Universität Siegen
Fachbereich Mathematik

Graphentheoretische Modellierung eines automatisierungstechnischen Echtzeitnetzwerks und Algorithmenentwurf zum Kommunikations-Scheduling

Diplomarbeit

Vorgelegt von
Uwe Nowak

Betreut von
Prof. Dr. Hartmut Ring
Frank Dopatka

Siegen, im November 2006

Inhaltsverzeichnis

Liste der Algorithmen

Symbolverzeichnis

$\mathcal{G}$ Der baumförmige Graph $(\mathbb{V}, \mathbb{E})$ des Netzwerks ..... Seite 53

$\mathcal{N}$ Das das Netzwerk repräsentierende Viertupel $(\mathcal{G}, \Gamma, Duplex, Cast)$ ..... Seite 53

$\mathcal{P}(M)$ Potenzmenge einer Menge ..... Seite 4

$\mathbb{D}$ Menge der Devices ..... Seite 53

$\mathbb{E}$ Menge der Kanten (Kabel) im Netzwerk ..... Seite 53

$\mathbb{S}$ Menge der Switches ..... Seite 53

$\mathbb{V}$ Menge der Knoten (Switches und Devices) im Netzwerk ..... Seite 53

Kapitel 1

Einleitung

1.1 Technischer Hintergrund

In dieser Arbeit soll ein neuartiger, in [DW06a] und [DW06b] dargestellter Ansatz zur Entwicklung eines echtzeitfähigen automatisierungstechnischen Netzwerks auf Ethernet-Basis theoretisch modelliert und formal bewiesen werden.

Bisher existieren verschiedene Ethernet-Lösungen, die harte Echtzeitfähigkeit ermöglichen. Beispiele hierfür sind Ethernet PowerLink, ProfiNet oder EtherCAT. Diese Lösungen lassen sich im Wesentlichen in drei Klassen kategorisieren:

- Die erste Klasse verwendet Hubs und beinhaltete somit eine einzelne Kollisionsdomäne. Durch die Verwendung eines zentralen Arbiters mit einem globalen Schedule wird ein Zeitmultiplex-Verhalten (TDMA) auf dem Netzwerk durchgesetzt.

- Die zweite Klasse basiert auf der Verwendung hardwaretechnisch modifizierter Switches. Ein Vertreter dieser Klasse ist ProfiNet. ProfiNet-Switches lassen neben dem Transfer von Echtzeitdaten auch asynchrone Übertragung von Standard-Ethernet Frames zu. Sie ermöglichen für Echtzeitdaten eine sehr geringe Verzögerungszeit von 3 µs pro Switch.

- Die dritte Klasse entspricht einem Token-Ring-Ansatz. Dabei durchläuft ein einzelner Frame maximaler Größe das echtzeitfähige Netzwerk in Ring-Topologie.

In allen Ansätzen können in den echtzeitfähigen Teilnetzen jedoch nur echtzeitfähige Teilnehmer eingesetzt werden. Für die Verwendung anderer Teilnehmer ohne Kenntnis des Protokolls sind spezielle Gateways notwendig, um die Echtzeitfähigkeit des Netzwerks nicht zu stören.

Der in dieser Arbeit modellierte Ansatz soll ohne Bildung strikter Subnetze auskommen. Vorausgesetzt wird eine baumförmige Netzwerkstruktur mit einheitlicher Bandbreite. Des Weiteren müssen die Kommunikationsanforderungen (Requests) mit Echtzeitbedarf im Voraus bekannt sein. Es werden Algorithmen für die Erzeugung eines Schedules entwickelt, der lokal in den einzelnen Switches realisierbar ist.

Dabei werden die Netzwerktypen anhand der folgenden Merkmale unterschieden

- Während in Halbduplex-Netzwerken immer nur eine Kommunikation über ein Kabel möglich ist, ermöglicht ein Vollduplex-Netzwerk bidirektionale Kommunikation.

- Im Wesentlichen unterscheiden wir Unicast (Punkt-zu-Punkt-Verbindung), wo eine Kommunikation nur von einem Sender zu einem Empfänger verläuft und Multicast (Mehrpunktverbindung), bei der ein Sender gleichzeitig an mehrere Empfänger sendet und die Pakete in den Switches falls notwendig vervielfältigt werden. Broadcast-Netzwerke, bei dem ein Sender an alle anderen Devices gleichzeitig sendet, entsprechen der oben beschriebenen ersten Klasse.
- Die Längen der zu schedulenden Requests können alle gleich sein (natürliche Schedules), Zweierpotenzen bilden (gestaffelte Schedules) oder beliebig sein.

1.2 Überblick über diese Arbeit

In Kapitel 2 werden notwendige mathematische und graphentheoretische Grundbegriffe zusammengefasst.

Da Knoten- und Kantenfärbung für die Lösung des Problems für einheitliche Requestlängen bedeutend sind, werden unterschiedliche Färbungsalgorithmen in Kapitel 3 zusammengefasst und analysiert.

In den Abschnitten 4.1 bis 4.2.3 wird das Netzwerk mit den zugehörigen Requests modelliert. Die Netzwerkstruktur $\mathcal{G}$ selbst wird als baumförmiger, ungerichteter Graph modelliert. Dabei werden die Switches und Devices durch Knoten, die Kabel durch Kanten repräsentiert.

Jedem Request r wird im Unicast-Fall die Kommunikationslinie $KL(r)$ als eindeutiger Weg vom Quelldevice zum Zieldevice des Requests zugeordnet, im Multicast-Fall ein Kommunikationsbaum $KB(r)$ vom Quelldevice zu allen Zieldevices. Zwei Requests haben dann in Halbduplex-Netzwerken einen Konflikt, wenn sie eine Kante gemeinsam haben und in Vollduplex-Netzwerken, wenn sie eine Kante gemeinsam in dieselbe Richtung enthalten. Da der Graph selbst ungerichtet ist, ist die Richtung einer Kante ausschließlich innerhalb von Kommunikationslinien und Kommunikationsbäumen definiert.

Ein zentrales Ergebnis aus 4.2.3 ist, dass in Nicht-Vollduplex-Multicast-Netzwerken zwei Requests durch einen Knoten genau dann einen Konflikt haben, wenn sie einen Konflikt in diesem Knoten haben. Dadurch genügt es für diese Netzwerke, Konflikte lokal zu betrachten.

In Abschnitt 4.3 werden Schedules für eine Menge von Requests als Abbildung betrachtet, die jedem Request eine Startzeit zuordnen. Dabei dürfen zwei miteinander in Konflikt stehende Requests niemals so geschedulet sein, dass sich die Zeit ihrer Ausführung überschneidet. Es werden zwei Schedules als synchron bezeichnet, wenn sie miteinander verträglich sind, d.h. wenn es möglich ist durch Zusammenzufassen der Schedules einen Schedule für die Vereinigung der Requestmengen zu erzeugen.

In Abschnitt 4.5 werden schließlich Datenstrukturen zur Speicherung des Netzwerks und Algorithmen für diese Strukturen entwickelt.

In Kapitel 5 werden Algorithmen für die Lösung des Schedulingproblems für beliebige Requestlängen gegeben.

In Kapitel 6 werden Algorithmen für die Lösung des Schedulingproblems gegeben, falls alle Requests die Länge 1 haben. Werden Requests nur zu ganzzahligen Zeitpunkten geschedulet,

so gibt es immer Blöcke gleichzeitig ausgeführter Requests. Dafür wird der Begriff der Synchronisierbarkeit zweier Schedules eingeführt. Anschaulich sind zwei Schedules genau dann synchronisierbar, wenn sie durch Vertauschen der Blöcke synchronisiert werden können.

Für Halbduplex-Netzwerke ergibt sich der zentrale Satz, dass je zwei Schedules für die Requests durch einen Knoten synchronisierbar sind. Das Problem eines globalen Schedules lässt sich daher durch die Erstellung lokaler Schedules für Knoten und deren Synchronisation lösen.

In Kapitel 7 werden Algorithmen für die Lösung des Schedulingproblems gegeben, falls alle Requests als Länge eine Zweierpotenz haben. Diese Algorithmen sind auf Grund einer besonderen Datenstruktur sehr effizient. Des Weiteren können in Kapitel 5 gegebene Algorithmen verbessert werden, indem das Umschedulen von Requests unter bestimmten Bedingungen erlaubt wird.

Zuletzt wird ein bei der sequentiellen Färbung verwendeter Ansatz (Saturation-Largest-First) auf das Erstellen in gestaffelter Schedules übertragen.

1.3 Anmerkungen

Während in der Literatur Färbungen üblicherweise als Abbildungen in Mengen $\{1, \ldots, n\}$ betrachtet werden, werden sie in dieser Arbeit meistens als Abbildung in Mengen $\{0, \ldots, n-1\}$ definiert. Dieses liegt daran, dass sie im Verlauf der Arbeit für die Erstellung von natürlichen Schedules verwendet werden und dort die Farbenmenge $\{0, \ldots, n-1\}$ zwingend vorgegeben ist.

Es wäre auch möglich, Vollduplex-Netzwerke als gerichtete Graphen zu modellieren. Dann wären jedoch Halbduplex- und Vollduplex-Netzwerke völlig getrennt zu betrachten. So hingegen fallen viele Beweise zusammen, meistens ist der Vollduplex-Fall nur eine Ergänzung zum Halbduplex-Fall.

Des Weiteren hätten die Definitionen für Unicast- und Multicast-Netzwerke vereinheitlicht werden können. Dieses hätte die Redundanzen in den Abschnitten 4.1 bis 4.2.3 jedoch die Komplexität der Beweise deutlich erhöht und somit die Verständlichkeit der Arbeit verschlechtert.

Außerdem wurden einige Eigenschaften bewiesen, die für die Algorithmen nicht unmittelbar notwendig sind. Ziel dieser Arbeit ist neben der Analyse und dem Beweis der Algorithmen jedoch auch, eine mathematische Theorie und Begriffe zu den behandelten Themen aufzubauen.

Kapitel 2

Grundlagen

2.1 Mathematische Grundbegriffe

Definition 2.1.1 (Potenzmenge)**.** *Sei M eine Menge, dann ist die* Potenzmenge $\mathcal{P}(M) := \{A : A \subset M\}$ *die Menge aller Teilmengen von M.*

Definition 2.1.2. *Zu einer natürlichen Zahl n ist $\underline{n} := \{0, \dots, n-1\}$.*

Definition 2.1.3 (Disjunkte Überdeckung)**.** *Sei T eine Menge. Eine Mengenfamilie $\mathcal{T} = (T_i)_{i \in I}$ heißt* disjunkte Überdeckung *von T, wenn*

- *T wird von $\mathcal{T}$ überdeckt: $T = \bigcup_{i \in I} T_i$.*
- *Die Mengen von $\mathcal{T}$ sind disjunkt: $i \neq j \implies T_i \cap T_j = \emptyset$.*

Die Mengen T_i heißen Klassen*. Wir definieren eine Indexfunktion $ind : T \mapsto I$, die jedem $t \in T$ den eindeutig bestimmten Index $i \in I$ mit $t \in T_i$ zuordnet, d.h für alle $t \in T$ ist $t \in T_{ind(t)}$.*

Ist nicht eindeutig, auf welche disjunkte Überdeckung sich die Indexfunktion bezieht, so schreiben wir für die Indexfunktion bezüglich der Überdeckung $\mathcal{T}$ auch $ind_{\mathcal{T}}$ statt ind.

Beachte, dass eine disjunkte Überdeckung i.A. keine Klasseneinteilung ist, da bei einer disjunkten Überdeckung einzelne Klassen leer sein können.

Lemma 2.1.4. *Sei A eine Menge, $(B_i)_{i \in I}$ eine disjunkte Überdeckung von B und $f : A \mapsto B$ eine Abbildung. Dann ist $(f^{-1}(B_i))_{i \in I}$ eine disjunkte Überdeckung von A. Insbesondere ist $(f^{-1}\{b\})_{b \in B}$ eine disjunkte Überdeckung von A.*

Beweis. Zu zeigen:

- Es gilt $A = \bigcup_{i \in I} f^{-1}(B_i)$.
- Es ist $i \neq j \implies f^{-1}(B_i) \cap f^{-1}(B_j) = \emptyset$.

Es gilt

$$\bigcup_{i \in I} f^{-1}(B_i) = f^{-1}\left(\bigcup_{i \in I} B_i\right) = f^{-1}(B) = A.$$

Indirekter Beweis: Angenommen, es gäbe ein $x \in f^{-1}(B_i) \cap f^{-1}(B_j)$. Dann

$$\begin{aligned} & x \in f^{-1}(B_i) \cap f^{-1}(B_j) \\ \Longrightarrow\ & f(x) \in B_i \wedge f(x) \in B_j \\ \Longrightarrow\ & f(x) \in B_i \cap B_j \\ \Longrightarrow\ & B_i \cap B_j \neq \emptyset \\ \Longrightarrow\ & i = j \end{aligned}$$

Da offensichtlich $(\{b\})_{b \in B}$ eine disjunkte Überdeckung von B ist, ist $(f^{-1}\{b\})_{b \in B}$ eine disjunkte Überdeckung von A. □

Definition 2.1.5 (Restriktion einer Funktion). *Sei $f : A \mapsto D$ und $B \subset A$. Dann ist die* Restriktion *von f auf B definiert durch*

$$f_{|B} : B \mapsto D, \qquad b \to f(b).$$

Lemma 2.1.6 (Mehrfache Restriktion). *Sei $C \subset B \subset A$ und $f : A \mapsto D$ eine Abbildung. Dann ist $(f_{|B})_{|C} = f_{|C}$.*

Beweis. Es gilt für alle $c \in C$:

$$(f_{|B})_{|C}(c) = f_{|B}(c) = f(c) = f_{|C}(c)$$

□

Satz und Definition 2.1.7 (Injektive Fortsetzung einer Funktion). *Seien A, B, X Mengen und A, B endlich. Sei $f : A \mapsto X$ injektiv und $A \subset B$ mit $|B| \leq |X|$. Dann gibt es eine injektive Funktion $g : B \mapsto X$ mit $g_{|A} = f$. Die Funktion g heißt* injektive Fortsetzung *von f auf B.*

Beweis. Es ist $|f(A)| = |A|$ und somit

$$|X \setminus f(A)| = |X| - |f(A)| \geq |B| - |A| = |B \setminus A|.$$

Daher gibt es eine injektive Abbildung $f_1 : B \setminus A \mapsto X \setminus f(A)$. Definiere

$$g : B \mapsto X, \qquad x \to \begin{cases} f(x) & \text{falls } x \in A \\ f_1(x) & \text{falls } x \notin A \end{cases}$$

Dann ist g injektiv. Denn sei $x_1, x_2 \in B$ mit $x_1 \neq x_2$. Für $x_1, x_2 \in A$ gilt $g(x_1) = f(x_1) \neq f(x_2) = g(x_2)$ und für $x_1, x_2 \notin A$ ist $g(x_1) = f_1(x_1) \neq f_1(x_2) = g(x_2)$. Ansonsten sei o.B.d.A. $x_1 \in A$ und $x_2 \notin A$. Dann ist $g(x_1) = f(x_1) \in f(A)$ und $g(x_2) = f(x_2) \in X \setminus f(A)$, also ist $g(x_1) \neq g(x_2)$. □

2.2 Graphentheoretische Grundbegriffe

In dieser Einführung werden einige elementare Notationen, Definitionen und Sätze zusammengetragen. Diese sind u.A. in [Jun99], [Vol91] und [Die06] nachzulesen.

Definition 2.2.1 (Graph). *Seien V und E disjunkte Mengen und $g : E \mapsto V^{\underline{2}}$. Dann heißt das Tripel $G = (V, E, g)$ (ungerichteter)* Graph. *Die Elemente aus V heißen* Knoten *oder* Ecken*, die Elemente aus E heißen* Kanten *und g ist die* Randabbildung*, die jeder Kante ihre Randknoten zuordnet.*

Ein Graph $G = (V, E, g)$ heißt einfach*, wenn g injektiv ist. Dann lässt sich jede Kante $e \in E$ eindeutig mit ihren Randknoten $g(e) \in V^{\underline{2}}$ identifizieren. Wir setzen dann $E' = \{g(e) : e \in E\}$ und schreiben $G = (V, E')$ für $G = (V, E, g)$.*

Zur besseren Unterscheidung nennt man einen nicht einfachen Graphen auch Multigraph.

Bemerkung 2.2.2 (Beschränkung auf endliche Graphen). Ein Graph $G = (V, E, g)$ mit $|E| < \infty$ und $|V| < \infty$ heißt endlich. In dieser Arbeit werden wir uns auf endliche Graphen beschränken.

Definition 2.2.3 (Randpunkt, inzident, adjazent, benachbart). *Sei $G = (V, E, g)$ ein Graph.*

- *Ein Knoten v* inzidiert *(oder* ist inzident*) mit einer Kante e, wenn $v \in g(e)$ ist.*
- *Zwei Kanten e, f heißen* adjazent*, wenn sie einen gemeinsamen Randknoten haben, d.h. wenn es einen Knoten $v \in V$ gibt, der inzident mit e und f ist.*
- *Zwei Knoten v, w heißen* benachbart*, wenn sie durch eine Kante verbunden werden, d.h. wenn es eine Kante e mit $g(e) = \{v, w\}$ gibt.*

Die mit einer Kante e inzidenten Knoten sind die Randknoten von e. Eine Kante mit den Randknoten v und w bezeichnen wir mit vw. In einfachen Graphen ist $vw = \{v, w\} = wv$. In Multigraphen muss die Kante vw nicht eindeutig sein, dann sei vw eine beliebige Kante mit Randknoten v und w.

Ist $X, Y \subset V$ und $v \in X$, $w \in Y$, so heißt vw eine X-Y-Kante. Die Menge aller X-Y-Kanten in E wird mit $E(X, Y)$ bezeichnet. Wir schreiben auch $E(v, Y) := E(\{v\}, Y)$, $E(X, w) := E(X, \{w\})$ und $E(v, w) := E(\{v\}, \{w\})$. Weiter setzen wir $E(v) := E(v, V \setminus \{v\})$.

Zu einem Knoten $v \in V$ bezeichnen wir die Menge der mit v benachbarten Knoten mit $N(v) := \{w \in V : vw \in E\}$

Definition 2.2.4 (Schlinge, schlingenfreier Graph). *Sei G ein Graph. Eine Kante vv heißt* Schlinge*. Ein Graph ohne Schlinge heißt* schlingenfrei.

Definition 2.2.5 (Grad eines Knotens). *Sei v ein Knoten. Dann ist der* Grad $d(v) = |E(v)|$ *die Anzahl der mit v inzidenten Kanten.*

Lemma 2.2.6. *Für einen schlingenfreien Graphen $G = (V, E, g)$ gilt die Gleichung*

$$\sum_{v \in V} d(v) = 2|E|.$$

Beweis. In der Summe auf der linken Seite wird jede Kante e von beiden zu e inzidenten Knoten aus – also doppelt – gezählt. □

Definition 2.2.7 (Teilgraph, Untergraph). *Sei $G = (V, E, g)$ ein Graph. Ist $V' \subset V$ und $E' \subset E$, so dass $G'(V', E', g_{|E'})$ ein Graph ist, dann heißt G'* Teilgraph *von G (beachte, dass insbesondere alle Randknoten von Kanten aus E' in V' enthalten sein müssen). Wir schreiben $G' \subset G$. Der Teilgraph G' heißt* induziert *von V' in G, wenn $E' = \{xy : x, y \in V'\} \cap E$ ist. Dann schreiben wir $G' = G[V']$. Ein induzierter Teilgraph heißt auch* Untergraph.

Ist $G = (V, E)$ ein Graph und $v \in V$ ein Knoten, so schreiben wir für den Graphen, der durch Entfernen von v und allen mit v inzidenten Kanten aus G entsteht auch $G - v := G[V \setminus \{v\}]$.

Definition 2.2.8 (Pfad). *Sei $G = (V, E, g)$ ein Graph. Ein* Pfad $P = (\overline{x}, \overline{e})$ *in G besteht aus einer Folge $\overline{x} = (x_i)_{i=0,\dots,n}$ von Knoten aus V und einer Folge $\overline{e} = (e_i)_{i=1,\dots,n}$ von Kanten aus E, so dass für alle $i = 1, \dots, n$ gilt: $e_i = x_{i-1}x_i$.*

Die Knoten x_0 und x_n heißen Anfangs- *und* Endknoten *(bzw.* Randknoten*) von P, die Knoten $x_1, \dots, x_{n-1}$ heißen* innere Knoten. *Wir nennen P einen Pfad von x_0 nach x_n. Die Anzahl $l(P) := n$ der Kanten von P heißt* Länge *des Pfades.*

Ist $P = ((v_0, \dots, v_n), (e_1, \dots, e_n))$ ein Pfad von v_0 nach v_n, so ist der inverse Pfad $P^{-1} := ((v_n, \dots, v_0), (e_n, \dots, e_1))$ *ein Pfad von v_n nach v_0.*

Die Menge der Knoten bzw. Kanten des Pfades P wird mit $V_P = \{x_0, \dots, x_n\}$ bzw. $E_P = \{e_1, \dots, e_n\}$ bezeichnet.

Definition 2.2.9 (Weg). *Ein* Weg *ist ein Pfad mit paarweise verschiedenen Knoten.*

Offenbar enthält jeder Pfad zwischen zwei Knoten einen Weg zwischen diesen Knoten. Dieses bedeutet nicht, dass die Knoten im Pfad aufeinanderfolgend sind, z.B. kann ein Weg durch Entfernen einer Schlaufe entstehen.

Definition 2.2.10 (Teilweg). *Sei $P = ((v_0, \dots, v_n), (e_1, \dots, e_n))$ ein Weg. Für $0 \le i \le j \le n$ ist dann $Q = ((v_i, \dots, v_j), (e_{i+1}, \dots, e_j))$ ein* Teilweg *von P. Man schreibt $Q \lhd P$ und sagt, P enthält den Weg Q.*

Definition 2.2.11 (Verkettung von Pfaden). *Seien $P = ((x_0, \dots, x_n), (e_1, \dots, e_n))$ und $Q = ((x_n, \dots, x_{n+m}), (e_{n+1}, \dots, e_{n+m}))$ Pfade. Dann definiert man die* Verkettung $P + Q$ *von P und Q durch $P + Q = ((x_0, \dots, x_{n+m}), (e_1, \dots, e_{n+m}))$. Offenbar ist $P + Q$ ein Pfad.*

Falls definiert, verwenden wir auch die Schreibweisen $P + Q + R = P + (Q + R)$ etc.

Die Verkettung von Wegen P und Q ist genau dann ein Weg, wenn die Knoten beider Wege paarweise verschieden sind.

Lemma 2.2.12. *Sei $G = (V, E, g)$ ein Graph. Die Relation*

$$\sim \subset V \times V, \qquad v \sim w \iff \text{Es gibt einen Pfad von } v \text{ nach } w$$

ist eine Äquivalenzrelation.

Beweis. Reflexivität ist offensichtlich, denn der triviale Weg $P(v, \cdot)$ ist ein Weg von v nach v. Symmetrie folgt, denn ist P ein Weg von v nach w, so ist P^{-1} ein Weg von w nach v. Transitivität gilt, denn sei P ein Pfad von u nach v und Q von v nach w, so ist $P + Q$ ein Pfad von u nach w. □

Definition 2.2.13 (Zusammenhang, Zusammenhangskomponenten)**.** *Sei $G = (V, E, g)$ ein Graph und $V_1, \dots, V_n$ die Äquivalenzklassen bezüglich der in Lemma 2.2.12 definierten Äquivalenzrelation. Dann heißen $G[V_1], \dots, G[V_n]$* Zusammenhangskomponenten *von G.*

Ein Graph G heißt zusammenhängend*, wenn G nur aus einer Zusammenhangskomponente besteht, d.h. wenn es für je zwei Knoten $v, w \in V$ einen Pfad von v nach w in G gibt. Eine Knotenmenge $W \subset V$ heißt zusammenhängend (in G), wenn $G[W]$ zusammenhängend ist.*

Definition 2.2.14 (Bezeichnungen für Wege)**.** *Zu einem Weg $P = ((x_0, \dots, x_n), (e_1, \dots, e_n))$ vereinbaren wir als Notation*

$$\begin{aligned} &\text{Für } i = 0, \dots, n: && e_i^{P-} := x_{i-1}, \qquad e_i^{P+} := x_i \\ &\text{Für } i = 1, \dots, n: && x_i^{P-} := e_i \\ &\text{Für } i = 0, \dots, n-1: && x_i^{P+} := e_{i+1} \end{aligned}$$

Lemma 2.2.15 (Die Kanten eines Weges sind paarweise verschieden)**.** *Sei P ein Weg. Dann sind die Kanten von P paarweise verschieden.*

Beweis. Da die Knoten paarweise verschieden sind, gilt

$$\begin{aligned} & e_i = e_j \\ \implies & (x_{i-1} = x_{j-1} \wedge x_i = x_j) \vee (x_i = x_{j-1} \wedge x_{i-1} = x_j) \\ \implies & (i - 1 = j - 1 \wedge i = j) \vee (i = j - 1 \wedge i - 1 = j) \\ \implies & i = j \vee i = i - 2 \\ \implies & i = j. \end{aligned}$$

□

Da die Knoten und Kanten von P paarweise verschieden sind, ist – auch ohne Angabe des Index – zu $x \in V_P$ und $e \in E_P$ sowohl e^{P-}, e^{P+} als auch x^{P-}, x^{P+} wohldefiniert.

Lemma 2.2.16. *Sei G ein Graph und P und Q Wege in G. Dann gilt (ggf. wo die entsprechenden Symbole definiert sind)*

(i) *Für $e \in E_P$ ist $e^{P-} \neq e^{P+}$.*

(ii) *Für $x \in V_P$ ist $x^{P-} \neq x^{P+}$.*

(iii) *Für $e, f \in E_P$ gilt $e^{P-} = f^{P-} \iff e = f \iff e^{P+} = f^{P+}$.*

(iv) *Für $x, y \in V_P$ gilt $x^{P-} = y^{P-} \iff x = y \iff x^{P+} = y^{P+}$.*

(v) *Für $e \in E_P \cap E_Q$ gilt: $e^{P-} = e^{Q-} \iff e^{P+} = e^{Q+}$ und $e^{P-} = e^{Q+} \iff e^{P+} = e^{Q-}$.*

(vi) *Für $x \in V_P$ und $e \in E_P$ gilt $x = e^{P-} \iff x^{P+} = e$ und $x = e^{P+} \iff x^{P-} = e$.*

Beweis. Es sei $P = ((x_0, \ldots, x_n), (e_1, \ldots, e_n))$.

Zu (i): Sei $e = e_i$. Dann ist $e^{P-} = e_i^{P-} = x_{i-1} \neq x_i = e_i^{P+} = e^{P+}$.

Zu (ii): Sei $x = x_i$. Dann ist $x^{P-} = x_i^{P-} = e_i \neq e_{i+1} = x_i^{P+} = x^{P+}$.

Zu (iii): Sei $e = e_i$ und $f = e_j$. Dann ist $e = f \implies e^{P-} = f^{P-}$ und $e = f \implies e^{P+} = f^{P+}$.
Weiter gilt

$$e^{P-} = f^{P-} \implies e_i^{P-} = e_j^{P-} \implies x_{i-1} = x_{j-1} \implies i = j \implies e_i = e_j \implies e = f,$$
$$e^{P+} = f^{P+} \implies e_i^{P+} = e_j^{P+} \implies x_i = x_j \implies i = j \implies e_i = e_j \implies e = f.$$

Somit gelten die Äquivalenzen.

Zu (iv): Sei $x = x_i$ und $y = x_j$. Dann ist $x = y \implies x^{P-} = y^{P-}$ und $x = y \implies x^{P+} = y^{P+}$.
Weiter gilt

$$x^{P-} = y^{P-} \implies x_i^{P-} = x_j^{P-} \implies e_i = e_j \implies i = j \implies x_i = x_j \implies x = y,$$
$$x^{P+} = y^{P+} \implies x_i^{P+} = x_j^{P+} \implies e_{i+1} = e_{j+1} \implies i = j \implies x_i = x_j \implies x = y.$$

Somit gelten die Äquivalenzen.

Zu (v): Dieses folgt unmittelbar aus $\{e^{P-}, e^{P+}\} = e = \{e^{Q-}, e^{Q+}\}$.

Zu (vi): Sei $x = x_i$ und $e = e_j$. Dann gilt

$$x = e^{P-} \iff x_i = e_j^{P-} \iff i = j - 1 \iff i + 1 = j \iff x_i^{P+} = e_j \iff x^{P+} = e,$$
$$x = e^{P+} \iff x_i = e_j^{P+} \iff i = j \iff x_i^{P-} = e_j \iff x^{P-} = e.$$

□

2.2.1 Eigenschaften einfacher Graphen

Im Folgenden betrachten wir nur noch einfache Graphen $G = (V, E)$. Dann ist ein Pfad $P = (\overline{x}, \overline{e})$ durch die Folge $(x_0, \ldots x_n)$ seiner Knoten eindeutig festgelegt und die Kante e_i durch die Bezeichnung $e_i = x_{i-1}x_i$ eindeutig definiert. Wir schreiben dann auch $P = (x_0, \ldots, x_n)$ für P. Ist $x_0 \in X \subset V$ und $x_n \in Y \subset V$ so heißt P Pfad von X nach Y bzw. X-Y-Pfad. Sind alle Knoten $x_0, \ldots, x_n$ paarweise verschieden, so ist P ein Weg und heißt entsprechend Weg von X nach Y bzw. X-Y-Weg.

