4 = \{(1000, y_5)\}$$

$$L_5 = \{(11001, y_6)\}$$

$$L_6 = \{(100000, y_7)\}$$

$$L_7 = \{(1000000, y_8)\}$$

Der 1-Bit-Trie für diese Präfixe ist in Abb. 4.5 dargestellt. Man kann leicht nachrechnen, dass  $A[0][2] = 2^{2-0} \cdot n(0) = 4 \cdot 1 = 4$ ,  $A[2][5] = 2^{5-2} \cdot n(2) = 8 \cdot 2 = 16$  und  $A[5][7] = 2^{7-5} \cdot n(5) = 4 \cdot 1 = 4$  gilt. Diese Werte stimmen mit dem tatsächlichen Speicherplatzbedarf

**Berechnung der  $M[\ell][k]$  für Multibit-Tries**  
**Eingabe:** Werte  $A[\ell'][\ell]$  für  $0 \leq \ell' < \ell \leq m$ , Anzahl  $r$  von Längenklassen  
**Ausgabe:** Werte  $M[\ell][k]$  und  $P[\ell][k]$  für  $0 \leq \ell \leq m$ ,  $1 \leq k \leq r$

```

for  $\ell = 0$  to  $m$  do
  for  $k = 1$  to  $r$  do
    if  $\ell = 0$  then  $M[\ell][k] := 0$ ;  $P[\ell][k] := 0$ ;
    else if  $k = 1$  then  $M[\ell][k] := A[0][\ell]$ ;  $P[\ell][k] := 0$ ;
    else
       $M[\ell][k] := \min_{0 \leq \ell' < \ell} M[\ell'][k - 1] + A[\ell'][\ell]$ ;
       $P[\ell][k] :=$  das  $\ell'$ , für das sich in der vorigen Zeile das Minimum ergab;
    fi;
  od;
od;

```

Abbildung 4.6: Dynamische Programmierung zur Minimierung des Speicherplatzbedarfs für einen Multibit-Trie mit  $r$  Schichten.

der drei Schichten im Trie mit den Längenklassen  $L_2$ ,  $L_5$  und  $L_7$  überein (vgl. Abb. 4.4). ♦

Betrachten wir nun den Multibit-Trie, der sich ergibt, wenn man die ursprünglichen Längenklassen  $L_1, \dots, L_\ell$  (für  $\ell = 0$  ist das die leere Menge) mittels Präfix-Expansion so umwandelt, dass höchstens  $k$  nicht-leere Längenklassen verbleiben und  $L_\ell$  die nicht-leere Längenkategorie mit der grössten Länge ist. Bezeichne mit  $M[\ell][k]$  den minimalen Speicherplatz für die Schichten in so einem Trie ab der Wurzel bis einschliesslich der Schicht, die  $L_\ell$  entspricht. Es gilt:

$$M[\ell][1] = A[0][\ell] \quad (\text{für alle } \ell \geq 1) \quad (4.4)$$

$$M[0][k] = 0 \quad (\text{für alle } k \geq 1) \quad (4.5)$$

$$M[\ell][k] = \min_{0 \leq \ell' < \ell} M[\ell'][k - 1] + A[\ell'][\ell] \quad (\text{für alle } \ell > 0, k > 1) \quad (4.6)$$

Gleichung (4.4) ist richtig, da der Speicherplatzbedarf für die Wurzel genau  $A[0][\ell] = 2^\ell \cdot n(0) = 2^\ell$  ist, wenn  $L_\ell$  die kleinste nicht-leere Längenkategorie ist. Gleichung (4.5) sagt nur aus, dass ein leerer Trie auch keinen Speicher benötigt. Gleichung (4.6) stimmt, da alle Möglichkeiten  $\ell'$  für die Wahl der nächstkleineren Längenkategorie (nach  $L_\ell$ ) berücksichtigt werden und für jede solche Möglichkeit der Speicherplatzbedarf gleich der Summe

- des optimalen Speicherplatzbedarfs für die Trie-Schichten zu Längenkategorien bis einschliesslich  $L_{\ell'}$  mit höchstens  $k - 1$  nicht-leeren Längenkategorien (ergibt  $M[\ell'][k - 1]$ ) und
- des Speicherplatzbedarfs für die Schicht der Längenkategorie  $L_\ell$  (der gerade  $A[\ell'][\ell]$  ist)

ist. Mit einem Algorithmus analog zu dem in Abb. 4.3 können alle Werte  $M[\ell][k]$  effizient berechnet werden, siehe Abb. 4.6. Wenn  $m$  die grösste Präfixlänge in der ursprünglichen Routing-Tabelle war, dann kann man aus  $M[m][r]$  den minimalen Speicherplatzbedarf eines Multibit-Tries mit  $r$  Schichten für diese Routing-Tabelle ablesen. Die optimale Wahl der nicht-leeren Längenkategorien kann man wieder aus den Variablen  $P[\ell][k]$  bestimmen.

Das Aufbauen des 1-Bit-Trie  $T_1$  und die Berechnung der Werte  $n(i)$  und  $A[\ell'][\ell]$  können in Zeit  $O(NW + W^2)$  realisiert werden. Die Berechnung der Werte  $M[\ell][k]$  dauert  $O(rW^2)$

Zeit. Insgesamt erhalten wir also in Zeit  $O(NW + rW^2)$  eine optimale Wahl von  $r$  nicht-leeren Längenklassen, so dass der Speicherplatzbedarf des sich ergebenden Multibit-Trie mit  $r$  Schichten minimal ist.

#### 4.4 Weitere Ansätze zur Realisierung des IP Prefix Lookup

In einer Arbeit von Feldmann und Muthukrishnan [FM00] wurde eine Datenstruktur für IP Prefix Lookup vorgestellt, die einen Tradeoff zwischen der Worst-Case-Laufzeit für die Suche nach dem längsten passenden Präfix und der Worst-Case-Laufzeit für Aktualisierungen der Routing-Tabelle ermöglicht. In dieser Arbeit wird auch die allgemeinere Problemvariante untersucht, bei der Pakete nicht nur anhand der Zieladresse, sondern aufgrund mehrerer Parameter (z.B. Adresse des Senders, Pakettyp, etc.) klassifiziert werden müssen.

Bei den dort vorgestellten Verfahren wird eine Datenstruktur für *Bereichsanfragen* (engl. *range location*) zu Hilfe genommen. Für eine gegebene Unterteilung der ganzen Zahlen von 0 bis  $U$  in Teilintervalle ermöglicht eine Datenstruktur für Bereichsanfragen, für jedes  $k \in \{0, 1, \dots, U\}$  effizient zu entscheiden, in welchem der Intervalle  $k$  liegt. Die besten bekannten dynamischen Datenstrukturen für Bereichsanfragen werden in [AT00] diskutiert.

Andere Vorschläge für die Realisierung des IP Prefix Lookup mittels Trie-basierter Datenstrukturen findet man in [DBCP97] und [NK98]. Auch zu hardware-basierten Verfahren (mittels *content-addressable memory*) gibt es Untersuchungen.

#### Referenzen

- [AT00] A. Andersson and M. Thorup. Tight(er) bounds on dynamic searching and priority queues. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing STOC'00*, pages 335–342, 2000.
- [DBCP97] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. *ACM Computer Communications Review*, 27(4):3–15, 1997. ACM SIGCOMM'97, September 1997.
- [FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with  $O(1)$  worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.
- [FM00] A. Feldmann and S. Muthukrishnan. Tradeoffs for packet classification. In *Proc. IEEE INFOCOM*, pages 1193–1202, 2000.
- [NK98] S. Nilsson and G. Karlsson. Fast address look-up for Internet routers. In *Proceedings of the IEEE Conference on Broadband Communications*. IEEE Press, 1998.
- [SV99] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*, 17(1):1–40, February 1999.
- [WVTP97] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed IP routing lookup. *ACM Computer Communications Review*, 27(4):25–36, 1997. ACM SIGCOMM'97, September 1997.



## Kapitel 5

# Online-Zugangskontrolle und Routing in ATM-Netzen

ATM (Asynchronous Transfer Mode) ist ein Übertragungsmodus, bei dem alle Daten in Zellen gleicher Grösse (53 Bytes) asynchron übertragen werden. Die Zellen vieler verschiedener Datenströme können sich die Kapazität eines Links teilen, indem die Zellen nacheinander über den Link übertragen werden (Zeitmultiplex). Asynchron bedeutet hier, dass die Zellen eines Datenstroms nicht notwendigerweise periodisch Zugang zu dem Link bekommen. Das Routing in ATM-Netzen wird mittels kurzer Identifier (*Virtual Channel Identifier* und *Virtual Path Identifier*) in den Zellen realisiert. Beim Aufbau einer Verbindung für einen Datenstrom wird auf allen Links auf einem Pfad vom Sender zum Empfänger die entsprechende Bandbreite reserviert, die dann diesem Datenstrom garantiert zur Verfügung steht. Dabei darf die Summe der Bandbreitenanforderungen von Datenströmen durch denselben Link natürlich nicht die Kapazität des Links überschreiten. Deshalb ist in einem ATM-Netz *Zugangskontrolle* nötig, d.h. man muss einen Teil der Verbindungswünsche ablehnen, um die Bandbreitengarantien für die akzeptierten Verbindungen einhalten zu können.

In der Realität ist die Situation etwas komplexer, da die einzelnen Datenströme nicht dauernd mit derselben Rate senden, sondern gelegentlich Bursts mit hohen Raten haben und dazwischen zeitweise mit einer geringeren Rate senden (man denke etwa an eine MPEG-komprimierte Videoübertragung). Auch kann ein Datenstrom neben der Bandbreitenreservierung noch weitere Anforderungen haben, z.B. eine Grenze für die akzeptable Verzögerung bei interaktiver Kommunikation. Das Anbieten solcher Gütegarantien bezüglich eines oder mehrerer Parameter wird oft als *guaranteed quality of service* bezeichnet. Wir wollen uns hier aber auf einer abstrakteren Ebene mit ATM-Netzen beschäftigen und nur Bandbreitenreservierung berücksichtigen. Wir nehmen an, dass jeder Verbindungswunsch den Sender, den Empfänger und eine Bandbreitenanforderung spezifiziert. Das Netzwerk muss für jeden Verbindungswunsch entscheiden, ob die Verbindung abgelehnt oder akzeptiert wird (Zugangskontrolle). Für akzeptierte Verbindungen muss der Pfad vom Sender zum Empfänger festgelegt werden, den der Datenstrom nimmt (Routing). Die Entscheidungen für einen Verbindungswunsch müssen getroffen werden, ohne Wissen über zukünftige Verbindungswünsche zu haben; ein solches Szenario nennt man *online* (im Gegensatz zu einem Offline-Szenario, bei dem der Algorithmus die vollständige Information über alle Eingabedaten kennt). Dabei will man ein Kriterium optimieren, das von den akzeptierten Verbindungswünschen abhängt: beispielsweise die Anzahl der akzeptierten Verbindungswünsche oder die Summe der Band-

breitenanforderungen der akzeptierten Verbindungswünsche. Wir betrachten im Folgenden das letztere Kriterium; dieses entspricht einer Maximierung des Profits des Netzbetreibers, wenn die Kosten einer Verbindung proportional zur dafür reservierten Bandbreite sind.

Im nächsten Abschnitt geben wir ein Modell für Online-Zugangskontrolle und Routing in Netzen mit Bandbreitenreservierung an und erklären, wie wir die Qualität eines Algorithmus für diese Problemstellung bewerten wollen. Danach lernen wir einen Algorithmus für dieses Problem kennen und analysieren ihn.

## 5.1 Problemmodellierung

Das Netzwerk ist durch einen gerichteten Graphen  $G = (V, E)$  gegeben, der aus  $|V| = n$  Knoten und  $|E| = m$  Kanten (Links) besteht. Jeder Link  $e \in E$  hat eine Kapazität  $u(e)$ . Nun werden nacheinander  $k$  Verbindungswünsche  $\beta_1, \beta_2, \dots, \beta_k$  an das Netzwerk gestellt. Die Sequenz dieser  $k$  Verbindungswünsche bezeichnen wir mit  $\beta = (\beta_1, \dots, \beta_k)$ . Jeder Verbindungswunsch  $\beta_i$  ist durch ein Tupel mit vier Komponenten gegeben:

$$\beta_i = (s_i, t_i, b_i, \rho_i)$$

Dabei ist  $s_i \in V$  der Sender,  $t_i \in V$  der Empfänger,  $b_i$  die Übertragungsrate (Bandbreitenanforderung) und  $\rho_i$  der Gewinn, den der Netzbetreiber bei Akzeptierung des Verbindungswunsches  $\beta_i$  erhält. Wir nehmen der Einfachheit halber an, dass akzeptierte Verbindungen unbegrenzt lange aktiv bleiben. Ferner nehmen wir an, dass  $\rho_i = n \cdot b_i$  gilt, dass also der Profit einer Verbindung proportional zur Übertragungsrate ist (der Faktor  $n$  ist eine Normalisierung, die sich später als nützlich erweist).

Realistischere Modelle lassen sich mit ähnlichen Methoden behandeln und analysieren. Man kann zum Beispiel Verbindungen zulassen, die endliche Dauern haben oder deren Übertragungsrate sich über die Dauer der Verbindung ändert. Man kann auch erlauben, dass der Profit um einen gewissen Faktor von  $n \cdot b_i$  abweicht. Wir wollen hier aber nicht auf diese Verallgemeinerungen eingehen, da die grundsätzliche Technik (Routing entlang kürzester Pfade bezüglich einer exponentiellen Kostenfunktion für die Links) dieselbe ist.

Ein Algorithmus für Online-Zugangskontrolle und Routing bearbeitet die Verbindungswünsche  $\beta_1, \dots, \beta_k$  nacheinander in dieser Reihenfolge und muss für jeden Verbindungswunsch  $\beta_i$  ohne Wissen über die späteren Verbindungswünsche entscheiden, ob  $\beta_i$  akzeptiert oder abgelehnt wird. Falls  $\beta_i$  akzeptiert wird, so muss ein Pfad  $P_i$  von  $s_i$  nach  $t_i$  in  $G$  bestimmt werden, entlang dem  $\beta_i$  geroutet wird. Zu jeder Zeit muss für jede Kante  $e \in E$  gelten, dass die Summe der Übertragungsraten  $b_i$  aller akzeptierten Verbindungswünsche, deren Pfad  $P_i$  die Kante  $e$  enthält, höchstens  $u(e)$  ist. Verbindungswünsche, die bereits akzeptiert wurden, können nachträglich nicht mehr abgebrochen werden.

Sei  $A(\beta)$  die Summe der Profite  $\rho_i$  aller Verbindungswünsche, die der Online-Algorithmus  $A$  bei Eingabe  $\beta$  akzeptiert. Sei  $OPT(\beta)$  die Summe der Profite aller akzeptierten Verbindungswünsche einer optimalen Lösung  $A^*$  für  $\beta$ , d.h. einer Auswahl von akzeptierten Verbindungswünschen und Pfaden für diese Verbindungswünsche, so dass die Kapazitätsschranken eingehalten werden und die Summe der Profite aller akzeptierten Verbindungswünsche maximal ist. Man beachte, dass kein Online-Algorithmus immer die optimale Lösung liefern kann, da ihm die zukünftigen Verbindungswünsche nicht bekannt sind.

Wir sagen, dass der Online-Algorithmus  $A$  *kompetitive Rate*  $c \geq 1$  hat, wenn für jede Eingabe  $\beta$  gilt, dass  $OPT(\beta)/A(\beta) \leq c$  ist. Einen solchen Algorithmus nennen wir auch

$c$ -kompetitiv. Der Profit, den ein  $c$ -kompetitiver Online-Algorithmus für Zugangskontrolle und Routing liefert, ist immer höchstens um Faktor  $c$  kleiner als der optimale Profit. Die kompetitive Rate bei Online-Algorithmen ist äquivalent zur Approximationsrate bei Offline-Algorithmen.

## 5.2 Der Algorithmus für Zugangskontrolle und Routing

Sei  $\mu = 2n + 2$  und sei  $u_{\min} = \min_{e \in E} u(e)$ . Wir benötigen die Annahme, dass für jeden Verbindungswunsch  $\beta_i$  gilt, dass

$$b_i \leq \frac{u_{\min}}{\log \mu}. \quad (5.1)$$

Das heisst, jede Verbindung kann höchstens einen  $1/\log \mu$ -Anteil der Kapazität einer Kante belegen. Da der Trend zu Netzen mit immer höherer Linkbandbreite geht, ist diese Annahme in der Praxis oft erfüllt.

Die *normalisierte Last* auf Kante  $e$  direkt vor Bearbeitung von  $\beta_j$  durch den Online-Algorithmus (d.h. die Summe der Bandbreitenanforderungen  $b_i$  aller zu diesem Zeitpunkt akzeptierten Verbindungswünsche, deren Pfad  $P_i$  die Kante  $e$  enthält, geteilt durch die Kapazität  $u(e)$  dieser Kante) wird mit  $\lambda_e(j)$  bezeichnet. Mit  $\lambda_e(k+1)$  wird die normalisierte Last auf Kante  $e$  nach Bearbeitung aller  $k$  Verbindungswünsche bezeichnet.

Der Algorithmus lässt sich nun wie folgt spezifizieren.

- Sei  $\beta_j$  der nächste zu bearbeitende Verbindungswunsch. Definiere für jede Kante  $e \in E$ :

$$c_e(j) = u(e) \cdot (\mu^{\lambda_e(j)} - 1) \quad (5.2)$$

- Weise nun jeder Kante die Kosten

$$w_j(e) = \frac{b_j}{u(e)} c_e(j)$$

zu und berechne einen kürzesten Pfad  $P$  von  $s_j$  nach  $t_j$  bezüglich dieser Kantenkosten  $w_j(e)$  (zum Beispiel mit dem Algorithmus von Dijkstra, siehe Kapitel 3.1).

- Falls der kürzeste Pfad  $P$  Länge höchstens  $\rho_j$  hat, d.h.  $\sum_{e \in P} \frac{b_j}{u(e)} c_e(j) \leq \rho_j$ , dann akzeptiere den Verbindungswunsch  $\beta_j$  und weise  $\beta_j$  den Pfad  $P_j = P$  zu; andernfalls, lehne den Verbindungswunsch  $\beta_j$  ab.

Die Grundidee des Algorithmus ist also, für jede Kante einen Kostenwert zu definieren, der exponentiell in der normalisierten Kantenlast ist, und einen Verbindungswunsch genau dann zu akzeptieren, wenn der kürzeste Pfad vom Sender zum Empfänger bezüglich dieser Kostenwerte nicht teurer ist als der Profit des Verbindungswunsches.

### 5.2.1 Analyse des Algorithmus

Wir teilen die Analyse des Algorithmus in drei Lemmas auf. Das erste Lemma setzt den vom Algorithmus nach Bearbeitung von  $j$  erreichten Profit mit den Werten  $c_e(j+1)$  in Beziehung.

**Lemma 5.1** Sei  $0 \leq j \leq k$  und sei  $A_j$  die Menge der Indizes der vom Algorithmus akzeptierten Verbindungsanfragen unter  $\beta_1, \beta_2, \dots, \beta_j$ . Dann gilt:

$$2 \log \mu \sum_{i \in A_j} \rho_i \geq \sum_{e \in E} c_e(j+1)$$

**Beweis.** Wir beweisen das Lemma per Induktion nach  $j$ . Für  $j = 0$  steht auf beiden Seiten der Ungleichung 0, die Ungleichung ist daher richtig (Induktionsanfang). Nehmen wir nun an, die Ungleichung sei für alle Indizes kleiner als  $j$  richtig und betrachten wir die Bearbeitung von  $\beta_j$ . Falls  $\beta_j$  abgelehnt wird, so ändern sich beide Seiten der Ungleichung nicht und sie bleibt weiterhin gültig. Wenn  $\beta_j$  akzeptiert wird, so wird die linke Seite der Ungleichung um  $2\rho_j \log \mu$  grösser und die rechte Seite um  $\sum_{e \in P_j} (c_e(j+1) - c_e(j))$ . Es reicht also zu zeigen, dass

$$\sum_{e \in P_j} (c_e(j+1) - c_e(j)) \leq 2\rho_j \log \mu$$

gilt, dann muss die Ungleichung aus der Behauptung des Lemmas auch für  $j$  gelten.

Nach Definition von  $c_e(j)$  gilt für  $e \in P_j$ :

$$\begin{aligned} c_e(j+1) - c_e(j) &= u(e) \cdot \left( \mu^{\lambda_e(j) + \frac{b_j}{u(e)}} - \mu^{\lambda_e(j)} \right) \\ &= u(e) \mu^{\lambda_e(j)} \left( \mu^{\frac{b_j}{u(e)}} - 1 \right) \\ &= u(e) \mu^{\lambda_e(j)} \left( 2^{\frac{b_j}{u(e)} \log \mu} - 1 \right) \end{aligned}$$

Aufgrund der Annahme (5.1) gilt  $\frac{b_j}{u(e)} \log \mu \leq 1$ . Da für  $0 \leq x \leq 1$  gilt, dass  $2^x - 1 \leq x$ , erhalten wir:

$$\begin{aligned} c_e(j+1) - c_e(j) &\leq u(e) \mu^{\lambda_e(j)} \frac{b_j}{u(e)} \log \mu \\ &= b_j \mu^{\lambda_e(j)} \log \mu \\ &= b_j \left( \frac{c_e(j)}{u(e)} + 1 \right) \log \mu \\ &= \log \mu \left( \frac{b_j}{u(e)} c_e(j) + b_j \right) \\ &= \log \mu (w_j(e) + b_j) \end{aligned}$$

Wenn wir die so erhaltene Ungleichung über alle  $e \in P_j$  aufsummieren und  $\sum_{e \in P_j} w_j(e) \leq \rho_j$  verwenden (dies gilt, weil  $\beta_j$  akzeptiert wurde), so erhalten wir:

$$\sum_{e \in P_j} (c_e(j+1) - c_e(j)) \leq \log \mu (\rho_j + |P_j| \cdot b_j).$$

Da  $|P_j| \cdot b_j \leq n \cdot b_j = \rho_j$  ist, folgt

$$\sum_{e \in P_j} (c_e(j+1) - c_e(j)) \leq \log \mu (\rho_j + \rho_j) = 2\rho_j \log \mu$$

und damit die Induktionsbehauptung.  $\square$

Als nächstes wollen wir eine obere Schranke für den Profit derjenigen Verbindungswünsche herleiten, die in der optimalen Lösung akzeptiert werden, aber nicht in der vom Algorithmus berechneten Lösung.

**Lemma 5.2** *Sei  $Q$  die Menge der Indizes von Verbindungswünschen, die in der optimalen Lösung  $A^*$  akzeptiert werden, aber nicht vom Online-Algorithmus. Sei  $\ell = \max Q$ . Dann gilt  $\sum_{j \in Q} \rho_j < \sum_{e \in E} c_e(\ell)$ .*

**Beweis.** Für  $j \in Q$  bezeichnen wir mit  $P_j^*$  den Pfad, den die optimale Lösung dem Verbindungswunsch  $\beta_j$  zugewiesen hat. Da  $\beta_j$  vom Online-Algorithmus abgelehnt wurde und da die Werte  $c_e(j)$  mit  $j$  monoton wachsen, gilt:

$$\rho_j < \sum_{e \in P_j^*} w_j(e) = \sum_{e \in P_j^*} \frac{b_j}{u(e)} c_e(j) \leq \sum_{e \in P_j^*} \frac{b_j}{u(e)} c_e(\ell)$$

Wenn wir diese Ungleichung über alle  $j \in Q$  aufsummieren, erhalten wir:

$$\begin{aligned} \sum_{j \in Q} \rho_j &< \sum_{j \in Q} \sum_{e \in P_j^*} \frac{b_j}{u(e)} c_e(\ell) \\ &= \sum_{e \in E} \sum_{j \in Q: e \in P_j^*} \frac{b_j}{u(e)} c_e(\ell) \\ &= \sum_{e \in E} c_e(\ell) \sum_{j \in Q: e \in P_j^*} \frac{b_j}{u(e)} \\ &\leq \sum_{e \in E} c_e(\ell) \end{aligned}$$

Die letzte Ungleichung ist richtig, da die optimale Lösung die Kapazitätsschranken einhalten muss und daher  $\sum_{j \in Q: e \in P_j^*} \frac{b_j}{u(e)} \leq 1$  gelten muss.  $\square$

Der Profit der optimalen Lösung ist beschränkt durch den Profit der Verbindungswünsche mit Index in  $Q$  plus den Profit der Verbindungswünsche mit Index in  $A_k$ , der vom Algorithmus berechneten Lösung. Wie die folgende Rechnung zeigt, folgt aus den Lemmas 5.1 und 5.2 bereits, dass  $OPT(\beta) \leq (2 \log \mu + 1) \cdot A(\beta)$  gilt.

$$\begin{aligned} OPT(\beta) &\leq \sum_{i \in Q} \rho_i + \sum_{i \in A_k} \rho_i \\ &< \sum_{e \in E} c_e(\ell) + \sum_{i \in A_k} \rho_i \\ &\leq \sum_{e \in E} c_e(k+1) + \sum_{i \in A_k} \rho_i \\ &\leq 2 \log \mu \sum_{i \in A_k} \rho_i + \sum_{i \in A_k} \rho_i \\ &= (2 \log \mu + 1) \sum_{i \in A_k} \rho_i = (2 \log \mu + 1) A(\beta) \end{aligned}$$

Wir müssen nur noch nachweisen, dass die vom Algorithmus berechnete Lösung die Kapazitätsschranken einhält.

**Lemma 5.3** Sei  $A_k$  die Menge der Indizes der vom Online-Algorithmus akzeptierten Verbindungswünsche. Für jede Kante  $e \in E$  gilt:  $\sum_{j \in A_k: e \in P_j} b_j \leq u(e)$ .

**Beweis.** Beweis durch Widerspruch. Nehmen wir an, dass der Algorithmus nach der Annahme des Verbindungswunsches  $\beta_j$  zum ersten Mal eine Kantenkapazität überschreitet. Sei  $e$  die betroffene Kante. Es muss

$$\lambda_e(j) > 1 - \frac{b_j}{u(e)}$$

gelten, da  $e$  in  $P_j$  enthalten sein muss und die normalisierte Last auf  $e$  nach der Annahme von  $\beta_j$  gleich  $\lambda_e(j) + \frac{b_j}{u(e)} > 1$  ist. Unter Verwendung der Definition von  $c_e(j)$  und Annahme (5.1) erhalten wir:

$$\frac{c_e(j)}{u(e)} = \mu^{\lambda_e(j)} - 1 > \mu^{1 - \frac{b_j}{u(e)}} - 1 \geq \mu^{1 - \frac{1}{\log \mu}} - 1 = \frac{\mu}{2} - 1 = \frac{2n + 2}{2} - 1 = n$$

Also gilt  $\sum_{e' \in P_j} w_j(e') \geq w_j(e) = \frac{b_j}{u(e)} c_e(j) > b_j n = \rho_j$ . Dies ist ein Widerspruch dazu, dass  $\beta_j$  angenommen und über den Pfad  $P_j$  geroutet wurde, da die Kosten des Pfades dann höchstens  $\rho_j = n \cdot b_j$  sein müssten.  $\square$

Die Ergebnisse können in dem folgenden Satz zusammengefasst werden.

**Satz 5.4** Der Online-Algorithmus für Zugangskontrolle und Routing überschreitet nie die Kantenkapazitäten und erzielt kompetitive Rate  $O(\log n)$ . Er erzielt für jede Sequenz  $\beta$  von Verbindungswünschen einen Profit  $A(\beta)$ , der höchstens um den Faktor  $2 \log(2n+2) + 1$  kleiner ist als der Profit einer optimalen Lösung.

### 5.3 Literaturhinweise

Der vorgestellte Algorithmus für Online-Zugangskontrolle und Routing in ATM-Netzen stammt von Awerbuch, Azar und Plotkin [AAP93], wobei dort die Problemvariante mit endlichen Dauern betrachtet wird. Die Darstellung hier orientiert sich an dem Überblicksartikel von Plotkin [Plo95].

#### Referenzen

- [AAP93] Baruch Awerbuch, Yossi Azar, and Serge Plotkin. Throughput-competitive on-line routing. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science FOCS'93*, pages 32–40, 1993.
- [Plo95] Serge Plotkin. Competitive routing of virtual circuits in atm networks. *IEEE Journal of Selected Areas in Communications*, 13:1128–1136, August 1995.

## Kapitel 6

# Wellenlängenzuteilung in optischen WDM-Netzen

In optischen Kommunikationsnetzen werden Daten mit Laserstrahlen durch Glasfaserkabel übertragen. Hauptvorteile der optischen Übertragung sind die extrem hohe Bandbreite, die Glasfaserkabel bieten, und die geringen Bitfehlerraten. Ein einziges Glasfaserkabel bietet eine Bandbreite im Bereich von 25–30 THz und ermöglicht damit Übertragungsraten bis zu mehreren Terabits ( $10^{12}$  Bits) pro Sekunde. Zum Beispiel könnten alle Telefongespräche, die weltweit zur selben Zeit geführt werden, problemlos durch ein einziges Glasfaserkabel übertragen werden.

Ein Problem bei der Nutzung dieser hohen Bandbreite ist, dass herkömmliche elektronische Komponenten Signale im Terahertz-Bereich nicht verarbeiten können. Abhilfe schafft die Technologie des Wellenlängen-Multiplex (WDM, *wavelength-division multiplexing*), bei der die Bandbreite eines Glasfaserkabels in Kanäle mit verschiedenen Wellenlängen aufgeteilt wird. Daten verschiedener Verbindungen können dann mittels Laserstrahlen verschiedener Wellenlängen gleichzeitig durch dasselbe Glasfaserkabel übertragen werden. Auf einer Wellenlänge werden Daten typischerweise mit Raten wie 2.4 Gbps (Gigabits pro Sekunde) oder 10 Gbps übertragen. Mit Hilfe optischer *Add-Drop-Multiplexer* (ADM) können einzelne Wellenlängen in ein optisches Netz eingespeist oder daraus extrahiert werden. Die elektronischen Komponenten des Netzes müssen dann nur die Bitrate einer einzelnen Wellenlänge verarbeiten können, was problemlos möglich ist.

Ferner gibt es frei konfigurierbare *optische Switches*, die die eingehenden Signale abhängig von ihrer Wellenlänge auf beliebige ausgehende Links weiterleiten können, ohne dass die Signale zwischenzeitlich in elektronische Form umgewandelt werden müssen. Die Weiterleitung ist damit so gut wie verzögerungsfrei. Die Wellenlänge eines Signals kann aber mit den heute verfügbaren optischen Switches bei der Weiterleitung nicht verändert werden.

Wenn ein Kommunikationsnetz aus Glasfaserkabeln und solchen optischen Switches aufgebaut ist, so kann eine Verbindung von Knoten  $u$  zu Knoten  $v$  eingerichtet werden, indem entlang eines Pfades von  $u$  nach  $v$  eine Wellenlänge für diese Verbindung reserviert wird und alle Switches entlang des Pfades so konfiguriert werden, dass die mit dieser Wellenlänge ankommenden Daten auf den nächsten Link des Pfades weitergeleitet werden.

Da der Weg der Daten durch das Netz nach Einrichtung der Verbindung nur noch von der Wellenlänge abhängt, wird diese Art des Routings auch als *Wellenlängenrouting* bezeichnet. Ein Vorteil dabei ist auch, dass das Netz völlig unabhängig von der Art der übertragenen Da-

ten und den verwendeten Protokollen ist: die Daten werden abhängig von ihrer Wellenlänge durch das Netz geleitet, unabhängig davon, ob es sich um Audio-Daten, eine Video-Übertragung oder Sonstiges handelt.

## 6.1 Problemdefinitionen

Wie oben beschrieben muss bei der Einrichtung einer Verbindung entlang eines Pfades eine Wellenlänge für die Verbindung reserviert werden. Die Zuordnung muss konfliktfrei sein, d.h. zwei Verbindungen, deren Pfade denselben Link enthalten, dürfen nicht dieselbe Wellenlänge erhalten. Die Anzahl der verfügbaren Wellenlängen ist eine beschränkte Ressource: die Kosten für das Netz sind umso grösser, je mehr Wellenlängen unterstützt werden müssen. Heute sind zum Beispiel Systeme mit höchstens etwa 80 Wellenlängen kommerziell verfügbar. Daher ist es sinnvoll, das Optimierungsproblem zu betrachten, für eine Menge von Verbindungsanfragen die Pfade und Wellenlängen so zuzuordnen, dass die Anzahl der nötigen Wellenlängen minimiert wird. Da man sich die Wellenlängen als Farben vorstellen kann, wird dieses Problem auch *Routing und Pfadfärbung* (engl. *routing and path coloring*, RPC) genannt. Wenn die Pfade eindeutig bestimmt (das ist in Netzen mit Baumtopologie der Fall) oder fest vorgegeben sind, spricht man einfach von *Pfadfärbung* (engl. *path coloring*, PC).

Das Netz wird als symmetrisch gerichteter Graph  $G = (V, E)$  modelliert, d.h. als gerichteter Graph, bei dem für jede Kante  $(u, v)$  auch die antiparallele Kante  $(v, u)$  enthalten ist. Eine Verbindungsanfrage  $r$  ist durch den Sender  $s_r$  und den Empfänger  $t_r$  bestimmt. Für die Einrichtung der Verbindung  $r$  muss ein gerichteter Pfad  $P(r)$  in  $G$  von  $s_r$  nach  $t_r$  und eine Farbe (Wellenlänge)  $w(r)$  bestimmt werden. Eine Zuordnung von Pfaden und Farben an Verbindungsanfragen heisst *konfliktfrei*, wenn für jede Kante  $e \in E$  und jede Farbe  $w$  höchstens eine Anfrage existiert, deren Pfad  $e$  enthält und der die Farbe  $w$  zugeordnet ist.

Somit lässt sich das Problem RPC wie folgt definieren.

Problem ROUTINGANDPATHCOLORING (RPC)

**Instanz:** symmetrisch gerichteter Graph  $G = (V, E)$ , Menge  $R$  von Verbindungsanfragen der Form  $r = (s_r, t_r)$  mit  $s_r, t_r \in V$

**Lösung:** konfliktfreie Zuordnung von gerichteten Pfaden  $P(r)$  und Farben  $w(r)$  für alle  $r \in R$

**Ziel:** minimiere die Anzahl verwendeter Farben

Für den Fall von eindeutig bestimmten oder vorgegebenen Pfaden erhalten wir das Problem PC.

Problem PATHCOLORING (PC)

**Instanz:** symmetrisch gerichteter Graph  $G = (V, E)$ , Menge  $P$  von gerichteten Pfaden in  $G$

**Lösung:** konfliktfreie Zuordnung von Farben  $w(p)$  für alle  $p \in P$

**Ziel:** minimiere die Anzahl verwendeter Farben

Die Probleme RPC und PC sind relevant, wenn ein optisches Netz entworfen wird oder wenn die verfügbaren Wellenlängen eines existierenden Netzes ausreichen, um alle Verbindungen einzurichten. Wenn die in einem existierenden Netz verfügbaren  $W$  Wellenlängen nicht ausreichen, um alle gewünschten Verbindungen konfliktfrei einzurichten, so möchte man zumindest eine möglichst grosse Anzahl der Verbindungsanfragen akzeptieren. Hier ergibt sich das so genannte MAXRPC-Problem.

Problem MAXIMUMROUTINGANDPATHCOLORING (MAXRPC)

**Instanz:** symmetrisch gerichteter Graph  $G = (V, E)$ , Menge  $R$  von Verbindungsanfragen der Form  $r = (s_r, t_r)$  mit  $s_r, t_r \in V$ , Anzahl  $W$  verfügbarer Wellenlängen

**Lösung:** Wahl einer Teilmenge  $R' \subseteq R$  und konfliktfreie Zuordnung von gerichteten Pfaden  $P(r)$  und Farben  $w(r) \in \{1, 2, \dots, W\}$  für alle  $r \in R'$

**Ziel:** maximiere  $|R'|$

Auch hier gibt es wieder die Problemvariante MAXPC, in der die Pfade bereits vorgegeben sind.

Der Spezialfall  $W = 1$  von MAXRPC und MAXPC führt zu einem sehr grundlegenden Optimierungsproblem, dem Finden einer maximalen Anzahl kantendisjunkter Pfade. Diese Problem wird als MEDP (engl. *maximum edge-disjoint paths*) bezeichnet, falls die Pfade vom Algorithmus zu wählen sind, und als MEDPwPP (engl. *MEDP with pre-specified paths*), falls die Pfade vorgegeben sind.

Problem MAXIMUMEDGEDISJOINTPATHS (MEDP)

**Instanz:** symmetrisch gerichteter Graph  $G = (V, E)$ , Menge  $R$  von Verbindungsanfragen der Form  $r = (s_r, t_r)$  mit  $s_r, t_r \in V$

**Lösung:** Wahl einer Teilmenge  $R' \subseteq R$  und Zuordnung von kantendisjunkten gerichteten Pfaden  $P(r)$  für alle  $r \in R'$

**Ziel:** maximiere  $|R'|$

Alle in diesem Abschnitt definierten Probleme können auch für ungerichtete Pfade in ungerichteten Graphen untersucht werden. Wir beschränken uns hier aber auf die Betrachtung der Varianten mit gerichteten Pfaden in symmetrisch gerichteten Graphen.

## 6.2 Überblick über bekannte Ergebnisse

Die im vorigen Abschnitt definierten Probleme wurden für verschiedene Netztopologien untersucht. Einige Topologien sind in Abb. 6.1 dargestellt. Nicht zu sehen ist hier die Topologie eines Baums von Ringen. Bäume von Ringen sind induktiv wie folgt definiert:

1. Ein einzelner Ring ist ein Baum von Ringen.
2. Wenn  $B_1$  und  $B_2$  Bäume von Ringen sind, so ist auch der Graph  $B$  ein Baum von Ringen, der aus  $B_1$  und  $B_2$  durch Identifizierung eines Knotens von  $B_1$  mit einem Knoten von  $B_2$  entsteht.

Da Ringe in optischen Netzen recht verbreitet sind, ist es interessant, Bäume von Ringen als Netztopologie zu betrachten.

Für manche Topologien konnten polynomielle Algorithmen gefunden werden, die immer optimale Lösungen liefern. In den meisten Topologien erwiesen sich die Probleme aber als  $\mathcal{NP}$ -schwer und es wurden Approximationsalgorithmen vorgeschlagen. Die Tabellen 6.1, 6.2 und 6.3 geben einen Überblick über bisher gefundene Ergebnisse. In den nächsten Abschnitten werden wir Algorithmen für einen Teil der Topologien genauer betrachten, wobei wir uns vor allem mit Pfadfärbungsalgorithmen befassen.