Für einen Weg $P = (x_0, \ldots, x_n)$ und $0 \leq i \leq j \leq n$ schreiben wir

$$x_iPx_j := (x_i, \ldots, x_j),$$
$$x_iP := (x_i, \ldots, x_n),$$
$$Px_j := (x_0, \ldots, x_j).$$

Weiter definieren wir

$$E_P(x_i) = \{(x_i)^{P-}, (x_i)^{P+}\} = \begin{cases} \{x_0x_1\} & \text{für } i = 0 \\ \{x_{i-1}x_i, x_ix_{i+1}\} & \text{für } 1 \leq i \leq n-1 \\ \{x_{n-1}x_n\} & \text{für } i = n \end{cases}.$$

Definition 2.2.17 (Schlaufe). *Ein Pfad $(x_0, \ldots, x_{n-1}, x_0)$ heißt Schlaufe, falls $(x_0, \ldots, x_{n-1})$ ein Weg ist. Ein Graph ohne Schlaufe heißt* schlaufenfrei.

Definition 2.2.18 (Baum). *Ein zusammenhängender schlaufenfreier Graph heißt* Baum. *Die Knoten mit Grad 1 heißen* Blätter.

Definition 2.2.19 (Benachbarte Mengen). *Sei $G = (V, E)$ ein Baum. Zwei disjunkte zusammenhängende Mengen $X, Y \subset V$ heißen* benachbart, *wenn es eine X-Y-Kante gibt.*

Satz 2.2.20 (Eigeschaften eines Baums). *Sei $G = (V, E)$ ein Baum. Dann gilt:*

(i) Ist G nicht-trivial (d.h. $|V| \geq 2$), so hat G mindestens zwei Blätter.

(ii) Zu je zwei verschiedenen Knoten $v, w \in V$ existiert ein eindeutig bestimmter Weg von v nach w in G.

(iii) Jeder zusammenhängende Untergraph von G ist ein Baum.

(iv) Seien $X, Y \subset V$ disjunkte Mengen und $G[X]$ und $G[Y]$ Bäume. Dann gibt es ein $x \in X$ und ein $y \in Y$, so dass jeder Weg von X nach Y in G den eindeutig bestimmten Weg P von x nach y in G enthält und jeder Weg von Y nach X den Weg P^{-1} enthält.

(v) Seien $x, y \in V$ benachbart und $z \notin \{x, y\}$. Sei P der Weg von x nach z und Q der Weg von y nach z. Dann ist $y \in V_P$ oder $x \in V_Q$.

(vi) Seien $X, Y \subset V$ zusammenhängede benachbarte Mengen und sei $z \notin X \cup Y$. Sei P ein X-z-Weg und und Q ein Y-z-Weg. Dann ist $y \in V_P$ oder $x \in V_Q$.

Beweis. Zu (i): Sei $P = (x_0, \ldots, x_n)$ ein Weg maximaler Länge n in G. Dann ist x_0 ein Blatt. Denn hat x_0 außer x_1 einen weiteren Nachbarn v, so ist $v \notin V_P$, da G schlaufenfrei ist, also wäre $(v, x_0, \ldots, x_n)$ ein Weg der Länge $n + 1$. Widerspruch.

Zu (ii): Seien $v, w \in V$. Da G zusammenhängend ist, existiert ein Pfad und somit ein Weg von v nach w. Seien nun $(v = x_0, x_1, \ldots, x_{n-1}, x_n = w)$ und $(v = y_0, y_1, \ldots, y_{m-1}, y_m = w)$ zwei verschiedene Wege von v nach w. Sei k der kleinste Index mit $x_k \neq y_k$. Sei $u = x_{k-1} = y_{k-1}$ und $u' = x_i = y_j$ der erste Knoten nach u, der beiden Wegen angehört. Dann ist $(u, x_k, \ldots, x_{i-1}, u', y_{j-1}, \ldots, y_k, u)$ eine Schlaufe. Widerspruch.

Zu (iii): Sei G' ein zusammenhängender Untergraph von G. Dann ist G' auch schlaufenfrei, da jede Schlaufe in G' eine Schlaufe in G wäre. Somit ist G' ein Baum.

Zu (iv): Die Fälle $X = \emptyset$ und $Y = \emptyset$ sind trivial. Sei $X, Y \neq \emptyset$. Dann gibt es einen kürzesten X-Y-Weg $P = (x = z_0, \ldots, z_n = y)$. Offenbar ist wegen der minimalen Länge $z_1, \ldots, z_n \notin X$ und $z_0, \ldots, z_{n-1} \notin Y$. Sei nun $v \in X$ und $w \in Y$. Dann gibt es einen komplett in $G[X]$ verlaufenden Weg P_X von v nach x und einen komplett in $G[Y]$ verlaufenden Weg P_Y von y nach w. Somit ist $P_X + P + P_Y$ definiert und der eindeutig bestimmte Weg von v nach w. Dieser enthält P.

Ist umgekehrt Q ein Weg von Y nach X, so ist Q^{-1} ein Weg von X nach Y, enthält also P. Somit enthält auch Q den Weg P^{-1} von y nach x.

Zu (v): Falls $x \notin V_Q$, so ist $xy + Q$ der eindeutig bestimmte Weg P von x nach z, also ist $y \in V_P$.

Zu (vi): Sei $\{x, y\}$ die X-Y-Kante mit $x \in X$ und $y \in Y$. Sei P ein Weg von v nach z und Q von w nach z. Es gibt einen komplett in X verlaufenden Weg P' von x nach v und einen komplett in Y verlaufenden Weg Q' von y nach w. Dann enthält $P + P'$ den Weg P'' von x nach z und $Q + Q'$ den Weg Q'' von y nach z als Teilweg. Ist nun $x \notin V_Q$, so ist wegen $x \notin Q'$ auch $x \notin Q''$. Nach (v) ist dann $y \in P''$, wegen $y \notin P'$ somit $y \in P$.

□

Definition 2.2.21 (Bezeichnungen für Bäume). *Sei $G = (V, E)$ ein Baum. Dann bezeichnet man mit $l(G)$ die maximale Länge eines Weges in G.*

Definition 2.2.22 (Isomorphie, Isomorphismus). *Ein* Isomorphismus *zwischen zwei einfachen Graphen $G = (V, E)$ und $G' = (V', E')$ ist eine bijektive Abbildung $\varphi : V \mapsto V'$, so dass gilt:*

$$vw \in E \iff \varphi(v)\varphi(w) \in E'.$$

Zwei Graphen heißen isomorph, *wenn ein* Isomorphismus *zwischen ihnen existiert.*

Definition 2.2.23 (Vollständiger Graph). *Ein einfacher und schlingenfreier Graph $G = (V, E)$ heißt* vollständig, *wenn je zwei Knoten von V benachbart sind, d.h. wenn gilt*

$$E = \{\{v, w\} : v, w \in V, v \neq w\}.$$

Da es zu jedem $n \in \mathbb{N}$ bis auf Isomorphie genau einen vollständigen Graphen gibt, bezeichnet man diesen auch mit K_n.

Definition 2.2.24 (Clique, Cliquenzahl). *Sei $G = (V, E)$ ein einfacher, schlingenfreier Graph. Eine* Clique C der Ordnung n *von G ist ein zu K_n isomorpher Untergraph von G. Eine Clique heißt* maximal, *wenn sie die größte Ordnung aller Cliquen von G hat. Die Ordnung einer maximalen Clique von G heißt* Cliquenzahl $cl(G)$ *von G.*

2.2.2 Hypergraphen

Am Rande werden in dieser Arbeit auch Hypergraphen von Bedeutung sein. Anschaulich gesprochen ist ein Hypergraph ein Graph, bei dem eine Kante eine beliebige Anzahl an Knoten – insbesondere mehr als zwei Knoten – verbinden kann.

Definition 2.2.25 (Hypergraph). *Seien V und E disjunkte Mengen und $g : E \mapsto \mathcal{P}(V)$. Dann heißt das Tripel $G = (V, E, g)$ (ungerichteter)* Hypergraph. *Man nennt V* Knoten, *E* Kanten *und die Elemente aus $g(e)$* Randknoten *oder Knoten der Kante e.*

Eine Kante e *verbindet* zwei Knoten v und w, wenn $v, w \in g(e)$ gilt. Entsprechend zu gewöhnlichen Graphen definiert man auch für Hypergraphen die Begriffe adjazent, inzident, benachbart usw.

2.2.3 Datenstrukturen für Graphen

Multigraphen $G = (V, E, g)$ können auf verschiedene Arten dargestellt werden.

Adjazenzmatrix Die Knoten werden mit 1 bis n durchnummeriert und in einer $n \times n$-Matrix $(a_{ij})_{1 \leq i,j \leq n}$ wird in der Komponente a_{ij} die Anzahl der Kanten von i nach j gespeichert. G ist genau dann einfach, wenn alle $a_{ij} \in \{0, 1\}$ sind und genau dann schlingenfrei, wenn alle $a_{ii} = 0$ sind. Offenbar benötigt diese Darstellung $\mathcal{O}(|V|^2)$ Speicherplatz.

- Das Finden aller Nachbarn eines Knotens hat die Komplexität $\mathcal{O}(|V|)$.
- Knoten können nicht hinzugefügt werden.
- Das Hinzufügen einer Kante ist in konstanter Zeit möglich.
- Das Testen, ob eine Kante existiert, benötigt konstante Zeit.

Adjazenzliste Alle Knoten werden in einer (doppelt) verketteten Liste oder einem Array gespeichert. Jeder Knoten v enthält eine Liste aller mit v inzidenten Kanten, wobei jede Kante eine Referenz auf ihre Randknoten enthält. Es ist ebenfalls möglich, dass nicht die Kante selbst, sondern nur der Nachbarknoten mit der Anzahl der Kanten zu diesem gespeichert wird. Diese Darstellung benötigt $\mathcal{O}(|V|+|E|)$ Speicherplatz. Werden Mehrfachkanten durch ihre Multiplizität gespeichert, so wird sogar nur $\mathcal{O}(|V| + |F|)$ Speicherplatz benötigt. Dabei ist $F = \{g(e) : e \in E\}$ die Menge der Kantentypen, d.h. die Menge der Kanten, wenn mehrere parallele Kanten durch eine Kante ersetzt werden.

- Das Finden aller Nachbarn eines Knotens v hat eine Komplexität $\mathcal{O}(d(v))$.
- Kanten können in konstanter Zeit hinzugefügt werden.
- Knoten können bei der Repräsentation der Knoten in einer Liste in konstanter Zeit hinzugefügt werden.

Für alle Darstellungen ist es ohne Zusatzaufwand möglich, den Grad $d(v)$ jedes Knotens zu speichern.

Die Umwandlung zwischen einer Darstellung als Adjazenzliste und einer Darstellung als Adjazenzmatrix ist in $\mathcal{O}(|V|^2)$ möglich. Dieses liegt im Wesentlichen daran, dass jedes der $|V|^2$ Elemente der Adjazenzmatrix gelesen bzw. geschrieben werden muss.

Kapitel 3

Graphenfärbung

3.1 Färbung von Graphen

Definition 3.1.1 (Knotenfärbung, Kantenfärbung)**.** *Sei $G = (V, E, g)$ ein (Multi)Graph oder (Multi)Hypergraph.*

- *Eine* Knotenfärbung *von G ist eine Abbildung $c : V \mapsto S$ in eine endliche Menge S, so dass für benachbarte Knoten $v, w \in V$ gilt $c(v) \neq c(w)$.*
- *Eine* Kantenfärbung *von G ist eine Abbildung $c : E \mapsto S$ in eine endliche Menge S, so dass für adjazente Kanten $e, f \in E$ gilt $c(e) \neq c(f)$.*

Mit $|c|$ wird die Anzahl $|c(V)|$ bzw. $|c(E)|$ der verwendeten Farben bezeichnet.

Für Knotenfärbung sind mehrfache Kanten irrelevant, da diese keinen Unterschied für die Nachbarschaft zweier Knoten machen. Für Kantenfärbung sind sie insofern relevant, als dass je zwei Kanten mit denselben Randknoten adjazent sind, also mit unterschiedlichen Farben gefärbt werden müssen.

Enthält G eine Schlinge, so gibt es eine Kante vv, für eine gültige Knotenfärbung c müsste also $c(v) \neq c(v)$ gelten. Somit ist die Knotenfärbung eines Graphen mit Schlinge nicht möglich. Daher seien alle Graphen als schlingenfrei vorausgesetzt.

Im Folgenden werden wichtige Eigenschaften zur Färbung zusammengetragen. Im Wesentlichen stammen die Sätze aus [Die06, Kapitel 4] und [Jun99, Kapitel 8].

Sei $G = (V, E)$ ein Graph und $c : V \mapsto S$ bzw. $c : E \mapsto S$ eine Knoten- bzw. Kantenfärbung von G. Dann heißt S die Menge der Farben. Relevant ist lediglich die Anzahl der Elemente von S, so meistens $S = \{0, \ldots, n-1\} = \underline{n}$ angenommen wird.

In der Regel sind wir an einer Färbung mit einer möglichst geringen Anzahl von Farben interessiert. Dieses führt zur folgenden Definition:

Definition 3.1.2 (Chromatischer Index, k-färbbar, k-Färbung)**.** *Sei $G = (V, E)$ ein schlingenfreier Graph. Dann definiert man den* knotenchromatischen Index $\chi(G)$ *bzw. den* kantenchromatischen Index $\chi'(G)$ *durch*

$$\chi(G) = \min\{k \in \mathbb{N} : \text{ Es existiert eine Knotenfärbung } c : V \mapsto \underline{k}\},$$
$$\chi'(G) = \min\{k \in \mathbb{N} : \text{ Es existiert eine Kantenfärbung } c : E \mapsto \underline{k}\}.$$

Ein Graph heißt k-knotenfärbbar, *wenn* $\chi(G) \leq k$ *ist und* k-kantenfärbbar, *wenn* $\chi'(G) \leq k$ *ist.*

Eine Knoten- bzw. Kantenfärbung, die maximal k *Farben benötigt, heißt* k-Knotenfärbung *bzw.* k-Kantenfärbung.

Definition 3.1.3 (Bipartiter Graph)**.** *Ein Graph* $G = (V, E)$ *mit* $\chi(G) \leq 2$ *heißt* bipartit.

Die Frage ist nun, wie sich der knoten- bzw. kantenchromatische Index eines Graphen und eine Färbung mit minimaler Anzahl an Farben bestimmen lässt.

Dazu führen wir folgende Notationen ein. Sei $G = (V, E)$ ein Graph. Der Grad eines Knotens sei mit $d(v)$ bezeichnet. Dann wird definiert

$$\begin{aligned} \text{Minimalgrad:} \quad & \delta(G) := \min\{d(v) : v \in V\}, \\ \text{Maximalgrad:} \quad & \Delta(G) := \max\{d(v) : v \in V\}, \\ \text{Maximale Anzahl paralleler Kanten:} \quad & \mu(G) := \max\{|E(v, w)| : v, w \in V\}. \end{aligned}$$

Ein Graph ist genau dann einfach, wenn $\mu(G) \leq 1$ ist.

Kantenfärbungen können auf Knotenfärbungen zurückgeführt werden.

Definition 3.1.4 (Kantengraph)**.** *Sei* $G = (V, E)$ *ein Graph. Der* Kantengraph $L(G) = (V', E')$ *ist definiert durch*

$$\begin{aligned} V' &= E, \\ E' &= \{\{e, f\} \in (V')^{\underline{2}} : e \cap f \neq \emptyset\}. \end{aligned}$$

Die Knoten von $L(G)$ *sind somit gerade die Kanten von* G*. Zwei Knoten von* $L(G)$ *sind genau dann benachbart, wenn sie als Kanten von* G *adjazent sind.*

Satz 3.1.5 (Kantenfärbung von G entspricht Knotenfärbung von $L(G)$)**.** *Sei* $G = (V, E)$ *ein Graph. Dann ist* $c : E \mapsto S$ *genau dann eine Kantenfärbung von* G*, wenn es eine Knotenfärbung von* $L(G)$ *ist.*

Beweis. Sei $c : E \mapsto S$. Dann gilt

$$\begin{aligned} & c \text{ ist Kantenfärbung von } G \\ \iff & \text{für in } G \text{ adjazente Kanten } e \text{ und } f \text{ gilt } c(e) \neq c(f) \\ \iff & \text{für in } L(G) \text{ benachbarte Knoten } e \text{ und } f \text{ gilt } c(e) \neq c(f) \\ \iff & c \text{ ist Knotenfärbung von } L(G). \end{aligned}$$

□

Üblicherweise wird Knotenfärbung auch schlicht als Färbung und der knotenchromatische Index eines Graphen kurz als *chromatischer Index* bezeichnet.

3.1.1 Graphenfärbung ist NP-vollständig

Im Folgenden wird nun der Beweis, dass Knotenfärbung NP-vollständig ist, skizziert. Dabei werden Grundkenntnisse und Begriffe wie *nicht-deterministische Turingmaschine* oder *boolesche Formeln* als bekannt vorausgesetzt.

Definition 3.1.6 (NP, NP-hart, NP-vollständig)**.** *Die Klasse P ist die Menge der Probleme, die eine deterministische Turingmaschine in polynomieller Zeit beantworten kann. Die Klasse NP ist die Menge der Probleme, die eine nicht-deterministische Turingmaschine in polynomieller Zeit positiv beantworten kann.*

Definition 3.1.7 (Polynomielle Reduktion)**.** *Ein Problem A heißt* polynomiell auf ein Problem B reduzierbar *(i.Z. $A \prec_p B$), wenn es einen Algorithmus φ mit polynomieller Laufzeit gibt, der aus einer Antwort auf B eine Antwort auf A berechnet.*

Diese Definition basiert auf der folgenden Idee: Gibt es einen polynomiellen Algorithmus ψ für B, so wäre die Hintereinanderausführung von φ nach ψ ein polynomieller Algorithmus für A. Also bedeutet $A \prec_p B$, dass $B \in P \implies A \in P$ gilt.

Definition 3.1.8 (NP-hart, NP-vollständig)**.** *Ein Problem A heißt* NP-hart*, wenn sich alle Probleme aus NP polynomiell auf A reduzieren lassen. Ein NP-hartes Problem $A \in NP$ heißt* NP-vollständig*.*

Definition 3.1.9 (Konjunktive Normalformen, 3-KNF)**.** *Wir betrachten nun eine spezielle Teilmenge der booleschen Formeln über einer Variablenmenge X. Es gibt*

- *Literale:* $Lit := \{x, \neg x : x \in X\}$.
- *Klauseln:*

$$Cl := \{\textit{Disjunktion von Literalen}\} = \left\{ \bigvee_{i=1}^{n} x_i : x_i \in Lit, i = 1, \ldots, n \right\}.$$

- *Konjunktive Normalform:*

$$KNF := \{\textit{Konjunktion von Klauseln}\} = \left\{ \bigwedge_{i=1}^{n} F_i : F_i \in Cl, i = 1, \ldots, n \right\}$$

- *3-KNF ist die Menge der Konjunktiven Normalformen, bei dem jede Klausel genau drei Literale enthält.*

Unter der Länge einer Formel $F \in$ 3-KNF verstehen wir die Anzahl der Klauseln von F.

Die Sätze und Beweise dieses Kapitels stammen im Wesentlichen aus [Aho88].

Satz und Definition 3.1.10 (3-SAT ist NP-vollständig)**.** *Eine Formel $F \in$ 3-KNF mit der Variablenmenge X heißt* erfüllbar*, wenn es eine Belegung der Variablen aus X so gibt, dass F wahr ist. Das Problem (3-SAT) ist, zu einer gegebenen Formel $F \in$ 3-KNF zu prüfen, ob F erfüllbar ist. Dann gilt: 3-SAT ist NP-vollständig.*

Satz 3.1.11 (Graphenfärbung ist NP-vollständing). *Das Entscheidungsproblem, ob ein gegebener Graph G mit n Farben färbbar – d.h. $\chi(G) \leq n$ – ist, ist NP-vollständig.*

Insbesondere gibt es – falls $P \neq NP$ – keinen polynomiellen Algorithmus zur Bestimmung von $\chi(G)$ und somit auch keinen polynomiellen Algorithmus zur Bestimmung einer Färbung mit minimaler Farbanzahl.

Beweis. Um zu zeigen, dass das Färbungsproblem (COL) NP-vollständig ist, zeigen wir 3-SAT$\prec_p$COL und COL $\in$ NP.

Zu 3-SAT$\prec_p$COL:

- Zu einer booleschen Formel $F \in$ 3-KNF mit n Variablen $x_1, \ldots, x_n$ und t Klauseln $F_1, \ldots, F_t$ konstruieren wir nun in polynomieller Zeit abhängig von $\max(n, t)$ einen Graphen $G = (V, E)$ zu F, der genau dann mit $n + 1$ Farben gefärbt werden kann, wenn F erfüllbar ist. Könnte nun in polynomieller Zeit getestet werden, ob der Graph $n+1$-färbbar wäre, so könnte in polynomieller Zeit geprüft werden, ob F erfüllbar ist.

  Da die Erfüllbarkeit jeder Formel aus 3-KNF mit $n = 3$ oder weniger Variablen durch Ausprobieren der $2^n \leq 8$ möglichen Belegungen für $x_1, \ldots, x_n$ in linearer Zeit in Abhängigkeit der Länge t getestet werden kann, wird o.B.d.A. $n \geq 4$ vorausgesetzt.

- *Definition des Graphen:* Wir definieren mit weiteren Symbolen $v_1, \ldots, v_n$ den Graphen $G = (V, E)$ durch

  $$\begin{aligned} V :=& \{x_i, \neg x_i, v_i : i = 1, \ldots, n\} \cup \{F_i : i = 1, \ldots, t\} \\ E :=& \{v_i v_j, v_i x_j, v_i(\neg x_j) : i \neq j\} \\ & \cup \{x_i(\neg x_i) : i = 1, \ldots, n\} \\ & \cup \{x_i F_j : x_i \text{ ist kein Literal von } F_j\} \cup \{(\neg x_i) F_j : \neg x_i \text{ ist kein Literal von } F_j\} \end{aligned}$$

  Insbesondere sind nun benachbart:

  - Je zwei verschiedene Knoten v_i, v_j.
  - Jedes Literal x_i und $\neg x_i$ mit jedem Knoten $v_j, j \neq i$.
  - Jedes Literal mit seiner Negierung.
  - F_j mit jedem Literal, welches nicht in F_j vorkommt.

  Nun soll versucht werden, eine Knotenfärbung $c : V \mapsto S$ von G mit $|S| = n+1$ Farben zu konstruieren.

- *Es gilt $c(x_j) = c(v_j)$ oder $c(\neg x_j) = c(v_j)$ für $j = 1, \ldots, n$:* Da je zwei verschiedene Knoten v_i, v_j benachbart sind, ist $G[v_1, \ldots, v_n]$ ein vollständiger Graph mit n Knoten und benötigt bei einer Färbung n verschiedene Farben. Nun gilt für alle j und $i \neq j$:

  $$\begin{aligned} v_i x_j \in E &\implies c(v_i) \neq c(x_j), \\ v_i(\neg x_j) \in E &\implies c(v_i) \neq c(\neg x_j), \\ x_j(\neg x_j) \in E &\implies c(x_j) \neq c(\neg x_j). \end{aligned}$$

  Wäre für ein j nun $c(x_j) \neq c(v_j)$ und $c(\neg x_j) \neq c(v_j)$, so würden mindestens $n + 2$ verschiedene Farben $c(v_1), \ldots, c(v_n), c(x_j), c(\neg x_j)$ verwendet. Da c jedoch nur $n + 1$

Farben verwendet, muss entweder $c(x_j) = c(v_j)$ oder $c(\neg x_j) = c(v_j)$ gelten. Der andere Knoten x_j bzw. $\neg x_j$ ist hingegen mit einer Farbe $s \notin \{c(v_1), \ldots, c(v_n)\}$ gefärbt.

- *Es gilt $c(F_j) \neq s$:* Sei $y_j \in \{x_j, \neg x_j\}$ das Literal, so dass $c(y_j) = c(v_j)$ und $c(\neg y_j) = s$ gilt.

  Es gibt insgesamt $2n$ Literale x_i bzw. $\neg x_i$. Da jede Klausel F_j drei Literale enthält, ist F_j zu den anderen $2n-3$ Literalen benachbart. Wegen $n \geq 4$ ist $2n - 3 \geq \frac{n}{2} + 1$, also gibt es zu jedem F_j mindestens ein i so, dass F_j zu y_i und $\neg y_i$ benachbart ist. Wegen $c(\neg y_i) = s$ folgt $c(F_j) \neq s$.

- *Enthält jedes F_j ein Literal y_j, so ist der Graph $n+1$-färbbar:* Nun enthalte F_j ein Literal y_i. Dann ist F_j nicht benachbart zu y_i.

  Des Weiteren ist F_j mit keinem Literal x der Farbe $c(y_i)$ benachbart. Denn sei x ein zu F_j benachbartes Literal, so gibt es ein k, so dass $x \in \{x_k, \neg x_k\} = \{y_k, \neg y_k\}$. Falls $x = y_k$, so ist $i \neq k$ und es gilt $c(x) = c(y_k) = c(v_k) \neq c(v_i) = c(y_i)$. Falls $x = \neg y_k$, so ist $c(x) = c(\neg y_k) = s \neq c(y_i)$.

  Daher ist F_j zu keinem Literal mit der Farbe $c(y_i)$ benachbart. Da F_j ausschließlich mit Literalen benachbart ist, hat kein Nachbarknoten von F_j die Farbe $c(y_i)$. Somit kann F_j die Farbe $c(F_j) := c(y_i)$ zugeordnet werden.

- *Enthält ein F_j kein Literal y_i, so ist der Graph nicht $n+1$-färbbar:* Enthält F_j kein Literal y_i, so ist F_j zu allen Literalen $y_1, \ldots, y_n$ benachbart, es muss also $c(F_j) \notin \{c(y_1), \ldots, c(y_n)\} = \{c(v_1), \ldots, c(v_n)\}$ gelten. Da jedoch schon $c(F_j) \neq s$ gilt, wird für F_j eine weitere $n+2$-te Farbe benötigt.

- *Es gilt 3-SAT$\prec_p$COL:* Der Graph G ist also genau dann mit $n+1$ Farben färbbar, wenn jede Klausel von F mindestens ein Literal y_i, also ein Literal x mit einer Farbe $c(x) \neq s$ enthält.

  Der Graph sei $n+1$-färbbar. Dann können die Literale x mit $c(x) \neq s$ mit wahr belegt werden und F ist erfüllt.

  Sei F erfüllbar. Dann sei $z_i \in \{x_i, \neg x_i\}$ das mit wahr belegte Literal. Nun enthält jede Klausel F_j ein solches Literal z_i. Dann definiere $c : V \mapsto \{1, \ldots, n+1\}$ durch

  $$\begin{aligned} &c(\neg z_i) := 0, \qquad i = 1, \ldots, n \\ &c(z_i) = c(v_i) := i, \qquad i = 1, \ldots, n \\ &c(F_j) := i \text{ für ein in } F_j \text{ enthaltenes Literal } z_i \end{aligned}$$

  Dann ist c eine Färbung mit $n+1$ Farben.

  Ist nun bekannt, ob der Graph mit $n+1$ Farben färbbar ist, so lässt sich unmittelbar entscheiden, ob F erfüllbar ist. Daher ist 3-SAT$\prec_p$COL.

Zu COL $\in$ NP: Offenbar kann der folgende nicht-deterministische Algorithmus in polynomieller Zeit eine positive Antwort liefern, wenn $G = (V, E)$ mit $n+1$ Farben färbbar ist.

- Rate eine Belegung $c(v)$ der Knoten V mit Werten aus $\underline{n}$.
- Gibt es zwei benachbarte Knoten v, w mit $c(v) = c(w)$, gebe $false$ zurück.
- Ansonsten gebe *true* zurück.

Könnte $\chi(G)$ in polynomieller Zeit bestimmt werden, so könnte in polynomieller Zeit bestimmt werden, ob $\chi(G) \leq n$ ist. Also wäre das Entscheidungsproblem in polynomieller Zeit lösbar, somit wegen der NP-Vollständigkeit P = NP. Eine Färbung mit minimaler Farbanzahl kann nicht in polynomieller Zeit berechnet werden, da sonst auch $\chi(G)$ in polynomieller Zeit berechnet werden könnte. □

Auf Grund der in Satz 3.1.5 gezeigten Beziehung von Knoten- und Kantenfärbung ergibt sich aus Satz 3.1.11 das folgende Korollar.

Korollar 3.1.12 (Kantenfärbung ist NP-vollständig)**.** *Das Entscheidungsproblem, ob ein Graph G mit n Farben knotenfärbbar ist, ist NP-vollständig.*

Insbesondere gibt es – falls $P \neq NP$ – keinen polynomiellen Algorithmus zur Bestimmung von $\chi'(G)$ und somit auch keinen polynomiellen Algorithmus zur Bestimmung einer Kantenfärbung mit minimaler Farbanzahl.

Bemerkung 3.1.13. Über Satz 3.1.11 hinaus haben Garey und Johnson bereits 1976 in [GJ76] gezeigt, dass – falls es einen polynomiellen Algorithmus gibt, der zu jedem Graphen eine Färbung mit höchstens $2\chi(G)$ Farben erzeugt – auch einen polynomiellen Algorithmus zur Bestimmung von $\chi(G)$ gibt. Insofern ist es sogar NP-vollständig, zu jedem Graphen eine Färbung mit maximal $2\chi(G)$ Farben zu finden.

In Abschnitt 3.2.2 wird diese Aussage weiter verschärft.

3.1.2 Abschätzungen für den chromatischen Index

Die Bestimmung des chromatischen Index eines Graphen ist NP-vollständig. Dennoch gibt es einige Abschätzungen für den chromatischen Index. Die folgenden Sätze stammen aus [Die06], [CH94] und [Vol91]

Satz 3.1.14. *Für jeden schlingenfreien Graphen G gilt $\chi(G) \leq 1 + \max\{\delta(H) | H \subset G\}$.*

Satz 3.1.15 (Brooks 1941)**.** *Sei G ein schlingenfreier Graph, dann gilt $\chi(G) \leq \Delta(G) + 1$.*

Satz 3.1.16 (Brooks 1941)**.** *Sei G ein zusammenhängender schlingenfreier Graph, der weder vollständig noch ein Kreis ungerader Länge ist, so gilt $\chi(G) \leq \Delta(G)$.*

Diese Abschätzungen können jedoch sehr grob sein. So gibt es 4-färbbare Graphen mit beliebig großem Maximalgrad.

Beispiel 3.1.17 (Färbung von Radgraphen)**.** Für $n \geq 3$ ist der Radgraph $W_n = (V_n, E_n)$ gegeben durch (vgl. Abbildung 3.1)

$$V_n = \{w, v_1, \ldots, v_n\}$$
$$E_n = \{v_1v_2, \ldots, v_{n-1}v_n, v_nv_1\} \cup \{wv_i : i = 1 \ldots, n\}.$$

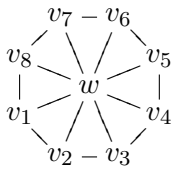


Abbildung 3.1: Der Radgraph W_8

Dann ist offenbar $\Delta(G) = d(w) = n$. Jedoch gilt

- Für gerade n ist $c : V \mapsto \{0, 1, 2\}$ gegeben durch

$$\begin{aligned} &c(w) = 0, \\ &c(v_1) = c(v_3) = \cdots = c(v_{n-1}) = 1, \\ &c(v_2) = c(v_4) = \cdots = c(v_n) = 1 \end{aligned}$$

  eine Färbung für W_n mit 3 Farben.

- Für ungerade n ist $c : V \mapsto \{0, 1, 2, 3\}$ gegeben durch

$$\begin{aligned} &c(w) = 0, \\ &c(v_1) = c(v_3) = \cdots = c(v_{n-2}) = 1, \\ &c(v_2) = c(v_4) = \cdots = c(v_{n-1}) = 2, \\ &c(v_n) = 3 \end{aligned}$$

  eine Färbung für W_n mit 4 Farben.

Es ist also für $n \geq 3$ jedes W_n 4-färbbar, jedoch gilt $\Delta(W_n) = n$.

Besser sind die Abschätzungen für den kantenchromatischen Index.

Satz 3.1.18 (Vinzig 1941)**.** *Sei $G = (V, E, g)$ ein Multigraph. Dann gilt*

$$\Delta(G) \leq \chi'(G) \leq \Delta(G) + \mu(G).$$

Insbesondere gilt für einfache Graphen

$$\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1.$$

Jedoch ist es bereits für einfache Graphen NP-vollständig zu bestimmen, ob $\Delta(G) = \chi'(G)$ oder $\Delta(G) = \chi'(G)+1$ gilt. Für bestimmte Unterklassen – insbesondere für bipartite Graphen – ist das Problem einfacher.

Satz 3.1.19. *Sei $G = (V, E, g)$ ein bipartiter Multigraph. Dann gilt $\chi'(G) = \Delta(G)$.*

In Abschnitt 3.4.3 werden effiziente Algorithmen zur exakten Kantenfärbung bipartiter Multigraph aufgeführt.

3.2 Übersicht über Färbungsalgorithmen

3.2.1 Übersicht

Obwohl es, falls P $\neq$ NP ist, nach Satz 3.1.11 und Korollar 3.1.12 keinen polynomiellen Algorithmus zur Bestimmung einer optimalen Knoten- bzw. Kantenfärbung geben kann, werden Algorithmen zur Graphenfärbung benötigt.

Prinzipiell unterscheidet man zwischen Heuristiken, die effizient sind, aber nicht immer eine optimale Färbung erzeugen und exakten Algorithmen, die in nicht-polynomieller Laufzeit eine exakte Färbung erzeugen. Während bei den exakten Algorithmen einzig die Laufzeit entscheidend ist, können bei den Heuristiken verschiedene Schwerpunkte gesetzt werden. Die heuristischen Algorithmen können sowohl auf ihre Laufzeit optimiert werden, als auch daraufhin, dass sie zumindest möglichst oft möglichst gute Färbungen erzeugen oder aber eine gewisse Approximationsgüte für die Färbung garantieren.

Definition 3.2.1 (Approximationsgüte)**.** *Ein Optimierungsproblem ist gegeben durch:*

- *Eine Menge* Π *von möglichen Eingaben*
- *Eine Größenfunktion* $N : \Pi \mapsto \mathbb{N}$
- *Eine Menge* $Sol(\pi)$ *von Lösungen zu jeder Eingabe* $\pi \in \Pi$
- *Eine Auswertungsfunktion* $t : Sol(\pi) \mapsto \mathbb{R}$ *zu jeder Eingabe* $\pi \in \Pi$

Das Optimum OPT *ist gegeben durch*

$$OPT(\pi) = \begin{cases} \inf\{t(s) : s \in Sol(\pi)\} & \text{bei Minimierungsproblemen} \\ \sup\{t(s) : s \in Sol(\pi)\} & \text{bei Maximierungsproblemen} \end{cases}$$

Sei A *ein Algorithmus der Funktion* $A : \Pi \mapsto \mathbb{R}$*. Dann wird die Approximationsgüte* κ_A *von* A *definiert durch*

$$\kappa_A(n) = \begin{cases} \sup\left\{\frac{t(A(\pi))}{OPT(\pi)} : \pi \in \Pi, N(\pi) = n\right\} & \text{bei Minimierungsproblemen} \\ \sup\left\{\frac{OPT(\pi)}{t(A(\pi))} : \pi \in \Pi, N(\pi) = n\right\} & \text{bei Maximierungsproblemen} \end{cases}$$

Oftmals wird für die Approximationsgüte nur eine obere Schranke in der Art $\kappa_A(n) \in \mathcal{O}(f(n))$ *oder* $\kappa_A(n) \leq f(n)$ *angegeben.*

Das Färbungsproblem lässt sich formal als Minimierungsproblem auffassen.

- Die Menge Π entspricht der Menge der Graphen.
- Die Menge $Sol(\pi)$ ist die Menge der gültigen Färbungen des Graphen π.
- Die Auswertungsfunktion t ordnet einer Färbung $s \in Sol(\pi)$ die Anzahl der verwendeten Farben zu.
- Das Optimum $OPT(\pi)$ ist die Anzahl der Farben einer Färbung mit minimaler Farbanzahl, also ist $OPT(\pi) = \chi(\pi)$ bei Knotenfärbung bzw. $OPT(\pi) = \chi'(\pi)$ bei Kantenfärbung.

Lediglich die Größenfunktion variiert. Für einen Graphen $\pi = G = (V, E)$ ist z.B. $N(G) = |V|$, $N(G) = |V| + |E|$ oder $N(G) = \Delta(G)$ möglich. Insofern wird die Approximationsgüte auch als Funktion von Π aufgefasst.

Definition 3.2.2 (Approximationsgüte von Färbungsalgorithmen). *Sei $c_A(G)$ die von A für einen Graphen G erzeugte Färbung. Dann heißt die Funktion κ_A*

$$\kappa_A(n) = \begin{cases} \sup\left\{\frac{|c_A(G)|}{\chi(G)} : N(G) = n\right\} & \text{für Knotenfärbung} \\ \sup\left\{\frac{|c_A(G)|}{\chi'(G)} : N(G) = n\right\} & \text{für Kantenfärbung} \end{cases}$$

Approximationsgüte *von A. Oft schreiben wir auch nur κ statt κ_A.*

Die Approximationsgüte misst, wie schlecht das Verhältnis der Anzahl der Farben des von A erzeugten Algorithmus zur Anzahl der Farben einer optimalen Färbung ist. Die obige Definition erlaubt, die Approximationsgüte als Funktion des Graphen anzugeben.

3.2.2 Knotenfärbung

Für Algorithmen zur Graphenfärbung eines Graphen $G = (V, E)$ gibt es unterschiedliche Ansätze. Aufbauend auf diesen gibt es sowohl exakte als auch heuristische Färbungsalgorithmen.

Sequentielle Färbung: Die Knoten werden in einer Reihenfolge $v_1, \ldots, v_n$ durchlaufen. Dann wird dem Knoten v_k basierend auf der bereits erzeugten partiellen Färbung der Knoten $v_1, \ldots, v_{k-1}$ eine Farbe zugeordnet, die noch kein Nachbarknoten von v_k hat.

Sequentielle Färbung mit Umfärben: Dieses Verfahren entspricht im Wesentlichen der sequentiellen Färbung. Sobald jedoch für ein Knoten eine neue Farbe eingeführt werden muss, wird versucht, eine bichromatische Zusammenhangskomponente so umzufärben, dass keine neue Farbe eingeführt werden muss.

Unabhängige Mengen: Es wird eine disjunkte Überdeckung der Knotenmenge V mit unabhängigen Mengen $V_1, \ldots, V_k$ gesucht. Dabei heißt eine Menge *unabhängig*, wenn sie keine benachbarten Knoten enthält. Anschließend werden alle Knoten einer Klasse der Überdeckung mit derselben Farbe gefärbt.

Genetische Algorithmen: In den letzten Jahren wurden vermehrt genetische Algorithmen zur Graphenfärbung entwickelt. Diese sollen in dieser Arbeit jedoch nicht betrachtet werden.

3.2.2.1 Überblick über Heuristische Algorithmen

Falls P $\neq$ NP, so besagt Bemerkung 3.1.13, dass kein polynomieller Algorithmus eine Approximationsgüte $\kappa \leq 2$ haben kann. Darüber hinaus zeigten Lund und Yannakakis 1993 in [LY93], dass es ein $\delta > 0$ gibt, so dass es keinen polynomiellen Algorithmus mit einer Approximationsgüte $\kappa < |V|^\delta$ gibt.[1]

[1] In [Has96] wird gezeigt, dass – falls NP $\neq$ ZPP gilt – diese Aussage für jedes $\delta < 1$ erfüllt ist, d.h. dass kein polynomieller Algorithmus mit einer Approximationsgüte $\kappa < |V|^{1-\varepsilon}$ für ein $\varepsilon > 0$ existiert.

Eine triviale obere Schranke für die Approximationsgüte ist $\kappa = |V|$, da $\chi(G) \geq 1$ und $|c_A(G)| \leq |V|$ ist. Die derzeit beste garantierte Approximationsgüte eines polynomiellen Algorithmus liefert Halldórson in [Hal93]. Er stellt einen auf unabhängigen Mengen basierenden Algorithmus Halldorson-Color vor mit einer Approximationsgüte

$$\kappa \in \mathcal{O}\left(\frac{|V|(\log\log|V|)^2}{(\log|V|)^3}\right).$$

Es existiert also eine große Lücke zwischen der Approximationsgüte, die derzeit mit polynomiellen Algorithmen erreicht wird und der Approximationsgüte, von der bekannt ist, dass sie NP-hart ist.

In der Praxis sind dennoch oft Ergebnisse empirischer Untersuchungen entscheidend. So bieten Algorithmen (z.B. DSATUR) einen guten Kompromiss aus schneller Laufzeit und relativ guter Färbung, obwohl sie lediglich die triviale Approximationsgüte $\kappa \in \mathcal{O}(|V|)$ garantieren.

In Abschnitt 3.3.1 werden einige auf sequentieller Färbung beruhende Heuristiken vorgestellt.

3.2.2.2 Überblick über Exakte Algorithmen

Exakte Algorithmen zur Graphenfärbung können – falls P $\neq$ NP – keine polynomielle Laufzeit haben. Die derzeit beste garantierte Laufzeit

$$\mathcal{O}\left(\left(\frac{4}{3}+\frac{3^{\frac{4}{3}}}{4}\right)^{|V|}\right) \approx \mathcal{O}(2,4150^{|V|})$$

liefert der in 2003 in [Epp03] vorgestellte, auf unabhängigen Mengen basierende Algorithmus Eppstein-Color. Interessant ist, dass dieser Algorithmus zuerst in der oben genannten Komplexität den chromatischen Index $\chi(G)$ des Graphen berechnet und anschließend aus den berechneten Daten in $\mathcal{O}(2^{|V|})$ eine explizite Färbung berechnet. Dieser Algorithmus hat eine sehr gute garantierte Worst-Case-Laufzeit, jedoch tritt diese auch nahezu immer ein. Somit sind in der Praxis oft andere Algorithmen besser.

Eine große Klasse exakter Färbungsalgorithmen basieren auf der sequentiellen Färbung. Diese probieren letztendlich systematisch alle möglichen Abbildungen auf eine optimale Färbung durch. Jedoch können oft große Teile des Suchbaums früh ausgeschlossen werden, so dass diese Algorithmen für kleine Graphen üblicherweise schnell arbeiten. Der erste solche Algorithmus wurde von Brown 1972 in [Bro72] vorgestellt. In Abschnitt 3.3.2 gehen wir näher auf exakte sequentielle Färbung ein.

Eine weitere, oft effiziente Methode zur exakten Graphenfärbung basiert ebenfalls auf unabhängigen Mengen. Das Modell unabhängiger Mengen wird in ein lineares ganzzahliges Optimierungsproblem ILP übersetzt, so dass das ILP genau dann optimal ist, wenn die zugehörige Färbung exakt ist. Da dieses ILP sehr viele Variablen hat, benötigt man zur Lösung die Technik der Spaltengenerierung. Einen Überblick über dieses Verfahren wird in Abschnitt 3.3.4 gegeben.

3.2.3 Kantenfärbung

Viele algorithmische Ansätze zur Knotenfärbung lassen sich auf Kantenfärbung übertragen. Im Wesentlichen gibt es die folgenden Algorithmen:

Sequentielle Färbung: Die Knoten werden in einer Reihenfolge $e_1, \ldots, e_n$ durchlaufen. Dann wird der Kante e_k basierend auf der bereits erzeugten partiellen Färbung der Kanten $e_1, \ldots, e_{k-1}$ eine Farbe zugeordnet, die noch kein Nachbarknoten von e_k hat.

Unabhängige Mengen: Es wird eine disjunkte Überdeckung der Kantenmenge E in Matchings $E_1, \ldots, E_k$ gesucht. Dabei heißt eine Kantenmenge *Matching*, wenn sie keine adjazenten Kanten enthält. Anschließend werden alle Kanten einer Klasse der Überdeckung mit derselben Farbe gefärbt.

Genetische Algorithmen: Auch für die Kantenfärbung gibt es genetische Algorithmen. Diese sollen jedoch wie bei der Knotenfärbung nicht Gegenstand dieser Arbeit sein.

Für die Kantenfärbung gibt es wichtige Resultate, auf die in Abschnitt 3.4 näher eingegangen wird.

- Ein Multigraph lässt sich in einer Laufzeit von $\mathcal{O}(|E|(\Delta(G) + |V|))$ mit maximal $\lfloor \frac{11}{10} \cdot \chi'(G) + \frac{4}{5} \rfloor$ Farben färben[2]. Insbesondere gibt es einen polynomiellen Algorithmus mit Approximationsgüte $\frac{4}{3}$.
- Ein Multihypergraph – und somit auch ein Multigraph – mit beschränkter Knotenanzahl lässt sich in $\mathcal{O}(|E|)$ exakt färben.
- In $\mathcal{O}(|V| + |E|)$ lässt sich eine optimale Kantenfärbung für einen bipartiten Graphen erzeugen.

3.3 Algorithmen zur Knotenfärbung

3.3.1 Heuristische sequentielle Färbung

Definition 3.3.1 (Partielle Färbung, Partielle Färbung der Ordnung p)**.** *Sei $G = (V, E)$ ein Graph und $W \subset V$. Eine Abbildung $c : W \mapsto S$ heißt* partielle Knotenfärbung *auf W, wenn c eine Knotenfärbung auf dem Untergraphen $G[W]$ ist, d.h. wenn keine zwei benachbarten Knoten aus W dieselbe Farbe zugeordnet bekommen. Die Abbildung heißt* partielle k-Knotenfärbung*, wenn $|S| = k$ ist.*

Für eine partielle Färbung auf W schreiben wir auch oft c_W. Ist $v_1, \ldots, v_n$ eine vorgegebene Anordnung der Knoten, so schreiben wir auch $c_p := c_{\{v_1,\ldots,v_p\}}$ und nennen c_p eine partielle Färbung der Ordnung p.

[2] Für eine reelle Zahl x ist $\lfloor x \rfloor := \max\{n \in \mathbb{Z} : n \leq c\}$ die größte ganze Zahl, die kleiner oder gleich x ist.

Definition 3.3.2 (Nachbarfarben eines Knotens)**.** *Zu einer partiellen Färbung c für W definieren wir die Menge der (verschiedenen) Nachbarfarben eines Knotens v als*

$$Nc_W(v) = \{c(w) : w \in W \cap N(v)\}.$$

Ist c eine partielle Färbung der Ordnung p, so schreiben wir $Nc_p(v)$. Ist die Menge W offensichtlich, so schreiben wir schlicht $Nc(v)$.

3.3.1.1 Allgemeine sequentielle Färbung

Viele Färbungsalgorithmen basieren auf sequentieller Färbung. Bei der sequentiellen Färbung werden die Knoten in einer bestimmten Reihenfolge durchlaufen und jedem Knoten wird eine Farbe zugewiesen, die im weiteren Verlauf nicht mehr verändert wird. Sequentielle Färbung ist somit ein Greedy-Algorithmus und wird auch oft als Greedy-Färbung bezeichnet.

Die unterschiedlichen sequentiellen Algorithmen unterscheiden sich in der Reihenfolge, in der die Knoten durchlaufen werden und der Wahl der Farbe für diesen Knoten. Der Pseudocode der allgemeinen sequentiellen Färbung steht in Algorithmus 1. Allerdings ist der Algorithmus nicht eindeutig spezifiziert, da die Reihenfolge, in der die Knoten durchlaufen werden, nicht festgelegt ist.

Algorithmus 1 : **Greedy-Seq-Color**
Sequentielle Färbung eines Graphen

Parameter : G: Ein Graph $G = (V, E)$ als Adjazenzmatrix oder Adjazenzliste.
Rückgabewert : Eine heuristische Färbung für den Graphen G.
Komplexität : $\mathcal{O}(|E| + |V|)$ falls G als Adjazenzliste vorliegt,
$\mathcal{O}(|V|^2)$ falls G als Adjazenzmatrix vorliegt.

```
1 Funktion Greedy-Seq-Color(G: Graph)
2     c : V ↦ ℕ0 ∪ {∞} ;                          // Färbungsabbildung als Array
3     foreach v ∈ V do c(v) := ∞ ;                // Zu Beginn ist jeder Knoten ungefärbt
4     foreach v ∈ V do
5         c(v) := min(ℕ \ Nc_p(v)) ;              // Färbung mit der kleinsten freien Farbe
6     return c;
7 end
```

Satz 3.3.3 (Sequentielle Färbung)**.** *Der Algorithmus Greedy-Seq-Color (siehe Algorithmus 1) erzeugt eine Knotenfärbung c mit maximal $\Delta(G) + 1$ Farben. Jeder Knoten hat eine Farbe $c(v) \leq d(v)$. Erfolgt die Auswahl des Knotens in Zeile 4 in konstanter Zeit, so hat der Algorithmus eine Komplexität von $\mathcal{O}(|E| + |V|)$, falls der Graph als Adjazenzliste vorliegt und $\mathcal{O}(|V|^2)$, falls der Graph als Adjazenzmatrix vorliegt.*

Beweis. Sei $v_1, \ldots, v_n$ die Reihenfolge, in der die Knoten durchlaufen werden.

Nach dem k-ten Schritt ist c eine partielle Färbung der Ordnung k und für jeden Knoten v_i, $i \leq k$ gilt $c(v_i) \leq d(v_i)$.

Induktionsanfang $k = 1$: Nach dem ersten Schritt ist c offenbar eine partielle 1-Färbung der Ordnung 1 mit $c(v_1) = 0 \leq d(v_1)$.

Induktionsschritt $k \to k+1$: Sei die Behauptung nach dem k-ten Schritt erfüllt. Dann gilt nach Induktionsvoraussetzung

$$\text{Für alle benachbarten } v_i, v_j \text{ mit } 1 \leq i < j \leq k: \qquad c(v_i) \neq c(v_j)$$

und wegen der Wahl von $c(v_{k+1})$

$$\text{Für alle zu } v_{k+1} \text{ benachbarten } v_i \text{ mit } 1 \leq i \leq k: \qquad c(v_i) \neq c(v_{k+1}),$$

also folgt:

$$\text{Für alle benachbarten } v_i, v_j \text{ mit } 1 \leq i < j \leq k+1: \qquad c(v_i) \neq c(v_j).$$

Somit ist nach dem $k+1$-ten Schritt c eine partielle Färbung der Ordnung $k+1$.

Des Weiteren gilt für alle $i \leq k$ nach Induktionsvoraussetzung $c(v_i) \leq \Delta(G)$. Nun ist v_{k+1} benachbart zu $d(v_{k+1})$ Knoten, somit ist $c(v_{k+1}) = \min(\mathbb{N}_0 \setminus Nc_p(v)) \leq d(v_{k+1}) \leq \Delta(G)$.

Nach dem n-ten Schritt ist also c eine Färbung für G mit Farben aus $\{0, \ldots, \Delta(G)\}$ und für alle v_i gilt $c(v_i) \leq d(v_i)$.

Zum Schluss ist somit c eine Färbung für V und für jeden Knoten $v_i \in V$ gilt $c(v_i) \leq d(v_i) \leq \Delta(G)$ und somit $c(v_i) \in \{0, \ldots, \Delta(G)\}$.

Zur Komplexität: Die Schleife wird für jeden Knoten $v \in V$ – also $|V|$ mal – durchlaufen. Zeile 5 lässt sich für v wie folgt implementieren.

1. Lege ein Array *used* mit den Feldern $[0, \ldots, d(v)]$ an und initialisiere es mit *false*.
2. Durchlaufe alle Nachbarknoten w von v und setze $used[c(w)]$ auf *true*, falls $c(w) \leq d(v)$.
3. Durchlaufe das Array von und setze $c(v)$ auf den kleinsten Wert s, für den $a[s] = false$ ist. Dieser existiert, da nach der vorherigen Überlegung $c(v_i) \leq d(v_i)$ ist.

Wurde der Graph in einer Adjazenzliste gespeichert, können diese Schritte für jeden Knoten v in $\mathcal{O}(|d(v)|)$ ausgeführt werden. Im Falle einer Adjazenzmatrix benötigt das Durchlaufen aller Nachbarknoten $\mathcal{O}(|V|)$ Schritte. Summiert über alle Knoten ergibt sich eine Gesamtkomplexität von

$$\text{für Adjazenzliste:} \qquad \sum_{v \in V} \mathcal{O}(d(v)) = \mathcal{O}\left(\sum_{v \in V} d(v)\right) = \mathcal{O}(2|E|) = \mathcal{O}(|E|),$$

$$\text{für Adjazenzmatrix:} \qquad \sum_{v \in V} \mathcal{O}(|V|) = |V| \cdot |V| = |V|^2.$$

Insgesamt ergibt sich somit eine Komplexität von $\mathcal{O}(|E| + |V|)$ bzw. $\mathcal{O}(|V|^2)$. □

Das Ergebnis der sequentiellen Färbung hängt stark von der Reihenfolge ab, in der die Knoten durchlaufen werden. So gibt es immer eine Knotenreihenfolge, bei der Greedy-Seq-Color eine optimale Färbung liefert. Nach [Tur04, Kapitel 5.1] gilt:

Satz 3.3.4 (Sequentielle Färbung liefert für eine Knotenreihenfolge optimale Färbung)**.** *Sei $G = (V, E)$ ein Graph. Dann gibt es eine Knotenreihenfolge $v_1, \ldots, v_n$ von V so, dass die sequentielle Färbung eine Färbung mit $\chi(G)$ Farben liefert.*

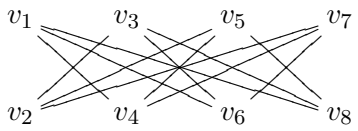


Abbildung 3.2: Für diesen bipartiten Graphen G_8 erzeugt Greedy-Seq-Color eine 4-Färbung.

Beweis. Es gibt eine Knotenfärbung $c : V \mapsto \underline{\chi(G)}$. Man ordne die Knoten $v_1, \ldots, v_n$ so, dass $0 = c(v_1) \leq c(v_2) \leq \cdots \leq c(v_n) = \chi(G) - 1$ ist. Dann gilt für die von Greedy-Seq-Color erzeugte Färbung c':

Für alle $v \in V$ gilt $c'(v) \leq c(v)$.

Induktionsanfang $i = 1$: Es ist

$$c'(v_1) := \min(\mathbb{N}_0 \setminus \{c'(x) : x \text{ ist Nachbar von } v_1\}) = \min(\mathbb{N}_0 \setminus \{\infty\}) = 0 \leq c(v_1).$$

Induktionsschritt $i - 1 \to i$: Sei die Bedingung für $1, \ldots, i-1$ erfüllt. Für alle zu v_i benachbarten Knoten v_j gilt nach Definition der Färbung $c(v_j) \neq c(v_i)$. Somit folgt für alle zu v_i benachbarten Knoten v_j mit $j < i$

$$c'(v_j) \leq c(v_j) < c(v_i),$$

also folgt

$$\begin{aligned} c'(v_i) :=& \min(\mathbb{N}_0 \setminus \{c'(x) : x \text{ ist Nachbar von } v_i\}) \leq \min(\mathbb{N}_0 \setminus \{\underbrace{c'(v_1), \ldots, c'(v_{i-1})}_{\in\{0,\ldots,c(v_i)-1\}}, \infty\}) \\ &\leq \min(\mathbb{N}_0 \setminus \{0, \ldots, c(v_i) - 1\}) = c(v_i). \end{aligned}$$

Somit ist c' eine Färbung und für alle i gilt $c'(v_i) \leq c(v_i) \leq c(v_n) = \chi(G) - 1$, also benötigt c' maximal $\chi(G)$ Farben. □

Bei ungeschickter Wahl der Knotenreihenfolge kann die erzeugte Färbung jedoch sehr viel mehr Farben als $\chi(G)$ benötigen. Das folgende an [CH94, Übung 6.2.1] angelehnte Beispiel zeigt, dass die Approximationsgüte κ nicht besser als $\frac{|V|}{2}$ ist.

Beispiel 3.3.5 (Sequentielle Färbung)**.** Sei für $n = 2m$ der Graph $G_n = (V_n, E_n)$ gegeben durch (vgl. Abbildung 3.2)

$$\begin{aligned} V_n &= \{v_1, \ldots, v_n\} \\ E_n &= \{v_{2i+1}v_{2j+2} : i, j = 0, \ldots, m-1,\ i \neq j\} \end{aligned}$$

Durchläuft der Algorithmus Greedy-Seq-Color die Knoten aus V in der natürlichen Reihenfolge $v_1, \ldots, v_n$, so entsteht die m-Färbung $c : V \mapsto \underline{m}$ mit $c(2i+1) = c(2i+2) = i$, $i = 0, \ldots, m-1$.

Denn im ersten Schritt ($i = 0$) sind noch keine Nachbarknoten von v_1 gefärbt, also wird

$$\begin{aligned} c(v_1) :=& \min(\mathbb{N}_0 \setminus \{c(x) : x \text{ ist Nachbar von } v_1\}) = \min(\mathbb{N}_0 \setminus \{c(v_4), c(v_6), \ldots, c(v_n)\}) \\ &= \min(\mathbb{N}_0 \setminus \{\infty\}) = \min(\mathbb{N}_0) = 0 \end{aligned}$$

und entsprechend gilt für v_2

$$\begin{aligned} c(v_2) &:= \min(\mathbb{N}_0 \setminus \{c(x) : x \text{ ist Nachbar von } v_2\}) = \min(\mathbb{N}_0 \setminus \{c(v_3), c(v_5), \ldots, c(v_{n-1})\}) \\ &= \min(\mathbb{N}_0 \setminus \{\infty\}) = \min(\mathbb{N}_0) = 0. \end{aligned}$$

Sei die Bedingung bis v_{2k} erfüllt. Dann gilt

$$\begin{aligned} c(v_{2k+1}) &:= \min(\mathbb{N}_0 \setminus \{c(x) : x \text{ ist Nachbar von } v_{2k+1}\}) = \min(\mathbb{N}_0 \setminus \{c(v_{2j+2}) : j \neq k\}) \\ &= \min(\mathbb{N}_0 \setminus \{c(v_2), \ldots, c(v_{2(k-1)+2}), c(v_{2(k+1)+2}), \ldots, c(v_{2m})\}) \\ &= \min(\mathbb{N}_0 \setminus \{0, \ldots, k-1, \infty\}) = k \\ c(v_{2k+2}) &:= \min(\mathbb{N}_0 \setminus \{c(x) : x \text{ ist Nachbar von } v_{2k+2}\}) = \min(\mathbb{N}_0 \setminus \{c(v_{2j+1}) : j \neq k\}) \\ &= \min(\mathbb{N}_0 \setminus \{c(v_1), \ldots, c(v_{2(k-1)+1}), c(v_{2(k+1)+1}), \ldots, c(v_{2m-1})\}) \\ &= \min(\mathbb{N}_0 \setminus \{0, \ldots, k-1, \infty\}) = k \end{aligned}$$

und somit die Aussage bis v_{2k+2}.

Induktiv folgt die Behauptung.

Für $n \to \infty$ konvergiert der Erwartungswert der Anzahl der verwendeten Farben der Greedy-Färbung gemittelt über alle Knotenanordnungen bei dieser Graphenfamilie nach [Big90] jedoch gegen 2. Allerdings gibt es Graphenfamilien, so dass der Greedy-Algorithmus für fast alle Knotenanordnungen schlechte Färbungen findet.

3.3.1.2 Sequentielle Färbung mit Umfärbung

In [MMI72, Abschnitt 2] wird eine Methode zur Verringerung der Farben bei sequentieller Färbung angegeben. Bei dieser Methode wird – falls für einen Knoten eine neue Farbe eingeführt werden muss – versucht, die bereits gefärbten Knoten so umzufärben, dass keine neue Farbe nötig ist.

Definition 3.3.6 (s,t-bipatiter Untergraph, s,t-Komponente)**.** *Sei $G = (V, E)$ ein Graph und $c : V \mapsto S$ eine Knotenfärbung. Seien $s, t \in S$ zwei verschiedene Farben und $V_s = c^{-1}(s)$, $V_t = c^{-1}(t)$ die Knoten mit der Farbe s bzw. t. Dann heißt $G[V_s \cup V_t]$ der s,t-*bipartite Untergraph *von G und eine Zusammenhangskomponente von $G[V_s \cup V_t]$ heißt s,t-*Komponente.