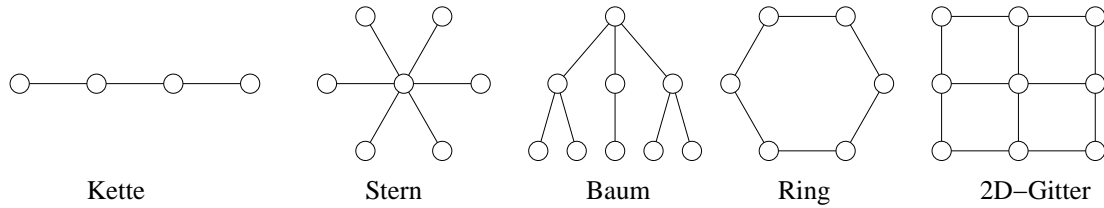


Abbildung 6.1: Einige Netztopologien

Ergebnisse für RPC und PC	
PC in Kette	polynomiell
PC in Stern	polynomiell
PC in Baum	$\mathcal{NP}$ -schwer, $\frac{5}{3}$ -Approximationsalgorithmus [EJK <sup>+</sup> 99]
PC in Ring	$\mathcal{NP}$ -schwer, $\frac{3}{2}$ -Approximationsalgorithmus [Kar80]
RPC in Ring	$\mathcal{NP}$ -schwer, 2-Approximationsalgorithmus [WW98]
RPC in 2D-Gitter	$\mathcal{NP}$ -schwer, $(\log \log n)^{O(1)}$ -Approximation [Rab96]
RPC in Baum von Ringen	$\mathcal{NP}$ -schwer, $\frac{10}{3}$ -Approximationsalgorithmus

Tabelle 6.1: Ergebnisse für RPC und PC

Ergebnisse für MEDP und MEDPwPP	
MEDP in Kette	polynomiell
MEDP in Stern	polynomiell
MEDP in Baum	$\mathcal{NP}$ -schwer, $\frac{5}{3}$ -Approximationsalgorithmus [EJ98]
MEDP in Ring	polynomiell
MEDPwPP in Ring	polynomiell
MEDP in 2D-Gitter	$\mathcal{NP}$ -schwer, $O(1)$ -Approximationsalgorithmus [KT95]
MEDP in allg. Graphen mit $m$ Kanten	$\mathcal{NP}$ -schwer, $O(\sqrt{m})$ -Approximationsalgorithmus [Kle96]

Tabelle 6.2: Ergebnisse für MEDP und MEDPwPP

Ergebnisse für MaxRPC und MaxPC	
MaxPC in Kette	polynomiell
MaxPC in Stern	polynomiell
MaxPC in Baum	$\mathcal{NP}$ -schwer, 2.22-Approximationsalgorithmus
MaxPC in Ring	$\mathcal{NP}$ -schwer, $\frac{e}{e-1} \approx 1.58$ -Approximationsalgorithmus
MaxRPC in Ring	$\mathcal{NP}$ -schwer, $\frac{e}{e-1} \approx 1.58$ -Approximationsalgorithmus
MaxRPC in 2D-Gitter	$\mathcal{NP}$ -schwer, $O(1)$ -Approximationsalgorithmus
MaxRPC in allg. Graphen mit $m$ Kanten	$\mathcal{NP}$ -schwer, $O(\sqrt{m})$ -Approximationsalgorithmus

Tabelle 6.3: Ergebnisse für MaxRPC und MaxPC

## 6.3 Algorithmen für Pfadfärbung

Zuerst betrachten wir Ketten und Bäume. In diesen Topologien sind die Pfade eindeutig bestimmt. Für eine gegebene Menge  $P$  von Pfaden bezeichnen wir für  $e \in E$  mit  $L(e)$  die Last der Kante  $e$ , d.h. die Anzahl der Pfade in  $P$ , die  $e$  enthalten. Sei  $L_{\max} = \max_{e \in E} L(e)$  die maximale Kantenlast. Offensichtlich ist  $L_{\max}$  eine untere Schranke für die Anzahl von Farben in einer optimalen Färbung der gegebenen Pfade.

### 6.3.1 Pfadfärbung in Ketten

In Ketten stellt sich heraus, dass es tatsächlich immer eine Färbung mit  $L_{\max}$  Farben gibt. Wir stellen uns die Knoten der Kette von links nach rechts aufsteigend durchnummeriert vor. Diejenigen Pfade, die in der Kette von links nach rechts laufen, sind völlig unabhängig von den Pfaden in der anderen Richtung. Daher können die Pfade in verschiedenen Richtungen unabhängig voneinander mit denselben Farben gefärbt werden. Der folgende, sehr einfache Algorithmus berechnet eine Färbung mit  $L_{\max}$  Farben für Pfade mit derselben Richtung.

- Bearbeite die Pfade in der Reihenfolge aufsteigender linker Endknoten. Ordne jedem Pfad die kleinste Farbnummer zu, so dass kein Konflikt mit einem bereits gefärbten Pfad erzeugt wird.

Um PC in Ketten zu lösen, wendet man diesen Algorithmus einfach nacheinander auf die Pfade beider Richtungen an.

**Satz 6.1** *Der angegebene Algorithmus berechnet für Pfade in Ketten in polynomieller Zeit eine optimale Färbung mit  $L_{\max}$  Farben.*

**Beweis.** Offensichtlich lässt sich der Algorithmus in polynomieller Zeit implementieren. Weiter ist klar, dass der Algorithmus mindestens  $L_{\max}$  Farben benötigt. Wir zeigen nun per Induktion nach der Anzahl bearbeiteter Pfade, dass höchstens  $L_{\max}$  Farben verwendet werden. Dabei betrachten wir o.B.d.A. die Pfade, die von links nach rechts laufen.

Zu Anfang (bevor ein Pfad gefärbt wurde) ist die Behauptung richtig. Nehmen wir an, dass die Induktionsbehauptung nach der Bearbeitung von  $k$  Pfaden richtig ist und sei  $p$  der  $(k + 1)$ -te Pfad, der bearbeitet wird. Sei  $(u, v)$  die erste Kante von  $p$ . Da alle vorher bearbeiteten Pfade ihre linken Endknoten nicht rechts von  $u$  haben, enthalten alle vorher bearbeiteten Pfade, die den Pfad  $p$  schneiden, ebenfalls die Kante  $(u, v)$ . Da höchstens  $L_{\max}$  Pfade die Kante  $(u, v)$  enthalten können, gibt es höchstens  $L_{\max} - 1$  Pfade, die bereits gefärbt sind und die  $p$  schneiden. Wenn  $p$  die kleinste freie Farbe zugeordnet wird, so liegt diese freie Farbe also immer unter den ersten  $L_{\max}$  Farben.  $\square$

### 6.3.2 Pfadfärbung in Bäumen

Im Gegensatz zu Ketten ist das Pfadfärbungsproblem in Bäumen  $\mathcal{NP}$ -schwer, sogar bereits in Binärbäumen. In Bäumen gibt es ausserdem nicht immer eine Färbung mit  $L_{\max}$  Farben. Beispielsweise sind in Abb 6.2 fünf Pfade in einem Baum dargestellt, für die  $L_{\max} = 2$  gilt, für deren Färbung aber drei Farben nötig sind.

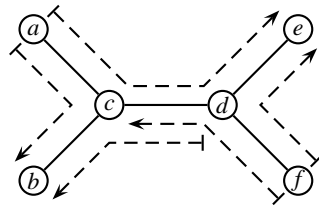


Abbildung 6.2: Instanz von PC in Bäumen

Wir erinnern uns, dass man die Präordernummerierung eines Baumes erhält, wenn man die Knoten der Baumes in der Reihenfolge durchnummeriert, in der sie von einem Präorderdurchlauf besucht werden (Abschnitt 3.2.1). Betrachten wir den folgenden Algorithmus für Pfadfärbung in Bäumen:

- Zu Anfang sind alle Pfade ungefärbt.
- Bearbeite die Knoten des Baumes in Reihenfolge ihrer Präordernummern.
- Bei Knoten  $v$  betrachte alle ungefärbten Pfade, die  $v$  berühren, in beliebiger Reihenfolge und ordne jedem die kleinste mögliche Farbnummer zu, so dass kein Konflikt entsteht.

**Satz 6.2** *Der angegebene Algorithmus berechnet für Pfade in Bäumen in polynomieller Zeit eine Färbung mit höchstens  $2L_{\max} - 1$  Farben. Da die optimale Farbanzahl mindestens  $L_{\max}$  ist, ist der Algorithmus ein 2-Approximationsalgorithmus.*

**Beweis.** Offensichtlich läuft der Algorithmus in polynomieller Zeit und liefert eine konfliktfreie Färbung. Wir beweisen mittels Induktion nach der Anzahl bearbeiteter Pfade, dass der Algorithmus höchstens  $2L_{\max} - 1$  Farben verwendet.

Zu Anfang (bevor ein Pfad gefärbt wurde) ist die Behauptung richtig. Nehmen wir an, dass die Induktionsbehauptung nach der Bearbeitung von  $k$  Pfaden richtig ist und sei  $p$  der  $(k + 1)$ -te Pfad, der bearbeitet wird. Sei  $v$  der Knoten, bei dem  $p$  bearbeitet wird. Da alle vorher gefärbten Pfade einen Knoten mit Präordernummer kleiner gleich der von  $v$  berühren, müssen alle vorher gefärbten Pfade, die den Pfad  $p$  schneiden, den Knoten  $v$  berühren und den Pfad  $p$  in einer zu  $v$  inzidenten Kante schneiden. Da  $p$  höchstens zwei zu  $v$  inzidente Kanten enthält, kann  $p$  also höchstens  $2(L_{\max} - 1)$  bereits gefärbte Pfade schneiden. Unter den ersten  $2L_{\max} - 1$  Farben findet man also immer eine, die  $p$  konfliktfrei zugewiesen werden kann.  $\square$

Der beste bekannte Algorithmus verwendet zur Pfadfärbung in Bäumen höchstens  $\lceil \frac{5}{3}L_{\max} \rceil$  Farben. Das Grundprinzip des Algorithmus ist ähnlich zu dem oben angegebenen, allerdings werden bei der Bearbeitung eines Knotens  $v$  die ungefärbten Pfade, die  $v$  berühren, alle zusammen betrachtet und mittels einer erheblich komplizierteren Methode so gefärbt, dass insgesamt höchstens  $\lceil \frac{5}{3}L_{\max} \rceil$  Farben verwendet werden und auf den zwei gerichteten Kanten zwischen zwei Knoten höchstens etwa  $\frac{4}{3}L_{\max}$  Farben. Wir wollen hier jedoch nicht näher auf diesen komplizierteren Algorithmus eingehen.

### 6.3.3 Pfadfärbung in Ringen

Wir betrachten das Problem RPC in Ringen. Für jede Verbindungsanfrage muss der Algorithmus einen der beiden möglichen Pfade (im oder gegen den Uhrzeigersinn) im Ring wählen und dann eine Farbe zuordnen. Die Minimierung der Farbanzahl ist auch hier  $\mathcal{NP}$ -schwer, doch es gibt wiederum einen sehr einfachen Algorithmus, der eine konstante Approximationsrate liefert:

- Wähle zwei beliebige benachbarte Knoten  $u$  und  $v$  im Ring und “schneide” die Kanten  $(u, v)$  und  $(v, u)$  durch. Man erhält so eine Kette mit Endknoten  $u$  und  $v$ .
- Betrachte die gegebenen Verbindungsanfragen als Pfade in der Kette und verwende den optimalen Algorithmus für Ketten aus Abschnitt 6.3.1.

**Satz 6.3** *Der angegebene Algorithmus ist ein 2-Approximationsalgorithmus für RPC in Ringen.*

**Beweis.** Sei  $S^*$  eine optimale Lösung für das Pfadfärbungsproblem mit  $OPT$  Farben und sei  $L^*$  die maximale Kantenlast der Pfade, die  $S^*$  den Verbindungsanfragen zuordnet. Sei  $L$  die maximale Kantenlast der Pfade, die der angegebene Algorithmus den Verbindungsanfragen zuweist. Man kann zeigen, dass  $L \leq 2L^*$  gilt (Übungsaufgabe). Da  $OPT \geq L^*$  gilt, folgt  $L \leq 2OPT$ . Der Algorithmus benötigt nur  $L$  Farben, um die Pfade in der Kette zu färben (Satz 6.1).  $\square$

Interessanterweise ist noch kein Approximationsalgorithmus für RPC in (symmetrisch gerichteten) Ringen bekannt, für den als Approximationsrate eine Konstante kleiner als 2 bewiesen werden konnte.

### 6.3.4 Pfadfärbung in Bäumen von Ringen

Betrachten wir schliesslich noch RPC in Bäumen von Ringen. Auch dieses Problem ist  $\mathcal{NP}$ -schwer, da RPC ja bereits für einen einzelnen Ring  $\mathcal{NP}$ -schwer ist. Die Idee, bei Ringen einfach einen beliebigen Ring “durchzuschneiden”, liefert auch hier einen einfachen Approximationsalgorithmus mit konstanter Approximationsrate:

- Wähle in jedem Ring des Baums von Ringen zwei beliebige benachbarte Knoten  $u$  und  $v$  und schneide die Kanten  $(u, v)$  und  $(v, u)$  durch. Der verbleibende Graph ist ein Baum.
- Verwende nun den Algorithmus für Pfadfärbung in Bäumen mit  $\frac{5}{3}L_{\max}$  Farben.

**Satz 6.4** *Der angegebene Algorithmus ist ein  $\frac{10}{3}$ -Approximationsalgorithmus für RPC in Bäumen von Ringen.*

**Beweis.** Sei  $S^*$  eine optimale Lösung für das Pfadfärbungsproblem mit  $OPT$  Farben und sei  $L^*$  die maximale Kantenlast der Pfade, die  $S^*$  den Verbindungsanfragen zuordnet. Sei  $L$  die maximale Kantenlast der Pfade, die der angegebene Algorithmus den Verbindungsanfragen zuweist. Man kann auch hier wieder zeigen, dass  $L \leq 2L^*$  gilt (Übungsaufgabe). Da  $OPT \geq L^*$  gilt, folgt  $L \leq 2OPT$ . Der Algorithmus benötigt höchstens  $\frac{5}{3}L \leq \frac{10}{3}OPT$  Farben, um die Pfade im Baum zu färben.  $\square$

```

Algorithmus A
Eingabe: Graph  $G$ , Menge  $P$  von Pfaden, Anzahl  $W$  von Farben
Ausgabe: disjunkte Teilmengen  $P_1, \dots, P_W$  von  $P$  (jedes  $P_i$  kantendisjunkt)
begin
  for  $i = 1$  to  $W$  do
    begin
       $P_i := A_1(G, P)$ ;
       $P := P \setminus P_i$ ;
    end
  end

```

Abbildung 6.3: Reduktion von MaxPC auf MEDPwPP

## 6.4 Reduktion von MaxRPC auf MEDP

In diesem Abschnitt wollen wir ein allgemeines Verfahren (verwendet unter anderem in [AAF<sup>+</sup>96]) kennen lernen, mit dem man Approximationsalgorithmen für MaxRPC bzw. für MaxPC in einer Netztopologie erhält, wenn man bereits einen  $\rho$ -Approximationsalgorithmus für MEDP bzw. MEDPwPP in dieser Topologie hat. Wir betrachten dieses Verfahren am Beispiel von MaxPC und MEDPwPP, für MaxRPC und MEDP funktioniert es analog.

Sei  $A_1$  ein  $\rho$ -Approximationsalgorithmus für MEDPwPP, d.h. für jede Menge  $P$  von gerichteten Pfaden in einem symmetrisch gerichteten Graphen  $G$  liefert  $A_1(G, P)$  eine Menge  $P' \subseteq P$  von Pfaden, die kantendisjunkt sind, so dass  $|P'|$  mindestens  $z^*/\rho$  Pfade enthält, wobei  $z^*$  die Anzahl von Pfaden in einer optimalen Lösung für MEDPwPP ist. Wir erhalten einen Approximationsalgorithmus  $A$  für MaxPC, indem wir bei einer Instanz mit  $W$  Farben den Algorithmus  $A_1$  einfach  $W$ -mal nacheinander aufrufen, wie in Abb. 6.3 angegeben.

**Satz 6.5** *Wenn  $A_1$  ein  $\rho$ -Approximationsalgorithmus für MEDPwPP ist, so hat der Algorithmus  $A$  aus Abb. 6.3 Approximationsrate höchstens*

$$\frac{1}{1 - e^{-1/\rho}} \leq \rho + 1$$

für MaxPC.

**Beweis.** Sei  $a_i = |P_i|$  für  $i = 1, 2, \dots, W$ . Die von Algorithmus  $A$  berechnete Lösung für MaxPC besteht aus  $a_1 + a_2 + \dots + a_W$  Pfaden. Sei  $O^*$  die Menge von Pfaden in einer optimalen Lösung für das MaxPC-Problem, und sei  $OPT = |O^*|$  die Anzahl von Pfaden in  $O^*$ . Da  $A_1$  ein  $\rho$ -Approximationsalgorithmus für MEDPwPP ist, gilt für  $k = 1, 2, \dots, W$ :

$$a_k \geq \frac{OPT - (a_1 + a_2 + \dots + a_{k-1})}{\rho W} \quad (6.1)$$

Wenn der Algorithmus  $A_1$  zum  $k$ -ten mal aufgerufen wird, sind nämlich höchstens  $a_1 + a_2 + \dots + a_{k-1}$  der Pfade aus  $O^*$  bereits einer Menge  $P_i$  zugeordnet, und da die restlichen mindestens  $OPT - (a_1 + a_2 + \dots + a_{k-1})$  Pfade in  $O^*$  mit  $W$  Farben gefärbt werden können, gibt es in  $P$  zu diesem Zeitpunkt noch eine Menge von  $(OPT - (a_1 + a_2 + \dots + a_{k-1}))/W$  kantendisjunkten Pfaden.

Wir behaupten nun, dass für  $k = 1, \dots, W$  gilt:

$$a_1 + a_2 + \dots + a_k \geq OPT \cdot \left(1 - \left(1 - \frac{1}{\rho W}\right)^k\right) \quad (6.2)$$

Wir beweisen die Ungleichung (6.2) mittels Induktion nach  $k$ . Für  $k = 1$  folgt die Ungleichung direkt aus (6.1).

Sei nun  $k > 1$  und nehmen wir an, dass die Ungleichung (6.1) für  $k - 1$  richtig ist. Wir können wie folgt rechnen:

$$\begin{aligned} a_1 + \dots + a_k &\geq a_1 + \dots + a_{k-1} + \frac{OPT - (a_1 + a_2 + \dots + a_{k-1})}{\rho W} \quad (\text{wegen (6.1)}) \\ &= (a_1 + \dots + a_{k-1}) \left(1 - \frac{1}{\rho W}\right) + \frac{OPT}{\rho W} \\ &\geq OPT \cdot \left(1 - \left(1 - \frac{1}{\rho W}\right)^{k-1}\right) \cdot \left(1 - \frac{1}{\rho W}\right) + \frac{OPT}{\rho W} \quad (\text{Induktionsannahme}) \\ &= OPT \cdot \left(1 - \left(1 - \frac{1}{\rho W}\right)^k\right) \end{aligned}$$

Mit  $k = W$  erhalten wir also, dass der Algorithmus  $A$  mindestens

$$OPT \cdot \left(1 - \left(1 - \frac{1}{\rho W}\right)^W\right)$$

Pfade mit den  $W$  zur Verfügung stehenden Farben färbt. Da die Folge  $(1 - \frac{1}{n})^n$  bekanntermaßen mit wachsendem  $n$  von unten gegen  $e^{-1}$  konvergiert, gilt:

$$1 - \left(1 - \frac{1}{\rho W}\right)^W = 1 - \left(\left(1 - \frac{1}{\rho W}\right)^{\rho W}\right)^{1/\rho} \geq 1 - (e^{-1})^{1/\rho} = 1 - e^{-1/\rho}$$

Damit folgt, dass Algorithmus  $A$  Approximationsrate höchstens  $1/(1 - e^{-1/\rho})$  hat. Da  $e^x \geq 1 + x$  gilt, folgt  $e^{1/\rho} \geq 1 + \frac{1}{\rho} = \frac{\rho+1}{\rho}$  und damit  $e^{-1/\rho} \leq \frac{\rho}{\rho+1}$ . Damit erhalten wir

$$\frac{1}{1 - e^{-1/\rho}} \leq \frac{1}{1 - \frac{\rho}{\rho+1}} = \rho + 1$$

und haben so den Satz vollständig bewiesen.  $\square$

Der Satz 6.5 gilt auch für MaxRPC und MEDP (statt für MaxPC und MEDPwPP). Der Beweis für diese modifizierte Aussage funktioniert völlig analog.

Der Satz 6.5 besagt, dass unabhängig von der Topologie des Netzes die Approximationsrate für MaxPC (bzw. MaxRPC) schlimmstenfalls um eine kleine Konstante schlechter ist als die für MEDPwPP (bzw. MEDP). Im wesentlichen reicht es also, für jede Topologie die Probleme MEDP und MEDPwPP gut zu lösen und dann mit dem obigen Verfahren Algorithmen für MaxRPC und MaxPC zu erhalten.

Wenn der Algorithmus  $A_1$  für MEDP oder MEDPwPP sogar die optimale Lösung berechnet, also ein 1-Approximationsalgorithmus ist, so erhält man mit Satz 6.5 Approximationsrate  $1/(1 - 1/e) \approx 1.58$  für MaxRPC oder MaxPC. Dies ist zum Beispiel in Ringen der Fall, da man dort MEDP und MEDPwPP in polynomieller Zeit optimal lösen kann.