Satz und Definition 3.3.7 ($s \leftrightarrow t$-Umfärbung)**.** *Sei $G = (V, E)$ ein Graph und $c : V \mapsto S$ eine Knotenfärbung. Sei $H = (V_H, E_H)$ eine s,t-Komponente. Dann ist c' definiert durch*

$$c'(v) = \begin{cases} c(v) & \text{falls } v \notin V_H \\ s & \text{falls } v \in V_H \wedge c(v) = t \\ t & \text{falls } v \in V_H \wedge c(v) = s \end{cases}$$

eine Knotenfärbung. Diese entsteht durch Vertauschen der Farben s und t auf einer s,t-Komponente und heißt durch $s \leftrightarrow t$-Umfärbung auf H entstandene Färbung.

Beweis. Seien v, w zwei benachbarte Knoten. Falls $v, w \notin V_H$ oder $v, w \in V_H$ so ist $c'(v) = c'(w) \iff c(v) = c(w)$, also $c'(v) \neq c'(w)$. Sei nun o.B.d.A. $v \in V_H$ und $w \notin V_H$. Falls $c(w) = s$ oder $c(w) = t$, so wäre w in einer anderen s,t-Zusammenhangskomponente, könnte also nicht mit v benachbart sein. Somit ist $c'(w) = c(w) \notin \{s,t\}$ und $c'(v) \in \{s,t\}$, also ist $c'(v) \neq c'(w)$. In jedem Fall ist für benachbarte Knoten v, w auch $c'(v) \neq c'(w)$, somit ist c' eine Färbung. □

Die Idee der sequentiellen Färbung mit Umfärben ist nun die folgende: Sei c eine partielle k-Färbung der Ordnung $p-1$. Kann nun v_p mit einer der Farben aus $\underline{k}$ gefärbt werden, so färbt man v_p wie in der normalen sequentiellen Färbung mit der kleinstmöglichen Farbe, d.h. ist $Nc_{p-1}(v_p) \neq \underline{k}$, so setzt man $c(v_p) = \min(\mathbb{N}_0 \setminus Nc_{p-1}(v_p))$.

Kann hingegen v_p mit keiner der Farben aus $\underline{k}$ gefärbt werden, so ist v_p mit mindestens einem Knoten jeder Farbe benachbart. Sei T die Menge der Farben, mit der genau ein Nachbarknoten von v_p gefärbt ist. Es gebe nun Farben $s, t \in T$ mit $s \neq t$, so dass eine s,t-Komponente von $G[\{v_1, \ldots, v_{p-1}\}]$ genau einen zu v_p adjazenten Knoten w hat, o.B.d.A. mit $c(w) = s$. Dann führt man eine $s \leftrightarrow t$-Umfärbung auf dieser s,t-Komponente durch. Da w der einzige zu v_p adjazente Knoten mit $c(w) = s$ war und nun die Farbe t hat, gibt es keinen zu v_p benachbarten Knoten mit Farbe s mehr, und somit kann v_p mit der Farbe s gefärbt werden.

Gibt es hingegen keine solche s,t-Komponente für Farben $s, t \in T$, so muss v_p mit der neuen Farbe k gefärbt werden.

Satz 3.3.8 (Sequentielle Färbung mit Umfärben)**.** *Die Auswahl in Zeile 5 des in Algorithmus 2 dargestellten Algorithmus* Greedy-SeqI-Color *erfolge in konstanter Zeit. Dann erzeugt der Algorithmus eine Knotenfärbung c in $\mathcal{O}(\chi(G)|E|) \subset \mathcal{O}(\Delta(G)|E|) \subset \mathcal{O}(|V||E|)$*[3]

Beweis. Der Beweis der Korrektheit verläuft im Wesentlichen analog zum Beweis für normale sequentielle Färbung. Sei $v_1, \ldots, v_n$ die Reihenfolge, in der die Knoten durchlaufen werden. Dann ist nach dem k-ten Schritt c eine partielle $m+1$-Färbung der Ordnung k.

Induktionsanfang $k = 1$: Nach dem ersten Schritt ist das die Behauptung trivialerweise erfüllt. Denn es ist $j = m = 0$ und somit c mit $c(v_1) = 0$ eine partielle 1-Färbung.

Induktionsschritt $k \mapsto k+1$: Sei die Behauptung nach dem k-ten Schritt erfüllt.

Fall 1: Nach Zeile 6 ist $j \leq m$. Dann erfolgt die Wahl von $c(v_{k+1})$ wie bei der gewöhnlichen sequentiellen Färbung und m bleibt unverändert, somit ist c nach dem Schleifendurchlauf eine partielle $m+1$-Färbung der Ordnung $k+1$.

Fall 2: Nach Zeile 6 ist $j = m+1$.

Fall 2a: Die Bedingung in Zeile 10, dass eine solche Austauschkomponente H existiert, ist wahr. Dann wird eine $s \leftrightarrow t$-Umfärbung dieser Komponente H vorgenommen. Nach Satz 3.3.7 bleibt c eine partielle $m+1$-Färbung der Ordnung k. Falls w die Farbe s hatte und nun $c(w) = t$ gilt, existiert nach der Umfärbung kein zu v_{k+1} benachbarter Knoten der Farbe s mehr. Nach der Zuweisung $c(v_{k+1}) := s$

[3] In einigen Quellen wird behauptet, dass dieser Algorithmus lineare Zeit benötigt. Dieses ist in dieser allgemeinen Form nicht korrekt. Vielmehr basiert die Idee des Umfärbens auf *Kempe-Ketten*, welche in einer Beweisidee des Vier-Farben-Satzes verwendet wurden. Dort wird der Algorithmus für planare Graphen verwendet und es ist immer $\chi(G) \leq 4$ (bzw. zur Zeit der Idee $\chi(G) \leq 5$) gesichert.

Algorithmus 2 : **Greedy-SeqI-Color**
Sequentiellen Färbung eines Graphen mit Umfärben

Parameter : G: Ein Graph $G = (V, E)$ als Adjazenzliste.
Rückgabewert : Eine heuristische Färbung für den Graphen G.
Komplexität : $\mathcal{O}(\chi(G)|E|) \subset \mathcal{O}(\Delta(G)|E|) \subset \mathcal{O}(|V||E|)$.

```
1  Funktion Greedy-SeqI-Color(G: Graph)
2      c : V ↦ ℕ ∪ {∞} ;                                  // Färbungsabbildung als Array
3      foreach v ∈ V do c(v) := ∞ ;                  // Zu Beginn sind alle Knoten ungefärbt
4      m := −1 ;                                          // Größte bisher verwendete Farbe
5      foreach v ∈ V do
6          k := min(ℕ_0 \ Nc(v)) ;                          // Kleinste freie Farbe für v
7          if k ≤ m then c(v) = k ;                   // Keine neue Farbe wird benötigt
8          else
9              Suche Farben s, t ∈ T so, dass es eine s,t-Komponente H mit nur einem zu v
               benachbarten Knoten gibt;
10             if es gibt eine solche Komponente H then
11                 Definiere w als der zu v benachbarte Knoten in H;
12                 foreach x ∈ H do                                  // s ↔ t-Umfärbung von H
13                     if c(x) = s then c(x) := t else c(x) := s;
14                 if c(w) = s then c(v) := t else c(v) := s ;                  // Färben von v
15             else                          // v muss mit einer neuen Farbe gefärbt werden
16                 c(v) := k;
17                 m := k
18     return c;
19 end
```

in Zeile 14 ist somit c eine partielle $m+1$-Färbung der Ordnung k. Falls w die Farbe t hatte, ist die Überlegung analog.

Fall 2b: Die Bedingung in Zeile 10, dass eine solche Austauschkomponente H existiert, ist falsch. Dann wird $c(v_{k+1})$ wie bei der gewöhnlichen sequentiellen Färbung definiert. Nach Zeile 16 ist c wegen $j = m+1$ also eine partielle $m+2$-Färbung der Ordnung $k+1$. Wegen der Definition von $m := j$ in der darauf folgenden Zeile gilt somit die Behauptung.

Eine Implementierung in der angegebenen Laufzeit steht in [SDK83, S. 415ff]. □

Im Gegensatz zur einfachen sequentiellen Färbung ist Greedy-Seql-Color für bipartite Graphen exakt, insbesondere auch für den in Abbildung 3.2 dargestellten Graphen. Obwohl der Algorithmus im Durchschnitt bessere Ergebnisse als Greedy-Seq-Color liefert, gibt es Graphen, für den er schlechtere Färbungen erzeugt (Vgl. [SDK83, S. 415]). In [Joh74] zeigte Johnson mit dem folgenden Beispiel, dass die Approximationsgüte dennoch nicht besser als $\frac{|V|}{3}$ sein kann.

Beispiel 3.3.9 (Sequentielle Färbung mit Umfärben)**.** Sei für $n = 3m$, $m \in \mathbb{N}$ der Graph $G_n = (V_n, E_n)$ gegeben durch (vgl. Abbildung 3.3)

$$V_n = \{u_i, v_i, w_i : 0 \leq i \leq m-1\}$$
$$E_n = \{a_i b_j, a_i c_j, b_i c_j : i \neq j\}$$

Dann ist $\chi(G_n) = 3$, denn offenbar ist G_n für $n \geq 3$ nicht bipartit und $c : V_n \mapsto \{0,1,2\}$, $c(a_i) = 0$, $c(b_i) = 1$, $c(c_i) = 2$ ist eine 3-Färbung für G_n. Färbt Greedy-Seql-Color die Knoten jedoch in der Reihenfolge $a_0, b_0, c_0, a_1, b_1, c_1, \ldots, a_{m-1}, b_{m-1}, c_{m-1}$, so benötigt er $m = \frac{|V|}{3}$ Farben.

Ist c' die vom Algorithmus erzeugte Färbung, dann ist $c'(a_k) = c'(b_k) = c'(c_k) = k$ und zu keinem Zeitpunkt ein Umfärben möglich. Dieses lässt sich induktiv begründen: Offenbar färbt der Algorithmus a_0, b_0 und c_0 mit der Farbe 0. Soll nun a_{k+1} gefärbt werden, so ist nach Induktionsvoraussetzung $c'(a_i) = c'(b_i) = c'(c_i) = i$ für $0 \leq i \leq k$. Dann ist für je zwei Farben $s, t \leq k$ der s-t-bipartite Untergraph gerade

$$G_{s,t}(\{b_s, b_t, c_s, c_t\}, \{b_s c_t, b_t c_s\}).$$

Dieser besteht aus den beiden von $\{b_s, c_t\}$ bzw. $\{b_t, c_s\}$ induzierten Zusammenhangskomponenten. Da a_{k+1} zu jedem der Knoten b_s, b_t, c_s, c_t benachbart ist, hat keine Zusammenhangskomponente von $G_{s,t}$ nur einen zu a_{k+1} benachbarten Knoten, somit ist eine $s \leftrightarrow t$-Umfärbung nicht möglich. Daher muss a_{k+1} mit der Farbe $k+1$ gefärbt werden.

Die Überlegungen für b_{k+1} und c_{k+1} sind analog.

3.3.1.3 Methoden zur Wahl der Knotenreihenfolge

Viele effiziente Algorithmen basieren auf einer geschickten Wahl der Knotenreihenfolge bei der sequentiellen Färbung. Sie lassen sich durch die Technik des Umfärbens weiter Verbessern.

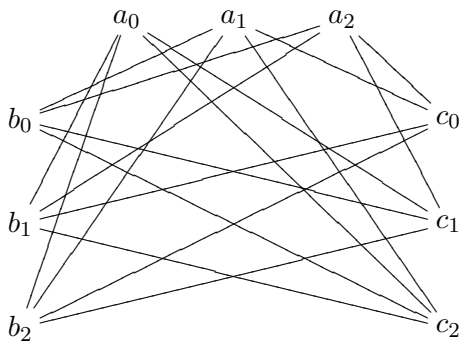


Abbildung 3.3: Für diese 3-färbbare Graphenfamilie G_{3m} erzeugt Greedy-Seql-Color m-Färbungen

Bei der Wahl der Knotenreihenfolge unterscheidet man zwischen *dynamischer* und *statischer* Ordnung. Bei der statischen Ordnung wird zu Beginn die Reihenfolge der Knoten festgelegt, anschließend werden die Knoten gefärbt. Bei der dynamischen Ordnung wird während der Laufzeit auf Basis der bisher erzeugten partiellen Färbung der nächste Knoten gewählt. Generell lässt sich mittels dynamischer Ordnung eine bessere Färbung erzeugen, da für die Wahl des nächsten Knotens mehr Informationen zur Verfügung stehen.

Nun werden verschiedene Möglichkeiten der Sortierung kurz vorgestellt.

Zufällige Ordnung: Die Knoten werden zu Beginn des Algorithmus zufällig geordnet. Die Ordnung kann in $\mathcal{O}(|V|)$ bestimmt werden und ist statisch.

Largest First: Die Knoten werden so geordnet, dass die Folge $d(v_1), \ldots, d(v_n)$ der Knotengrade monoton fallend ist. Dieses entspricht dem Welsch-Powell-Algorithmus und wurde zuerst 1967 in [WP67] vorgestellt. Die Ordnung kann in $\mathcal{O}(|V| \log |V|)$ berechnet werden und ist statisch.

Greedy Largest First: v_1 ist der Knoten mit maximalem Grad und die anderen Knoten werden so sortiert, dass v_i mindestens so viele Nachbarknoten in $v_1, \ldots, v_{i-1}$ hat wie jeder Knoten $v_{i+1}, \ldots, v_n$. Diese Ordnung kann nach [KJ85] in $\mathcal{O}(\min(|V|^2, |E| \log |V|))$ bestimmt werden und ist statisch.

Smallest Last: Die Knoten werden so geordnet, dass $d(v_i)$ minimal in $G[\{v_1, \ldots, v_i\}]$ ist. Dieser Algorithmus wurde zuerst 1972 in [MMI72] vorgestellt. Die Ordnung kann nach [MB83] in $\mathcal{O}(|E| + |V|)$ bestimmt werden und ist statisch.

Saturation Largest First: Im ersten Schritt wird der Knoten mit maximalem Grad gefärbt. In jedem Schritt wird der ungefärbte Knoten gewählt, dessen Sättigungsgrad maximal ist. Der Sättigungsgrad eines Knotens v ist die Anzahl der unterschiedlich gefärbten, zu v adjazenten Knoten (siehe Definition 3.3.10). Der Algorithmus wurde zuerst 1979 als DSATUR in [Bré79] vorgestellt. Die Ordnung ist dynamisch.

Diese Algorithmen wurden sowohl theoretisch als auch empirisch untersucht. Für alle möglichen Knotenordnungen gibt es sowohl mit als auch ohne Umfärben 3-färbbare Graphen, für die die sequentielle Färbung $\mathcal{O}(|V|)$ Farben benötigt (für Beispiele zur Smallest Last Ordnung mit und ohne Umfärben siehe [Joh74]). Als bester Algorithmus dieser Klasse gilt der Algorithmus DSATUR. Daher soll dieser im folgenden Abschnitt genauer untersucht werden.

3.3.1.4 Saturation Largest First – DSATUR und DSATUR mit Umfärben

Definition 3.3.10 (Sättigungsgrad). *Sei $G = (V, E)$ ein Graph, $W \subset V$, $c : W \mapsto S$ eine partielle Färbung und $v \in V$ ein Knoten. Der Sättigungsgrad $d_s(v)$ von v bezüglich c ist definiert als die Anzahl an unterschiedlichen Farben von gefärbten Nachbarknoten von v, d.h. es ist $d_s(v) := |Nc_W(v)|$.*

Der Saturation-Largest-First-Algorithmus, zuerst von D. Brélaz als DSATUR in [Bré79] veröffentlicht, wählt in jedem Schritt einen noch nicht gefärbten Knoten, dessen Sättigungsgrad (engl. Saturation) bezüglich der bisherigen Färbung maximal ist. Teilweise wird zusätzlich gefordert, dass bei gleichem Sättigungsgrad mehrerer Knoten der Knoten mit maximalem Sättigungsgrad und größtmöglichem Grad gewählt wird.

Die Idee ist, dass der Knoten mit dem höchsten Sättigungsgrad in einem gewissen Sinne am schwierigsten zu färben ist. In jedem Schritt wird also versucht, das schwierigste Problem zu lösen, bevor für dieses durch evtl. willkürliche, schlechte Färbung einfach zu färbender Nachbarknoten weitere Bedingungen entstehen.

Algorithmus 3 : **DSATUR**
Sequentielle Färbung mit Saturation-Largest-First-Ordnung

Parameter : G: Ein Graph $G = (V, E)$ als Adjazenzliste.
Rückgabewert : Eine heuristische Färbung für den Graphen G.
Komplexität : $\mathcal{O}(\min(|V|^2, |E| \log |V|))$.

```
1  Funktion DSATUR(G: Graph)
2      c : V ↦ ℕ₀ ∪ {∞} ;                          // Färbungsabbildung als Array
3      foreach v ∈ V do c(v) := ∞ ;        // Zu Beginn ist jeder Knoten ungefärbt
4      W := V ;                  // W ist die Menge der bereits gefärbten Knoten
5      while W ≠ ∅ do
6          if V = W then                                  // Erster Schleifendurchlauf
7              Wähle v ∈ W mit maximalem Grad d(v);
8          else                                          // Weitere Schleifendurchläufe
9              Wähle v ∈ W mit maximalem Sättigungsgrad d_s(v);
10         c(v) := min(ℕ₀ \ {c(x) : x ist Nachbar von v}) ;          // Färben von v
11         W := W \ {v};
12     return c;
13 end
```

Satz 3.3.11 (DSATUR). *Der in Algorithmus 3 beschriebene Algorithmus DSATUR färbt die Knoten eines Graphen in $\mathcal{O}(\min(|V|^2, |E| \log |V|))$.*

Beweis. Der Algorithmus DSATUR erzeugt eine Färbung, da er eine Form der sequentiellen Färbung ist. Die Laufzeit folgt nach [KJ85, S. 415]. □

Nach [Bré79] ist DSATUR exakt für bipartite Graphen. Dennoch können 3-färbbare Graphen schlecht gefärbt werden. Das folgende Beispiel wurde in leicht abgewandelter Form von Spinnrad und Yijayan 1985 in [SV85] vorgestellt.

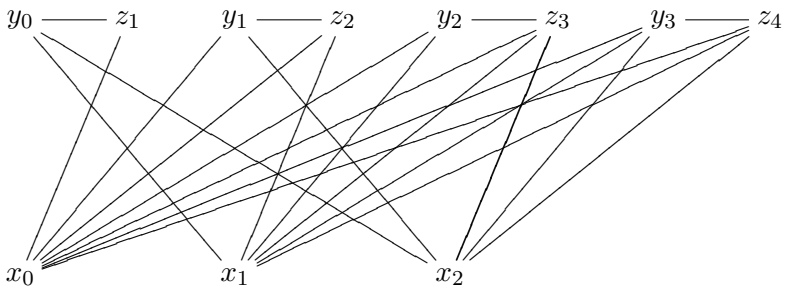


Abbildung 3.4: Für diese 3-färbbare Graphenfamilie G_n erzeugt DSATUR n-Färbungen.

Beispiel 3.3.12. Sei für $n \in \mathbb{N}$ der Graph $G_n = (V_n, E_n)$ gegeben durch (vgl. Abbildung 3.4)

$$V_n = X_n \cup Y_n \cup Z_n, \qquad E_n = E_n^1 \cup E_n^2 \cup E_n^3$$

mit

$$\begin{aligned} X_n &= \{x_0, \ldots, x_{n-2}\}, & E_n^1 &= \{y_i z_{i+1} : 0 = 1, \ldots, n-1\}, \\ Y_n &= \{y_0, \ldots, y_{n-1}\}, & E_n^2 &= \{x_i y_j : i \neq j\}, \\ Z_n &= \{z_1, \ldots, z_n\}, & E_n^3 &= \{x_i z_j : i < j\}. \end{aligned}$$

Dann ist $\chi(G_n) = 3$, denn zwischen Knoten innerhalb einer der Mengen X_n, Y_n, Z_n existiert keine Kante, somit ist $c : V_n \mapsto \{0, 1, 2\}$, $c(x_i) = 0$, $c(y_i) = 1$, $c(z_i) = 2$ eine 3-Färbung für den Graphen G_n.

Der Graph G_n hat $3n - 1$ Knoten. Für den Grad der Knoten ergibt sich:

$$d(x_i) = 2n - 1 - i \qquad d(y_i) = \begin{cases} n & \text{falls } i = n-1 \\ n-1 & \text{sonst} \end{cases} \qquad d(z_i) = \begin{cases} i & \text{falls } i = n \\ i+1 & \text{sonst} \end{cases}$$

Verzichtet man im Algorithmus DSATUR auf die Bedingung, dass im ersten Schritt und bei gleichem Sättigungsgrad der Knoten mit maximalen Grad gewählt wird, so ist $s := y_0, z_1, x_0, y_1, z_2, x_1, \ldots, x_{n-2}, y_{n-1}, z_n$ eine zulässige Knotenreihenfolge für den (modifizierten) DSATUR und die Knoten x_i, y_i und z_i werden mit der Farbe i gefärbt. Insgesamt wird also eine $n + 1$-Färbung erzeugt.

- Fügt man nun die Bedingung hinzu, dass der DSATUR mit dem Knoten größten Grades beginnt, so muss y_0 der Knoten mit höchstem Grad sein. Es ist $d_{G_n}(y_0) = n - 1$ und $\Delta(G_n) = d_{G_n}(x_0) = 2n - 1$. Daher fügt man zu dem Graphen weitere n Knoten $\{w_1, \ldots, w_n\}$ und Kanten $\{w_1 y_0, \ldots, w_n y_0\}$ hinzu. Sei G'_n der so entstandene Graph. Dieser hat $4n - 1$ Knoten. Dann bleibt $\chi(G'_n) = 3$, da die obige 3-Färbung c für G_n durch $c(w_i) = 0$ für alle $i = 1, \ldots, n$ auf G'_n erweitert werden kann. Außerdem ist y_0 wegen $d_{G'_n}(y_0) = 2n - 1$ ein Knoten maximalen Grades und somit $s, w_1, \ldots, w_n$ eine zulässige Knotenreihenfolge. Für diese erzeugt DSATUR eine $n + 1$-Färbung.
- Wird hingegen die Auswahlregel so verschärft, dass DSATUR in jedem Schritt bei gleichem Sättigungsgrad den Knoten mit maximalem Grad wählt, so muss der Graph wie folgt erweitert werden: Es werden $n + 1$ Knoten $W_n = \{w_1, \ldots, w_{n+1}\}$ und $2n - 2$

Knoten $U_n = \{u_1, \ldots, u_{2n-2}\}$ hinzugefügt. Dann werden Kanten $E'_n = \{y_i w_j : (i,j) \neq (n-1, n+1)\}$ und $E''_n = \{z_i u_j : j = 1, \ldots, 2n-1-i\} \cup \{z_n u_n\}$ hinzugefügt. Der so entstandene Graph G''_n hat $6n-2$ Knoten und obige Färbung c kann durch $c(w_i) := 1$, $c(u_i) := 0$ zu einer 3-Färbung für G'' erweitert werden. Des Weiteren wurden die Kanten E'_n bzw. E''_n gerade so gewählt, dass jeder Knoten aus Y_n bzw. Z_n den Grad $2n$ hat. Es lässt sich zeigen, dass $s, w_1, \ldots, w_{n+1}, u_1, \ldots, u_{2n-2}$ eine für DSATUR gültige Knotenordnung ist. Daher erzeugt der Algorithmus wie oben eine $n+1$-Färbung.

Somit lässt sich eine Graphenfamilie G_n mit $\mathcal{O}(n)$ Knoten finden, für die DSATUR $n+1$ Farben benötigt. Daher ist die Approximationsgüte des Algorithmus nicht besser als $\mathcal{O}(|V|)$.

Auch der DSATUR-Algorithmus lässt sich durch bichromatisches Umfärben zu DSATURI erweitern. Nach [KJ85, S. 415] hat DSATURI eine Laufzeit von $\mathcal{O}(|V||E|)$.

3.3.2 Exakte sequentielle Färbung

Auch wenn – falls P $\neq$ NP ist – kein exakter polynomieller Algorithmus für Färbung existiert, ist exakte Färbung oft wichtig. Im Folgenden sollen sequentielle Backtracking-Algorithmen zur exakten Färbung von Graphen vorgestellt werden.

3.3.2.1 Einfacher rekursiver Algorithmus

Wegen $\chi(G) \leq |V|$ genügt es, für alle möglichen $c : V \mapsto \underline{|V|}$ zu testen, ob c eine Färbung ist. Alle Färbungen mit minimaler Farbanzahl sind optimal. In der Praxis geht es nun darum, nur einen möglichst kleinen Teilraum des Suchraums (der Größe $|V|^{|V|}$) zu durchsuchen, der mit Sicherheit eine optimale Färbung enthält. Es geht also darum, möglichst große Bereiche des Suchraums möglichst früh auszuschließen.

Das algorithmische Vorgehen ist dabei folgendermaßen: Die Knoten werden in einer Reihenfolge $v_1, \ldots, v_n$ durchlaufen. Ordne für jede partielle Färbung der Ordnung p und dem Knoten v_{p+1} alle möglichen Farben $s \in \underline{|V|}$ zu, so dass eine partielle Färbung der Ordnung $p+1$ entsteht. Verfahre mit allen diesen partiellen Färbungen der Ordnung $p+1$ entsprechend. Auf diese Art wird der Suchraum baumartig durchlaufen.

Nun gibt es offensichtliche Überlegungen, wann bestimmte partielle Färbungen bzw. die von diesen partiellen Färbungen ausgehenden Teile des Suchbaums nicht weiter untersucht werden müssen:

- Aus einer partiellen Färbung der Ordnung p entsteht genau dann eine partielle Färbung der Ordnung $p+1$, wenn v_{p+1} eine Farbe zugeordnet wird, die noch kein Nachbarknoten von v_{p+1} hat.
- Sollte eine partielle Färbung bereits mindestens so viele Farben benötigen, wie die bisher beste gefundene Färbung, so kann in diesem Teil des Suchbaums keine bessere Färbung existieren.
- Lemma 3.3.13 besagt, dass es – von einer partiellen k-Färbung der Ordnung p ausgehend – für v_{p+1} genügt, alle möglichen Farben aus $\underline{k+1}$ zu testen.

- Wurde zu Beginn eine untere Grenze lb für $\chi(G)$ bestimmt (z.B. die Größe einer maximalen Clique), so kann der Algorithmus abbrechen, sobald eine Färbung mit lb Farben gefunden wurde.

Das folgende Lemma ist anschaulich plausibel und wird daher in den Quellen nicht begründet. Da es für die meisten exakten Algorithmen jedoch wesentlich ist, soll es in dieser Arbeit formal bewiesen werden.

Lemma 3.3.13. *Sei $G = (V, E)$ ein Graph. Dann gibt es für jede Reihenfolge $v_1, \ldots, v_n$ der Knoten eine optimale Färbung $c : V \mapsto \underline{\chi(G)}$ mit $c(v_i) \leq \max\{c(v_1), \ldots, c(v_{i-1})\} + 1$. Insbesondere ist $c(v_i) < i$.*

Beweis. Sei $m = \chi(G)$ und $c' : V \mapsto \underline{m}$ eine optimale Färbung für G. Wir zeigen, dass es eine Permutation $\sigma : \underline{m} \mapsto \underline{m}$ gibt, so dass $c := \sigma \circ c'$ die Behauptung erfüllt.

Man definiert für jede Permutation $\pi : \underline{m} \mapsto \underline{m}$

$$p(\pi) = \max\{k : \text{Für } i \leq k \text{ gilt } \pi(c(v_i)) \leq \max\{\pi(c(v_1)), \ldots, \pi(c(v_{i-1}))\} + 1\}.$$

Dann wähle man diejenige Permutation σ, für die $p(\sigma)$ maximal ist.

Falls $p(\sigma) = n$ ist, so erfüllt σ die Behauptung.

Ansonsten sei $k := p(\sigma)$. Nun wird eine Permutation σ' mit $p(\sigma') > k$ konstruiert. Da dieses ein Widerspruch zur Wahl von σ ist, kann dieser Fall nicht eintreten.

Zur Konstruktion: Definiere $M := \max\{\sigma \circ c(v_1), \ldots, \sigma \circ c(v_k)\}$.

- *Konstruktion von σ':* Nach der Wahl von k ist $M + 1 < \sigma \circ c(v_{k+1})$, also $\sigma \circ c(v_{k+1}) \geq M + 2$. Da c eine optimale Färbung ist, ist c surjektiv. Wegen der Surjektivität von σ gibt es somit ein v_l mit $\sigma \circ c(v_l) = M + 1$. Sei τ die Transposition von $\sigma \circ c(v_{k+1})$ und $\sigma \circ c(v_l)$. Definiere dann $\sigma' := \tau \circ \sigma$.

- *Es ist $M = \max\{\sigma'(c(v_1)), \ldots, \sigma'(c(v_k))\}$:* Für $i \leq k$ gilt $c(v_i) \notin \{c(v_l), c(v_{k+1})\}$, denn sonst wäre
$$\sigma \circ c(v_i) \in \{\sigma \circ c(v_l), \sigma \circ c(v_{k+1})\} \implies \sigma \circ c(v_i) > M,$$
was ein Widerspruch zur Wahl von M ist. Also gilt
$$\forall i \leq k : \sigma'(c(v_i)) = \tau \circ \sigma(c(v_i)) = \sigma(c(v_i))$$
und insbesondere $M = \max\{\sigma'(c(v_1)), \ldots, \sigma'(c(v_k))\}$.

- *Es ist $p(\sigma') > k$:* Es gilt für alle $i \leq k$
$$\begin{aligned}\sigma'(c(v_i)) &= \sigma(c(v_i)) \\ &\leq \max\{\sigma(c(v_1)), \ldots, \sigma(c(v_{i-1}))\} + 1 \\ &= \max\{\sigma'(c(v_1)), \ldots, \sigma'(c(v_{i-1}))\} + 1\end{aligned}$$
und weiter
$$\sigma'(c(v_{k+1})) = \sigma(c(v_l)) = M + 1 = \max\{\sigma'(c(v_1)), \ldots, \sigma'(c(v_k))\} + 1.$$
Somit ist $p(\sigma') \geq k + 1 > k$.

Dieses ist wegen $p(\sigma') > p(\sigma)$ ein Widerspruch zur Wahl von σ.

Sei nun c eine optimale Färbung mit $c(v_i) \leq \max\{c(v_1), \ldots, c(v_{i-1})\} + 1$. Dann gilt $c(v_1) = 0 < 1$ und induktiv

$$c(v_i) \leq \max\{c(v_1), \ldots, c(v_{i-1})\} + 1 < (i-1) + 1 = i.$$

□

Aus den obigen Überlegungen folgt der in Algorithmus 4 dargestellte rekursive Prototyp des sequentiellen Backtracking-Algorithmus.

Satz 3.3.14. *Der Algorithmus Recursive-Seq-Color erzeugt eine exakte Färbung in $\mathcal{O}(|V|!)$.*

Beweis. Der Algorithmus durchsucht alle möglichen Abbildungen und wählt die beste Färbung aus. Dabei werden nur Abbildungen ausgeschlossen, von denen die bereits geführten Überlegungen sichergestellt haben, dass sie keine optimale Färbung sein können. Insofern befindet sich unter den durchsuchten Abbildungen eine optimale Färbung, diese findet der Algorithmus. Da insgesamt maximal die $|V|!$ verschiedenen Abbildungen $c : V \mapsto \mathbb{N}_0$ mit $c(v_i) \in \underline{i}$ untersucht werden, wird der Funktionskörper von rekColor maximal $|V|!$ mal durchlaufen. Somit folgt eine Komplexität von $\mathcal{O}(|V|!)$. □

3.3.2.2 Verfeinerung der Sequentiellen Färbung

Brown stellte 1972 in [Bro72] einen zu Recursive-Seq-Color ähnlichen, iterativen Algorithmus vor. Dieser funktioniert folgendermaßen: Ausgehend von einer Ordnung der Knoten von G werden abwechselnd Vorwärts- und Rückwärtsoperationen durchgeführt.

In den Vorwärtsoperationen werden der Reihe nach die Knoten gefärbt, bis entweder alle Knoten gefärbt sind oder für einen Knoten keine Färbung möglich ist. Falls alle Knoten gefärbt sind, so versucht der Algorithmus im Folgenden eine Färbung mit weniger Farben zu finden. Daher kann es auch eintreten, dass ein Knoten nicht gefärbt werden kann.

In der Rückwärtsoperation wird der letzte Knoten gesucht, der mit einer anderen Farbe gefärbt werden kann. Von diesem Knoten aus wird wieder die Vorwärtsoperation begonnen. Wird irgendwann der erste Knoten in der Rückwärtsoperation erreicht, so terminiert der Algorithmus, da eine Umfärbung des ersten Knotens keine bessere Färbung ermöglicht.

Der originale Algorithmus von Brown wurde in den folgenden Jahren weiter verbessert. Eine erste Verbesserung lieferte Brown in [Bro72] selbst. Weitere (fehlerhafte) Algorithmen kamen 1975 von Christofides in [Chr75] und 1979 von Brelaz in [Bré79]. Der letzte Algorithmus wurde 1983 von Peemöller in [Pee83] korrigiert. Eine allgemeine Darstellung und Zusammenfassung der entwickelten Algorithmen gaben Kubale und Jackowski in [KJ85]. Diese Darstellung entspricht Algorithmus 5.

In dieser allgemeinen Darstellung wurden gegenüber Browns Algorithmus sowohl die Vorwärtsoperation als auch die Rückwärtsoperation weiter verbessert. Dabei ist lb eine untere und ub eine obere Grenze für den chromatischen Index und r der größte Wert, so dass die Knoten $v_1, \ldots, v_{r-1}$ mit Farben aus $\underline{ub-1}$ gefärbt werden können.

Algorithmus 4 : **Recursive-Seq-Color**
Exakte rekursive sequentielle Färbung

Parameter : G: Ein Graph $G = (V, E)$ als Adjazenzliste.
Rückgabewert : Eine exakte Färbung für den Graphen G.
Komplexität : $\mathcal{O}(|V|!)$.

```
1  Funktion Recursive-Seq-Color(G: Graph)
2      c : V ↦ ℕ0 ∪ {∞} ;                          // Abbildung für die Färbung
3      opt := beliebige Färbung für G ;            // Bisher beste gefundene Färbung
4      lb := untere Grenze für χ(G) ;              // Z.B. eine maximale Clique
5      rekColor(V);
6      return c;
7  end
```

Globale Variablen : c: partielle Färbung
opt: bisher beste gefundene Färbung
lb: untere Grenze für $\chi(G)$
Parameter : V_r: Menge der noch nicht gefärbten Knoten

```
8  Funktion rekColor(V_r: Knotenmenge)
9      if |opt| > lb then      // Die bisher beste Färbung muss nicht optimal sein
10         if V_r = ∅ then opt := c;                // Alle Knoten gefärbt
11         else
12             Wähle v ∈ V_r;
13             q := Anzahl der verwendeten Farbe der partiellen Färbung c;
               // Generiere rekursiv alle aufbauenden partiellen Färbungen
14             foreach s ∈ (q + 1) ∩ (|opt| − 1) \ Nc(v) do
15                 c(v) := s;
16                 rekColor(V_r \ {v});
17                 c(v) := ∞;
18 end
```

Algorithmus 5 : **Iterative-Seq-Color**
Exakte iterative sequentielle Färbung

Parameter : G: Ein Graph $G = (V, E)$ als Adjazenzliste.
Rückgabewert : Eine exakte Färbung für den Graphen G.
Komplexität : $\mathcal{O}((v(|V|) + r(|V|)) \cdot |V|!)$, falls $forward \in \mathcal{O}(v(|V|))$ und $backward \in \mathcal{O}(r(|V|))$.

```
1  Funktion Iterative-Seq-Color(G: Graph)
2      r := 1 ;                                   // Index des zu färbenden Knotens
3      c : {1,...,n} ↦ ℕ_0 ;                      // Aktueller partieller Schedule
4      opt : {1,...,n} ↦ ℕ_0 ;                    // Der bisher beste Schedule
5      cp := ∅ ;                                  // Menge der Current Predecessors von v_k
6      for i := 1,...,n do fc_i := ∅ ;            // fc_i ist Menge der für v_i zulässigen Farben
7      lb := untere Grenze für die Farbanzahl;
8      ub := obere Grenze für die Farbanzahl + 1;
9      Wähle Knotenreihenfolge v_1,...,v_n;
10     while true do
11         r := forward(r);
12         if ub = lb then return c ;             // Optimale Färbung wurde gefunden
13         r := backward(r);
14         if r ≤ 1 then return c ;               // Beim ersten Knoten angelangt
15 end
16 Funktion forward(r: Integer)
17     for i := r,...,n do
18         Evtl: Bestimme neue Ordnung v_i,...,v_n;
19         Bestimme fc_i ;                        // Menge der möglichen Farben für v_i
20         if fc_i = ∅ then return i ;            // v_i kann nicht gefärbt werden
21         c(v_i) := min(fc_i) ;                  // Färbe v_i mit der kleinstmöglichen Farbe
       // Die aktuelle Färbung c ist vollständig und die bisher beste
22     opt := c;
23     ub := |opt|;
       // Versuche den letzten Knoten höchster Farbe umzufärben
24     return min{i : c(v_i) = ub − 1};
25 end
26 Funktion backward(r: Integer)
27     bestimme die Menge cp der möglichen Vorgänger;
28     while cp ≠ ∅ do
29         i := max(cp) ;                         // Versuche, den letztmöglichen Vorgänger umzufärben
30         cp := cp \ {i};
31         Evtl: Modifiziere cp;
32         fc_i := fc_i \ {c(v_i)} ;              // Färbungen mit Farbe c(v_i) schon analysiert
33         if fc_i ≠ ∅ then return i ;            // v_i kann umgefärbt werden
34     return 0;
35 end
```

In der Vorwärtsoperation wird den Knoten v_i, $i = r, r+1, \ldots, n$ sukzessive die Menge fc_i (feasible colors) der möglichen Farben für v_i zugeordnet und v_i mit der kleinsten möglichen Farbe gefärbt. Diese Operation wird beendet, wenn ein Knoten nicht mehr gefärbt werden kann (da $fc_i = \emptyset$) oder wenn der letzte Knoten gefärbt wurde ($i = n$).

Falls $fc_i = \emptyset$, so wird $r := i$ gesetzt, da die Knoten $v_1, \ldots, v_{i-1}$ mit Farben aus $\underline{ub-1}$ färbbar sind.
Falls der letzte Knoten gefärbt wurde, so wurde eine Färbung mit weniger als ub Farben gefunden. Diese ist die aktuelle beste Färbung und der Wert für ub kann verringert werden. Ist $ub = lb$, so ist der Algorithmus beendet. Ansonsten wird anschließend ebenfalls der Wert für r angepasst, d.h. es wird der erste Knoten (z.B. v_l) mit der höchsten Farbe $ub-1$ gesucht und $r := l$ gesetzt, da die Knoten $v_1, \ldots, v_{l-1}$ mit Farben aus $\underline{ub-2}$ färbbar sind.

Somit ist sichergestellt, dass nach der Vorwärtsoperation r so gewählt ist, dass $v_1, \ldots, v_{r-1}$ den Farben aus $\underline{ub-1}$ gefärbt werden können, v_r jedoch nicht mehr.

In der Rückwärtsoperation sei cp (Abkürzung für Current Predecessors) die Menge der Knoten aus $v_1, \ldots, v_{r-1}$, deren Umfärbung einen Einfluss auf die Färbung von $v_r, v_{r+1}, \ldots$ haben kann. Der Algorithmus sucht den größten solchen Knoten v_l. Ist dieses v_1, so ist der Algorithmus beendet, da ein Umfärben von v_1 keine bessere Färbung ermöglicht.

Da mit der bestehenden partiellen Färbung $c(v_1), \ldots, c(v_{l-1}), c(v_l)$ keine bessere Färbung möglich ist, wird die aktuelle Farbe $c(v_l)$ aus der Menge der möglichen Farben fc_l von v_l entfernt. Anschließend wird $r := l$ gesetzt. Dadurch wird im folgenden Vorwärtsschritt mit einer neuen Farbe für v_l begonnen.

Die expliziten Realisierungen unterscheiden sich nun in der konkreten Implementierung der folgenden Blöcke:

1. Die Initialisierung in den Zeilen 7 bis 9.
2. Die Neuordnung der Knoten in Zeile 18.
3. Die Bestimmung der möglichen Farben fc_i in Zeile 19.
4. Die Bestimmung und Veränderung der Menge cp in Zeile 27 bzw. Zeile 31.

Erst nach einer konkreten Implementierung für Blöcke lässt sich die Korrektheit beweisen und die Komplexität bestimmen. Zur Komplexität des Algorithmus folgt jedoch nach [KJ85]:

Satz 3.3.15. *Die Komplexität der Vorwärts- und Rückwärtsoperation seien Polynome $v(|V|)$ bzw. $r(|V|)$. Dann hat der Algorithmus Iterative-Seq-Color (vgl. Algorithmus 5) eine Komplexität von $\mathcal{O}((v(|V|) + r(|V|)) \cdot |V|!)$.*

Beweis. Es wird vorausgesetzt, dass der Algorithmus die in Lemma 3.3.13 nachgewiesene Eigenschaft nutzt. Dann müssen maximal $|V|!$ verschiedenen Abbildungen $c : V \mapsto \mathbb{N}_0$ mit $c(v_i) \in \underline{i}$ auf eine exakte Färbung untersucht werden. Dementsprechend wird die Schleife von Zeile 10 bis Zeile 14 maximal $|V|!$ mal durchlaufen. □

Im Folgenden werden die verschiedenen Implementierungen vorgestellt.

Browns Algorithmus

Es wird der von Brown in [Bro72] vorgestellte Algorithmus betrachtet. Viele Feinheiten (z.B. das Überspringen von Knoten in der Rückwärtsoperation oder das dynamische Umordnen der Knoten) sind in diesem Algorithmus noch nicht entwickelt.

1. Die Knoten werden nach der Greedy-Largest-First-Methode sortiert. Es wird $lb = 1$ und $ub = |V|$ gesetzt.

2. Die zu Beginn gewählte Ordnung ist statisch und bleibt erhalten.

3. Sei c die bisher berechnete partielle Färbung der Ordnung $p-1$ und $q = |c|$ die Anzahl der von c verwendeten Farben. Dann ist

$$fc_i = (\underline{q+1}) \cap (\underline{ub-1}) \setminus Nc(v_i).$$

4. Es wird $cp = \{1, \ldots, r-1\}$ gewählt.

Die Berechnung eines fc_i benötigt $d(v_i)$ Schritte, innerhalb einer Vorwärtsoperation also maximal $2|E|$ Schritte und eventuell weitere $\mathcal{O}(|V|)$ Operationen, um die neue, bessere Färbung zu speichern. Somit ergibt sich für die Vorwärtsoperation eine Komplexität von $\mathcal{O}(|V|+|E|)$. Der triviale Rückwärtsschritt erfolgt in konstanter Zeit. Daher hat der Algorithmus nach Satz 3.3.15 eine Komplexität von $\mathcal{O}((|V| + |E|) \cdot |V|!)$

Browns Algorithmus mit Look-Ahead

Brown zeigte bereits selbst in [Bro72] eine mögliche Verbesserung seines Algorithmus, indem er einen Look-Ahead-Mechanismus zur Verbesserung der Vorwärtsoperation einführte. Die Schritte 1, 2 und 4 bleiben wie in Browns Algorithmus. In Schritt 3 wird ein Look-Ahead-Mechanismus eingebracht: Im einfachen Algorithmus von Brown sind alle Farben $(\underline{q+1}) \cap (\underline{ub-1})$ zulässig, die noch kein zu v_i benachbarter Knoten hat, d.h. die nicht in $Nc(v_i)$ sind. Der Look-Ahead-Mechanismus besagt nun: Existiert ein noch nicht gefärbter Knoten v_j mit $j > i$, der nur mit einer Farbe s gefärbt werden kann, so ist s keine zulässige Farbe für v_i, da dann v_j nicht mehr gefärbt werden könnte. Für die Menge pc_i der für v_i unzulässigen Farben ergibt sich

$$pc_i := Nc(v) \cup \{s : \exists v_j \in N(v_i) \text{ mit } j > i, \text{ welches nur mit } s \text{ gefärbt werden kann}\}$$

und somit ist $fc_i = (\underline{q+1}) \cap (\underline{ub-1}) \setminus pc_i$.

Die Vorwärtsoperation hat auch mit Look-Ahead-Mechanismus eine Komplexität von $\mathcal{O}(|V|+|E|)$, jedoch ist die Laufzeit auf Grund eines größeren multiplikativen Faktors länger. Dafür werden weniger Rückwärtsoperation benötigt. Für den gesamten Algorithmus verbleibt die Laufzeit bei $\mathcal{O}((|V| + |E|) \cdot |V|!)$

Christofides Algorithmus

Christofides stellte in [Chr75] eine Idee zur Verbesserung der Rückwärtsoperation des Algorithmus von Brown vor. Sein Algorithmus ist jedoch fehlerhaft, so dass hier die korrigierte Version aus [KJ85] vorgestellt wird.

Die Schritte 1, 2 und 3 bleiben wie in Browns Algorithmus. Für eine gegebene Ordnung $v_1, \ldots, v_n$ der Knoten ist ein *monotoner Weg* von v_r nach v_j ein Weg $(v_r, v_{i_1}, \ldots, v_{i_{l-1}}, v_j)$ so dass $r < i_1 < \ldots < i_{l-1} < j$ ist. Zu jedem Knoten v_r ist die *Vorgängermenge* p_r definiert durch

$$p_r := \{v_j : \text{Es gibt einen } \textit{monotonen Weg} \text{ von } v_j \text{ nach } v_r\}.$$

Dann wird die Menge cp in Zeile 27 bestimmt durch $cp := cp \cup p_r$.

Die Vorgängermengen p_r lassen sich nach Wahl einer Knotenordnung berechnen und müssen im weiteren Verlauf nicht modifiziert werden. Zur Berechnung betrachte man den zu G knotengleichen gerichteten Graphen G', in dem jede ungerichtete Kante $\{v_i, v_j\} \in E_G$ durch die gerichtete Kante (v_i, v_j) mit $i < j$ ersetzt wird. Sei H die transitive Hülle von G'.[4] Dann ist

$$p_r = \{v_j : (v_j, v_r) \in E_H\}.$$

Somit kann die Rückwärtsoperation in $\mathcal{O}(|V|)$ implementiert werden.

Da die Vorwärtsoperation eine Laufzeit von $\mathcal{O}(|V|+|E|)$ hat, ist die Laufzeit des Algorithmus wie schon bei den vorherigen Algorithmen $\mathcal{O}((|V| + |E|) \cdot |V|!)$.

Brelaz Algorithmus

Brelaz stellte 1979 in [Bré79] einen fehlerhaften Algorithmus vor, der 1983 in [Pee83] von Peemöller korrigiert wurde. Im Wesentlichen verbesserte Brelaz die Initialisierung sowie die Rückwärtsoperation.

In der Initialisierung in Schritt 1 wird eine heuristische Färbung c' mittels DSATUR oder DSATURI mit Umfärben gesucht. Die Knotenordnung dieser heuristischen Färbung wird dann für den exakten Färbungsalgorithmus als statische Knotenordnung verwendet. Die ersten Knoten $IC := \{v_1, \ldots, v_r\}$ induzieren eine *initiale Clique*. Außerdem wird $ub := |c'|$ definiert.

Die Schritte 2 und 3 bleiben wie bei Browns Algorithmus.

Die Rückwärtsoperation in Schritt 4 wird durch eine dynamische Veränderung der Menge cp in Zeile 31 verbessert.

Sei c eine partielle Färbung der Ordnung $p-1$. Dann sei $M_p = N(v_p) \cap \{v_1, \ldots, v_{p-1}\}$ die Menge der zu v_p benachbarten Knoten mit kleinerem Index. Zu jeder Farbe $s \in Nc(v_p)$ gibt es nun eine nicht-leere Farbklassen $M_{p,s} = M_p \cap c^{-1}(s)$. Für jede dieser Farbklassen $M_{p,s}$ sei $r_s = \min(M_{p,s})$. Die Menge der Repräsentanten von M_p ist dann

$$R_p := \{r_s : s \in Nc(v_p)\} \setminus IC.$$

[4] Diese lässt sich z.B. mit dem Algorithmus von Floyd-Warschall in $\mathcal{O}(|V|^3)$ bestimmen. Alternativ gibt es effizientere Algorithmen (z.B. [Sch83])) mit einer Komplexität von $\mathcal{O}(|V||E|+|V|+|E|)$ und weit besseren praktischen Laufzeiteigenschaften.

In Zeile 27 wird nun $cp := cp \cup R_r$ definiert. Während der Rückwärtsoperation in Zeile 31 wird $cp = cp \cup R_i$ gesetzt. Dann kann die Rückwärtsoperation in $\mathcal{O}(|V| + |E|)$ implementiert werden, der gesamte Algorithmus hat also eine Laufzeit von $\mathcal{O}((|V| + |E|) \cdot |V|!)$.

3.3.3 Graphenfärbung mittels unabhängigen Mengen

Die Graphenfärbung mittels unabhängiger Mengen soll in diesem Kapitel nur skizziert werden. Insbesondere wird auf formale Beweise verzichtet.

Definition 3.3.16 (unabhängige Menge, maximal unabhängige Menge)**.** *Sei $G = (V, E)$ ein Graph. Eine Menge $W \subset V$ heißt* unabhängig in G*, wenn keine zwei Knoten aus W benachbart sind.*

Eine unabhängige Menge W heißt maximal*, wenn für alle unabhängigen Mengen W' mit $W \subset W'$ gilt: $W = W'$.*

Eine Menge W ist also genau dann maximal unabhängig, wenn jeder Knoten $v \notin W$ zu einem Knoten $w \in W$ benachbart ist. Man beachte aber, dass es sehr wohl zwei maximal unabhängige Mengen unterschiedlicher Mächtigkeit geben kann.

Nun kann man einen Graphen färben, indem man für die Knotenmenge V eine Überdeckung $(V_s)_{s \in S}$ findet. Anschließend können alle Knoten einer Menge V_s mit derselben Farbe gefärbt werden. Liegt ein Knoten v in verschiedenen Mengen V_s, so kann er mit einer beliebigen solchen Farbe gefärbt werden.

Ein einfacher, auf unabhängigen Mengen basierender heuristischer Färbungsalgorithmus ist der in Algorithmus 6 dargestellte Algorithmus Greedy-IS-Color. In jedem Schritt sucht dieser eine maximal unabhängige Menge und färbt alle Knoten mit der aktuellen Farbe. Anschließend werden die gefärbten Knoten entfernt und der Algorithmus auf den Restgraphen angewandt.

Offenbar hängt die genaue Laufzeit und Güte des in Algorithmus 6 dargestellten Algorithmus Greedy-IS-Color von der Erzeugung der unabhängigen Menge ab. Ähnlich zur sequentiellen Färbung gibt es hier viele unterschiedliche Heuristiken.

Ein einfacher Algorithmus für das Finden einer maximal unabhängigen Menge ist der in Algorithmus 7 dargestellte Algorithmus Greedy-MaxIS. Auch dort gibt es unterschiedliche Möglichkeiten zur Wahl des Knotens in Zeile 5.

Johnson schlug 1974 in [Joh74] vor, zur Auswahl der unabhängigen Menge im Algorithmus Greedy-IS-Color den Algorithmus Greedy-MaxIS zu verwenden, wobei in Zeile 5 der Knoten v gewählt wird, dessen Grad in $G[U]$ minimal ist. Dieser Algorithmus heißt Johnson-Color.

Nach [Wig83] kann Johnson-Color in $\mathcal{O}(|V|^2)$ implementiert werden und erzeugt für nichttriviale Graphen eine Färbung mit maximal $3\frac{|V|}{\log_{\chi(G)}|V|}$ Farben. Somit ist seine Approximationsgüte

$$\kappa \leq \frac{3|V|}{\log_{\chi(G)}|V|} \leq \frac{3|V|}{\log_2|V|}.$$

Algorithmus 6 : **Greedy-IS-Color**
Färbung eines Graphen mit unabhängigen Mengen

Parameter : G: Ein Graph $G = (V, E)$ als Adjazenzmatrix oder Adjazenzliste.
Rückgabewert : Eine heuristische Färbung für den Graphen G.
Komplexität : Wird Greedy-MaxIS zum Finden der unabhängigen Menge verwendet:
$\mathcal{O}(|V|^2)$, wenn der Graph als Adjazenzliste vorliegt,
$\mathcal{O}(|V|^3)$, wenn der Graph als Adjazenzmatrix vorliegt.

```
1  Funktion Greedy-IS-Color(G: Graph)
2      c : V ↦ ℕ0 ∪ {∞} ;                          // Färbungsabbildung als Array
3      foreach v ∈ V do c(v) := ∞ ;                // Zu Beginn ist jeder Knoten ungefärbt
4      i := 0 ;                                    // Farbe der aktuellen unabhängigen Menge
5      U := V ;                                    // Noch nicht gefärbte Knoten
6      while U ≠ ∅ do                              // es sind noch nicht alle Knoten gefärbt
7          W := unabhängige Menge in G[U];
8          foreach v ∈ W do c(v) := i ;            // Färbe alle Knoten aus W
9          U := U \ W ;                            // Die Knoten von W sind nun gefärbt
10         i := i + 1 ;                            // Wähle Farbe für weitere Knoten
11     return c;
12 end
```

Algorithmus 7 : **Greedy-MaxIS**
Finden einer maximal unabhängigen Menge

Parameter : G: Ein Graph $G = (V, E)$ als Adjazenzmatrix oder Adjazenzliste.
Rückgabewert : Eine maximal unabhängige Menge des Graphen G.
Komplexität : $\mathcal{O}(|V|)$, falls der Graph als Adjazenzliste vorliegt,
$\mathcal{O}(|V|^2)$, falls der Graph als Adjazenzmatrix vorliegt.

```
1  Funktion Greedy-MaxIS(G: Graph)
2      W := ∅ ;                                    // Knoten der unabhängigen Menge
3      U := V ;                                    // Mögliche zu W hinzufügbare Knoten
4      while U ≠ ∅ do                              // W ist noch nicht maximal unabhängig
5          Wähle v ∈ U;
6          W := W ∪ {v};
7          U := U \ ({v} ∪ N(v)) ;                 // v und N(v) nicht mehr zu W hinzufügbar
8      return W;
9  end
```

3.3.4 Graphenfärbung mit linearer Programmierung

3.3.4.1 Grundlagen der linearen Programmierung

Lineare Programmierung ist ein wesentliches Hilfsmittel bei vielen Optimierungsproblemen. Im Folgenden soll ein kurzer Überblick über die Grundzüge der linearen Programmierung gegeben werden. Nähere Informationen und Beweise finden sich in Lehrbüchern zur linearen Optimierung, z.B. in [IC94].

Definition 3.3.17 (Lineares Programm (LP)). *Seien $m, n \in \mathbb{N}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$ (aufgefasst als Spaltenvektoren) und $A \in \mathbb{R}^{m \times n}$. Ein lineares Programm (LP) in kanonischer Form ist ein Problem der Form*

$$\begin{aligned} &\textit{Maximiere } c^\top x \\ &\textit{mit Bedingungen } Ax \leq b, \\ &\qquad x \geq 0. \end{aligned} \tag{3.1}$$

Dabei sind Ungleichungen für Vektoren komponentenweise zu lesen.

Definition 3.3.18. *Für LPs gelten folgende Bezeichnungen:*

- *Ein LP heißt* zulässig, *wenn es ein $x \in \mathbb{R}^n$ gibt, welches die Bedingungen $Ax \leq b$, $x \geq 0$ erfüllt. Ist ein LP nicht zulässig, so heißt es* unzulässig.
- *Ein LP heißt* beschränkt, *wenn die Zielfunktion $c^\top x$ beschränkt ist, ansonsten heißt das LP unbeschränkt.*

Andere Formen von Optimierungsproblemen sind äquivalent zur kanonischen Form. Zum Beispiel lassen sich durch die Einführung von *Schlupfvariablen* die Bedingungen als Gleichungen formulieren, durch Negation der Komponenten von c entsteht ein Minimierungsproblem. Stellt man zusätzlich die Bedingung, dass $x \in \mathbb{Z}^n$ ist, so bezeichnet man das LP auch als ganzzahliges lineares Programm (ILP).

Definition 3.3.19 (Duales Optimierungsproblem). *Sei ein lineares Optimierungsproblem P* (primales Problem) *in der Form von* (3.1) *gegeben. Das zu P duale Problem D ist*

$$\begin{aligned} &\textit{Minimiere } b^\top y \\ &\textit{mit Bedingungen } A^\top y \geq c, \\ &\qquad y \geq 0. \end{aligned}$$

Satz 3.3.20 (Starker Dualitätssatz, von Neumann (1947)). *Für ein primales Problem P und das zu P gehörige duale Problem D gilt genau eine der folgenden vier Eigenschaften.*

- *Sowohl P als auch D sind zulässig und die optimalen Werte stimmen überein, d.h. es gilt*
$$\max\{c^\top x : Ax \leq b, x \geq 0\} = \min\{b^\top y : A^\top y \geq c, y \geq 0\}. \tag{3.2}$$
- *P ist unzulässig und D ist unbeschränkt.*
- *P ist unbeschränkt und D ist unzulässig.*
- *Sowohl P als auch D ist zulässig.*

3.3.4.2 Konstruktion des linearen Programms zur Graphenfärbung

Die Techniken dieses Unterkapitels stammen aus [MT96] und werden hier nur kurz zusammengefasst. Für eine ausführlichere Arbeit über das Thema Graphenfärbung mittels linearer Programmierung siehe [Tom05].

Sei S die Menge aller maximal unabhängigen Mengen in $G = (V, E)$. Nach den Überlegungen aus Abschnitt 3.3.3 ergibt sich eine Färbung unmittelbar aus einer Überdeckung $\mathcal{T}$ von V mit unabhängigen Mengen. Die Mächtigkeit von $\mathcal{T}$ entspricht dann der Anzahl der benötigten Farben. Für jedes $s \in S$ gebe es eine binäre Variable x_s, die besagt, ob die Menge s in der Überdeckung enthalten ist ($x_s = 1$) oder nicht ($x_s = 0$).

Nun ist folgendes Minimierungsproblem (UM) zu lösen:

$$\begin{aligned} &\text{Minimiere } \sum_{s \in S} x_s \\ &\text{mit Bedingungen } \forall v \in V : \sum_{s \in S, v \in s} x_s \geq 1, \\ &\qquad\qquad \forall s \in S : x_s \in \{0, 1\}. \end{aligned}$$

Die Zielfunktion berechnet die Mächtigkeit der Überdeckung, zählt also die Anzahl der benötigten Farben. Die erste Bedingung stellt sicher, dass jeder Knoten v in einer Menge $s \in \mathcal{T}$ enthalten ist, also dass $\mathcal{T}$ tatsächlich eine Überdeckung ist. Die zweite Bedingung ist trivial und besagt, dass jede Menge entweder in der Überdeckung enthalten ist oder nicht.

Hat man nun das Optimierungsproblem UM gelöst, so hat man eine Überdeckung $\mathcal{T}$ der Knotenmenge mit einer minimalen Anzahl an Mengen. Anschließend erhält man eine optimale Färbung, indem man jedem Knoten $v \in V$ die eindeutige Farbe irgendeines s mit $v \in s$ zuweist.