**Referenzen**

- [AAF<sup>+</sup>96] Baruch Awerbuch, Yossi Azar, Amos Fiat, Stefano Leonardi, and Adi Rosén. Online competitive algorithms for call admission in optical networks. In *Proceedings of the 4th Annual European Symposium on Algorithms ESA'96*, LNCS 1136, pages 431–444, 1996.
- [EJ98] Thomas Erlebach and Klaus Jansen. Maximizing the number of connections in optical tree networks. In *Proceedings of the 9th Annual International Symposium on Algorithms and Computation ISAAC'98*, LNCS 1533, pages 179–188, 1998.
- [EJK<sup>+</sup>99] Thomas Erlebach, Klaus Jansen, Christos Kaklamanis, Milena Mihail, and Pino Persiano. Optimal wavelength routing on directed fiber trees. *Theoretical Computer Science*, 221:119–137, 1999. Special issue of ICALP'97.
- [Kar80] Iskandar A. Karapetian. On the coloring of circular arc graphs. *Journal of the Armenian Academy of Sciences*, 70(5):306–311, 1980. (in Russian)
- [Kle96] Jon Kleinberg. *Approximation algorithms for disjoint paths problems*. PhD thesis, MIT, 1996.
- [KT95] Jon Kleinberg and Éva Tardos. Disjoint paths in densely embedded graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science FOCS'95*, pages 52–61, 1995.
- [Rab96] Yuval Rabani. Path coloring on the mesh. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science FOCS'96*, pages 400–409, 1996.
- [WW98] Gordon Wilfong and Peter Winkler. Ring routing and wavelength translation. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms SODA'98*, pages 333–341, 1998.

## 6.5 Optimaler Multicast in optischen WDM-Netzen

In diesem Kapitel wollen wir uns wieder mit dem Problem von Routing und Pfadfärbung (RPC) beschäftigen, diesmal jedoch für Mengen  $R$  von Verbindungsanfragen, die eine ganz besondere Eigenschaft haben: Wir nehmen an, dass bei allen Verbindungsanfragen in  $R$  derselbe ausgezeichnete Knoten  $s$  der Sender ist. Die Empfänger aller  $k$  Verbindungsanfragen sind verschiedene Knoten  $t_1, \dots, t_k$  des Netzes. Instanzen von Pfadfärbungsproblemen mit solchen Mengen von Verbindungsanfragen werden auch als *Multicast*-Instanzen bezeichnet. Man will wieder versuchen, den Anfragen konfliktfrei Pfade und Farben zuzuordnen, so dass die Anzahl verwendeter Farben minimiert wird.

Als Anwendung könnte man sich vorstellen, dass  $s$  ein Hochleistungs-Server ist, der an die Knoten  $t_1$  bis  $t_k$  jederzeit ohne Verzögerung grosse Datenmengen ausliefern können muss. (Dabei können an verschiedene Knoten  $t_i$  durchaus verschiedene Daten geschickt werden, diesbezüglich ist der Begriff *Multicast* hier etwas irreführend.)

Anders als in den vorhergehenden Abschnitten lassen wir jetzt ausserdem *beliebige gerichtete Graphen* zur Modellierung des optischen Netzes zu, nicht nur symmetrisch gerichtete Graphen.

Für eine gegebene Menge  $R$  von Anfragen bezeichnen wir jede Zuordnung von gerichteten Pfaden an die Anfragen in  $R$  als *Routing*. Mit  $L^*(R)$  bezeichnen wir die maximale Kantenlast bei demjenigen Routing für die Anfragen aus  $R$ , das die maximale Kantenlast minimiert. Mit  $W^*(R)$  bezeichnen wir die minimale Farbanzahl, die nötig ist, um den Verbindungsanfragen in  $R$  konfliktfrei Pfade und Farben zuzuordnen. Wir werden nun einen Algorithmus für Multicast-Instanzen kennen lernen, der effizient ein Routing und eine Färbung der Pfade liefert, so dass gleichzeitig die maximale Kantenlast und die Farbanzahl minimiert werden. Der Algorithmus stammt aus [BHP98].

**Satz 6.6** *Seien ein gerichteter Graph  $G = (V, E)$  und eine Menge  $R$  von Anfragen gegeben, wobei alle Anfragen denselben Sender  $s$  haben. Es gibt einen polynomiellen Algorithmus, der den Anfragen in  $R$  konfliktfrei Pfade und Farben zuweist, so dass die maximale Kantenlast  $L^*(R)$  und die verwendete Farbanzahl  $W^*(R)$  ist. Dabei gilt immer  $L^*(R) = W^*(R)$ .*

Ein Beispiel für eine Multicast-Instanz und eine optimale Lösung mit 2 Farben ist in Abb. 6.4 dargestellt. Die Instanz besteht aus vier Anfragen, die jeweils den Knoten  $s$  mit einem der grauen Knoten verbinden. Die zwei verwendeten Farben sind durch durchgezogene bzw. gestrichelte Linien repräsentiert. Man kann leicht sehen, dass hier  $L^*(R) = W^*(R) = 2$  gilt.

Man beachte, dass die Gleichung  $L^*(R) = W^*(R)$  bei allgemeinen Instanzen nicht richtig ist. In Abb. 6.2 war beispielsweise eine Instanz mit fünf Pfaden in einem Baum dargestellt, bei der  $L^*(R) = 2$  und  $W^*(R) = 3$  galt.

In den folgenden Abschnitten werden wir nach und nach die Details des Algorithmus erarbeiten, dessen Existenz in Satz 6.6 behauptet wird. Dieser Algorithmus macht intensiven Gebrauch von einer Unteroutine zur Berechnung des maximalen Flusses von einem Knoten  $s$  zu einem Knoten  $t$  in einem Netzwerk mit Kantenkapazitäten (siehe Kapitel 1.3.3).

### 6.5.1 Minimierung der maximalen Kantenlast

In diesem Abschnitt betrachten wir zuerst das einfachere Problem, den gegebenen Anfragen Pfade zuzuordnen, so dass die maximale Kantenlast minimiert wird. Die Zuweisung von

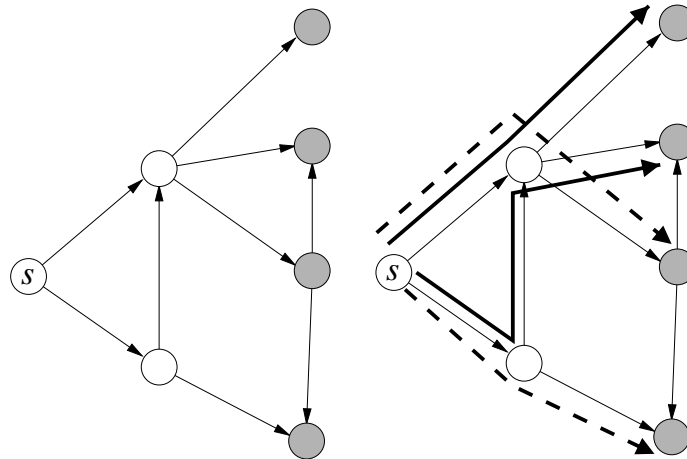


Abbildung 6.4: Multicast-Instanz (links) und optimale Lösung mit 2 Farben (rechts).

Farben ignorieren wir vorerst einfach.

Sei eine Multicast-Instanz durch einen gerichteten Graphen  $G = (V, E)$  und eine Menge  $R$  von  $k$  Anfragen gegeben, die alle den Knoten  $s \in V$  als Sender haben. Sei  $T = \{t_1, \dots, t_k\}$  die Menge der Empfänger der  $k$  Anfragen in  $R$ .

Betrachten wir zuerst das Problem, wie man für ein gegebenes  $L$  testen kann, ob ein Routing mit maximaler Kantenlast  $L$  existiert. Dabei beobachten wir, dass ein Routing vergleichbar ist mit einem Fluss, der von der Quelle  $s$  zu den Empfängern  $t_1, \dots, t_k$  der Anfragen fließt und der bei jedem Empfänger  $t_i$  genau eine Flusseinheit abliefern. Wir können den Test daher auf ein Flussproblem in einem Graphen  $G' = (V', E')$  zurückführen, den wir wie folgt aus  $G$  erhalten:

- Wir fügen einen neuen Quellknoten  $s'$  und eine Kante  $(s', s)$  mit Kapazität  $\infty$  ein.
- Wir fügen eine neue Senke  $t'$  und  $k$  Kanten  $(t_i, t')$ , für  $1 \leq i \leq k$ , jeweils mit Kapazität 1 ein.
- Die Kapazität aller Kanten von  $G$  setzen wir auf  $L$ .

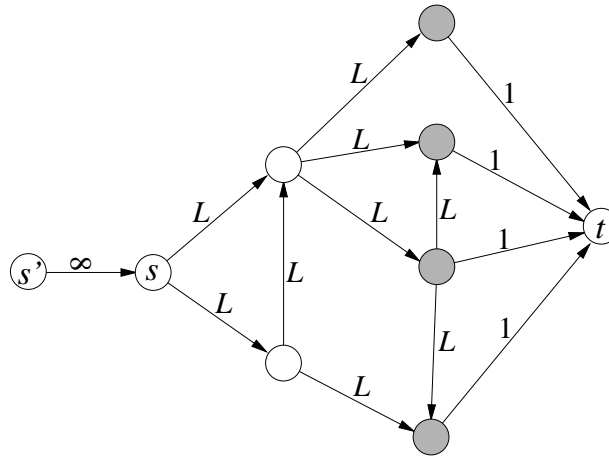
Für die Instanz aus Abb. 6.4 ist der Graph  $G'$  in Abb. 6.5 dargestellt.

Wir behaupten nun, dass das folgende Lemma gilt.

**Lemma 6.7** *Es gibt genau dann ein Routing mit maximaler Kantenlast höchstens  $L$ , wenn es im Graphen  $G'$  einen Fluss von  $s'$  nach  $t'$  mit Wert genau  $k$  gibt.*

**Beweis.** Die Korrektheit des Lemmas kann man sich wie folgt klar machen:

- Wenn es ein Routing mit Kantenlast höchstens  $L$  gibt, so kann man den Fluss auf jeder Kante von  $G$  gleich der Anzahl Pfade durch diese Kante setzen, den Fluss auf den Kanten  $(t_i, t')$  auf 1, und den Fluss auf der Kante  $(s', s)$  auf  $k$ . Man erhält auf diese Weise einen gültigen Fluss von  $s'$  nach  $t'$  mit Wert  $k$ .
- Umgekehrt kann man aus einem gültigen Fluss von  $s'$  nach  $t'$  mit Wert  $k$  (bei dem ohne Einschränkung angenommen werden darf, dass der Fluss auf jeder Kante ganzzahlig ist,

Abbildung 6.5: Flussproblem im Graphen  $G'$  für den Test, ob  $L^*(R) \leq L$ .

siehe Abschnitt 1.3.3) ein Routing mit maximaler Kantenlast höchstens  $L$  bekommen, indem man nacheinander einzelne Pfade  $p$  von  $s'$  nach  $t'$ , die nur Kanten mit positivem Fluss enthalten, berechnet, den Teilpfad von  $p$  von Knoten  $s$  bis zum letzten Knoten  $t_i$  vor  $t$  als Routing für die Anfrage von  $s$  nach  $t_i$  verwendet, entlang des Pfades  $p$  den Fluss um 1 erniedrigt, und schliesslich das Verfahren wiederholt, bis das Routing für alle Anfragen bestimmt ist.

Damit ist das Lemma bewiesen. □

Der optimale Wert  $L^*(R)$  liegt irgendwo zwischen 1 und  $k$ . Wir können einfach alle Werte von 1 bis  $k$  für  $L$  ausprobieren und gemäss Lemma 6.7 jeweils mittels eines Flussalgorithmus testen, ob es ein Routing mit Last höchstens  $L$  gibt. Das kleinste  $L$ , für das wir ein Routing finden, muss dann gleich  $L^*(R)$  sein. Da es sehr effiziente polynomielle Algorithmen für das Flussproblem gibt, ist die Gesamtlaufzeit ebenfalls polynomiell. Um Rechenzeit zu sparen, können wir die lineare Suche im Bereich  $[1, k]$  nach  $L^*(R)$  auch durch eine Binärsuche ersetzen und benötigen dann nur  $O(\log k)$  Aufrufe des Flussalgorithmus.

### Eine Schnittbedingung

Im nächsten Abschnitt benötigen wir dann noch eine Aussage über die Kapazität von Schnitten in  $G'$ , die wir hier noch herleiten wollen. In Kapitel 1.3.3 haben wir bereits das berühmte Max-Flow-Min-Cut-Theorem kennen gelernt, das wir hier noch einmal wiederholen:

**Satz 6.8 (Ford und Fulkerson, 1956)** *Der maximale Wert eines Flusses in  $G$  ist gleich der minimalen Kapazität eines Schnittes in  $G$ .*

Da der maximale Wert eines Flusses in  $G'$  für  $L = L^*(R)$  gleich  $k$  ist, folgt also, dass jeder Schnitt in  $G'$  Kapazität mindestens  $k$  haben muss.

Für disjunkte Knotenmengen  $A$  und  $B$  bezeichnen wir im Folgenden mit  $m(A, B)$  immer die Anzahl der Kanten, die im aktuell betrachteten Graphen von  $A$  nach  $B$  laufen.

Sei durch  $S \subseteq V$  und  $\bar{S} = V \setminus S$  eine Unterteilung von  $V$  in zwei disjunkte Teilmengen gegeben. Betrachten wir den Schnitt  $C = (\{s'\} \cup S, \{t'\} \cup \bar{S})$  in  $G'$ . Wir beobachten:

- Falls  $s \in \bar{S}$ , so ist die Kapazität von  $C$  gleich  $\infty$ , da die Kante  $(s', s)$  im Schnitt liegt.
- Andernfalls ist die Kapazität von  $C$  gleich  $|S \cap T| + L \cdot m(S, \bar{S})$ .

Da jeder Schnitt in  $G'$  (für  $L = L^*(R)$ ) Kapazität mindestens  $k$  haben muss, folgt das folgende Lemma.

**Lemma 6.9** *Für alle  $S \subseteq V$  mit  $s \in S$  gilt  $|S \cap T| + L^*(R) \cdot m(S, \bar{S}) \geq k$ .*

## 6.5.2 Minimierung der Farbanzahl

Nun sind wir bereit, zusätzlich zum Routing auch das Färbungsproblem zu betrachten. Wir werden auch dieses Problem auf ein Flussproblem in einem Graphen  $G''$  zurückführen. Die Grundidee dabei ist, dass wir für jede Farbe  $j$  eine Kopie  $G_j$  des Graphen  $G$  erzeugen, so dass der Fluss in  $G_j$  gerade den Pfaden entspricht, die die Farbe  $j$  zugeordnet bekommen.

Nehmen wir an, dass wir testen wollen, ob es eine konfliktfreie Zuordnung von Pfaden und Farben mit höchstens  $W$  Farben gibt. Der Graph  $G'' = (V'', E'')$  wird wie folgt konstruiert.

- Wir beginnen mit  $W$  disjunkten Kopien des Graphen  $G = (V, E)$ , wobei die  $j$ -te Kopie als  $G_j = (V_j, E_j)$  bezeichnet wird. Für einen Knoten  $v \in V$  bezeichnen wir mit  $v^{(j)}$  die Kopie von  $v$  in  $G_j$ . Die Kopien der Knoten von  $T$  in  $G_j$  werden mit  $T_j$  bezeichnet. Alle Kanten in den Kopien  $G_j$  erhalten Kapazität 1.
- Wir fügen einen neuen Knoten  $s''$  ein und die Kanten  $(s'', s^{(j)})$  für  $1 \leq j \leq W$ , jeweils mit Kapazität  $\infty$ .

Diese Kanten erlauben, dass die Pfade, die bei  $s''$  starten, sich eine beliebige Farbe  $j$  "aussuchen" können und dann effektiv in  $G_j$  bei  $s^{(j)}$  beginnen.

- Wir fügen  $k$  neue Knoten  $t'_i$ ,  $1 \leq i \leq k$ , ein und für jeden solchen Knoten  $t'_i$  die  $W$  Kanten  $(t_i^{(j)}, t'_i)$  mit Kapazität jeweils 1.

Die neuen Knoten  $t'_i$  "sammeln" die Pfade aus allen  $G_j$  auf, die jeweils bei  $t_i^{(j)}$  ankommen. Dies ist nötig, da vorab nicht klar ist, welche der  $W$  Farben der Pfad für die Anfrage mit Empfänger  $t_i$  letztendlich bekommen soll. In der Lösung soll dann natürlich in genau einer der  $W$  Kopien von  $G$  tatsächlich ein Pfad zu  $t_i^{(j)}$  führen.

- Wir fügen einen neuen Knoten  $t''$  und die  $k$  Kanten  $(t'_i, t'')$  für  $1 \leq i \leq k$  ein, jeweils mit Kapazität 1.

Diese Kanten sorgen dafür, dass der ganze Fluss am Ende zu  $t''$  laufen kann, wobei von jedem  $t'_i$  aber nur genau eine Flusseinheit kommen darf (da in der Lösung ja nur ein Pfad für die Anfrage mit Empfänger  $t_i$  enthalten sein darf).

Für die Instanz aus Abb. 6.4 ist der Graph  $G''$  für  $W = 2$  in Abb. 6.6 dargestellt, wobei alle nicht beschrifteten Kanten Kapazität 1 haben.

Wir behaupten nun, dass das folgende Lemma gilt.

**Lemma 6.10** *Es gibt genau dann eine konfliktfreie Zuordnung von Pfaden und höchstens  $W$  Farben an die Anfragen in einer Multicast-Instanz, wenn es im für  $W$  Farben konstruierten Graphen  $G''$  einen Fluss von  $s''$  nach  $t''$  mit Wert genau  $k$  gibt.*

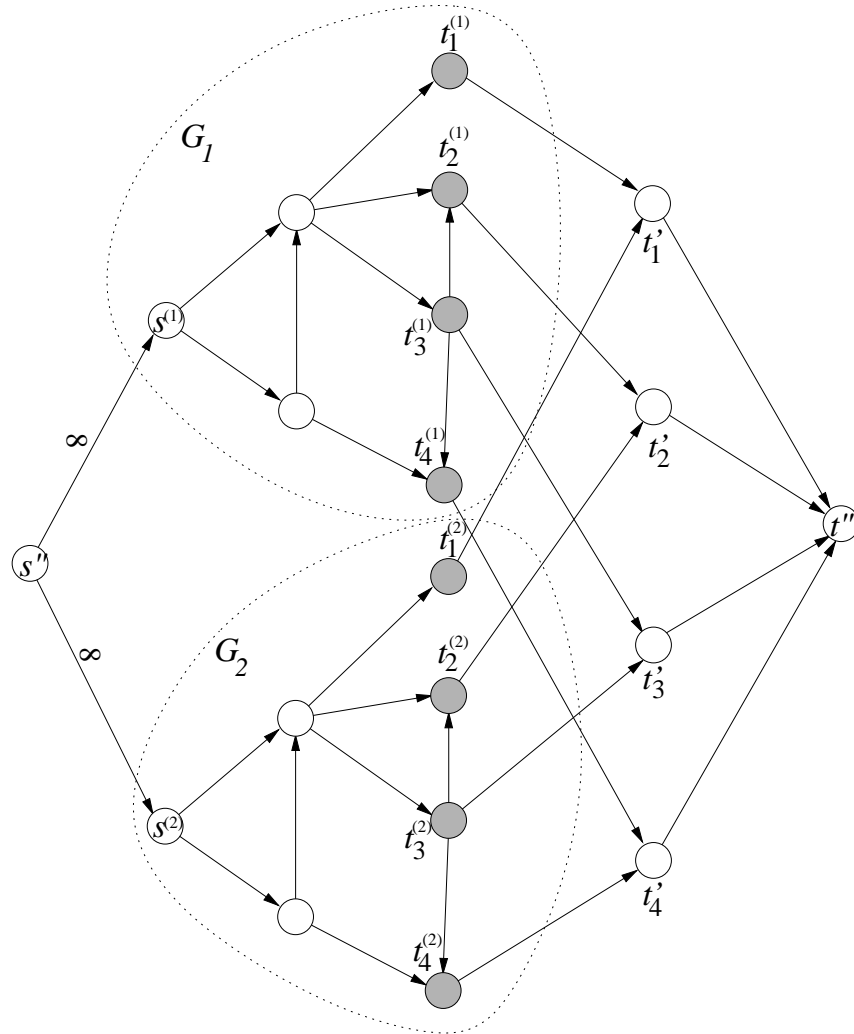
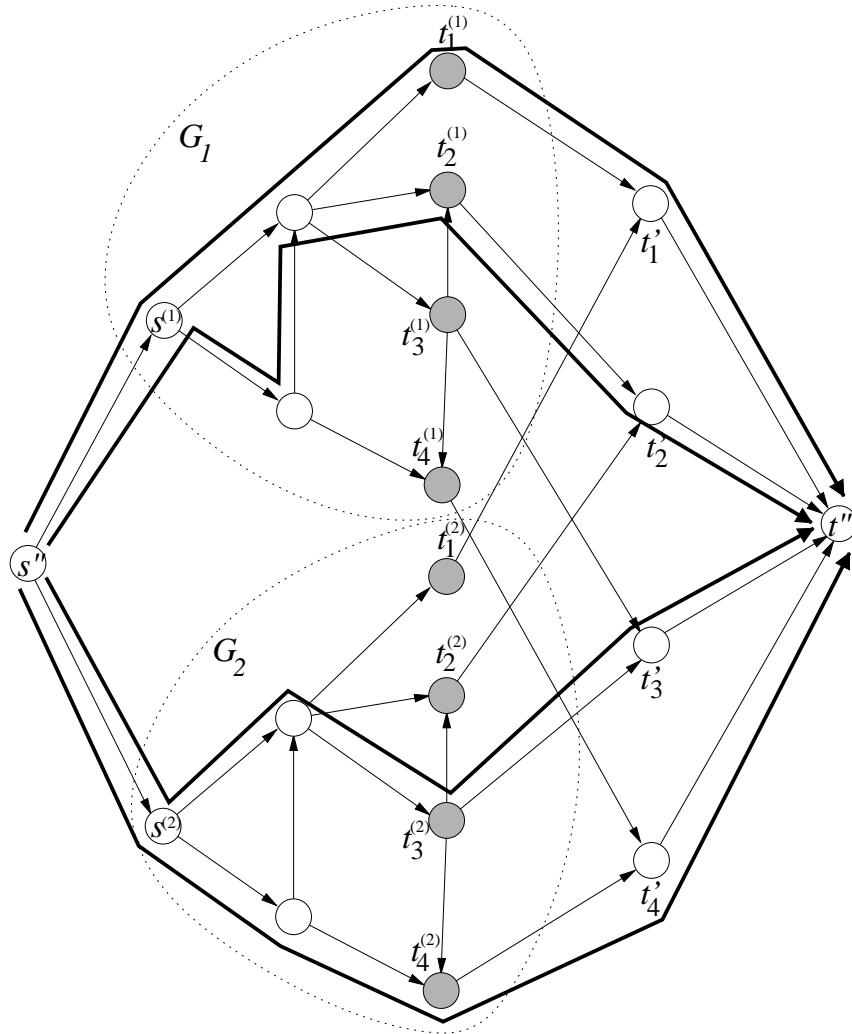


Abbildung 6.6: Flussproblem im Graphen  $G''$  für den Test, ob  $W^*(R) \leq W$ .