Die Reduzierung auf maximal unabhängige Mengen stellt kein Problem da. Dadurch wird die Anzahl der benötigten Variablen drastisch reduziert, und jede Überdeckung $\mathcal{T}$ der Knoten V mit unabhängigen Mengen lässt sich durch Hinzufügen von Knoten zu den unabhängigen Mengen $t \in \mathcal{T}$ in eine Überdeckung mit gleich vielen maximal unabhängigen Mengen umwandeln.

3.3.4.3 Lösung durch Spaltengenerierung

Da es jedoch immer noch sehr viele maximal unabhängige Mengen gibt, besitzt das Minimierungsproblem (UM) eine sehr große Menge an Variablen, so dass eine direkte Lösung praktisch nicht möglich ist. Dieses Problem wird mittels Spaltengenerierungs-Ansatz gelöst. Das bedeutet, dass man mit einer Teilmenge der Variablen arbeitet und bei Bedarf – d.h. falls die Lösung auf der Teilmenge der Variablen keine global optimale Lösung ist – neue Variablen generiert.

Das Hauptproblem besteht darin, eine Regel festzulegen, wann neue Variablen generiert werden müssen. Dieses sollte so selten wie möglich der Fall sein, da nach jeder Variablenbelegung ein neues ILP gelöst werden muss, also Rechenzeit benötigt wird. Eine weitere Schwierigkeit besteht darin, dass für die Spaltengenerierung duale Variablen benötigt werden. Während

dieses bei gewöhnlichen LPs kein Problem ist, ist die Generierung dualer Variablen bei gleichzeitiger Erhaltung der Ganzzahligkeit schwierig.

Für diesen Fall wird der folgende Algorithmus verwendet:

1. UM ist ein ganzzahliges lineares Programm. Um die für die Spaltengenerierung notwendigen dualen Variablen zu erhalten, wird zunächst die zu UM gehörige Relaxation UMR betrachtet. Bei der Relaxation eines ILPs wird die Forderung nach der Ganzzahligkeit (hier $x_s \in \{0,1\}$) der Variablen durch die Forderung nach nicht-Negativität ($x_s \geq 0$) ersetzt. Die Relaxation UMR ist

$$\begin{aligned} \text{Minimiere } & \sum_{s \in S} x_s \\ \text{mit Bedingungen } \forall v \in V : & \sum_{s \in S, v \in s} x_s \geq 1, \\ & \forall s \in S : x_s \geq 0. \end{aligned}$$

2. Begonnen wird mit einer Menge $\overline{S} \subset S$ unabhängiger Mengen.

3. Dann betrachtet man UMR über der Menge $\overline{S}$ anstelle von S. Dieses gibt eine Lösung für UMR und für jede Bedingung in UMR eine duale Variable π_v.

4. Zur Überprüfung, ob die Menge $\overline{S}$ erweitert werden muss, wird nun über den dualen Variablen π_v, $v \in V$ das ganzzahlige lineare Programm GUM

$$\begin{aligned} \text{Maximiere } & \sum_{v \in V} \pi_v z_v \\ \text{mit Bedingungen } & \forall \{v,w\} \in E : z_v + z_w \leq 1, \\ & \forall v \in V : z_v \in \{0,1\} \end{aligned}$$

   für gewichtete unabhängige Mengen gelöst.

5. Ist das Maximum der Zielfunktion größer als 1, dann entspricht jedes z_v mit Wert 1 einer zu $\overline{S}$ hinzuzufügenden Menge. In diesem Fall muss UMR erneut über der neuen Menge $\overline{S}$ gelöst werden und der Algorithmus ab Schritt 3 wiederholt werden.

   Ansonsten werden keine weiteren Variablengenerierungen mehr benötigt, d.h. das Optimum von UMR über $\overline{S}$ ist auch das Optimum von UMR über der Menge S aller unabhängigen Mengen.

Es gibt zwei wesentliche Schwierigkeiten in diesem Algorithmus:

- Da Problem GUM ist wieder ein ganzzahliges lineares Programm – wenn auch mit weniger Variablen – und als solches schwierig. Da es jedoch für jede Generierung neuer Variablen gelöst werden muss, werden effiziente Techniken benötigt.
- Zum Schluss des Algorithmus hat man eine optimale Lösung für die lineare Relaxation UMR. In der Regel werden die Variablen jedoch nicht ganzzahlig sein. Daher ist es notwendig, die Ganzzahligkeit der Lösung zu sichern.

Auf die genauen Lösungen dieser Probleme soll an dieser Stelle nicht eingegangen werden. Diese sind im Detail in [MT96] beschrieben.

3.3.5 Vor- und Nachbereitung des Graphen zur Färbung

Als weitere Optimierung kann der Graph vor der Färbung präpariert werden. So ist es möglich, Knoten zu entfernen, die auf jeden Fall gefärbt werden können. Die folgenden Vorschläge stammen aus [Tom05, Kapitel 3.3].

- Wurde eine untere Schranke lb für den chromatischen Index $\chi(G)$ gefunden, so können alle Knoten v mit $d(v) < lb$ entfernt werden, da diese im Nachhinein sicher mit einer Farbe aus $\underline{lb} \setminus Nc(v)$ gefärbt werden können.

- Sind v, w Knoten mit $N(w) \subset N(v)$ (insbesondere sind v und w nicht benachbart), so wird der Knoten w von dem Knoten v *dominiert* und kann entfernt werden. Zum Schluss kann w in jedem Fall mit derselben Farbe wie v gefärbt werden.

Diese Schritte lassen sich gegebenenfalls mehrfach wiederholen. Wurden in einem Schritt die Knoten W aus dem Graphen $G = (V, E)$ entfernt, so können obige Überlegungen rekursiv auf $G[V \setminus W]$ angewandt werden.

Nach der Färbung des durch die Funktion Precolor (siehe Algorithmus 8) entstandenen Graphen H müssen die entfernten Knoten gefärbt werden. Dieses geschieht, indem die Knoten sequentiell in umgekehrter Reihenfolge des Entfernens gefärbt und zu dem Graphen hinzugefügt werden. Dabei muss nach obigen Überlegungen niemals eine neue Farbe eingefügt werden.

Es kann sogar passieren, dass der Graph nach der Vorbereitung leer ist. Dann ist $lb = \chi(G)$ gewesen und L enthält die Knoten in einer Reihenfolge, so dass die sequentielle Färbung exakt ist. Jedoch führt nicht immer, wenn $lb = \chi(G)$ ist, die Vorbereitung zu einem leeren Graphen.

3.4 Algorithmen zur Kantenfärbung

In diesem Kapitel werden Algorithmen und Ergebnisse zur Kantenfärbung dargestellt. Da für unsere Anwendungszwecke der Kantenfärbung Multigraphen notwendig sind, werden in diesem Kapitel im Wesentlichen Multigraphen betrachtet.

3.4.1 Einfache Greedy-Färbungsalgorithmen

Der Algorithmus Greedy-Seq-Edgecolor entspricht dem in Algorithmus 1 dargestellten Algorithmus Greedy-Seq-Color zur Knotenfärbung. Nacheinander wird jeder Kante die kleinstmögliche Farbe zugeordnet.

Satz 3.4.1. *Zu einem Multigraphen $G = (V, E, f)$ erzeugt der in Algorithmus 9 angegebene Algorithmus Greedy-Seq-Edgecolor eine Kantenfärbung mit maximal $2\Delta(G)-1$ Farben. Erfolgt das Auswählen der Kante in Zeile 4 in konstanter Zeit, so beträgt die Laufzeit $\mathcal{O}(|E|\Delta(G))$.*

Algorithmus 8 : **Coloring-Preperation**
Vor- und Nachbearbeitung eines Graphen zur Färbung

Parameter : G: Ein Graph $G = (V, E)$ als Adjazenzliste.
Daten : *color*: Eine Funktion zur Graphenfärbung.
Rückgabewert : Eine Färbung für den Graphen G.
Komplexität : $\mathcal{O}(|V|^3) + \mathcal{O}(color)$.

```
1  Funktion Coloring-Preperation(G: Graph)
2      (H, L) := Precolor(G) ;                  // Entferne sicher zu färbende Knoten
3      Erzeuge Färbung c für den Graphen H mit der Funktion color;
4      Postcolor(H, L) ;                              // Färbe die entfernten Knoten
5      return c;
6  end
```

Komplexität : $\mathcal{O}(|V|^3)$.

```
7  Funktion Precolor(G: Graph)
8      L = ∅ ;                          // Liste zum Speichern der entfernten Knoten
9      H := G ;                             // Graph, aus dem Knoten entfernt werden
10     lb := untere Schranke für χ(G);
11     repeat                        // Solange, bis kein Knoten mehr entfernt wurde
12         n := |V_H|;
13         foreach v ∈ V_H mit d_H(v) < lb do   // Entferne alle Knoten mit d_H(v) < lb
14             H := H − v;
15             L := L ∪ {v};
16         foreach v ∈ V_H do                    // Entferne alle dominierten Knoten
17             foreach w ∈ V_H \ {v} do
18                 if N_H(w) ⊂ N_H(v) then        // Knoten w wird durch v dominiert
19                     H := H − w;
20                     L := L ∪ {w};
21     until n = |V_H|;
22     return (H, L);
23 end
```

Komplexität : $\mathcal{O}(|V| + |E|)$.

```
24 Funktion Postcolor(H: Graph, L: Liste)
25     foreach v ∈ L do          // In umgekehrter Reihenfolge des Einfügens in L
26         c(v) := min(ℕ_0 \ Nc_H(v));
27         H = G[V_H ∪ {v}];
28 end
```

Algorithmus 9 : **Greedy-Seq-Edgecolor**
Sequentielle Kantenfärbung eines Multigraphen

Parameter : G: Ein Multigraph $G = (V, E, f)$ als Adjazenzmatrix oder Adjazenzliste.
Rückgabewert : Eine heuristische Kantenfärbung für den Multigraphen G.
Komplexität : $\mathcal{O}(|E|\Delta(G))$.

```
1 Funktion Greedy-Seq-Edgecolor(G: Graph)
2     c : E ↦ ℕ ∪ {∞} ;                          // Färbungsabbildung als Array
3     foreach e ∈ E do c(e) := ∞ ;          // Zu Beginn ist jede Kante ungefärbt
4     foreach e ∈ E do
5         c(e) := kleinstmögliche Farbe, die keine zu e adjazente Kante hat;
6     return c;
7 end
```

Beweis. Die Korrektheit sieht man analog zur Korrektheit der sequentiellen Knotenfärbung (siehe Satz 3.3.3). Soll nun eine Kante $e = vw$ gefärbt werden, so sind sowohl v als auch w zu maximal $\Delta(G) - 1$ weiteren Kanten inzident, also ist e zu maximal $2\Delta - 2$ Kanten adjazent. Daher kann e mit einer der Farben $\{0, \ldots, 2\Delta(G) - 2\}$ gefärbt werden.

Nun werden $|E|$ Kanten jeweils in $\mathcal{O}(\Delta(G))$ gefärbt. Somit folgt die Gesamtkomplexität. □

Um einen zu Greedy-IS-Color (vgl. Algorithmus 6) analogen Algorithmus zur Kantenfärbung zu definieren, wird der Begriff des *Matchings* benötigt. Ein Matching für Kanten entspricht einer unabhängigen Menge für Knoten.

Definition 3.4.2 (Matching, maximales Matching)**.** *Sei $G = (V, E, f)$ ein Multigraph. Eine Menge $F \subset E$ heißt* Matching *von G, wenn keine zwei Kanten aus F adjazent sind.*

Ein Matching F von G heißt maximal*, wenn für alle Matchings F' mit $F \subset F'$ gilt: $F = F'$.*

Mit dieser Definition lässt sich nun der dem Algorithmus Greedy-IS-Color entsprechende Kantenfärbungsalgorithmus beschreiben.

Seien $M_0, \ldots, M_{k-1}$ Matchings des Multigraphen $G = (V, E, f)$, so dass $(M_j)_{j \in \underline{k}}$ eine Überdeckung von E ist, d.h. dass jede Kante in mindestens einem Matching liegt. Dann lässt sich eine Kantenfärbung von G definieren, indem jeder Kante aus M_j die Farbe j zugewiesen wird. Ist eine Kante in mehreren Matchings enthalten, so wird ihr irgendeine dieser Farben zugewiesen. Die so definierte Kantenfärbung verwendet maximal k Farben. Umgekehrt definiert eine Kantenfärbung $c : E \mapsto \underline{k}$ gerade eine (disjunkte) Überdeckung $(c^{-1}(j))_{j \in \underline{k}}$ von E mit Matchings. Insbesondere folgt:

Satz 3.4.3. *Sei $G = (V, E, f)$ ein Multigraph. Die kleinste Zahl k, so dass es eine Überdeckung $(M_j)_{j \in \underline{k}}$ von E mit Matchings gibt, ist der kantenchromatische Index $k = \chi'(G)$. Wird für jede Kante $c(e) = j$ für ein j mit $e \in M_j$ zugewiesen, so ist c eine optimale Färbung für G. Diese Überdeckung kann disjunkt gewählt werden.*

Eine gute Übersicht über Heuristiken zur Kantenfärbung wird in [HDD03] gegeben.

Algorithmus 10 : **Greedy-Match-Edgecolor**
Kantenfärbung eines Multigraphen mit Matchings

Parameter : G: Ein Multigraph $G = (V, E, f)$ als Adjazenzmatrix oder Adjazenzliste.
Rückgabewert : Eine heuristische Kantenfärbung für den Multigraphen G.
Komplexität : $\mathcal{O}(|E|^2)$, wenn das Matching in $\mathcal{O}(|V|)$ bestimmt wird.

```
1  Funktion Greedy-Match-Edgecolor(G: Graph)
2      c : E ↦ N ∪ {∞} ;                          // Färbungsabbildung als Array
3      foreach e ∈ E do c(e) := ∞ ;               // Zu Beginn ist jede Kante ungefärbt
4      i := 0 ;                                   // Farbe des aktuellen Matchings
5      F := E ;                                   // Noch nicht gefärbte Kanten
6      while F ≠ ∅ do                             // Es sind noch nicht alle Kanten gefärbt
7          M := Matching in G = (V, F, f_|F);
8          foreach e ∈ M do c(e) := i ;           // Färbe alle Kanten aus M
9          F := F \ M ;                           // Die Kanten von M sind nun gefärbt
10         i := i + 1 ;                           // Wähle Farbe für die nächsten Kanten
11     return c;
12 end
```

3.4.2 Komplexität der Kantenfärbung

Obwohl Kantenfärbung NP-hart ist, ist Kantenfärbung von Multigraphen mit beschränkter Knotenanzahl zumindest theoretisch sehr effizient lösbar.

Satz 3.4.4 (Kantenfärbung von Graphen mit beschänkter Knotenzahl ist in $\mathcal{O}(|E|)$ lösbar)**.** *Sei c eine Konstante. Für Multigraphen $G = (V, E, f)$ mit $|V| \leq c$ gibt es einen Kantenfärbungsalgorithmus Bounded-Edgecolor mit Laufzeit $\mathcal{O}(|E|)$*

Beweis. In Satz 3.4.7 wird bewiesen, dass sogar Multihypergraphen mit beschränkter Knotenanzahl in $\mathcal{O}(|E|)$ exakt kantenfärbbar sind, insbesondere also auch Multigraphen. □

Die multiplikative Konstante steigt jedoch exponentiell mit der Anzahl der Knoten, daher ist der Algorithmus in der Praxis nicht einsetzbar.

Im Gegensatz zur Knotenfärbung gibt es für Kantenfärbung jedoch auch effiziente Heuristiken mit sehr guter Approximationsgüte.

In [NK90] zeigten Nishizeki und Kashiwagi einen sequentiellen Algorithmus zur Kantenfärbung mit sehr guter Approximationsgüte. Genauer:

Satz 3.4.5 (Nishizeki und Kashiwagi, 1990)**.** *Es gibt einen polynomiellen Algorithmus zur Kantenfärbung eines Multigraphen $G = (V, E, f)$, der maximal $\lfloor \frac{11}{10} \cdot \chi'(G) + \frac{4}{5} \rfloor$ Farben benötigt. Dieser im Folgenden mit NK-Edgecolor bezeichnete Algorithmus hat eine Komplexität von $\mathcal{O}(|E|(\Delta(G) + |V|))$.*

Korollar 3.4.6 (Approximationsgüte polynomieller Kantenfärbung)**.** *Es gibt einen polynomiellen Algorithmus zur Kantenfärbung mit Approximationsgüte $\kappa = \frac{4}{3}$. Dieser ist nur eine leichte Modifikation von NK-Edgecolor und soll im Folgenden ebenfalls als NK-Edgecolor bezeichnet werden.*

Beweis. Ein Multigraph mit $\chi'(G) = 1$ kann mit dem Greedy-Algorithmus in $\mathcal{O}(|V| + |E|)$ exakt gefärbt werden. Ein Multigraph mit $\chi'(G) = 2$ kann ebenfalls in $\mathcal{O}(|V| + |E|)$ exakt gefärbt werden, indem der bipartite Kantengraph in $\mathcal{O}(|V| + |E|)$ exakt gefärbt wird.

Für $\chi'(G) \geq 3$ gilt für die mit NK-Edgecolor erzeugte Kantenfärbung c

$$\begin{aligned}
\chi'(G) = 3: \qquad & |c(E)| \leq \left\lfloor \frac{11}{10} \cdot \chi'(G) + \frac{4}{5} \right\rfloor = \lfloor 4, 1 \rfloor = 4 = \frac{4}{3}\chi'(G) \\
\chi'(G) \geq 4: \qquad & |c(E)| \leq \left\lfloor \frac{11}{10} \cdot \chi'(G) + \frac{4}{5} \right\rfloor \leq \frac{11}{10} \cdot \chi'(G) + \frac{4}{5} \\
& = \left(\frac{11}{10} + \frac{4}{5\chi'(G)} \right) \chi'(G) \leq \left(\frac{11}{10} + \frac{4}{5\chi'(G)} \right) \chi'(G) \\
& \leq \left(\frac{11}{10} + \frac{1}{5} \right) \chi'(G) \leq \frac{13}{10}\chi'(G) \leq \frac{4}{3}\chi'(G)
\end{aligned}$$

Versucht man zuerst in $\mathcal{O}(|V| + |E|)$ eine 1- oder 2-Färbung zu erzeugen (ansonsten liefern die Algorithmen in dieser Zeit ein negatives Ergebnis) und färbt ansonsten den Graphen mit NK-Edgecolor in $\mathcal{O}(|V| + |E|)$, so hat der gesamte Algorithmus eine Komplexität von $\mathcal{O}(|V| + |E|)$ und benötigt maximal $\frac{4}{3}\chi'(G)$ Farben. □

3.4.3 Kantenfärbung bipartiter Multigraphen

Für bipartite Multigraphen ist das Problem der Kantenfärbung einfach. In diesen Graphen gilt die Gleichung $\chi'(G) = \Delta(G)$ (siehe [Vol91, S. 219]). Außerdem kann eine optimale Färbung effizient berechnet werden, denn 2001 stellte Cole in [COS01] einen $\mathcal{O}(|E| \log \Delta(G))$-Algorithmus zur Bestimmung einer optimalen Kantenfärbung mit $\Delta(G)$ Farben vor. Einen einfacheren $\mathcal{O}(|E| \log |E|)$-Algorithmus präsentierte Alon 2003 in [Alo03].

3.4.4 Kantenfärbung von Hypergraphen

In [Erl99, S. 47] wird bewiesen, dass Kantenfärbung von Multigraphen mit beschränkter Knotenanzahl in polynomieller Zeit möglich ist. Der folgende selbst geführte Beweis verallgemeinert diese Aussage auf Multihypergraphen.

Satz 3.4.7 (Exakte Kantenfärbung von Multihypergraphen in $\mathcal{O}(|E|)$ möglich)**.** *Sei c eine Konstante. Für Multihypergraphen $G = (V, E, f)$ mit $|V| \leq c$ gibt es einen Kantenfärbungsalgorithmus Bounded-Hyperedgecolor mit Laufzeit $\mathcal{O}(|E|)$.*

Beweis. Wir formulieren das Kantenfärbungsproblem mittels dem in Satz 3.4.3 dargestellten Zusammenhang als ganzzahliges lineares Programm. Wir sagen, zwei Kanten mit denselben Knoten haben denselben Typ. Entsprechend haben zwei Matchings von G denselben Typ, wenn alle ihre Kanten denselben Typ haben.

Da jeder Kantentyp einem Element der Potenzmenge $\mathcal{P}(V)$ entspricht, gibt es in einem Multigraphen mit $|V| \leq c$ Knoten maximal $k \leq |\mathcal{P}(V)| = 2^c$ Kantentypen $t_1, \ldots, t_k$. Der Multigraph kann daher als Vektor $g = (g_1, \ldots, g_k) \in {\mathbb{N}_0}^k$ aufgefasst werden, wobei es genau

g_j Kanten vom Typ t_j gibt[5]. Man beachte, dass in dieser Darstellung von G isolierte Ecken nicht vorkommen. Da jedoch isolierte Ecken für die Kantenfärbung irrelevant sind, können diese sowieso weggelassen werden.

Nun lässt sich jeder Matchingtyp B_i, $i = 1 \ldots, l$ als Vektor $b_i = (b_{i1}, \ldots, b_{ik}) \in \{0,1\}^k$ auffassen, wobei b_{ij} genau dann 1 ist, wenn eine Kante vom Typ t_j in B_i vorkommt. Für die Anzahl l der verschiedenen Matchingtypen gilt somit $l \leq 2^k$.

Das Kantenfärbungsproblem lässt sich nun als ganzzahliges lineares Programm (ILP) formulieren. Es soll eine disjunkte Überdeckung $(M_i)_{i \in \underline{r}}$ von E mit minimaler Anzahl r an Matchings gefunden werden. Die Variablen $x_1, \ldots, x_l$ repräsentieren die Häufigkeit eines Matchings vom Typ l. Die Summe der x_i ist somit gerade die zu minimierende Anzahl der verwendeten Farben.

$$\begin{aligned} &\text{Minimiere } \sum_{i=1}^{l} x_i \\ &\text{mit Bedingungen } \forall j = 1, \ldots, k : \sum_{i=1}^{l} x_i \cdot b_{ij} = g_j, \\ &\qquad \forall i = 1, \ldots, l : x_i \in \mathbb{N}_0. \end{aligned}$$

Die erste Bedingung stellt sicher, dass in allen Matchings zusammen genau g_j Kanten vom Typ t_j verwendet werden.

Die Werte k und 2^k hängen nur von der Konstanten c ab, sind also ebenfalls konstant. Somit ist sowohl die Dimension $l \leq 2^k$ des ILP als auch die Anzahl k der Bedingungen beschränkt. Daher lassen sich die l verschiedenen Matchingtypen in konstanter Zeit erzeugen. Dann können die Vektoren $b_i = (b_{i1}, \ldots, b_{ik})$ und $g = (g_1, \ldots, g_k)$ in $\mathcal{O}(|V| + |E|) = \mathcal{O}(|E|)$ generiert werden. Nach [Eis03] kann ein ganzzahliges lineares Programm mit beschränkter Anzahl an Variablen und Bedingungen in linearer Zeit abhängig von der Länge der binären Kodierung der Eingabe gelöst werden.

Für natürliche Zahlen steigt die Länge der binären Kodierung logarithmisch mit der Größe der Zahl. Wegen $g_i \leq |E|$ steigt die Länge der Kodierung jedes g_i, also auch von g, mit $\log(|E|)$. Da die übrige Eingabe des ILP gleich bleibt, gilt für die Länge s der binären Kodierung der Eingabe $s \in \mathcal{O}(\log |E|)$. Nach [Eis03] lässt sich dieses ILP in $\mathcal{O}(s) = \mathcal{O}(\log |E|)$ lösen.

Die Komplexität des Algorithmus ergibt sich somit aus der Konstruktion des ILP in $\mathcal{O}(|E|)$. □

Der Algorithmus ist jedoch praktisch nicht verwendbar, da die Konstante l schon für kleine Schranken c zu groß ist. Für maximal c Knoten ist für Multihypergraphen $l = 2^{2^c}$ und für Multigraphen $l = 2^{\frac{1}{2}c(c+1)}$. Für einen Multihypergraphen mit 32 Knoten ist diese Konstante somit schon $2^{2^{32}} \approx 3,1032 \cdot 10^{1292913986}$, für einen Multigraphen $2^{16 \cdot 33} \approx 8,7869 \cdot 10^{158}$.

[5] Für nicht-Hypergraphen entsprechen die Werte von g_j genau den Einträgen der Adjazenzmatrix, falls der Graph in einer solchen gespeichert ist.

Kapitel 4

Modellierung

4.1 Beschreibung des Modells

Im Folgenden wird nun ein graphentheoretisches Modell für das Netzwerk erstellt. In dem Netzwerk gibt es:

- Eine endliche Menge $\mathbb{S}$ von Switches und eine endliche Menge $\mathbb{D}$ von Devices, die wir zu einer Knotenmenge $\mathbb{V} = \mathbb{S} \cup \mathbb{D}$ zusammenfassen.
- Eine endliche Menge $\mathbb{E}$ von Netzwerkkabeln. Dabei wird ein Netzwerkkabel zwischen zwei Knoten x und y als Kante $\{x, y\} \in \mathbb{V}^{\underline{2}}$ zwischen den verbundenen Knoten repräsentiert.
- Ein Attribut $Duplex \in \{VD, HD\}$, wobei VD ein Vollduplex-Netzwerk und HD ein Halbduplex-Netzwerk repräsentiert.
- Ein Attribut $Cast \in \{UC, MC\}$, wobei UC für Unicast und MC für Multicast steht.
- Eine endliche Menge Γ von Requests. Jedes Request r bekommt durch $\delta : \Gamma \mapsto \mathbb{R}^{>0}$ eine echt positive Übertragungsdauer $\delta(r)$ zugeordnet und verläuft von einem Quelldevice $Quelle(r)$ zu Zieldevices $Ziel(r)$. Im Unicastfall ist $Ziel(r) \in \mathbb{D}$ ein einzelnes Device, im Multicastfall ist $Ziel(r) \subset \mathbb{D}$ eine Menge von Devices.

Da das zu simulierende Netzwerk baumförmig ist wird gefordert, dass $\mathcal{G} = (\mathbb{V}, \mathbb{E})$ ein Baum ist. Das so konstruierte Modell des Netzwerks wird im Folgenden mit dem Tupel $\mathcal{N} = (\mathcal{G}, \Gamma, Duplex, Cast)$ bezeichnet.

Bemerkung 4.1.1. Teilweise verwenden wir die Bezeichnungen VD-Netzwerk oder HD-Netzwerk und meinen damit, dass $Duplex = VD$ oder $Duplex = HD$ erfüllt ist. Entsprechend verwenden wir UC-Netzwerk oder MC-Netzwerk oder auch Kombinationen wie VD-MC-Netzwerk.

Bemerkung 4.1.2 (Broadcast-Netzwerke)**.** Neben Unicast- und Multicast-Netzwerken gibt es noch Broadcast-Netzwerke. In diesen verläuft ein Request immer von einem Device d zu allen anderen Devices. Die hier behandelten Themen sind für Broadcast-Netzwerke trivial, daher werden Broadcast-Netzwerke im Folgenden nicht näher betrachtet und in Abschnitt 4.3.3 kurz beschrieben.

Da das Netzwerk baumförmig ist, gibt es zwischen je zwei Devices immer genau einen Weg. Daher gibt es in diesem Zusammenhang kein Routing-Problem und die Menge der Kabel, über die ein Request Daten übermittelt, ist eindeutig definiert. Im Unicastfall bilden die von einem Request verwendeten Kanten einen Weg. Dieser wird als Kommunikationslinie bezeichnet. Im Multicastfall bilden die von einem Request verwendeten Kanten hingegen einen Baum. Diesen werden wir (in etwas abgewandelter Form) als Kommunikationsbaum bezeichnen.

Im Folgenden werden wir definieren, wann zwei Requests einen Konflikt haben. Im Wesentlichen sagen wir, dass zwei verschiedenen Requests in Konflikt stehen, wenn sie

- in Halbduplex-Netzwerken über dasselbe Kabel Daten übertragen,
- in Vollduplex-Netzwerken über dasselbe Kabel in dieselbe Richtung Daten übertragen.

Ziel ist es dann, einen Zeitplan (Schedule) für die Ausführung der Requests zu erstellen, so dass niemals zwei in Konflikt stehende Requests zur selben Zeit ausgeführt werden.

4.2 Requests

4.2.1 Kommunikationslinien

Kommunikationslinien werden im Verlauf dieser Arbeit die von einem Unicast-Request verwendeten Knoten und Kanten repräsentieren. In diesem Abschnitt werden Eigenschaften von Kommunikationslinien hergeleitet.

Definition 4.2.1 (Kommunikationslinie)**.** *Die* Kommunikationslinie $KL(v, w)$ *von einem Knoten v zu einem Knoten w ist der eindeutig bestimmte Weg von v nach w in $\mathcal{G}$.*

Definition 4.2.2 (Konflikt von Kommunikationslinien)**.** *Seien K und L zwei Kommunikationslinien und $e \in \mathbb{E}$ eine Kante. Dann sagt man, K und L haben einen* Konflikt in e *(i.Z. $K \leftrightsquigarrow_e L$), wenn gilt:*

- *In HD-Netzwerken: $e \in E_K \cap E_L$.*
- *In VD-Netzwerken: $e \in E_K \cap E_L$ und $e^{K-} = e^{L-}$ (bzw. äquivalent $e^{K+} = e^{L+}$).*

Dann definiert man

$$\begin{aligned}
\text{für } v \in \mathbb{V}: \quad & K \leftrightsquigarrow_v L \iff \exists e \in E(v) : K \leftrightsquigarrow_e L, \\
\text{für } V \subset \mathbb{V}: \quad & K \leftrightsquigarrow_V L \iff \exists v \in V : K \leftrightsquigarrow_v L, \\
& K \leftrightsquigarrow L \iff K \leftrightsquigarrow_{\mathbb{V}} L.
\end{aligned}$$

Man sagt

- *K und L haben einen* Konflikt *im Knoten v, wenn $K \leftrightsquigarrow_v L$,*
- *K und L haben einen* Konflikt *in V, wenn $K \leftrightsquigarrow_V L$,*
- *K und L haben einen* Konflikt*, wenn $K \leftrightsquigarrow L$.*

Entsprechend nennt man K und L konfliktfrei *in e,* konfliktfrei *in v,* konfliktfrei *in V oder* global konfliktfrei*, wenn sie dort keinen Konflikt haben.*

Offensichtlich sind zwei Kommunikationslinien konfliktfrei, wenn sie keinen Knoten bzw. keine Kante gemeinsam haben.