**Beweis.** Der Beweis der Korrektheit dieses Lemmas ist ähnlich zu dem von Lemma 6.7:

- Nehmen wir an, es gibt eine konfliktfreie Zuordnung von Pfaden und höchstens  $W$  Farben an die Anfragen. Wir setzen den Fluss auf allen Kanten in  $G''$  anfangs auf 0. Für jeden Pfad  $p$  mit Farbe  $j$ , der einer Anfrage mit Sender  $s$  und Empfänger  $t_i$  zugeordnet ist, erhöhen wir den Fluss auf der Kante  $(s'', s^{(j)})$ , auf dem  $p$  entsprechenden Pfad in  $G_j$  und auf den Kanten  $(t_i^{(j)}, t_i')$  und  $(t_i', t'')$  um Eins. Man erhält auf diese Weise einen gültigen Fluss von  $s''$  nach  $t''$  mit Wert  $k$ .
- Umgekehrt kann man aus einem gültigen Fluss von  $s''$  nach  $t''$  mit Wert  $k$  (bei dem wieder ohne Einschränkung angenommen werden darf, dass der Fluss auf jeder Kante ganzzahlig ist) folgendermassen eine konfliktfreie Zuordnung von Pfaden und höchstens  $W$  Farben zu den Anfragen in  $R$ . Man berechnet nacheinander einzelne Pfade  $p$  von  $s''$  nach  $t''$ , die nur Kanten mit positivem Fluss enthalten. Von jedem solchen Pfad  $p$  streicht man die erste Kante  $(s'', s^{(j)})$  und die letzten beiden Kanten  $(t_i^{(j)}, t_i')$  und  $(t_i', t'')$ .

Abbildung 6.7: Fluss mit Wert 4 im Graphen  $G''$  für  $W = 2$ .

Man erhält so einen Pfad  $p^j$  von  $s^{(j)}$  nach  $t_i^{(j)}$  in  $G_j$ . Schliesslich ordnet man der Anfrage von  $s$  nach  $t_i$  den  $p^j$  entsprechenden Pfad von  $s$  nach  $t_i$  in  $G$  und die Farbe  $j$  zu. Nun erniedrigen wir den Fluss auf allen Kanten von  $p$  um 1 und wiederholen das Verfahren, bis ein Fluss mit Wert 0 übrig bleibt.

Da bei einem gültigen Fluss mit Wert  $k$  für jeden Knoten  $t_i'$  genau eine eingehende Kante ( $t_i^{(j)}$ ) Fluss 1 haben muss, finden wir mit dem angegebenen Verfahren genau einen Pfad für jede der gegebenen Verbindungsanfragen.

Damit ist das Lemma bewiesen. □

Zur Veranschaulichung des Lemmas ist in Abb. 6.7 ein Fluss mit Wert 4 in dem Graphen  $G''$  für  $W = 2$  aus Abb. 6.6 dargestellt. Man sieht, dass der Fluss in  $G''$  genau den gefärbten Pfaden der optimalen Lösung für die Instanz in Abb. 6.4 entspricht.

Der optimale Wert  $W^*(R)$  liegt zwischen 1 und  $k$ . Analog zu Abschnitt 6.5.1 kann man daher  $W^*(R)$  und das zugehörige Routing und die Farbzuteilung berechnen, indem man

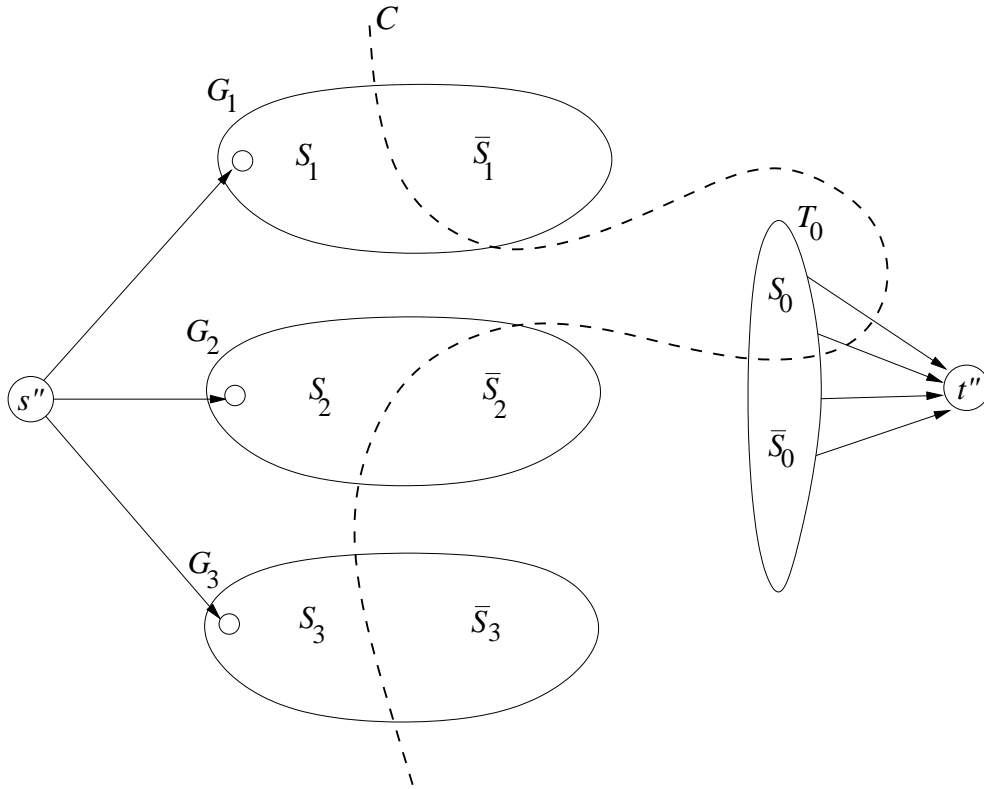


Abbildung 6.8: Skizze zum Beweis von Lemma 6.11

$O(\log k)$  Aufrufe eines Flussalgorithmus durchführt.

### 6.5.3 Gleichheit von $L^*(R)$ und $W^*(R)$

In den vorigen Abschnitten haben wir gesehen, dass man sowohl ein Routing mit maximaler Kantenlast  $L^*(R)$  als auch ein Routing und die zugehörige Pfadfärbung mit minimaler Farbanzahl  $W^*(R)$  effizient mit Hilfe eines Flussalgorithmus berechnen kann. Daraus folgt aber noch nicht, dass  $L^*(R) = W^*(R)$  gilt. Dies wollen wir daher im Folgenden noch beweisen.

Dazu müssen wir nur zeigen, dass es in dem Netzwerk  $G''$ , das wir für  $W = L^*(R)$  konstruieren, tatsächlich einen Fluss mit Wert  $k$  von  $s''$  nach  $t''$  gibt. Mit Hilfe des Max-Flow-Min-Cut-Theorems folgt das, wenn wir zeigen können, dass jeder Schnitt in diesem Netzwerk  $G''$  Kapazität mindestens  $k$  hat.

**Lemma 6.11** *Betrachte das Netzwerk  $G'' = (V'', E'')$ , das für  $W = L^*(R)$  konstruiert wird. Sei  $S \subseteq (V'' \setminus \{s'', t''\})$  und  $\bar{S} = V'' \setminus (S \cup \{s'', t''\})$  eine beliebige Partition von  $V'' \setminus \{s'', t''\}$ . Dann hat der Schnitt  $C = (S \cup \{s''\}, \bar{S} \cup \{t''\})$  Kapazität mindestens  $k$ .*

**Beweis.** Sei  $S_j = S \cap V_j$  und  $\bar{S}_j = \bar{S} \cap V_j$ , für  $1 \leq j \leq W$ . Wir setzen  $T_0 = \{t'_1, t'_2, \dots, t'_k\}$  und definieren  $S_0 = S \cap T_0$  und  $\bar{S}_0 = \bar{S} \cap T_0$ . Abbildung 6.8 zeigt eine Skizze zur Veranschaulichung dieser Bezeichnungen und der folgenden Argumente.

Falls für irgendein  $j$  der Knoten  $s^{(j)}$  nicht in  $S_j$  enthalten ist, so hat der Schnitt  $C$  Kapazität  $\infty$ . Im Folgenden nehmen wir daher an, dass für alle  $j$ ,  $1 \leq j \leq W$ , der Knoten  $s^{(j)}$  in  $S_j$  enthalten ist.

Die Kapazität  $c(C)$  des Schnittes  $C$  lässt sich nun durch folgende Formel angeben:

$$c(C) = \sum_{j=1}^W m(S_j, \bar{S}_j) + \sum_{j=1}^W m(S_j, \bar{S}_0) + |S_0|$$

Da wir  $W = L^*(R)$  gesetzt haben, folgt aus Lemma 6.9, dass für alle  $j$ ,  $1 \leq j \leq W$ , die Ungleichung  $|S_j \cap T_j| + W \cdot m(S_j, \bar{S}_j) \geq k$  gilt. Diese Ungleichung teilen wir durch  $W$  und erhalten:

$$\frac{1}{W}|S_j \cap T_j| + m(S_j, \bar{S}_j) \geq \frac{k}{W} \quad (6.3)$$

Nun können wir  $c(C)$  wie folgt abschätzen:

$$\begin{aligned} c(C) &= \sum_{j=1}^W m(S_j, \bar{S}_j) + \sum_{j=1}^W m(S_j, \bar{S}_0) + |S_0| \\ &= \sum_{j=1}^W \left( m(S_j, \bar{S}_j) + m(S_j, \bar{S}_0) + \frac{1}{W}|S_0| \right) \\ &\quad (\text{wir verwenden nun, dass } m(S_j, S_0) \leq |S_0| \text{ gilt}) \\ &\geq \sum_{j=1}^W \left( m(S_j, \bar{S}_j) + m(S_j, \bar{S}_0) + \frac{1}{W}m(S_j, S_0) \right) \\ &\geq \sum_{j=1}^W \left( m(S_j, \bar{S}_j) + \frac{1}{W}m(S_j, T_0) \right) \\ &= \sum_{j=1}^W \left( m(S_j, \bar{S}_j) + \frac{1}{W}|S_j \cap T_j| \right) \\ &\quad (\text{nun setzen wir Ungleichung (6.3) ein}) \\ &\geq \sum_{j=1}^W \frac{k}{W} = k \end{aligned}$$

Damit erhalten wir  $c(C) \geq k$  und haben das Lemma bewiesen.  $\square$

Lemma 6.11 zeigt, dass jeder Schnitt in  $G''$  für  $W = L^*(R)$  Kapazität mindestens  $k$  hat. Daher gibt es in  $G''$  einen Fluss von  $s''$  nach  $t''$  mit Wert  $k$ , und nach Lemma 6.10 gibt es dann eine konfliktfreie Zuordnung von Pfaden und höchstens  $L^*(R)$  Farben an die Anfragen in  $R$ . Da immer  $W^*(R) \geq L^*(R)$  gilt, erhalten wir somit, dass für Multicast-Instanzen in beliebigen gerichteten Graphen immer  $W^*(R) = L^*(R)$  gilt. Damit haben wir alle Behauptungen von Satz 6.6 vollständig bewiesen.

## Referenzen

[BHP98] Bruno Beauquier, Pavol Hell, and Stéphane Pérennes. Optimal wavelength-routed multicasting. *Discrete Applied Mathematics*, 84:15–20, 1998.

## Kapitel 7

# Ausfallsicherheit von Netzen

Ein wichtiges Ziel beim Entwurf der Topologie von Kommunikationsnetzen ist Ausfallsicherheit. Man möchte vermeiden, dass der Ausfall eines Netzknotens oder einer Verbindung (z.B. durch den Bruch eines Glasfaserkabels) zur Folge hat, dass Teile des Netzes von anderen Teilen abgeschnitten werden und nicht mehr miteinander kommunizieren können. In diesem Kapitel werden wir sehen, wie man solche Anforderungen graphentheoretisch formalisieren kann und wie man einige sich daraus ergebende Probleme algorithmisch sehr effizient lösen kann.

### 7.1 Der Knoten- und Kantenzusammenhang von Graphen

Zwei Eigenschaften eines Graphen, die mit der Ausfallsicherheit des entsprechenden Kommunikationsnetzes im obigen Sinn zu tun haben, sind *Knotenzusammenhang* und *Kantenzusammenhang*. Wir betrachten im Folgenden diese Eigenschaften nur für ungerichtete Graphen. Es sei aber angemerkt, dass diese Konzepte auch auf gerichtete Graphen angewendet werden können.

**Definition 7.1 (Kantendisjunkte Pfade)** Sei  $G = (V, E)$  ein ungerichteter Graph und seien  $x$  und  $y$  zwei verschiedene Knoten von  $G$ . Zwei Pfade  $p_1$  und  $p_2$  von  $x$  nach  $y$  heißen kantendisjunkt, falls sie keine Kante gemeinsam haben.

**Definition 7.2 (Kantenzusammenhang)** Ein ungerichteter Graph  $G = (V, E)$  heißt  $k$ -fach kantenzusammenhängend (engl. *k-edge-connected*), falls es für jedes Paar  $(x, y)$  verschiedener Knoten von  $G$  mindestens  $k$  kantendisjunkte Pfade von  $x$  nach  $y$  gibt. Der Kantenzusammenhang von  $G$  ist das maximale  $k$ , für das  $G$   $k$ -fach kantenzusammenhängend ist.

**Definition 7.3 (Knotendisjunkte Pfade)** Sei  $G = (V, E)$  ein ungerichteter Graph und seien  $x$  und  $y$  zwei verschiedene Knoten von  $G$ . Zwei Pfade  $p_1$  und  $p_2$  von  $x$  nach  $y$  heißen knotendisjunkt, falls sie ausser dem Startknoten  $x$  und dem Endknoten  $y$  keinen Knoten gemeinsam haben.

**Definition 7.4 (Knotenzusammenhang)** Ein ungerichteter Graph  $G = (V, E)$  heißt  $k$ -fach knotenzusammenhängend (engl. *k-connected*), falls es für jedes Paar  $(x, y)$  verschiedener Knoten von  $G$  mindestens  $k$  knotendisjunkte Pfade von  $x$  nach  $y$  gibt. Der Knotenzusammenhang von  $G$  ist das maximale  $k$ , für das  $G$   $k$ -fach knotenzusammenhängend ist.

Man beachte, dass Knotenzusammenhang und Kantenzusammenhang für  $k = 1$  äquivalent sind und dass die 1-fach knoten- bzw. kantenzusammenhängenden Graphen genau die Graphen sind, die wir bisher einfach als *zusammenhängend* bezeichnet haben.

Graphen, die 2-fach knotenzusammenhängend sind, werden im Englischen als *biconnected graphs* bezeichnet. Der vollständige Graph mit  $n$  Knoten ist  $(n - 1)$ -fach knoten- und kantenzusammenhängend.

Bezüglich Ausfallsicherheit von Netzen interessiert uns, ob ein Graph noch zusammenhängend ist, wenn eine Teilmenge der Knoten oder der Kanten ausfällt. Betrachten wir hierzu die folgenden Definitionen.

**Definition 7.5 (Trennende Knotenmenge)** Für zwei Knoten  $s$  und  $t$  in einem ungerichteten Graphen  $G = (V, E)$  heisst eine Menge  $T \subseteq (V \setminus \{s, t\})$  eine  $s$ - $t$ -trennende Knotenmenge, falls in  $G \setminus T$  kein Pfad von  $s$  nach  $t$  existiert. Eine Menge  $T \subseteq V$  heisst trennende Knotenmenge, falls  $T$  für zwei Knoten  $s$  und  $t$  eine  $s$ - $t$ -trennende Knotenmenge ist.

**Definition 7.6 (Trennende Kantenmenge)** Für zwei Knoten  $s$  und  $t$  in einem ungerichteten Graphen  $G = (V, E)$  heisst eine Menge  $F \subseteq E$  eine  $s$ - $t$ -trennende Kantenmenge, falls in  $G \setminus F$  kein Pfad von  $s$  nach  $t$  existiert. Eine Menge  $F \subseteq E$  heisst trennende Kantenmenge, falls  $F$  für zwei Knoten  $s$  und  $t$  eine  $s$ - $t$ -trennende Kantenmenge ist.

Die folgende Beziehung zwischen der Grösse einer  $s$ - $t$ -trennenden Knoten- bzw. Kantenmenge und der Anzahl knoten- bzw. kantendisjunkter Pfade von  $s$  nach  $t$  ist ein klassisches Ergebnis der Graphentheorie.

**Satz 7.7 (Menger, 1927)**

**Knotenversion:** Für zwei nicht benachbarte Knoten  $s$  und  $t$  in einem ungerichteten Graphen  $G$  ist die maximale Anzahl knotendisjunkter Pfade von  $s$  nach  $t$  gleich der minimalen Grösse einer  $s$ - $t$ -trennenden Knotenmenge.

**Kantenversion:** Für zwei Knoten  $s$  und  $t$  in einem ungerichteten Graphen  $G$  ist die maximale Anzahl kantendisjunkter Pfade von  $s$  nach  $t$  gleich der minimalen Grösse einer  $s$ - $t$ -trennenden Kantenmenge.

Der Satz von Menger kann als Vorläufer des Max-Flow-Min-Cut-Theorems von Ford und Fulkerson betrachtet werden, mit dessen Hilfe er auch leicht bewiesen werden kann. Aus dem Satz von Menger lassen sich die folgenden Aussagen ableiten.

**Satz 7.8 (Whitney, 1932)**

- (i) Ein ungerichteter Graph  $G$  mit mindestens  $k + 1$  Knoten ist genau dann  $k$ -fach knotenzusammenhängend, wenn jede trennende Knotenmenge in  $G$  mindestens  $k$  Knoten enthält.
- (ii) Ein ungerichteter Graph  $G$  ist genau dann  $k$ -fach kantenzusammenhängend, wenn jede trennende Kantenmenge in  $G$  mindestens  $k$  Kanten enthält.