Lemma 4.2.3. *Seien K und L Kommunikationslinien.*

(i) Sei $\mathcal{N}$ ein Vollduplex-Netzwerk und $e \in E_K \cap E_L$ eine Kante mit $e^{K-} = e^{L+}$. Dann sind K und L konfliktfrei in e und in den Randknoten von e.

(ii) Sei P ein nicht-leerer Weg in $\mathcal{G}$ mit $P \lhd K$ und $P \lhd L$. Dann haben K und L in jedem Knoten und jeder Kante von P einen Konflikt.

(iii) Sei P ein nicht-leerer Weg in $\mathcal{G}$ mit $P \lhd K$ und $P^{-1} \lhd L$. In einem HD-Netzwerk haben K und L in jedem Knoten und jeder Kante von P einen Konflikt. In einem VD-Netzwerk sind K und L in jedem Knoten und jeder Kante von P konfliktfrei.

(iv) Sei P ein nicht-leerer Weg, so dass K und L entweder P oder P^{-1} enthalten. Haben K und L in einem Knoten oder einer Kante von P einen Konflikt, so haben sie in jedem Knoten und jeder Kante von P einen Konflikt.

Beweis. Zu (i): Hätten K und L einen Konflikt in e, so wäre $e^{K+} = e^{L+} = e^{K-} \neq e^{K+}$. Widerspruch. Somit sind K und L konfliktfrei in e.

Annahme: Es gebe einen Randknoten v von e, in dem K und L einen Konflikt haben. Diesen haben sie in einer zu v adjazenten Kante $f \in E_K \cap E_L$ mit $f \neq e$. Sei $v = e^{K+}$ (der Fall $v = e^{K-}$ ist analog). Dann ist

$$v = e^{K+} \implies e = v^{K-} \implies f = v^{K+} \implies v = f^{K-}$$

Da K und L konfliktfrei in e sind, ist $v \neq e^{L+}$. Somit folgt

$$v \neq e^{L+} \implies v = e^{L-} \implies e = v^{L+} \implies f = v^{L-} \implies v = f^{L+}.$$

Wegen $f^{K-} = f^{L+}$ sind somit K und L konfliktfrei in f. Widerspruch.

Zu (ii): Für eine beliebige Kante $e \in E_P$ gilt $e \in E_K \cap E_L$ und $e^{K-} = e^{P-} = e^{L-}$. Somit haben K und L sowohl in HD-Netzwerken als auch in VD-Netzwerken einen Konflikt in e. Wegen der Beliebigkeit von e haben K und L in jeder Kante von P einen Konflikt, da jeder Knoten von P zu einer Kante von P inzident ist, also auch in jedem Knoten von P.

Zu (iii): Sei $P = (x_0, \ldots, x_n)$. Wegen $E_P = E_{P^{-1}}$ gilt $E_P \subset E_K \cap E_L$. In HD-Netzwerken haben somit K und L in jeder Kante $e \in E_P$ wegen $e \in E_K \cap E_L$ einen Konflikt, also auch in jedem Knoten von P.

In VD-Netzwerken gilt für jede Kante $e = x_i x_{i+1} \in E_P$ hingegen $e^{K-} = e^{P-} = x_i = e^{(P^{-1})+} = e^{L+}$, somit sind K und L nach (i) konfliktfrei in e und ihren Randknoten. Da jeder Knoten von P zu einer Kante von P inzident ist, sind K und L konfliktfrei in jeder Kante und jedem Knoten von P.

Zu (iv): Dieses ist eine Zusammenfassung der Ergebnisse von (ii) und (iii).

□

Satz 4.2.4 (Lokale Konfliktfreiheit impliziert globale Konfliktfreiheit)**.**

(i) Haben zwei Kommunikationslinien K und L einen gemeinsamen Knoten $v \in V_K \cap V_L$ und dort keinen Konflikt, so sind sie global konfliktfrei.

(ii) Sei $V \subset \mathbb{V}$ zusammenhängend. Enthalten zwei Kommunikationslinien K und L Knoten aus V und haben keinen Konflikt in V, so sind sie global konfliktfrei.

Beweis. Zu (ii): Hätten K und L einen Konflikt, so hätten sie diesen in einem Knoten w. Da K und L konfliktfrei in V sind, ist $w \notin V$. Dann enthalten K und L beide einen Knoten aus den disjunkten Mengen V und $\{w\}$. Da $\mathcal{G}[\{w\}]$ zusammenhängend ist, sind sowohl $\mathcal{G}[V]$ als auch $\mathcal{G}[\{w\}]$ Bäume. Somit gibt es nach Satz 2.2.20(iv) ein $v \in V$, so dass K und L den eindeutig bestimmten Weg P von v nach w oder P^{-1} von w nach v in $\mathcal{G}$ enthalten. Da K und L einen Konflikt in w haben, haben sie nach Lemma 4.2.3(iv) einen Konflikt in v, also auch in V. Widerspruch.

Zu (i): Dieses folgt aus (ii) mit $V = \{v\}$.

□

Korollar 4.2.5. *Seien K und L Kommunikationslinien, die den nicht-leeren Weg P oder P^{-1} enthalten. Dann gilt*

- *Ist Duplex $= VD$ und $P \lhd K$, $P^{-1} \lhd L$ oder umgekehrt, so sind K und L global konfliktfrei.*
- *Ansonsten haben K und L einen Konflikt.*

Beweis. Im ersten Fall sind K und L nach Lemma 4.2.3 konfliktfrei in jedem (gemeinsamen) Knoten von P, nach Satz 4.2.4 somit auch global konfliktfrei. Ansonsten haben K und L nach Lemma 4.2.3 einen Konflikt in jedem Knoten von P. □

Satz 4.2.6 (Konfliktknoten bilden einen Weg)**.** *Seien K und L zwei Kommunikationslinien. Sei V die Menge der Knoten und E die Menge der Kanten, in denen K und L einen Konflikt haben. Dann gibt es einen Weg P mit $V_P = V$ und $E_P = E$.*

Beweis. Haben K und L keinen Konflikt, so ist die Aussage für den leeren Weg erfüllt.

Sonst sei $K = (x_0, \ldots, x_n)$ und x_i der erste und x_j der letzte Knoten von K, in denen K und L einen Konflikt haben. Dann ist $P = (x_i, \ldots, x_j)$ der eindeutig bestimmte Weg von x_i nach x_j. Wir zeigen $V_P = V$ und $E_P = E$.

Wegen $x_i, x_j \in V_K \cap V_L$ gilt $P \lhd K$ und $P \lhd L$ bzw. $P^{-1} \lhd L$. Da K und L nicht konfliktfrei in x_i sind, haben sie nach Lemma 4.2.3 in jedem Knoten und jeder Kante von P einen Konflikt. Somit gilt $E_P \subset E$ und $V_P \subset V$.

Zu $V \subset V_P$: Sei x ein Knoten, in denen K und L einen Konflikt haben. Dann ist $x \in V_K$, also $x = x_k$ für ein k. Wegen der Wahl von x_i und x_j ist $i \leq k \leq j$, also $x_k \in V_P$.

Zu $E \subset E_P$: Sei e eine Kante, in denen K und L einen Konflikt haben, so ist $e \in E_K$, also $e = x_k x_{k+1}$ für ein k. Dann haben K und L einen Konflikt in x_k und x_{k+1}, also gilt $i \leq k < j$ und somit $e \in E_P$. □

4.2.2 Kommunikationsbäume

Kommunikationsbäume sind spezielle Kommunikationsmengen und werden im Verlauf dieser Arbeit die von einem Multicast-Request verwendeten Knoten und Kanten repräsentieren. In diesem Abschnitt werden ihre Eigenschaften näher untersucht. Einige Sätze entsprechen analogen Sätzen über Kommunikationslinien, daher verlaufen die Beweise oft ähnlich.

Definition 4.2.7 (Kommunikationsmenge)**.** *Eine* Kommunikationsmenge K *ist eine Menge von Kommunikationslinien.*

Definition 4.2.8 (Konflikt von Kommunikationsmengen)**.** *Zwei Kommunikationsmengen K und L haben einen* Konflikt in der Kante e *(i.Z. $K \leftrightsquigarrow_e L$), wenn sie zwei Kommunikationslinien enthalten, die einen Konflikt in e haben. Formaler ist*

$$K \leftrightsquigarrow_e L \iff \exists K' \in K \text{ und } L' \in L \text{ mit } K' \leftrightsquigarrow_e L'.$$

Analog zu Definition 4.2.2 definiert man einen Konflikt in v, einen Konflikt in V bzw. einen globalen Konflikt.

Für eine Kommunikationsmenge K definieren wir

$$V_K = \bigcup_{K' \in K} V_{K'}, \qquad E_K = \bigcup_{K' \in K} E_{K'}$$

und nennen V_K die Knoten und E_K die Kanten von K.

Offensichtlich sind zwei Kommunikationsmengen K und L konfliktfrei, wenn sie keinen Knoten bzw. keine Kante gemeinsam haben, d.h. wenn $V_K \cap V_L = \emptyset$ bzw. $E_K \cap E_L = \emptyset$ gilt.

Definition 4.2.9 (Kommunikationsbaum)**.** *Der* Kommunikationsbaum $KB(v, W)$ *von einem Knoten v zu einer Knotenmenge W ist die Menge*

$$KB(v, W) := \{KL(v, w) : w \in W\}$$

der Kommunikationslinien von v zu den Knoten aus W. Der Knoten v heißt Wurzel *von K.*

Satz 4.2.10 (Kommunikationsbaum ist ein Baum)**.** *Sei K ein Kommunikationsbaum und $G_K := (V_K, E_K)$. Dann ist $G_K = \mathcal{G}[V_K]$ und G_K ein Baum.*

Beweis. Sei v die Wurzel von K. Für eine Kante $e = \{x, y\}$ gilt

$$e \in E_K \implies \exists K' \in K : e \in E_{K'} \implies x, y \in V_{K'} \implies x, y \in V_K$$

und somit $E_K \subset (V_K)^{\underline{2}}$. Daher ist G_K ein Graph und wegen $V_K \subset \mathbb{V}$ und $E_K \subset \mathbb{E}$ ein Teilgraph von $\mathcal{G}$.

Nun ist zu zeigen, dass G_K von $\mathcal{G}$ und V_K induziert wird, d.h. dass für $\{x, y\} \in \mathbb{E}$ mit $x, y \in V_K$ auch $\{x, y\} \in E_K$ gilt.

Sei $x \in KL(v, v_1)$ und $y \in KL(v, v_2)$. Dann enthalten $KL(v, v_1)$ bzw. $KL(v, v_2)$ den eindeutigen Weg P von v nach x bzw. Q von v nach y.

- Falls $x = y_j \in V_Q$, dann ist $Qx + xy$ der in $\mathcal{G}$ eindeutig bestimmte Weg Q von v nach y, also $\{x, y\} \in E_Q \subset E_{KL(v,v_2)} \subset E_K$.
- Falls $y = x_j \in V_P$, so ist analog $\{x, y\} \in E_K$.
- Falls $y \notin V_P$ und $x \notin V_Q$, dann ist $Q + yx$ der in $\mathcal{G}$ eindeutig bestimmte Weg P von v nach x, also $\{x, y\} \in E_P \subset E_{KL(v,v_1)} \subset E_K$ (und insbesondere $y \in V_P$, daher tritt dieser Fall nie ein).

Somit ist $G_K = \mathcal{G}[V_K]$. Insbesondere ist G_K ein schlaufenfreier Graph.

Weiter ist G_K zusammenhängend. Denn seien $x, y \in V_K$ beliebig, so gibt es Kommunikationslinien $K_1 = KL(v, v_1)$ bzw. $K_2 = KL(v, v_2)$ in K mit $x \in V_{K_1}$ und $y \in V_{K_2}$. Dann ist $xK_1^{-1}v + vK_2y$ ein Pfad von x nach y.

Somit ist $G_K = \mathcal{G}[V_K]$ zusammenhängend und schlaufenfrei, also ein Baum. □

Satz 4.2.11. *Sei K ein Kommunikationsbaum. Dann gilt:*

(i) Ist $L_1, L_2 \in K$ und $e \in E_{L_1} \cap E_{L_2}$, so gilt $e^{L_1-} = e^{L_2-}$ und $e^{L_1+} = e^{L_2+}$.

(ii) Ist $L_1, L_2 \in K$ und $w \in V_{L_1} \cap V_{L_2}$, so gilt $w^{L_1-} = w^{L_2-}$.

Beweis. Sei v die Wurzel von K.

Zu (i): Sei $e = \{x, y\} \in E_{L_1} \cap E_{L_2}$ mit $x = e^{L_1-}$ und $y = e^{L_1+}$. Dann sind L_1y und L_2y beide der eindeutig bestimmte Weg P von v nach y in $\mathcal{G}$. Somit folgt

$$e^{L_1-} = e^{L_1y-} = e^{P-} = e^{L_2y-} = e^{L_2-},$$
$$e^{L_1+} = e^{L_1y+} = e^{P+} = e^{L_2y+} = e^{L_2+}.$$

Zu (ii): Ist $w \in V_{L_1} \cap V_{L_2}$, so sind L_1w und L_2w der eindeutig bestimmte Weg P von v nach w in $\mathcal{G}$, also ist

$$w^{L_1-} = w^{(L_1w)-} = w^{P-} = w^{(L_2)w-} = e^{L_2-}.$$

□

Aufgrund von Satz 4.2.11 ist die folgende Definition wohldefiniert:

Definition 4.2.12. *Sei K ein Kommunikationsbaum. Dann lässt sich für einen Knoten $v \in V_K$ und eine Kante $e \in E_K$ definieren*

$$e^{K-} = e^{L-} \text{ für ein } L \in K \text{ mit } e \in E_L,$$
$$e^{K+} = e^{L+} \text{ für ein } L \in K \text{ mit } e \in E_L,$$
$$v^{K-} = v^{L-} \text{ für ein } L \in K \text{ mit } v \in V_L,$$
$$v^{K+} = \{v^{L+} : L \in K \text{ mit } v \in V_L\}.$$

Satz 4.2.13. *Seien K und L Kommunikationsbäume. Dann ist $K \leftrightsquigarrow_e L$ genau dann, wenn:*

- *In HD-Netzwerken: $e \in E_K \cap E_L$.*
- *In VD-Netzwerken: $e \in E_K \cap E_L$ und $e^{K-} = e^{L-}$ (bzw. äquivalent $e^{K+} = e^{L+}$).*

Beweis.

- Zur Hinrichtung in (i) und (ii): Haben K und L einen Konflikt in e, so gibt es $K' \in K$ und $L' \in L$ mit $K' \leftrightsquigarrow_e L'$. Dann gilt $e \in E'_K \subset E_K$ und $e \in E'_L \subset E_L$ und somit $e \in E_K \cap E_L$. In Vollduplex-Netzwerken gilt nach Definition 4.2.12 weiter

$$e^{K-} = e^{K'-} = e^{L'-} = e^{L-}.$$

- Zur Rückrichtung in (i): Gibt es ein $e \in E_K \cap E_L$, so gibt es Kommunikationslinien $K' \in K$ und $L' \in L$ mit $e \in E_{K'}$ und $e \in E_{L'}$. Also ist $e \in E_{K'} \cap E_{L'}$ und somit haben K' und L' einen Konflikt in e, also auch K und L.

- Zur Rückrichtung in (ii): Gibt es ein $e \in E_K \cap E_L$ mit $e^{K-} = e^{L-}$, so gibt es Kommunikationslinien $K' \in K$ bzw. $L' \in L$ mit $e \in E_{K'}$ und $e^{K'-} = e^{K-}$ bzw. $e \in E_{L'}$ mit $e^{L'-} = e^{L-}$. Also ist

$$e \in E_{K'} \cap E_{L'} \qquad \text{und} \qquad e^{K'-} = e^{K-} = e^{L-} = e^{L'-}$$

und somit haben K' und L' einen Konflikt in e, also auch K und L.

□

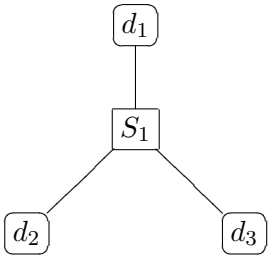


Abbildung 4.1: Vollduplex-Netzwerk mit Kommunikationsbäumen $KB(d_2, \{d_1, d_3\})$ und $KB(d_3, \{d_1, d_2\})$. Die Kommunikationsbäume sind lokal konfliktfrei in den gemeinsamen Kanten $\{S_1, d_2\}$ und $\{S_1, d_3\}$ sowie den gemeinsamen Knoten d_2 und d_3, jedoch nicht global konfliktfrei.

Bemerkung 4.2.14 (Lokale Konfliktfreiheit impliziert keine globale Konfliktfreiheit)**.**
Für den Unicast-Fall gilt nach Satz 4.2.4, dass zwei Kommunikationslinien, die in einem gemeinsamen Knoten keinen Konflikt haben, global konfliktfrei sind. Diese Aussage basiert für Vollduplex-Netzwerke im Wesentlichen darauf, dass zwei Kommunikationslinien, die eine Kante in entgegengesetzte Richtung enthalten, konfliktfrei sind. Diese Aussage gilt für Kommunikationsbäume jedoch nicht. Denn in Abbildung 4.1 enthalten $KB(d_2, \{d_1, d_3\})$ und $KB(d_3, \{d_1, d_2\})$ die Kante $\{d_3, S_1\}$ in unterschiedlichen Richtungen, haben jedoch einen Konflikt in $\{S_1, d_1\}$. Entsprechend enthalten auch beide Kommunikationsbäume den Knoten d_2 (bzw. d_3), haben dort keinen Konflikt, aber sind dennoch nicht global konfliktfrei.

Da diese Aussage jedoch von elementarer Bedeutung ist, betrachten wir im Folgenden ausschließlich den Halbduplexfall. Dort lässt sich die Aussage aus Satz 4.2.4 und darauf aufbauend ein Großteil der Modellierung auf Kommunikationsbäume verallgemeinern. Daher wird im Folgenden $\mathcal{N}$ immer als Halbduplex-Netzwerk vorausgesetzt.

Satz 4.2.15 (Lokale Konfliktfreiheit impliziert globale Konfliktfreiheit)**.** *Für ein Halbduplex-Netzwerk $\mathcal{N}$ gilt:*

(i) Enthalten zwei Kommunikationsbäume K und L den Knoten v und haben dort keinen Konflikt, so sind sie global konfliktfrei.

(ii) Sei $V \subset \mathbb{V}$ zusammenhängend. Enthalten zwei Kommunikationsbäume K und L Knoten aus V und haben keinen Konflikt in V, so sind sie global konfliktfrei.

Beweis. Zu (ii): Sei $G_K = (V_K, E_K)$ und $G_L = (V_L, E_L)$. Es gibt Knoten $v_K \in V \cap V_K$ und $v_L \in V \cap V_L$.

Annahme: K und L haben einen Konflikt in einem Knoten $w \notin V$. Nach Satz 2.2.20(iv) gibt es ein $v \in V$, so dass jeder Weg von V nach w den eindeutigen nicht-leeren Weg $P = (v = x_0, x_1, \ldots, x_{n-1}, x_n = w)$ von v nach w enthält. Da G_K ein Baum ist, enthält G_K einen Weg von v_K nach w und somit auch P. Entsprechend enthält G_L einen Weg von v_L nach w, also auch P.

Somit ist $vx_1 \in E_P \subset E_K \cap E_L$, also haben K und L nach Satz 4.2.13 einen Konflikt in vx_1, also auch in v und somit in V. Widerspruch.

Zu (i): Dieses folgt aus (i) mit $V = \{v\}$.

□

Satz 4.2.16 (Konfliktknoten bilden einen Baum)**.** *Sei $\mathcal{N}$ ein Halbduplex-Netzwerk und K und L zwei Kommunikationsbäume. Sei V die Menge der Knoten und E die Menge der Kanten, in denen K und L einen Konflikt haben. Dann ist $G = (V, E)$ ein Baum.*

Beweis. Haben K und L in einer Kante e einen Konflikt, so haben sie in den inzidenten Knoten einen Konflikt. Also ist $E \subset V^{\underline{2}}$. Somit ist G ein Graph. Als Untergraph von $\mathcal{G}$ ist G schlaufenfrei.

Wir zeigen nun, dass G zusammenhängend ist. Sei $G_K = (V_K, E_K)$ und $G_L = (V_L, E_L)$. Seien $v, w \in V \subset V_K \cap V_L$ beliebig und $P = (v = x_0, x_1, \ldots, x_{n-1}, x_n = w)$ der eindeutig bestimmte Weg von v nach w in $\mathcal{G}$. Da G_K und G_L Bäume und somit zusammenhängend sind, ist P sowohl in G_K als auch in G_L enthalten. Daher folgt $E_P \subset E_K \cap E_L$, also haben K und L nach Satz 4.2.13 in jeder Kante $e \in E_P$ einen Konflikt und es ist $E_P \subset E$. Da jeder Knoten aus V_P zu einer Kante aus E_P inzident ist, haben K und L in jedem Knoten aus V_P einen Konflikt und es gilt $V_P \subset V$. Daher enthält G den Weg P von v nach w, ist also zusammenhängend. □

4.2.3 Requests

Definition 4.2.17 (Kommunikationslinie und Kommunikationsbaum eines Requests)**.**
Zu UC-Requests wird die Kommunikationslinie $KL(r) = KL(Quelle(r), Ziel(r))$ definiert.
Zu MC-Requests wird der Kommunikationsbaum $KB(r) = KB(Quelle(r), Ziel(r))$ definiert.
Zur Vereinheitlichung definiert man

$$KLB(r) := \begin{cases} KL(r) & \text{falls } Cast = UC, \\ KB(r) & \text{falls } Cast = MC. \end{cases}$$

Definition 4.2.18 (Konflikt von Requests). *Die Konfliktrelation $\rightsquigarrow_e \subset \Gamma^2$ zwischen Requests wird definiert durch*

$$r \rightsquigarrow_e s \iff r \neq s \wedge KLB(r) \rightsquigarrow_e KLB(s).$$

Analog zu Definition 4.2.2 definiert man einen Konflikt in v, einen Konflikt in V bzw. einen globalen Konflikt sowie die Konfliktfreiheit.

Definition 4.2.19 (Knoten- und Kantenmengen von Requests). *Für ein Request r definiert man die Knoten- und Kantenmengen*

$$V_r := V_{KLB(r)}, \qquad E_r := E_{KLB(r)}.$$

Für $v \in V_r$ und $e \in E_r$ ist

$$v^{r-} := v^{KLB(r)-}, \qquad v^{r+} := v^{KLB(r)+},$$
$$e^{r-} := e^{KLB(r)-}, \qquad e^{r+} := e^{KLB(r)+}.$$

Dabei beachte man, dass v^{r+} in UC-Netzwerken eine Kante und in MC-Netzwerken eine Menge von Kanten ist.

Satz 4.2.20. *Zwei verschiedene Requests r und s haben genau dann einen Konflikt in einer Kante e, wenn gilt:*

- *In HD-Netzwerken: Es ist $e \in E_r \cap E_s$.*
- *In VD-Netzwerken: Es ist $e \in E_r \cap E_s$ und $e^{r-} = e^{s-}$.*

Beweis. Dieses folgt unmittelbar aus Definition 4.2.18, Definition 4.2.2 und Satz 4.2.13. □

Lemma 4.2.21. *Der Graph $G_r = (V_r, E_r)$ ist zusammenhängend.*

Beweis. Sei $K = KLB(r)$, dann ist $G_r = (V_K, E_K)$ in Multicast-Netzwerken nach Satz 4.2.10 und in Unicast-Netzwerken als Graph eines Weges zusammenhängend. □

Definition 4.2.22 (Requests durch eine Knotenmenge). *Zu $V \subset \mathbb{V}$ bzw. $E \subset \mathbb{E}$ wird die Menge der Requests, deren Kommunikationslinie einen Knoten aus V bzw. eine Kante aus E enthält, bezeichnet mit*

$$\Gamma_V := \{r \in \Gamma : V \cap V_r \neq \emptyset\},$$
$$\Gamma_E := \{r \in \Gamma : E \cap E_r \neq \emptyset\}.$$

Für einen Knoten $v \in \mathbb{V}$ bzw. eine Kante $e \in \mathbb{E}$ schreiben wir auch $\Gamma_v := \Gamma_{\{v\}}$ bzw. $\Gamma_e := \Gamma_{\{e\}}$. Offensichtlich ist $\Gamma_{V \cup W} = \Gamma_V \cup \Gamma_W$.

Lemma 4.2.23. *Sei $r \in \Gamma_v \cap \Gamma_w$ und P der eindeutig bestimmte Weg von v nach w. Für jedes $x \in V_P$ und $e \in E_P$ ist dann $r \in \Gamma_x$ und $r \in \Gamma_e$.*

Beweis. Es ist $v, w \in V_r$ und $G_r = (V_r, E_r)$ zusammenhängend. Daher enthält G_r den eindeutig bestimmten Weg von v nach w. Somit ist $V_P \subset V_r$ und $E_P \subset E_r$ und es gilt

$$x \in V_P \implies x \in V_r \implies \{x\} \cap V_r \neq \emptyset \implies r \in \Gamma_x$$
$$e \in E_P \implies e \in E_r \implies \{e\} \cap E_r \neq \emptyset \implies r \in \Gamma_e$$

□

Die folgenden Aussagen gelten nicht für VD-MC-Netzwerke (siehe Abbildung 4.1). Dieses ist der entscheidende Grund, warum große Teile der Überlegungen dieser Arbeit nicht auf VD-MC-Netzwerke übertragbar sind.

Satz 4.2.24 (Lokale Konfliktfreiheit äquivalent zu globaler Konfliktfreiheit)**.** *Sei $\mathcal{N}$ kein VD-MC-Netzwerk. Dann gilt für eine zusammenhängende Menge $V \subset \mathbb{V}$ und $r, s \in \Gamma_V$*

$$r \rightsquigarrow_V s \iff r \rightsquigarrow s.$$

Beweis. Die Hinrichtung ist trivial. Die Rückrichtung folgt für UC-Netzwerke aus Satz 4.2.4, für HD-MC-Netzwerke aus Satz 4.2.15. □

Später werden oft Eigenschaften über verschiedene Requests aus $\Gamma_V \cap \Gamma_W$ benötigt. Diese werden in den folgenden beiden Lemmas zusammengefasst.

Lemma 4.2.25. *Sei $\mathcal{N}$ kein VD-MC-Netzwerk.*

(i) Seien $V, W \subset \mathbb{V}$ disjunkte, zusammenhängende Mengen. Es seien zwei verschiedene Requests $r, s \in \Gamma_V \cap \Gamma_W$ gegeben.

- *Falls $Duplex = HD$, so haben r und s einen Konflikt in V und in W.*
- *Falls $Duplex = VD$, so haben r und s entweder einen Konflikt in V und in W oder sind global konfliktfrei.*

(ii) Ist eine weiteres Requests t mit Knoten aus V und W gegeben, so stehen mindestens zwei der Requests r, s, t in Konflikt.

Beweis. Zu (i): Es gibt nach Voraussetzung ein $v_r \in V_r \cap V$ und ein $w_r \in V_r \cap W$ bzw. $v_s \in V_s \cap V$ und ein $w_s \in V_s \cap W$. Da $\mathcal{G}[V]$ und $\mathcal{G}[W]$ Bäume sind, gibt es nach Satz 2.2.20(iv) ein $v \in V$ und $w \in W$, so dass jeder V-W-Weg den eindeutig bestimmten nicht-leeren Weg $P = (v = x_0, x_1, \ldots, x_{n-1}, x_n = w)$ von v nach w und entsprechend jeder W-V-Weg den Weg P^{-1} von w nach v in $\mathcal{G}$ enthält.

Da $G_r = (V_r, E_r)$ zusammenhängend ist enthält er einen V-W-Weg von v_r nach w_r. Daher enthält G_r auch den Weg P. Analog enthält $G_s = (V_s, E_s)$ den Weg P und es folgt $E_P \subset E_r \cap E_s$. Also ist $\{vx_1, x_{n-1}w\} \subset E_P \subset E_r \cap E_s$, nach Definition 4.2.2 bzw. Satz 4.2.13 haben r und s somit in Halbduplex-Netzwerken einen Konflikt in vx_1 und $x_{n-1}w$, also auch in $v \in V$ und $w \in W$.

Anderenfalls ist $\mathcal{N}$ ein HD-UC-Netzwerk. Enthalten dann $KL(r)$ und $KL(s)$ beide denselben Weg P oder P^{-1}, so haben sie nach Korollar 4.2.5 einen Konflikt in jedem Knoten von P, somit auch in $v \in V$ und in $w \in W$. Ansonsten ist $P \lhd KL(r)$, $P^{-1} \lhd KL(s)$ oder umgekehrt. Nach Korollar 4.2.5 sind $KL(r)$ und $KL(s)$ dann global konfliktfrei.

Zu (ii): In HD-Netzwerken haben bereits r und s einen Konflikt. Sei das Netzwerk ein VD-UC-Netzwerk. Dann enthält analog zu oben auch $KL(t)$ entweder P oder P^{-1}. Somit enthalten mindestens zwei der Kommunikationslinien $KL(r)$, $KL(s)$ und $KL(t)$ denselben Weg P oder P^{-1}. Nach Lemma 4.2.3(ii) haben diese dann einen Konflikt in jedem Knoten von P, also in v und in w und somit in V und in W.