Wenn also ein Kommunikationsnetz nach dem Ausfall von  $r$  beliebigen Knoten noch zusammenhängend sein soll, so muss die Topologie ein  $(r + 1)$ -fach knotenzusammenhängender Graph sein. Wenn das Netz nach dem Ausfall von  $s$  beliebigen Kanten noch zusammenhängend sein soll, so muss die Topologie ein  $(s + 1)$ -fach kantenzusammenhängender Graph sein.

Aus algorithmischer Sicht sind daher die folgenden Arten von Problemen interessant:

- Für einen gegebenen Graphen und eine Zahl  $k$  testen, ob der Graph  $k$ -fach knotenzusammenhängend oder  $k$ -fach kantenzusammenhängend ist. Dieses Problem ist polynomiell lösbar.
- Für einen gegebenen Graphen die grösste Zahl  $k$  bestimmen, so dass der Graph  $k$ -fach knoten- bzw. kantenzusammenhängend ist. Dieses Problem ist ebenfalls polynomiell lösbar.
- Einen gegebenen Graphen, der nicht  $k$ -fach knoten- bzw. kantenzusammenhängend ist, durch Einfügen minimal vieler zusätzlicher Kanten  $k$ -fach knoten- bzw. kantenzusammenhängend machen (engl. *connectivity augmentation*). Dieses Problem ist im Fall von Kantenzusammenhang für beliebiges  $k$  polynomiell lösbar. Im Fall von Knotenzusammenhang sind bisher nur für kleine Werte von  $k$  ( $k \leq 4$ ) polynomielle Algorithmen bekannt. Die gewichtete Variante, bei der jede potentielle Kante ein bestimmtes Gewicht hat und man das Gesamtgewicht der eingefügten Kanten minimieren will, ist schon für  $k = 2$   $\mathcal{NP}$ -schwer, sowohl für Knotenzusammenhang als auch für Kantenzusammenhang. Für  $k = 1$  ist das gewichtete Problem dagegen gleichwertig zur Berechnung eines minimalen Spannbaums (Kapitel 2.1) und daher polynomiell lösbar.

In den folgenden Abschnitten wollen wir effiziente Algorithmen für einige dieser Probleme kennenlernen, wobei wir vor allem den 2-fachen Knotenzusammenhang genauer betrachten werden.

## 7.2 Die Blockstruktur von Graphen

Im Folgenden sei  $G = (V, E)$  immer ein ungerichteter Graph mit mindestens 3 Knoten. Wir interessieren uns in diesem Abschnitt dafür, alle maximal grossen Teilgraphen von  $G$  zu bestimmen, die 2-fach knotenzusammenhängend sind. Früher haben wir ja bereits Zusammenhangskomponenten eines Graphen betrachtet, d.h. die maximal grossen Teilgraphen, die 1-fach knotenzusammenhängend sind.

**Definition 7.9** *Ein Knoten  $a$  in  $G$  heisst Artikulationsknoten, wenn  $\{a\}$  eine trennende Knotenmenge in einer Zusammenhangskomponente von  $G$  ist.*

Offensichtlich ist ein Graph  $G$  mit mindestens 3 Knoten genau dann 2-fach knotenzusammenhängend, wenn er zusammenhängend ist und keinen Artikulationsknoten enthält. Ausserdem gilt auch, dass ein Graph  $G$  mit mindestens 3 Knoten genau dann 2-fach knotenzusammenhängend ist, wenn  $G$  keine isolierten Knoten enthält und jedes Paar von Kanten auf einem gemeinsamen einfachen Kreis liegt.

Für  $e_1, e_2 \in E$  definieren wir  $e_1 \equiv e_2$ , falls  $e_1$  und  $e_2$  auf einem gemeinsamen einfachen Kreis liegen. Man kann leicht zeigen, dass durch  $\equiv$  eine Äquivalenzrelation erzeugt wird, d.h.  $E$  wird durch  $\equiv$  in disjunkte Teilmengen  $E_1, E_2, \dots, E_h$  für ein geeignetes  $h$  partitioniert, wobei für  $e, f \in E_i$  immer  $e \equiv f$  gilt. Sei  $G_i = (V_i, E_i)$  für  $1 \leq i \leq h$  der von  $E_i$  induzierte Teilgraph von  $G$ . Diese Teilgraphen  $G_i$  werden *Blöcke* genannt. Die Blöcke, die mindestens zwei Kanten enthalten, sind gerade die maximalen 2-fach knotenzusammenhängenden Teilgraphen von  $G$ . In Abb. 7.1 ist das Beispiel eines Graphen  $G$  mit drei Blöcken dargestellt, wobei die Kanten der Blöcke durchgezogen, gestrichelt bzw. gepunktet gezeichnet sind.

Das folgende Lemma setzt die Blockstruktur eines Graphen mit den Artikulationsknoten in Beziehung.

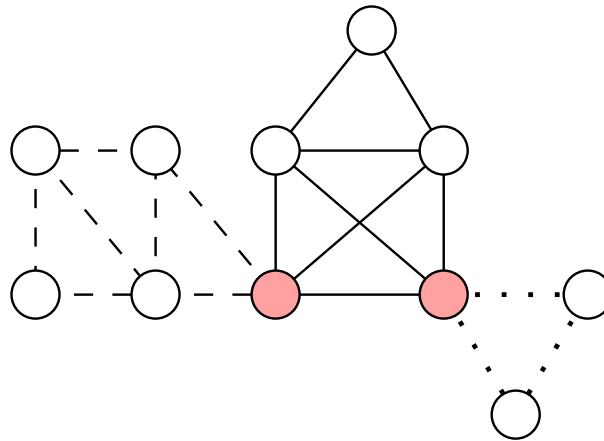


Abbildung 7.1: Beispiel für Blöcke eines Graphen.

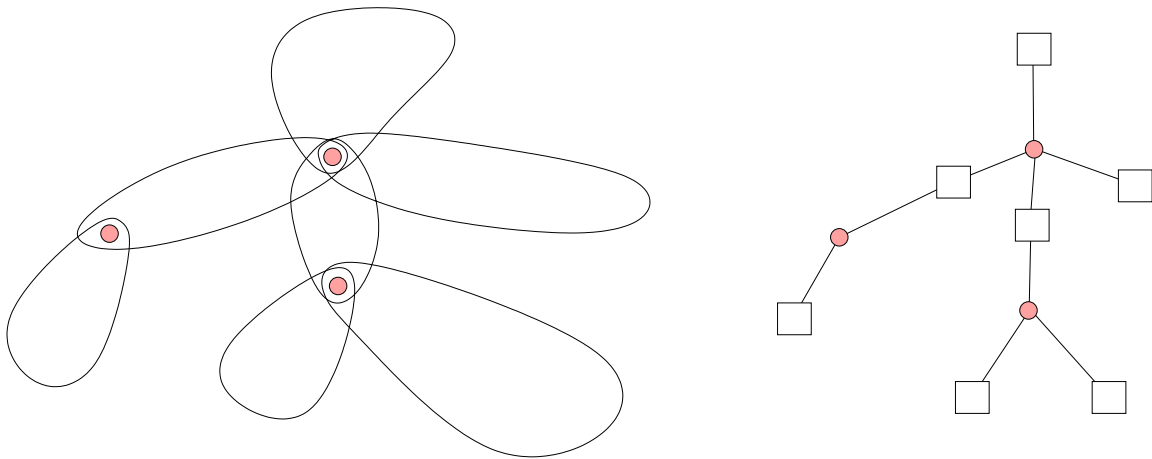


Abbildung 7.2: Die Blockstruktur eines Graphen bildet einen Baum.

**Lemma 7.10** Seien  $G_i = (V_i, E_i)$  für  $1 \leq i \leq h$  die Blöcke von  $G$ .

- (a) Für  $i \neq j$  gilt  $|V_i \cap V_j| \leq 1$ .
- (b) Ein Knoten  $a \in V$  ist genau dann ein Artikulationsknoten, wenn  $V_i \cap V_j = \{a\}$  für geeignete  $i \neq j$  gilt.

Für den Graphen aus Abb. 7.1 kann man leicht überprüfen, dass es genau zwei Artikulationsknoten gibt, die ausserdem jeweils im Schnitt zweier Blöcke liegen.

Die Blockstruktur von  $G$  kann man durch einen neuen Graphen, den *Blockstrukturgraphen*  $B(G) = (V', E')$ , beschreiben.  $B(G)$  enthält für jeden Artikulationsknoten  $a$  von  $G$  einen Knoten  $v_a$  und für jeden Block  $b$  von  $G$  einen Knoten  $v_b$ . Zwei Knoten  $v_a$  und  $v_b$  sind in  $G'$  genau dann verbunden, wenn der Block  $b$  den Artikulationsknoten  $a$  enthält. In Abb. 7.2 ist links die Blockstruktur eines Graphen skizziert und rechts der zugehörige Blockstrukturgraph gezeichnet, wobei die Knoten für Blöcke viereckig dargestellt sind.

**Lemma 7.11** Der Blockstrukturgraph  $B(G)$  eines ungerichteten Graphen  $G$  enthält keine Kreise.

**Beweis.** Wenn  $B(G)$  einen Kreis enthalten würde, so könnte man in  $G$  einen einfachen Kreis finden, der Kanten aus verschiedenen Blöcken enthält.  $\square$

Nun haben wir genügend über die Eigenschaften der Blöcke eines Graphen gelernt, um einen effizienten Algorithmus zu ihrer Berechnung entwerfen zu können.

### 7.2.1 Berechnung der Blöcke mittels Tiefensuche

Wir werden sehen, dass man die Blöcke von  $G = (V, E)$  mittels einer modifizierten Tiefensuche (DFS, *depth-first search*) berechnen kann. Dieser Algorithmus wurde bereits 1972 von Tarjan gefunden [Tar72]. Wir nehmen hier der Einfachheit halber an, dass  $G$  zusammenhängend ist. Die Blöcke eines nicht zusammenhängenden Graphen kann man berechnen, indem man den Algorithmus auf jede Zusammenhangskomponente getrennt anwendet.

Eine Tiefensuche besucht von einem Startknoten  $s \in V$  aus alle Knoten von  $G$  nach den folgenden Regeln, wobei nebenbei die Kanten von  $G$  in *Baumkanten* (die am Ende einen Spannbaum von  $G$ , den so genannten DFS-Baum, ergeben) und *Rückwärtskanten* eingeteilt werden.

- Am Anfang ist  $s$  der aktuelle Knoten und nur  $s$  ist als *besucht* markiert.
- Falls der aktuelle Knoten  $v$  einen unbesuchten Nachbarknoten  $w$  hat, so wird die Tiefensuche bei  $w$  fortgesetzt, d.h.  $w$  wird als *besucht* markiert und ist nun der aktuelle Knoten. Die Kante  $(v, w)$  ist eine Baumkante und  $v$  der *Vater* von  $w$  im DFS-Baum.
- Falls der aktuelle Knoten  $v$  keine unbesuchten Nachbarn hat, so wird der Vater von  $v$  zum aktuellen Knoten. Falls  $v$  keinen Vater hat, so muss  $v = s$  gelten und die Tiefensuche ist beendet.

Die Tiefensuche wird als rekursive Prozedur implementiert, die beim aktuellen Knoten einmal über alle Nachbarkanten läuft und dabei für jede Kante zu einem unbesuchten Knoten einen rekursiven Aufruf macht.

Man kann während der Tiefensuche allen Knoten  $v \in V$  Präordernummern  $pre[v]$  in der Reihenfolge zuordnen, in der die Knoten von der Tiefensuche besucht werden, wobei man die Nummerierung mit  $pre[s] = 1$  beginnt. Die Kanten  $e \in E$ , die keine Baumkanten werden, betrachten wir als *Rückwärtskanten*, die von dem Knoten mit grösserer Präordernummer zu dem Knoten mit kleinerer Präordernummer gerichtet sind. (Wir betrachten Baumkanten und Rückwärtskanten als gerichtete Kanten.)

Abbildung 7.3 zeigt (a) einen ungerichteten Graphen, (b) die Einteilung der Kanten in Baumkanten (durchgezogen) und Rückwärtskanten (gestrichelt) durch eine DFS mit Startknoten  $s$ , und (c) eine gewurzelte Darstellung des sich daraus ergebenden DFS-Baumes (wobei Baumkanten als abwärts gerichtet und Rückwärtskanten als aufwärts gerichtet aufzufassen sind).

Mittels einer nur geringfügig modifizierten Tiefensuche in  $G$  kann für jeden Knoten  $v$  zusätzlich der folgende Wert ermittelt werden:

- $low[v] =$  minimale Präordernummer  $pre[w]$  eines Knotens  $w$ , der von  $v$  aus über  $\geq 0$  Baumkanten abwärts, evtl. gefolgt von einer einzigen Rückwärtskante, erreicht werden kann.

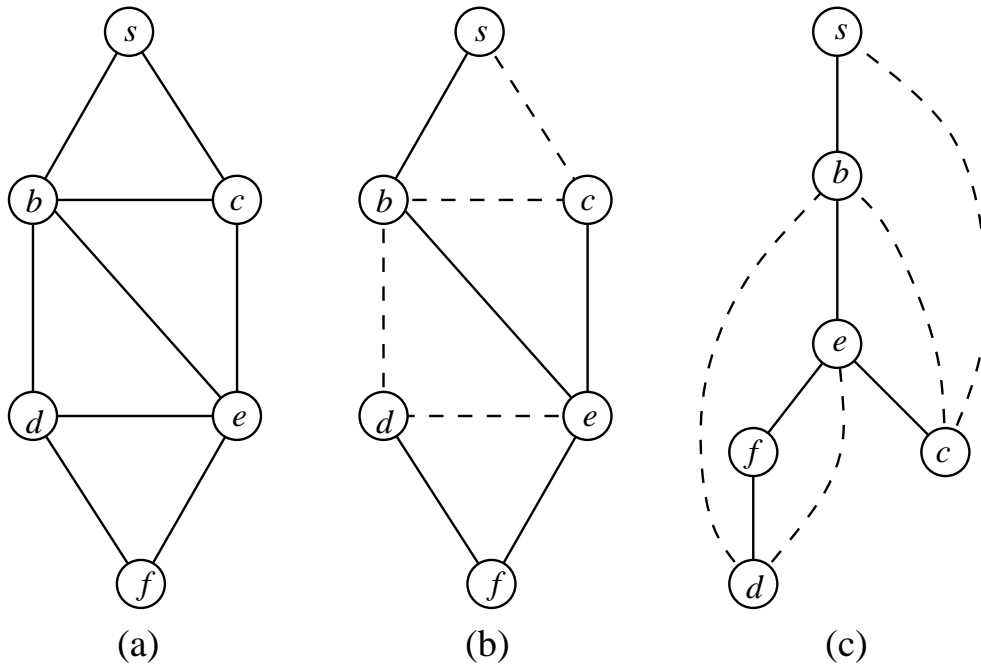


Abbildung 7.3: Tiefensuche in einem Graphen.

Die Berechnung der *low*-Werte geschieht einfach durch Ausnutzung der folgenden Tatsache:

$$low[v] = \min \left( \begin{aligned} & \{ pre[v] \} \cup \{ low[w] \mid w \text{ ist Kind von } v \text{ im DFS-Baum} \} \\ & \cup \{ pre[w] \mid \{v, w\} \text{ ist Rückwärtskante} \} \end{aligned} \right)$$

Genauer wird am Anfang der rekursiven DFS-Prozedur für einen Knoten  $v$  der Wert  $pre[v]$  bestimmt und  $low[v]$  mit  $pre[v]$  initialisiert; dann werden alle Nachbarkanten von  $v$  betrachtet: bei einer abwärtsgerichteten Baumkante zu einem Kind  $w$  erfolgt ein rekursiver Aufruf und anschließend, wenn  $low[w]$  bereits korrekt berechnet ist, wird  $low[v] := \min\{low[v], low[w]\}$  gesetzt; bei einer Rückwärtskante zu einem Knoten  $w$  setzen wir  $low[v] := \min\{low[v], pre[w]\}$ .

Das folgende Lemma sagt uns, wie wir anhand der Werte  $low[v]$  und  $pre[v]$  die Artikulationsknoten von  $G$  bestimmen können.

**Lemma 7.12** Sei  $G = (V, E)$  ein ungerichteter, zusammenhängender Graph und  $T$  ein DFS-Baum in  $G$ . Ein Knoten  $a \in V$  ist Artikulationsknoten genau dann, wenn entweder

- (a)  $a$  die Wurzel von  $T$  ist und  $\geq 2$  Kinder hat, oder
- (b)  $a$  nicht die Wurzel von  $T$  ist und es ein Kind  $b$  von  $a$  mit  $low[b] \geq pre[a]$  gibt.

**Beweis.** Sei  $s$  die Wurzel von  $T$ . Wenn  $s$  zwei Kinder hat, dann gibt es offensichtlich keine Kante in  $G$ , die einen Knoten im Teilbaum des ersten Kindes mit einem Knoten im Teilbaum eines anderen Kindes verbindet. Also sind das erste und das zweite Kind von  $s$  in verschiedenen Komponenten von  $G \setminus \{s\}$ . Folglich ist  $s$  ein Artikulationsknoten. Umgekehrt folgt aus der Annahme, dass  $s$  ein Artikulationsknoten ist, ebenfalls, dass  $s$  mindestens zwei Kinder in  $T$  hat. Damit ist (a) gezeigt.

Sei  $a$  ein Knoten ungleich  $s$  mit einem Kind  $b$ , für das  $low[b] \geq pre[a]$  gilt. Offensichtlich gehen alle Kanten in  $G$ , die einen Knoten im Teilbaum von  $b$  mit einem Knoten ausserhalb dieses Teilbaums verbinden, zum Knoten  $a$ . Also ist  $a$  ein Artikulationsknoten. Umgekehrt folgt aus der Annahme, dass  $a$  ein Artikulationsknoten ist, ebenfalls, dass  $a$  ein Kind  $b$  hat, so dass der Teilbaum von  $b$  genau alle Knoten einer Komponente von  $G \setminus \{a\}$  enthält. Für dieses Kind  $b$  muss dann  $low[b] \geq pre[a]$  gelten. Somit ist auch (b) gezeigt.  $\square$

Die Artikulationsknoten von  $G$  lassen sich also mittels einer leicht modifizierten DFS ermitteln. Um die Blöcke selbst ebenfalls während der DFS zu bestimmen, kann man so vorgehen: Es wird ein Stack  $S$  von Kanten verwaltet, der anfangs leer ist. Wenn eine Baumkante abwärts gefunden wird, so wird sie vor dem rekursiven Aufruf auf  $S$  abgelegt; wenn eine Rückwärtskante gefunden wird (von dem unteren Endknoten aus), so wird sie ebenfalls auf dem Stack abgelegt. Es ist leicht zu sehen, dass immer dann, wenn ein rekursiver Aufruf für Knoten  $w$  zu Knoten  $v$  zurückkehrt und  $low[w] \geq pre[v]$  gilt, die Kanten oben auf dem Stack bis einschliesslich der Kante  $\{v, w\}$  genau die Kanten des nächsten Blockes bilden.

Der sich ergebende Algorithmus zur Berechnung der Artikulationsknoten und Blöcke eines Graphen ist noch einmal detailliert in Abb. 7.4 dargestellt. Da für jeden Knoten und jede Kante nur konstanter Aufwand nötig ist, ist die Gesamtlaufzeit des Algorithmus  $O(|V| + |E|)$ .

**Satz 7.13** *Für einen gegebenen ungerichteten Graphen  $G = (V, E)$  kann man in Zeit  $O(|V| + |E|)$  die Artikulationsknoten und die Blöcke berechnen.*

Damit können wir also insbesondere in linearer Zeit testen, ob ein gegebener Graph 2-fach knotenzusammenhängend ist. Dies ist nämlich genau dann der Fall, wenn der Graph mindestens 3 Knoten enthält, zusammenhängend ist und aus einem einzigen Block besteht. Im nächsten Abschnitt werden wir sehen, wie wir mit Hilfe der Blockstruktur minimal viele Kanten in einen nicht 2-fach knotenzusammenhängenden Graphen einfügen können, um den Graphen 2-fach knotenzusammenhängend zu machen.

## Referenzen

- [Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

```

Algorithmus zur Berechnung der Blöcke eines Graphen
Eingabe: zusammenhängender, ungerichteter Graph  $G = (V, E)$ 
Ausgabe: Menge  $A$  der Artikulationsknoten von  $G$ , Menge  $\mathcal{B}$  der Blöcke von  $G$ 

procedure bicon(node v, node pv)
begin //  $v$  ist aktueller Knoten,  $pv$  sein Vater
    visited[ $v$ ] := true;
    pre[ $v$ ] := low[ $v$ ] := current;
    current++;
    int  $c := 0$ ; // zählt Anzahl Kinder von  $v$  im DFS-Baum
    for alle Nachbarkanten  $e = \{v, w\}$  von  $v$  do
        if visited[ $w$ ] = false then // Baumkante
            S.push( $e$ );
             $c++$ ;
            bicon( $w, v$ ); // rekursiver Aufruf
            low[ $v$ ] =  $\min(\textit{low}[v], \textit{low}[w])$ ;
            if low[ $w$ ]  $\geq$  pre[ $v$ ] and ( $v \neq s$  or  $c = 2$ ) then
                 $A := A \cup \{v\}$ ; //  $v$  ist Artikulationsknoten
            fi
            if low[ $w$ ]  $\geq$  pre[ $v$ ] then
                 $C := \emptyset$ ;
                repeat
                     $f := \textit{S.pop}()$ ;
                     $C := C \cup \{f\}$ ;
                until  $f = e$ ;
                 $\mathcal{B} := \mathcal{B} \cup \{C\}$ ; //  $C$  ist der nächste Block
            fi;
        else if pre[ $w$ ] < pre[ $v$ ] and  $w \neq pv$  then // Rückwärtskante
            S.push( $e$ );
            low[ $v$ ] =  $\min(\textit{low}[v], \textit{pre}[w])$ ;
        fi;
    od;
end;

// Hauptprogramm
for alle  $v \in V$  do visited[ $v$ ] := false; od;
 $s :=$  ein beliebiger Startknoten;
 $\textit{current} := 1$ ;  $A := \emptyset$ ;  $\mathcal{B} := \emptyset$ ;
 $S :=$  leerer Stack;
bicon( $s, s$ );
return ( $A, \mathcal{B}$ );

```

Abbildung 7.4: Algorithmus zur Berechnung der Blöcke eines Graphen in linearer Zeit mittels modifizierter Tiefensuche.

### 7.3 Erhöhung des Knotenzusammenhangs

Im vorigen Abschnitt haben wir gesehen, wie man mittels einer modifizierten Tiefensuche (DFS) die Blockstruktur eines Graphen berechnen kann. Wenn der Graph aus einem einzigen Block mit mehr als einer Kante besteht, so ist er bereits 2-fach knotenzusammenhängend. Andernfalls ist es ein interessantes Optimierungsproblem, den Graphen durch Einfügen möglichst weniger zusätzlicher Kanten 2-fach knotenzusammenhängend zu machen, um so die Ausfallsicherheit des Netzes zu erhöhen. Mit einem effizienten Algorithmus für dieses Problem, der auf Eswaran und Tarjan zurückgeht [ET76], wollen wir uns im Folgenden beschäftigen.

Problem BICONNECTIVITYAUGMENTATION

**Instanz:** ungerichteter Graph  $G = (V, E)$

**Lösung:** Kantenmenge  $E'$ , so dass  $G' = (V, E \cup E')$  2-fach knotenzusammenhängend ist

**Ziel:** minimiere  $|E'|$

Man beachte, dass der Graph  $G$  nicht unbedingt zusammenhängend sein muss.

Falls  $G$  bereits 2-fach knotenzusammenhängend ist, so können wir das mit dem Algorithmus aus dem vorigen Abschnitt in linearer Zeit feststellen, und die optimale Lösung ist die leere Kantenmenge  $E' = \emptyset$ . Daher nehmen wir an, dass  $G$  noch nicht 2-fach knotenzusammenhängend ist und mindestens 3 Knoten hat. (Ein einfacher Graph mit 2 Knoten kann nämlich nie 2-fach knotenzusammenhängend sein.)

Zuerst werden wir uns überlegen, wie wir aus der Blockstruktur von  $G$  eine gute untere Schranke für die Anzahl der einzufügenden Kanten ablesen können. Danach werden wir sehen, dass wir tatsächlich immer mit genau so vielen Kanten auskommen, wie die untere Schranke angibt.

#### 7.3.1 Untere Schranke für die Anzahl einzufügender Kanten

Wir betrachten den im vorigen Abschnitt definierten Blockstrukturgraphen  $B(G)$  von  $G$  und führen die folgenden Bezeichnungen ein:

- Für jeden Knoten  $v$  von  $G$  bezeichnen wir mit  $d(v)$  die Anzahl der Zusammenhangskomponenten von  $G \setminus \{v\}$ . Sei  $d = \max_{v \in V} d(v)$ . Wir beobachten: Falls  $G$  Artikulationsknoten enthält, so gilt  $d(a) = d$  genau für diejenigen Knoten  $a$  von  $G$ , die Artikulationsknoten sind und deren entsprechende Knoten  $v_a$  in  $B(G)$  maximalen Grad haben.
- Mit  $p$  bezeichnen wir die Anzahl von Blöcken, die genau einen Artikulationsknoten enthalten. Diese Blöcke entsprechen den Knoten mit Grad 1 im Blockstrukturgraphen, also den Blättern. Wir nennen diese Blöcke *Randblöcke*.
- Mit  $q$  bezeichnen wir die Anzahl derjenigen Zusammenhangskomponenten von  $G$ , die aus einem einzigen Knoten bestehen oder aus einem einzigen Block. Wir nennen diese Zusammenhangskomponenten *isolierte Blöcke*.

**Lemma 7.14** *Es sind mindestens  $\max\{d-1, \lceil p/2 \rceil + q\}$  Kanten nötig, um einen nicht 2-fach knotenzusammenhängenden Graphen  $G$  2-fach knotenzusammenhängend zu machen.*

**Beweis.** Sei  $v$  ein Knoten mit  $d(v) = d$ . Sei  $G'$  der 2-fach knotenzusammenhängende Graph, der aus  $G$  entsteht, wenn man die Menge  $E'$  der einzufügenden Kanten optimal wählt.  $G \setminus \{v\}$

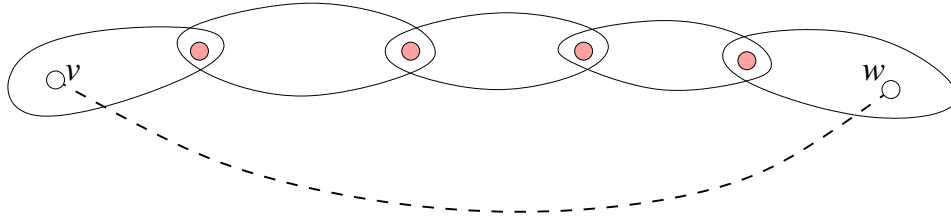


Abbildung 7.5: Der Graph  $G$  ist zusammenhängend und hat genau zwei Randblöcke. Er wird durch Einfügen der Kante  $\{v, w\}$  2-fach knotenzusammenhängend.

besteht aus  $d$  Zusammenhangskomponenten, aber  $G' \setminus \{v\}$  muss zusammenhängend sein. Um die  $d$  Komponenten in  $G'$  miteinander zu verbinden, sind mindestens  $d - 1$  Kanten nötig. Damit folgt  $|E'| \geq d - 1$ .

Jeder isolierte Block von  $G$  muss in  $G'$  über mindestens zwei neue Kanten mit anderen Komponenten verbunden sein, sonst wäre  $G'$  nicht 2-fach knotenzusammenhängend. In jedem Randblock von  $G$  muss in  $G'$  von mindestens einem Nicht-Artikulationsknoten eine neue Kante zu einem Knoten ausserhalb des Blocks vorhanden sein, sonst wäre der Randblock auch in  $G'$  noch ein Randblock. Also müssen die neuen Kanten mindestens  $2q + p$  "Enden" haben (in jedem der  $q$  isolierten Blöcke müssen mindestens zwei neue Kanten ihren Endknoten haben, in jedem der  $p$  Randblöcke mindestens eine neue Kante). Also muss  $E'$  mindestens  $(2q + p)/2$  Kanten enthalten, und da  $|E'|$  ganzzahlig ist, gilt  $|E'| \geq q + \lceil p/2 \rceil$ .  $\square$

### 7.3.2 Optimale Auswahl der einzufügenden Kanten

In diesem Abschnitt zeigen wir, dass die in Lemma 7.14 angegebene untere Schranke für die Anzahl einzufügender Kanten tatsächlich von einem Algorithmus erreicht werden kann, d.h.  $\max\{d - 1, \lceil p/2 \rceil + q\}$  neue Kanten reichen tatsächlich aus, um  $G$  2-fach knotenzusammenhängend zu machen. Der Beweis dieser Aussage gibt gleichzeitig den Algorithmus an, mit dem man die minimal vielen einzufügenden Kanten effizient berechnen kann.

**Satz 7.15** *Man kann jeden nicht 2-fach knotenzusammenhängenden Graphen  $G$  durch das Einfügen von  $\max\{d - 1, \lceil p/2 \rceil + q\}$  Kanten 2-fach knotenzusammenhängend machen.*

**Beweis.** Sei  $i = \max\{d - 1, \lceil p/2 \rceil + q\}$ . Wir beweisen die Aussage durch Induktion nach  $i$ .

Den Fall  $i = 1$  können wir leicht überprüfen. Jede Zusammenhangskomponente von  $G$  ist entweder ein isolierter Block (und trägt 1 zu  $q$  bei) oder besteht aus mehreren Blöcken (und trägt mindestens 2 zu  $p$  bei). Da  $\lceil p/2 \rceil + q \leq 1$ , muss  $G$  aus einer Zusammenhangskomponente bestehen. Da  $G$  noch nicht 2-fach knotenzusammenhängend ist, muss  $p = 2$  und  $q = 0$  gelten. Wenn wir eine Kante von einem Nicht-Artikulationsknoten  $v$  im ersten Randblock zu einem Nicht-Artikulationsknoten  $w$  im zweiten Randblock einfügen, so ist der resultierende Graph offensichtlich 2-fach knotenzusammenhängend: Jeder der bisherigen Artikulationsknoten kann nun entfernt werden, ohne dass der Graph in mehrere Komponenten zerfällt, siehe Abb. 7.5. Also ist die Behauptung für  $i = 1$  richtig.

Damit ist der Induktionsanfang erledigt. Nun nehmen wir an, dass die Behauptung für  $i$  richtig ist (d.h. im Fall  $\max\{d - 1, \lceil p/2 \rceil + q\} = i$  reichen tatsächlich  $i$  Kanten, um  $G$  2-fach knotenzusammenhängend zu machen), und beweisen unter dieser Annahme, dass sie auch für  $i + 1$  richtig ist (Induktionsschluss). Dabei können wir  $i + 1 \geq 2$  voraussetzen.

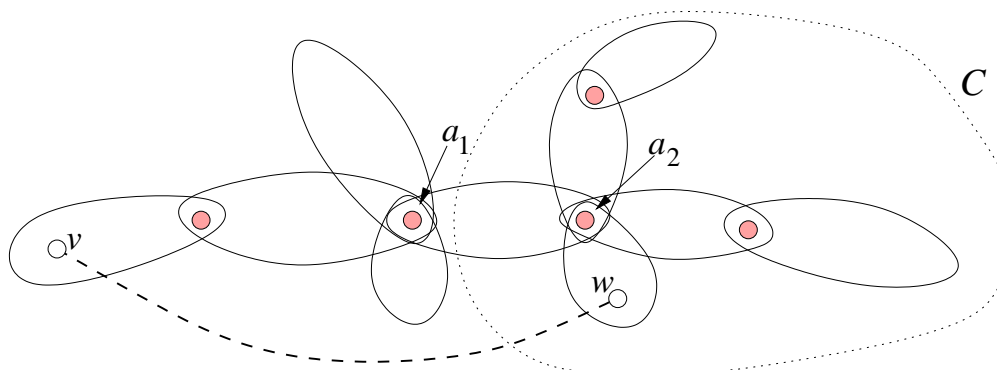


Abbildung 7.6: Der Graph  $G$  ist zusammenhängend und hat zwei Knoten  $a_1$  und  $a_2$  mit  $d(a_1) = d(a_2) = \lceil p/2 \rceil + 1$ . Es gilt  $d = 4$  und  $p = 6$ . Durch Einfügen der Kante  $\{v, w\}$  wird  $\max\{d - 1, \lceil p/2 \rceil + q\}$  um 1 kleiner.

Sei also  $G$  ein Graph, für den  $\max\{d - 1, \lceil p/2 \rceil + q\} = i + 1$  gilt. Wir machen eine Fallunterscheidung.

**Fall 1:**  $G$  ist nicht zusammenhängend.

Seien  $C_1$  und  $C_2$  zwei Komponenten von  $G$ . Jede Komponente von  $G$  ist entweder ein isolierter Block oder sie enthält einen Randblock. Wir fügen eine Kante in  $G$  ein, die einen Nicht-Artikulationsknoten in dem isolierten Block oder Randblock von  $C_1$  mit einem Nicht-Artikulationsknoten in dem isolierten Block oder Randblock von  $C_2$  verbindet. Man kann leicht nachprüfen, dass dadurch  $\lceil p/2 \rceil + q$  um genau 1 abnimmt. Ferner nimmt auch  $d$  um 1 ab, ausser in dem Fall, wo  $G$  aus lauter isolierten Blöcken besteht; in dem Fall ist aber  $d - 1 < q$  und  $\max\{d - 1, \lceil p/2 \rceil + q\}$  wird daher auch dann kleiner, wenn  $d$  gleich bleibt. Der resultierende Graph kann also nach der Induktionsannahme durch Einfügen von  $i$  Kanten 2-fach knotenzusammenhängend gemacht werden. Insgesamt reichen damit  $i + 1$  Kanten, und die Induktionsbehauptung ist gezeigt.

**Fall 2:**  $G$  ist zusammenhängend.

In diesem Fall muss  $q = 0$  gelten. Wir unterscheiden drei Unterfälle danach, ob  $d - 1$  grösser als, gleich oder kleiner als  $\lceil p/2 \rceil$  ist.

Wir beachten dabei, dass für jeden Artikulationsknoten  $a$  in jeder Komponente von  $B(G) \setminus \{v_a\}$  mindestens ein Randblock (bzw. ein in einem Randblock entsprechender Knoten in  $B(G)$ ) liegt. Daher muss für zwei verschiedene Artikulationsknoten  $a_1$  und  $a_2$  immer  $d(a_1) + d(a_2) \leq p + 2$  gelten.

**Fall 2.1:**  $d - 1 > \lceil p/2 \rceil$ . Es kann nur einen einzigen Artikulationsknoten  $a$  mit  $d(a) = d$  geben. Wir wählen zwei Randblöcke  $B_1$  und  $B_2$  von  $G$ , die nicht in derselben Komponente von  $G \setminus \{a\}$  liegen, und verbinden einen Nicht-Artikulationsknoten von  $B_1$  mit einem Nicht-Artikulationsknoten von  $B_2$ . Dadurch wird  $d(a)$  und damit auch  $d$  um 1 kleiner, da zwei Komponenten von  $G \setminus \{a\}$  nun durch die neue Kante verbunden sind. Da  $d - 1 > \lceil p/2 \rceil$  galt und  $p$  nicht grösser wird, wird auch  $\max\{d - 1, \lceil p/2 \rceil + q\}$  um 1 kleiner, und nach der Induktionsannahme reichen  $i$  weitere Kanten, um  $G$  2-fach knotenzusammenhängend zu machen.

**Fall 2.2:**  $d - 1 = \lceil p/2 \rceil$ . Es kann höchstens zwei Knoten  $a$  mit  $d(a) = d$  geben. Da  $\lceil p/2 \rceil = i + 1 \geq 2$ , muss  $p \geq 3$  gelten. Betrachten wir zuerst den Fall  $p = 3$ . Dann gilt  $d = 3$  und  $i + 1 = 2$ .  $G$  enthält genau einen Artikulationsknoten  $a$  mit  $d(a) = 3$ . Wenn wir zwei Nicht-Artikulationsknoten aus verschiedenen Randblöcken über eine Kante verbinden, so wird  $p$  zu 2 und  $d$  zu 2. Damit wird  $\max\{d - 1, \lceil p/2 \rceil + q\}$  zu 1, und nach der Induktionsannahme reicht eine weitere Kante, um  $G$  2-fach knotenzusammenhängend zu machen.

Sei nun  $p \geq 4$ . Sei  $a_1$  ein Knoten mit  $d(a_1) = d$ . Sei  $a_2$  ein zweiter Knoten mit  $d(a_2) = d$ , falls es einen solchen gibt. Der Fall, in dem  $a_2$  existiert, ist in Abb. 7.6 skizziert. Wegen  $p \geq 4$  gilt  $d = \lceil p/2 \rceil + 1 < p$ , also muss eine Komponente  $C$  von  $G \setminus \{a_1\}$  mindestens 2 der  $p$  Randblöcke enthalten. Falls  $a_2$  existiert, so muss  $a_2$  in  $C$  enthalten sein (siehe Abb. 7.6).

Sei  $B_1$  einer der Randblöcke in  $C$  und sei  $B_2$  ein Randblock, der nicht in  $C$  enthalten ist. Wir verbinden einen Nicht-Artikulationsknoten von  $B_1$  mit einem Nicht-Artikulationsknoten von  $B_2$ . Dadurch wird  $d$  um 1 kleiner und  $p$  um 2 kleiner. Also wird  $\max\{d - 1, \lceil p/2 \rceil + q\}$  um 1 kleiner, und wir können  $G$  nach Induktionsannahme mit  $i$  weiteren Kanten 2-fach knotenzusammenhängend machen.

**Fall 2.3:**  $d - 1 < \lceil p/2 \rceil$ . Es muss einen Artikulationsknoten  $a$  geben, so dass eine Komponente  $C$  von  $G \setminus \{a\}$  mindestens zwei Randblöcke enthält. Wir verfahren ähnlich zum vorigen Fall. Sei wieder  $B_1$  einer der Randblöcke in  $C$  und sei  $B_2$  ein Randblock, der nicht in  $C$  enthalten ist. Wir verbinden einen Nicht-Artikulationsknoten von  $B_1$  mit einem Nicht-Artikulationsknoten von  $B_2$ . Dadurch wird  $p$  um 2 kleiner und damit  $\max\{d - 1, \lceil p/2 \rceil + q\}$  um 1. Wir können  $G$  nach Induktionsannahme mit  $i$  weiteren Kanten 2-fach knotenzusammenhängend machen.

Damit ist der Induktionsschluss in allen Fällen gezeigt und die Behauptung des Satzes bewiesen.  $\square$

Es ist klar, dass man aus dem Beweis von Satz 7.15 leicht einen polynomiellen Algorithmus erhalten kann, der eine optimale Kantenmenge  $E'$  bestimmt. Es ist sogar möglich, diesen Algorithmus in linearer Zeit  $O(|V| + |E|)$  zu implementieren, das resultierende Programm ist jedoch sehr kompliziert [ET76].

## Referenzen

- [ET76] Kapali P. Eswaran and Robert Endre Tarjan. Augmentation problems. *SIAM J. Comput.*, 5(4):653–665, 1976.

## 7.4 Effiziente Berechnung des Kantenzusammenhangs

Nun betrachten wir das Problem, den Kantenzusammenhang eines gegebenen Graphen  $G = (V, E)$  zu bestimmen, d.h. die grösste Zahl  $k$  zu berechnen, so dass  $G$   $k$ -fach kantenzusammenhängend ist. Nach dem Satz von Whitney (Satz 7.8) wissen wir, dass  $G$  genau dann  $k$ -fach kantenzusammenhängend ist, wenn jede trennende Kantenmenge mindestens  $k$  Kanten enthält. Wenn wir also eine kleinste trennende Kantenmenge  $F^* \subseteq E$  in  $G$  berechnen, so liefert  $k = |F^*|$  genau den gesuchten Wert (d.h. den Kantenzusammenhang von  $G$ ). Im Folgenden werden wir einen Algorithmus für die effiziente Bestimmung einer kleinsten trennenden Kantenmenge kennen lernen.

Eine trennende Kantenmenge  $F$  nennen wir auch *Schnitt*, da das Entfernen von  $F$  den Graphen in mehrere Zusammenhangskomponenten “zerschneidet”. Wenn wir eine kleinste trennende Kantenmenge suchen, berechnen wir also einen *minimalen Schnitt* im gegebenen Graphen. Dabei können wir uns im Folgenden immer auf die Betrachtung von Schnitten beschränken, die genau aus den Kanten zwischen zwei disjunkten Teilmengen  $V_1, V_2 \subset V$  mit  $V_1 \cup V_2 = V$  bestehen; falls ein Schnitt zusätzliche Kanten enthält, so kann man ihn auf jeden Fall kleiner machen, indem man die zusätzlichen Kanten weglässt.

Wir hatten bereits bei der Betrachtung von Flussalgorithmen über Schnitte gesprochen, dort waren jedoch immer zwei Knoten  $s$  und  $t$  ausgezeichnet und es wurden nur solche Schnitte betrachtet, die  $s$  und  $t$  voneinander trennen. Im Unterschied dazu betrachten wir hier alle möglichen Schnitte, die den Graphen in zwei oder mehr Komponenten teilen.

Wir wollen minimale Schnitte auch in Graphen mit Kantengewichten berechnen. Dabei nehmen wir an, dass jeder Kante  $e$  von  $G$  ein Gewicht  $c(e)$  zugeordnet ist. Das Gewicht einer trennenden Kantenmenge  $F$  ist dann  $c(F) = \sum_{e \in F} c(e)$ . Für die Bestimmung des Kantenzusammenhangs eines Graphen können wir einfach das Gewicht aller Kanten auf 1 setzen, dann ist das Gewicht eines Schnittes gerade gleich der Anzahl Kanten im Schnitt.

Problem MINCUT

**Instanz:** ungerichteter Graph  $G = (V, E)$  mit Kantengewichten  $c : E \rightarrow \mathbb{R}^+$

**Lösung:** Kantenmenge  $F \subseteq E$ , so dass  $G \setminus F$  nicht zusammenhängend ist

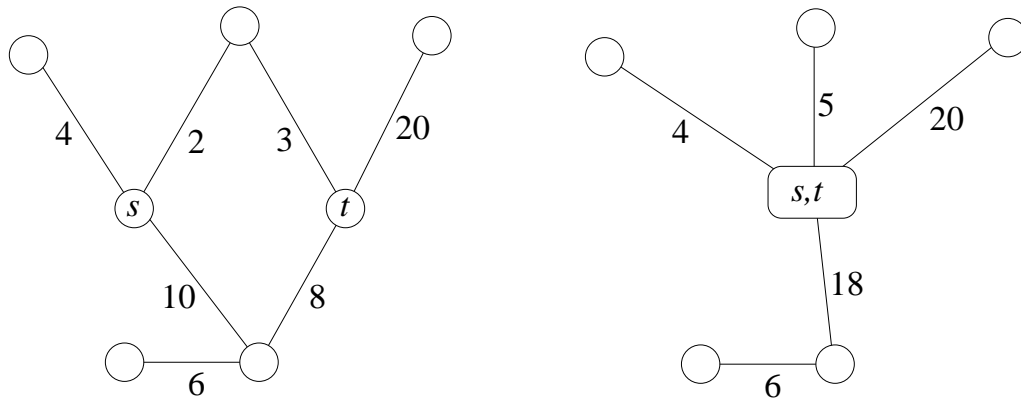
**Ziel:** minimiere  $c(F)$

Für gegebene Knoten  $s$  und  $t$  kann man mit Hilfe eines Flussalgorithmus auch leicht einen minimalen  $s$ - $t$ -Schnitt berechnen (also eine  $s$ - $t$ -trennende Kantenmenge mit minimalem Gewicht). Wenn wir nun den Knoten  $s$  beliebig wählen und dann alle  $|V| - 1$  Möglichkeiten für  $t$  durchprobieren und jeweils einen minimalen  $s$ - $t$ -Schnitt berechnen, so muss einer der  $|V| - 1$  Schnitte auch der gesuchte minimale Schnitt in  $G$  sein. Man könnte das Problem MINCUT also durch  $|V| - 1$  Aufrufe eines Flussalgorithmus lösen. Das Problem MINCUT ist somit auf jeden Fall in polynomieller Zeit optimal lösbar.

Wir betrachten einen Algorithmus, der den minimalen Schnitt in  $G$  sehr effizient ohne Verwendung eines Flussalgorithmus berechnet. Die wesentliche Grundidee des Algorithmus stammt von Nagamochi und Ibaraki [NI92], eine einfachere Darstellung und Analyse wurde von Stoer und Wagner gefunden [SW97].

### 7.4.1 Ein einfacher Min-Cut-Algorithmus

Die Grundidee des Algorithmus ist es, zuerst einen minimalen  $s$ - $t$ -Schnitt  $C_1$  für beliebige (nicht vorgegebene) Knoten  $s$  und  $t$  zu berechnen, dann  $s$  und  $t$  zu verschmelzen und in dem

Abbildung 7.7: Verschmelzung zweier Knoten  $s$  und  $t$ .

so erhaltenen kleineren Graphen rekursiv einen minimalen Schnitt  $C_2$  zu berechnen. Am Ende wird dann derjenige der beiden Schnitte  $C_1$  und  $C_2$  ausgegeben, der das kleinere Gewicht hat.

Beim Verschmelzen zweier Knoten  $s$  und  $t$  zu einem neuen Knoten  $v$  erhält  $v$  dabei die folgenden inzidenten Kanten:

- Falls  $s$  über Kante  $e_1$  und  $t$  über Kante  $e_2$  zu einem Knoten  $w \neq s, t$  adjazent war, so erhält  $v$  eine Kante zu  $w$  mit Gewicht  $c(e_1) + c(e_2)$ .
- Falls  $s$  über Kante  $e_1$  zu  $w \neq s, t$  adjazent war, aber nicht  $t$ , so erhält  $v$  eine Kante zu  $w$  mit Gewicht  $c(e_1)$ .
- Falls  $t$  über Kante  $e_1$  zu  $w \neq s, t$  adjazent war, aber nicht  $s$ , so erhält  $v$  eine Kante zu  $w$  mit Gewicht  $c(e_1)$ .

Ein Beispiel für die Verschmelzung zweier Knoten ist in Abb. 7.7 dargestellt.

Wenn der Graph  $H$  aus  $G$  durch mehrere Knotenverschmelzungen entstanden ist, so sagen wir, dass ein Knoten  $v$  von  $G$  in einem Knoten  $w$  von  $H$  *enthalten* ist, falls  $v = w$  oder falls  $w$  durch Verschmelzen zweier Knoten  $w_1$  und  $w_2$  entstanden ist, wobei  $v$  in  $w_1$  oder  $w_2$  enthalten war. Jeden Schnitt  $F$  in  $H$  können wir auch als Schnitt in  $G$  auffassen, indem wir einfach jede Kante  $e \in F$  durch die Menge der Kanten in  $G$  ersetzen, aus denen  $e$  entstanden ist.

**Lemma 7.16** *Seien  $s$  und  $t$  zwei Knoten in  $G$  und sei  $H$  der Graph, der aus  $G$  durch Verschmelzen von  $s$  und  $t$  entsteht. Sei  $C_1$  ein minimaler  $s$ - $t$ -Schnitt in  $G$  und sei  $C_2$  ein minimaler Schnitt in  $H$ . Dann ist derjenige der beiden Schnitte, der das kleinere Gewicht hat, ein minimaler Schnitt in  $G$ .*

**Beweis.** Offensichtlich entsprechen die Schnitte in  $H$  genau denjenigen Schnitten in  $G$ , die  $s$  und  $t$  nicht voneinander trennen. Falls es in  $G$  einen minimalen Schnitt gibt, der ein  $s$ - $t$ -Schnitt ist, so ist  $C_1$  ein minimaler Schnitt in  $G$ . Falls es keinen solchen minimalen Schnitt gibt, so werden  $s$  und  $t$  durch den minimalen Schnitt nicht getrennt, und folglich liefert  $C_2$  einen minimalen Schnitt in  $G$ .  $\square$

Der Algorithmus geht nun in  $|V| - 1$  Phasen vor und berechnet dabei in Phase  $i$  einen Schnitt  $C_i$ , der für gewisse Knoten  $s$  und  $t$  im aktuellen Graphen ein minimaler  $s$ - $t$ -Schnitt ist. Die Ausgabe des Algorithmus ist dann derjenige Schnitt mit minimalem Gewicht unter

den  $|V| - 1$  berechneten Schnitten. Am Ende jeder Phase werden die Knoten  $s$  und  $t$  im aktuellen Graphen miteinander verschmolzen, so dass der Graph in jeder Phase um einen Knoten kleiner wird.

Betrachten wir das Vorgehen des Algorithmus in einer Phase  $i$ , wobei  $H = (V', E')$  den Graphen am Anfang der Phase bezeichne. Für eine Teilmenge  $A \subset V'$  und einen Knoten  $y \in V' \setminus A$  bezeichnen wir mit  $w(A, y)$  die Summe der Gewichte aller Kanten zwischen einem Knoten aus  $A$  und dem Knoten  $y$ . Zunächst wird ein beliebiger Knoten  $a$  des Graphen  $H$  ausgewählt und  $A = \{a\}$  gesetzt. Solange  $A \neq V'$ , fügt der Algorithmus nun jeweils denjenigen Knoten  $z \in V' \setminus A$  in  $A$  ein, der unter allen Knoten  $y \in V' \setminus A$  den maximalen Wert  $w(A, y)$  hat.

Bezeichne mit  $s$  den vorletzten in der aktuellen Phase in  $A$  eingefügten Knoten und mit  $t$  den letzten in  $A$  eingefügten Knoten. Der Schnitt  $C_i$  ist dann einfach als die Menge der Nachbarkanten von  $t$  in  $H$  definiert. Als Schnitt in  $G$  interpretiert, besteht  $C_i$  aus denjenigen Kanten von  $G$ , die einen in  $t$  enthaltenen Knoten mit einem nicht in  $t$  enthaltenen Knoten verbinden. Am Ende der Phase werden die Knoten  $s$  und  $t$  zu einem neuen Knoten verschmolzen.

Nach  $|V| - 1$  Phasen ist der Graph auf einen einzigen Knoten geschrumpft. Der Algorithmus terminiert und gibt denjenigen der  $|V| - 1$  berechneten Schnitte  $C_i$  mit minimalem Gewicht aus.

### Effiziente Implementierung und Laufzeit

Eine effiziente Implementierung des Algorithmus realisiert jede Phase ähnlich zum Algorithmus von Prim zur Berechnung minimaler Spannbäume (siehe Abb. 7.8): die Knoten aus  $V' \setminus A$  werden in einer Priority-Queue verwaltet, wobei die Priorität von Knoten  $y \in V' \setminus A$  genau  $w(A, y)$  ist. Zu Anfang haben die Nachbarknoten  $v$  von  $a$  also Priorität  $c(\{a, v\})$  und alle anderen Knoten in der Priority-Queue Priorität 0. Dann wird in jedem Schritt derjenige Knoten  $z$  mit maximaler Priorität aus der Priority-Queue entfernt und in  $A$  eingefügt. Nun muss lediglich für jeden Nachbarn  $v$  von  $z$  in der Priority-Queue die Priorität um das Gewicht der Kante zwischen  $z$  und  $v$  erhöht werden, so dass die Priorität aller Knoten  $y$  in der Priority-Queue wieder mit  $w(A, y)$  für die um einen Knoten vergrößerte Menge  $A$  übereinstimmt.

Die Priority-Queue muss das Extrahieren des Elementes mit der grössten Priorität sowie das Erhöhen der Priorität eines Elementes effizient unterstützen. Diese Anforderungen sind dieselben wie beim Algorithmus von Prim, nur dass wir jetzt statt DeleteMin und DecreasePriority die Operationen DeleteMax und IncreasePriority benötigen, die sich jedoch völlig analog implementieren lassen. Wenn für die Priority-Queue eine Implementierung mit Fibonacci-Heaps gewählt wird, kann damit die Laufzeit für eine Phase des Min-Cut-Algorithmus wie die Laufzeit des Algorithmus von Prim mit  $O(|E| + |V| \log |V|)$  abgeschätzt werden. Da insgesamt  $|V| - 1$  Phasen nötig sind, führt das zu einer Gesamtlaufzeit von  $O(|V||E| + |V|^2 \log |V|)$  für den Min-Cut-Algorithmus.

### Korrektheit des Algorithmus

Abschliessend wollen wir noch die Korrektheit des vorgestellten Min-Cut-Algorithmus beweisen. Die wesentliche Erkenntnis ist, dass der in Phase  $i$  berechnete Schnitt  $C_i$  für die beiden in dieser Phase zuletzt in  $A$  eingefügten Knoten  $s$  und  $t$  einen minimalen  $s$ - $t$ -Schnitt darstellt.

**Algorithmus für eine Phase des Min-Cut-Algorithmus**  
**Eingabe:** ungerichteter Graph  $H = (V', E')$  mit Kantengewichten  $c : E \rightarrow \mathbb{R}$   
**Ausgabe:** Knoten  $t$ , dessen Nachbarkanten ein minimaler  $s$ - $t$ -Schnitt sind

```

 $a :=$  beliebiger Startknoten in  $V'$ ;
 $A := \{a\}$ ;
p_queue  $Q$ ; { Priority-Queue für Knoten }
for alle inzidenten Kanten  $e = \{a, v\}$  von  $a$  do  $Q.$ Insert( $v, c(e)$ ); od;
for alle Knoten  $v \in V'$  mit  $v \neq a$  und  $v \notin Q$  do  $Q.$ Insert( $v, 0$ ); od;
 $t := a$ ;
while  $Q$  nicht leer do
   $t := Q.$ DeleteMax();
   $A := A \cup \{t\}$ ;
  for alle inzidenten Kanten  $e = \{t, w\}$  von  $t$  do
    if  $w \in Q$  then
       $Q.$ IncreasePriority( $w, Q.$ prio( $w$ ) +  $c(e)$ );
    fi;
  od;
od;
return  $t$ ;

```

Abbildung 7.8: Implementierung einer Phase des Min-Cut-Algorithmus

**Lemma 7.17** Sei  $H = (V', E')$  der Graph am Anfang einer Phase. Seien  $s$  und  $t$  die in der Phase zuletzt in  $A$  eingefügten Knoten. Dann bilden die Nachbarkanten von  $t$  einen minimalen  $s$ - $t$ -Schnitt in  $H$ .

**Beweis.** Sei  $T$  die Menge der Nachbarkanten von  $t$ . Sei  $C$  ein beliebiger  $s$ - $t$ -Schnitt in  $H$ . Wir zeigen, dass  $c(C) \geq c(T)$  ist. Daraus folgt dann, dass  $T$  ein minimaler  $s$ - $t$ -Schnitt ist.

Seien  $V_1$  und  $V_2$  die beiden Knotenmengen, so dass  $C$  die Menge aller Kanten zwischen  $V_1$  und  $V_2$  ist. Wir bezeichnen einen Knoten  $v \neq a$  als *aktiv*, falls  $v$  und der direkt vor  $v$  in  $A$  eingefügte Knoten in verschiedenen  $V_i$  ( $i = 1, 2$ ) sind.

Bezeichne mit  $A_v$  die Menge der Knoten, die vor  $v$  in  $A$  eingefügt wurden, und mit  $C_v$  den Schnitt, den  $C$  eingeschränkt auf die Knoten  $A_v \cup \{v\}$  ergibt.

Wir beweisen per Induktion nach der Anzahl aktiver Knoten, dass für jeden aktiven Knoten die Ungleichung

$$w(A_v, v) \leq c(C_v) \tag{7.1}$$

gilt. Der Knoten  $t$  ist offensichtlich ein aktiver Knoten, und die Aussage  $w(A_t, t) \leq c(C_t)$  liefert genau die Behauptung  $c(C) \geq c(T)$ , da  $c(T) = w(A_t, t)$  und  $c(C_t) = c(C)$  gilt.

Für den ersten aktiven Knoten ist (7.1) mit Gleichheit erfüllt. Das liefert uns den Induktionsanfang. Nehmen wir nun an, dass (7.1) für alle aktiven Knoten bis einschliesslich  $v$  richtig ist. Sei  $u$  der nächste aktive Knoten. Es gilt:

$$w(A_u, u) = w(A_v, u) + w(A_u \setminus A_v, u) =: \alpha$$

Da der Algorithmus den Knoten  $v$  vor  $u$  aus der Priority-Queue geholt hat, muss  $w(A_v, u) \leq w(A_v, v)$  gelten. Aus der Induktionsannahme wissen wir  $w(A_v, v) \leq c(C_v)$ . Also gilt:

$$\alpha \leq c(C_v) + w(A_u \setminus A_v, u)$$

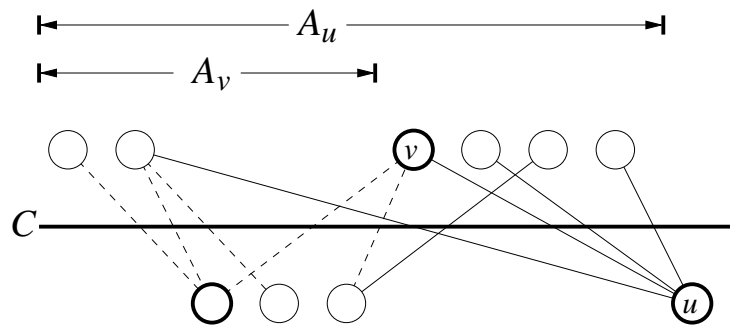


Abbildung 7.9: Skizze zum Beweis von Lemma 7.17. Die Knoten sind von links nach rechts in der Reihenfolge gezeichnet, in der sie in  $A$  eingefügt werden. Der Schnitt  $C$  ist durch die horizontale Linie angedeutet. Knoten oberhalb der Linie liegen in  $V_1$ , Knoten unterhalb in  $V_2$ . Kanten innerhalb  $V_1$  oder  $V_2$  sind nicht gezeichnet. Aktive Knoten sind dick umrandet. Im Schnitt  $C_u$  (der alle gezeichneten Kanten enthält) liegen mindestens die Kanten von  $C_v$  (gestrichelt gezeichnet) sowie die Kanten von  $u$  zu Knoten in  $A_u \setminus A_v$ .

Alle Kanten von  $A_u \setminus A_v$  zu  $u$  liegen im Schnitt  $C_u$ , aber nicht in  $C_v$  (siehe auch Abb. 7.9). Daher gilt  $c(C_v) + w(A_u \setminus A_v, u) \leq c(C_u)$ . Folglich ist (7.1) auch für  $u$  richtig und die Induktionsbehauptung damit gezeigt.  $\square$

Nun können wir leicht die Korrektheit des Algorithmus zeigen.

**Satz 7.18** *Der vorgestellte Min-Cut-Algorithmus berechnet den minimalen Schnitt in einem Graphen  $G = (V, E)$  mit Kantengewichten in Zeit  $O(|V||E| + |V|^2 \log |V|)$ .*

**Beweis.** Wir beweisen die Korrektheit per Induktion nach der Knotenzahl des Graphen. Für  $|V| = 2$  liefert der Algorithmus offensichtlich einen korrekten minimalen Schnitt. Nehmen wir nun an, dass der Algorithmus für alle Graphen mit höchstens  $n - 1$  Knoten richtig ist und sei  $G$  ein Graph mit  $n$  Knoten. Nach Lemma 7.17 berechnet der Algorithmus in Phase 1 korrekt einen minimalen  $s$ - $t$ -Schnitt. Sei  $H$  der Graph, der aus  $G$  durch Verschmelzung von  $s$  und  $t$  entsteht. Nach der Induktionsannahme liefert der kleinste unter allen in den weiteren Phasen berechneten Schnitten einen minimalen Schnitt in  $H$ . Mit Lemma 7.16 folgt, dass der kleinste unter allen berechneten Schnitten ein minimaler Schnitt in  $G$  ist.  $\square$

Mit Hilfe des Min-Cut-Algorithmus können wir natürlich insbesondere den Kantenzusammenhang eines Graphen in Zeit  $O(|V||E| + |V|^2 \log |V|)$  berechnen.

## Referenzen

- [NI92] Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM J. Disc. Math.*, 5(1):54–66, February 1992.
- [SW97] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, July 1997.