□

Lemma 4.2.26. *Sei $\mathcal{N}$ kein VD-MC-Netzwerk. Seien $V, W \subset \mathbb{V}$ benachbarte zusammenhängende Mengen. Dann haben zwei Request $r \in \Gamma_V$ und $s \in \Gamma_W$ genau dann einen Konflikt, wenn sie einen Konflikt in V oder in W haben.*

Beweis. Haben r und s einen Konflikt in V oder in W, so haben sie einen Konflikt.

Umgekehrt sei nun $r \leftrightsquigarrow s$. Sei $\{v, w\}$ die V-W-Kante. Dann haben r und s diesen Konflikt in einem Knoten z. Ist $z \in V \cup W$, so ist die Aussage bewiesen. Ansonsten enthält $G_r = (V_r, E_r)$ einen V-z-Weg P und $G_s = (V_s, E_s)$ einen W-z-Weg Q. Nach Satz 2.2.20(vi) ist dann $v \in V_Q$ oder $w \in V_P$. Falls $v \in V_Q \subset V_s$, so enthalten r und s Knoten aus V. Nach Satz 4.2.24 haben sie dann in V einen Konflikt, da sie nicht global konfliktfrei sind. Falls $w \in V_P \subset V_r$, so folgt analog, dass r und s einen Konflikt in W haben. □

4.2.4 Kenngrößen für Requests

In diesem Abschnitt werden wichtige Kenngrößen für Requestmengen eingeführt. Diese erlauben Abschätzungen für die Länge der Schedules oder die Komplexität bei deren Erstellung.

Zu $e \in \mathbb{E}$ und einer Requestmenge R ist die Last $\Psi_R(e)$ bezüglich R definiert als die Anzahl der durch e verlaufenden Requests, die Lastdauer $\Phi_R(e)$ bezüglich R definiert als die Gesamtdauer aller durch e verlaufenden Requests.

In Vollduplex-Netzwerken wird die Last und Lastdauer noch getrennt in die verschiedenen Richtungen betrachtet werden. Den größeren Wert bezeichnet man dann als gerichtete Last $\overrightarrow{\Psi_R}(e)$ bzw. gerichtete Lastdauer $\overrightarrow{\Phi_R}(e)$ der Kante e.

Definition 4.2.27 (Last, Lastdauer)**.** *Sei $e = \{v, w\} \in \mathbb{E}$ und $R \subset \Gamma$. Dann definiert man:*

$$
\begin{aligned}
\textit{Last:} \qquad & \Psi_R(e) := |R \cap \Gamma_e| \\
\textit{Maximale Last:} \qquad & \Psi_R := \max_{e \in \mathbb{E}} \Psi_R(e) \\
\textit{Lastdauer:} \qquad & \Phi_R(e) := \sum_{r \in R \cap \Gamma_e} \delta(r) \\
\textit{Maximale Lastdauer:} \qquad & \Phi_R := \max_{e \in \mathbb{E}} \Phi_R(e)
\end{aligned}
$$

Für Vollduplex-Netzwerke ist es wichtig, in welche Richtung Requests eine Kante e durchlaufen. Für die gerichtete Last der Kante $e = \{v, w\}$ betrachtet man die Anzahl der Requests r mit $e^{r-} = v$ und die Anzahl der Requests mit $e^{r-} = w$. Das Maximum der beiden Werte ist die gerichtete Last der Kante e. Entsprechend definiert man die gerichtete Lastdauer einer Kante. Dort wird die Dauer aller Requests mit $e^{r-} = v$ bzw. mit $e^{r-} = w$ summiert und die gerichtete Lastdauer als Maximum der beiden Werte gewählt. Formaler:

$$\begin{aligned}
\textit{Gerichtete Last:} \qquad & \overrightarrow{\Psi_R}(e) := \max\left\{|\{r \in R \cap \Gamma_e : e^{r-} = x\}| : x \in \{v, w\}\right\} \\
\textit{Maximale gerichtete Last:} \qquad & \overrightarrow{\Psi_R} := \max_{e \in \mathbb{E}} \overrightarrow{\Psi_R}(e) \\
\textit{Gerichtete Lastdauer:} \qquad & \overrightarrow{\Phi_R}(e) := \max\left\{ \sum_{r \in R \cap \Gamma_e, e^{r-} = x} \delta(r) : x \in \{v, w\} \right\} \\
\textit{Maximale gerichtete Lastdauer:} \qquad & \overrightarrow{\Phi_R} := \max_{e \in \mathbb{E}} \overrightarrow{\Phi_R}(e)
\end{aligned}$$

Zur Vereinfachung der Notation setzt man $\Psi(e) := \Psi_\Gamma(e)$ und $\Psi := \Psi_\Gamma$ sowie die entsprechende Definition für die Lastdauer bzw. die gerichtete Last und gerichtete Lastdauer.

Lemma 4.2.28. *In Vollduplex-Netzwerken gelten die Ungleichungen:*

$$\begin{aligned}
(i) \qquad & \overrightarrow{\Psi_R}(e) \geq \frac{1}{2}\Psi_R(e) \\
(ii) \qquad & \overrightarrow{\Psi_R} \geq \frac{1}{2}\Psi_R \\
(iii) \qquad & \overrightarrow{\Phi_R}(e) \geq \frac{1}{2}\Phi_R(e) \\
(iv) \qquad & \overrightarrow{\Phi_R} \geq \frac{1}{2}\Phi_R
\end{aligned}$$

Beweis. Sei $e = \{v, w\}$.

Zu (i): Es gilt die Ungleichungskette

$$\begin{aligned}
\Psi_R(e) &= |R \cap \Gamma_e| = |\{r \in R \cap \Gamma_e : e^{r-} = v\}| + |\{r \in R \cap \Gamma_e : e^{r-} = w\}| \\
&\leq \overrightarrow{\Psi_R}(e) + \overrightarrow{\Psi_R}(e) = 2\overrightarrow{\Psi_R}(e).
\end{aligned}$$

Zu (ii): Es gilt

$$\overrightarrow{\Psi_R} = \max_{e \in \mathbb{E}} \overrightarrow{\Psi_R}(e) \geq \max_{e \in \mathbb{E}} \frac{1}{2}\Psi_R(e) = \frac{1}{2}\max_{e \in \mathbb{E}} \Psi_R(e) = \frac{1}{2}\Psi_R.$$

Zu (iii): Analog zu (i).

Zu (iv): Analog zu (ii).

□

Definition 4.2.29 (Anzahl der Konflikte)**.** *Zu $R \subset \Gamma$ ist die Anzahl Θ_R der Konflikte aus R definiert durch*

$$\Theta(R) = |\{\{r, s\} \in R : r \leftrightsquigarrow s\}|.$$

Wir definieren

$$\Theta := \Theta(\Gamma), \qquad \Theta(V) := \Theta(\Gamma_V), \qquad \Theta(v) := \Theta(\Gamma_v), \qquad \Theta(e) = \Theta(\Gamma_e).$$

Mit Θ^Σ wird die Summe aller $\Theta(v), v \in \mathbb{V}$ bezeichnet, d.h.

$$\Theta^\Sigma := \sum_{v \in \mathbb{V}} \Theta(v).$$

Man beachte, dass $\Theta^\Sigma \geq \Theta$ ist und im Allgemeinen keine Gleichheit gilt.

Lemma 4.2.30. *Für $v \in \mathbb{V}$ gelten die Abschätzungen:*

$$\begin{aligned}
&(i) && \Theta(e) \leq \frac{1}{2}\Psi(e)(\Psi(e)-1) \leq 2\Theta(e) + \Psi(e) \\
&(ii) && \Theta(e) \leq \frac{1}{2}\Psi(e)^2 \\
&(iii) && \Theta(v) \leq \sum_{e \in E(v)} \Theta(e) \leq \frac{1}{2} \sum_{e \in E(v)} \Psi(e)^2 \leq \frac{1}{2}\Delta(\mathcal{G})\Psi^2 \\
&(iv) && \Theta(R) \leq \frac{1}{2}|R|^2 \\
&(v) && \Theta^\Sigma = \sum_{e \in \mathbb{E}} \Theta(e) + \Theta
\end{aligned}$$

Beweis. Zu (i): Zwei Requests $r, s \in \Gamma_e$ können in e nur dann einen Konflikt haben, wenn sie verschieden sind. Es gibt genau $\binom{\Psi(e)}{2} = \frac{1}{2}\Psi(e)(\Psi(e)-1)$ verschiedene Requests in Γ_e. Somit folgt die linke Ungleichung. Im Fall von HD-Netzwerken gilt sogar Gleichheit. Falls das Netzwerk ein VD-Netzwerk ist, so haben je zwei Requests, die e in dieselbe Richtung enthalten, dort einen Konflikt. Es gibt somit mindestens $\Theta(e) \geq \frac{1}{2}\overrightarrow{\Psi}(e)(\overrightarrow{\Psi}(e)-1)$ Konflikte in e. Daraus folgt

$$\frac{1}{2}\Psi(e)(\Psi(e)-1) \leq \overrightarrow{\Psi}(e)(2\overrightarrow{\Psi}(e)-1) = \overrightarrow{\Psi}(e)(\overrightarrow{\Psi}(e)-1) + \overrightarrow{\Psi}(e) \leq 2\Theta(e) + \Psi(e)$$

Zu (ii): Dieses folgt unmittelbar aus der linken Ungleichung von (i).

Zu (iii): Zwei Requests haben in v genau dann einen Konflikt, wenn sie ihn in einer zu v inzidenten Kante haben. Daraus folgt die erste Ungleichung. Die übrigen ergeben sich aus (ii) und

$$\sum_{e \in E(v)} \Psi(e)^2 \leq \sum_{e \in E(v)} \Psi^2 \leq \Delta(\mathcal{G})\Psi^2.$$

Zu (iv): Haben je zwei verschiedene Requests $r, s \in R$ einen Konflikt, so gibt es $\frac{1}{2}|R|(|R|-1) \leq \frac{1}{2}|R|^2$ Konflikte.

Zu (v): Seien $E_{r,s}$ bzw. $V_{r,s}$ die Kanten bzw. Knoten, in denen r und s einen Konflikt haben. Dann ist $G_{r,s} = (E_{r,s}, V_{r,s})$ ein Weg bzw. ein Baum und somit $|E_{r,s}| = |V_{r,s}| - 1$. Dann gilt

$$\Theta^\Sigma = \sum_{v \in \mathbb{V}} \Theta(v) = \sum_{\substack{r,s \in \Gamma \\ r \leftrightsquigarrow s}} |V_{r,s}| = \sum_{\substack{r,s \in \Gamma \\ r \leftrightsquigarrow s}} |E_{r,s}| + \Theta = \sum_{e \in \mathbb{E}} \Theta(e) + \Theta.$$

□

Lemma 4.2.31. *Ist M eine obere Schranke für die Anzahl der Zieldevices eines MC-Requests und im UC-Netzwerk $M := 1$, so ist weiter*

$$
\begin{aligned}
(i) \quad & \Theta^{\Sigma} \leq \sum_{v \in \mathbb{V}} |\Gamma_v|^2 \\
(ii) \quad & \Theta^{\Sigma} \leq \sum_{v \in \mathbb{V}} d(v)\Psi^2 \leq \Delta(\mathcal{G}) \cdot |\mathbb{V}| \cdot \Psi^2 \\
(iii) \quad & \Theta^{\Sigma} \leq M \cdot \Theta \cdot (l(\mathcal{G}) + 1) \\
(iv) \quad & \Theta^{\Sigma} \leq M \cdot |\Gamma|^2 \cdot (l(\mathcal{G}) + 1) \\
(v) \quad & \sum_{e \in \mathbb{E}} \Psi(e) = \sum_{r \in \Gamma} |E_r| \leq M \cdot |\Gamma| \cdot l(\mathcal{G})
\end{aligned}
$$

Beweis. Zu (i): Nach Lemma 4.2.30(iv) folgt

$$\Theta^{\Sigma} = \sum_{v \in \mathbb{V}} \Theta(v) = \sum_{v \in \mathbb{V}} \Theta(\Gamma_v) \leq \sum_{v \in \mathbb{V}} |\Gamma_v|^2.$$

Zu (ii): Nach Lemma 4.2.30(iii) folgt wegen $d(v) \leq \Delta(\mathcal{G})$

$$\Theta^{\Sigma} = \sum_{v \in \mathbb{V}} \Theta(v) \leq \sum_{v \in \mathbb{V}} d(v)\Psi^2 \leq \Delta(\mathcal{G}) \cdot |\mathbb{V}| \cdot \Psi^2.$$

Zu (iii): Eine Kommunikationslinie enthält maximal $l(\mathcal{G})+1$ Knoten, ein Request somit maximal $M \cdot (l(\mathcal{G}) + 1)$ Knoten. Daher können zwei Requests einen Konflikt in maximal $M \cdot (l(\mathcal{G}) + 1)$ Knoten haben. Durch Aufsummierung über alle Konflikte folgt die Ungleichung.

Zu (iv): Dieses folgt wegen $\Theta = \Theta(\mathbb{V}) \leq |\Gamma^2_{\mathbb{V}}| = |\Gamma|^2$ aus (iii).

Zu (v): In der linken Summe wird jedes Request für jede Kante, in der es vorkommt, einmal gezählt, also $|E_r|$-mal. Somit gilt die erste Gleichung. Die zweite gilt unmittelbar wegen $|E_r| \leq M \cdot l(\mathcal{G})$

□

Lemma 4.2.32. *Sei $\mathcal{N}$ das Netzwerk und $v \in \mathbb{V}$. Dann gilt*

$$\sum_{e \in E(v)} \Psi(e)^2 \in \mathcal{O}(|\Theta(v)| + |\Gamma_v|).$$

Beweis. Für $e \in E(v)$ sei $R_{in}(e) := \{r \in \Gamma_e : e^{r+} = v\}$ als die Menge aller durch e in v hinein laufenden Requests und $R_{out}(e) := \{r \in \Gamma_e : e^{r-} = v\}$ die Menge aller durch e aus v heraus laufenden Requests. Dann ist $\Gamma_e = R_{in} \cup R_{out}$.

Dann gilt

- In Halbduplex-Netzwerken haben $r \neq s$ einen Konflikt in e genau dann, wenn $(r, s) \in \Gamma_e^2 \supset (R_{in}(e))^2 \cup (R_{out}(e))^2$.
- In Vollduplex-Netzwerken haben $r \neq s$ einen Konflikt in e genau dann, wenn $(r, s) \in (R_{in}(e))^2 \cup (R_{out}(e))^2$.

Insbesondere gilt immer : $(r,s) \in (R_{in}(e))^2 \cup (R_{out}(e))^2 \implies r \leftrightsquigarrow_e s \vee r = s$ und somit

$$|R_{in}(e)|^2 + |R_{out}(e)|^2 \leq |\{(r,s) \in \Gamma_e^2 : r \leftrightsquigarrow_e s \vee r = s\}|. \tag{4.1}$$

Weiter gilt $2|R_{in}(e)||R_{out}(e)| \leq |R_{in}(e)|^2 + |R_{out}(e)|^2$ wegen

$$\begin{aligned} & 0 \leq (|R_{in}(e)| - |R_{out}(e)|)^2 \\ \iff & 0 \leq |R_{in}(e)|^2 - 2|R_{in}(e)||R_{out}(e)| + |R_{out}(e)|^2 \\ \iff & 2|R_{in}(e)||R_{out}(e)| \leq |R_{in}(e)|^2 + |R_{out}(e)|^2. \end{aligned} \tag{4.2}$$

Somit ist

$$\begin{aligned} \Psi(e)^2 = |\Gamma_e|^2 &= |R_{in}(e) \cup R_{out}(e)|^2 \\ &= (|R_{in}(e)| + |R_{out}(e)|)^2 = |R_{in}(e)|^2 + 2|R_{in}(e)||R_{out}(e)| + |R_{out}(e)|^2 \\ &\overset{(4.2)}{\leq} 2|R_{in}(e)|^2 + 2|R_{out}(e)|^2 \leq 2(|R_{in}(e)|^2 + |R_{out}(e)|^2) \\ &\overset{(4.1)}{\leq} 2|\{(r,s) \in \Gamma_e^2 : r \leftrightsquigarrow_e s \vee r = s\}| \\ &= 2|\{(r,s) \in \Gamma_e^2 : r \leftrightsquigarrow_e s\}| + 2|\{(r,s) \in \Gamma_e^2 : r = s\}| \\ &= 2|\{(r,s) \in \Gamma_e^2 : r \leftrightsquigarrow_e s\}| + 2|\Gamma_e|. \end{aligned}$$

Es folgt

$$\begin{aligned} \sum_{e \in E(v)} \Psi(e)^2 &\leq 2 \sum_{e \in E(v)} \left(|\{(r,s) \in \Gamma_e^2 : r \leftrightsquigarrow_e s\}| + |\Gamma_e| \right) \\ &= 2 \sum_{e \in E(v)} |\{(r,s) \in \Gamma_e^2 : r \leftrightsquigarrow_e s\}| + 2 \sum_{e \in E(v)} |\Gamma_e| =: 2S_1 + 2S_2. \end{aligned}$$

Zur Abschätzung der Summe S_1: Dort wird jedes Tupel $(r,s) \in \Gamma_v^2$ für jede Kante $e \in E(v)$ gezählt, in denen r und s einen Konflikt haben. Da zwei Requests in maximal $(M+1)$ Kanten einen Konflikt haben, gilt

$$S_1 = \sum_{(r,s) \in \Gamma_v^2} |\{e \in E(v) : r \leftrightsquigarrow_e s\}| \leq |\Theta(v)| \cdot (M+1).$$

Zur Abschätzung der Summe S_2: Es wird jedes Request $r \in \Gamma_v$ für jede zu v inzidente Kante $e \in E_r$ gezählt. Somit ist

$$S_2 = \sum_{r \in \Gamma_v} |V_r \cap E(v)| \leq |\Gamma_v| \cdot (M+1).$$

Es folgt

$$\sum_{e \in E(v)} \Psi(e)^2 \leq 2(M+1) \cdot (|\Theta(v)| + |\Gamma_v|) \in \mathcal{O}(|\Theta(v)| + |\Gamma_v|).$$

□

4.3 Schedules

4.3.1 Allgemeine Eigenschaften

Ein *Schedule* α für eine Menge R von Requests ist eine Abbildung, die jedem Request $r \in R$ einen Zeitpunkt $\alpha(r)$ zuordnet, an dem seine Ausführung beginnt. Der Request wird nun während des Zeitintervalls $[\alpha(r), \alpha(r) + \delta(r))$ ausgeführt. Zwei in Konflikt stehende Requests dürfen dabei niemals gleichzeitig ausgeführt werden.

Definition 4.3.1. *Sei $R \subset \Gamma$ eine Menge von Requests und $\alpha : R \mapsto \mathbb{R}^{\geq 0}$. Dann definiert man eine Abbildung*

$$\widehat{\alpha} : R \mapsto \mathcal{P}(\mathbb{R}), \qquad \widehat{\alpha}(r) := [\alpha(r), \alpha(r) + \delta(r)).$$

Offenbar gilt

$$\begin{aligned} &\widehat{\alpha}(s) \neq \emptyset, \\ &\alpha(r) = \alpha(s) \wedge \delta(r) = \delta(s) \iff \widehat{\alpha}(r) = \widehat{\alpha}(s). \end{aligned}$$

Definition 4.3.2 (Schedule, Länge, Aktivitätsintervall)**.** *Sei $R \subset \Gamma$. Eine Abbildung $\alpha : R \mapsto \mathbb{R}^{\geq 0}$ heißt* Schedule für R*, wenn für $r, s \in R$ gilt:*

$$r \leftrightsquigarrow s \implies \widehat{\alpha}(r) \cap \widehat{\alpha}(s) = \emptyset.$$

Sei α ein Schedule für R. Dann heißt $|\alpha| := \max\{\alpha(r) + \delta(r) : r \in R\}$ Länge *von α und $\widehat{\alpha}(r)$* Aktivitätsintervall *von r.*

Für einen Schedule α ist das Aktivitätsintervall $\widehat{\alpha}(r)$ eines Requests r gerade die Zeitspanne, während der das Request r ausgeführt wird.

Satz 4.3.3 (Maximale Lastdauer ist untere Schranke für die Länge eines Schedules)**.** *Sei $R \subset \Gamma$ und α ein Schedule für R.*

(i) In HD-Netzwerken ist die maximale Lastdauer eine untere Schranke für die Länge des Schedules, d.h. es ist $|\alpha| \geq \Phi_R$.

(ii) In VD-Netzwerken ist die maximale gerichtete Lastdauer eine untere Schranke für die Länge des Schedules, d.h. es ist $|\alpha| \geq \overrightarrow{\Phi}_R$.

Beweis. Zu (i): Wäre $|\alpha| < \Phi_R$, so gäbe es eine Kante $e \in \mathbb{E}$ mit $|\alpha| < \Phi_R(e)$. Dann gibt es zwei verschiedene durch e verlaufende Requests r und s mit $\widehat{\alpha}(r) \cap \widehat{\alpha}(s) \neq \emptyset$. Jedoch haben r und s wegen $e \in E_r \cap E_s$ einen Konflikt in e. Widerspruch zur Definition des Schedules.

Zu (ii): Wäre $|\alpha| < \overrightarrow{\Phi}_R$, so gäbe es eine Kante $e = \{v, w\} \in \mathbb{E}$ mit $|\alpha| < \overrightarrow{\Phi}_R(e)$. Sei nun $R_v = \{r \in R : e \in E_r \wedge e^{r-} = v\}$ und o.B.d.A.

$$\overrightarrow{\Phi}_R(e) = \sum_{\substack{r \in R \\ e \in E_r, e^{r-} = x}} \delta(r) = \sum_{r \in R_v} \delta(r).$$

Nun haben je zwei verschiedene Requests $r, s \in R_v$ einen Konflikt. Wegen $|\alpha| < \overrightarrow{\Phi}_R(e)$ gibt es $r, s \in R_v$ mit $\widehat{\alpha}(r) \cap \widehat{\alpha}(s) \neq \emptyset$. Widerspruch zur Definition des Schedules.

□

Definition 4.3.4 (Synchrone Schedules). *Seien $R, S \subset \Gamma$. Zwei Schedules $\alpha : R \mapsto \mathbb{R}^{\geq 0}$ und $\beta : S \mapsto \mathbb{R}^{\geq 0}$ heißen* synchron, *wenn gilt*

$$\begin{aligned} &(i) && \alpha_{|R\cap S} = \beta_{|R\cap S}, \\ &(ii) && r \in R, s \in S \text{ haben einen Konflikt} \implies \widehat{\alpha}(r) \cap \widehat{\beta}(s) = \emptyset. \end{aligned}$$

Eine Menge von Schedules heißt synchron, *wenn je zwei Schedules synchron sind.*

Definition 4.3.5. *Zwei Mengen $R, S \subset \Gamma$ heißen* konfliktfrei, *wenn kein $r \in R \setminus S$ und $s \in S \setminus R$ mit $r \leftrightsquigarrow s$ existiert, d.h. wenn für zwei in Konflikt stehende Requests $r, s \in R \cup S$ gilt: $r, s \in R$ oder $r, s \in S$.*

Satz 4.3.6. *Seien $R, S \subset \Gamma$ konfliktfreie Mengen und α ein Schedule für R und β ein Schedule für S. Dann sind α und β genau dann synchron, wenn $\alpha_{|R\cap S} = \beta_{|R\cap S}$ gilt.*

Beweis. Die Hinrichtung ist trivial. Sei nun α ein Schedule für R und β ein Schedule für S mit $\alpha_{|R\cap S} = \beta_{|R\cap S}$. Dann bleibt die Bedingung (ii) für Synchronität zu zeigen. Sei $r \in R$ und $s \in S$ mit $r \leftrightsquigarrow s$. Nach Voraussetzung ist dann $r \in S$ oder $s \in R$. Falls $r \in S$, so ist $\widehat{\alpha}(r) \cap \widehat{\beta}(s) = \widehat{\beta}(r) \cap \widehat{\beta}(s) = \emptyset$. Falls $s \in R$, so folgt die Behauptung analog. □

Lemma 4.3.7 (Induzierter Schedule). *Sei $S \subset R \subset \Gamma$ und $\alpha : R \mapsto \mathbb{R}^{\geq 0}$ ein Schedule. Dann ist $\alpha_{|S} : S \mapsto \mathbb{R}^{\geq 0}$ ein Schedule für S mit der Länge $|\alpha_{|S}| \leq |\alpha|$.*

Der so definierte Schedule heißt von α auf S induzierter Schedule.

Beweis. Es gilt für $r, s \in S$

$$r \leftrightsquigarrow s \implies \widehat{\alpha}(r) \cap \widehat{\alpha}(s) = \emptyset \implies \widehat{\alpha_{|S}}(r) \cap \widehat{\alpha_{|S}}(s) = \emptyset.$$

Offenbar ist weiter

$$\begin{aligned} |\alpha_{|S}| &= \max\{\alpha_{|S}(r) + \delta(r) : r \in S\} \\ &= \max\{\alpha(r) + \delta(r) : r \in S\} \\ &\leq \max\{\alpha(r) + \delta(r) : r \in R\} \\ &= |\alpha|. \end{aligned}$$

□

Satz 4.3.8 (Synchronität induzierter Schedules). *Sei $S, T \subset R \subset \Gamma$ und α ein Schedule für R . Dann sind die Schedules $\alpha_{|S}$ und $\alpha_{|T}$ synchron.*

Sind $S_i \subset R \subset \Gamma$ für $i \in I$, so sind die Schedules $\alpha_{|S_i}$ synchron.

Beweis. Nach Lemma 2.1.6 ist

$$(\alpha_{|S})_{|S\cap T} = \alpha_{|S\cap T} = (\alpha_{|T})_{|S\cap T}.$$

Weiter gilt:

$$s \in S, t \in T \text{ haben einen Konflikt } \implies \widehat{\alpha}(s) \cap \widehat{\alpha}(t) = \emptyset \implies \widehat{\alpha_{|S}}(s) \cap \widehat{\alpha_{|T}}(t) = \emptyset.$$

Ist $S_i \subset R \subset \Gamma$ für $i \in I$, so sind für je zwei $j, k \in I$ die Schedules $\alpha_{|S_j}$ und $\alpha_{|S_k}$ synchron, also ist die Menge der Schedules $\alpha_{|S_i}$, $i \in I$ synchron. □

Satz und Definition 4.3.9 (Synchrone Vereinigung zweier Schedules)**.** *Seien $R, S \subset \Gamma$ und $\alpha : R \mapsto \mathbb{R}^{\geq 0}$ und $\beta : S \mapsto \mathbb{R}^{\geq 0}$ synchrone Schedules für R und S. Dann ist*

$$\alpha \uplus \beta : R \cup S \mapsto \mathbb{R}^{\geq 0}, \qquad \alpha \uplus \beta(r) = \begin{cases} \alpha(r) & \textit{falls } r \in R \\ \beta(r) & \textit{falls } r \notin R \end{cases}$$

ein Schedule für $R \cup S$ mit $|\alpha \uplus \beta| = \max(|\alpha|, |\beta|)$. Es ist $\alpha = (\alpha \uplus \beta)_{|R}$ und $\beta = (\alpha \uplus \beta)_{|S}$.

Der Schedule $\alpha \uplus \beta$ heißt synchrone Vereinigung von α und β.

Beweis. Wir zeigen, dass $\alpha \uplus \beta$ wohldefiniert und ein Schedule ist.

- *$\alpha \uplus \beta$ ist wohldefiniert:* Es tritt für $r \in R \cup S$ genau einer der Fälle $r \in R$ oder $r \notin R$ ein. Falls $r \notin R$, so ist $r \in S$, also ist $\beta(r)$ definiert. Falls $r \in R$, so ist offenbar $\alpha(r)$ definiert.
- *$\alpha \uplus \beta$ ist ein Schedule:*

  Wegen der Synchronität von α und β gilt für $r \in R \cap S$

  $$\alpha \uplus \beta(r) = \alpha(r) = \beta(r).$$

  Für in Konflikt stehende Requests $r, s \in R \cup S$ bleibt zu zeigen:

  $$\widehat{\alpha \uplus \beta}(r) \cap \widehat{\alpha \uplus \beta}(s) \neq \emptyset.$$

  Falls $r \in R$ und $s \in S$, so gilt dieses nach Definition der Synchronität. Falls $r, s \in R$ so gilt

  $$\widehat{\alpha \uplus \beta}(r) \cap \widehat{\alpha \uplus \beta}(s) = \widehat{\alpha}(r) \cap \widehat{\alpha}(s) \neq \emptyset.$$

  Falls $r, s \in S$ so gilt analog

  $$\widehat{\alpha \uplus \beta}(r) \cap \widehat{\alpha \uplus \beta}(s) = \widehat{\beta}(r) \cap \widehat{\beta}(s) \neq \emptyset.$$

- *Die induzierten Schedules sind α bzw. β:* Nun ist für $r \in R$

  $$(\alpha \uplus \beta)_{|R}(r) = \alpha \uplus \beta(r) = \alpha(r)$$

  und für $s \in S$

  $$(\alpha \uplus \beta)_{|S}(s) = \alpha \uplus \beta(s) = \beta(s).$$

  Somit folgt $(\alpha \uplus \beta)_{|R} = \alpha$ und $(\alpha \uplus \beta)_{|S} = \beta$.

- *Es ist* $|\alpha \uplus \beta| = \max(|\alpha|, |\beta|)$. Unmittelbar sieht man nun

$$\begin{aligned}
|\alpha \uplus \beta| &= \max\{\alpha \uplus \beta(r) + \delta(r) : r \in R \cup S\} \\
&= \max\{\{\alpha \uplus \beta(r) + \delta(r) : r \in R\} \cup \{\alpha \uplus \beta(r) + \delta(r) : r \in S\}\} \\
&= \max\{\{\alpha(r) + \delta(r) : r \in R\} \cup \{\beta(r) + \delta(r) : r \in S\}\} \\
&= \max(\max\{\alpha(r) + \delta(r) : r \in R\}, \max\{\beta(r) + \delta(r) : r \in S\}) \\
&= \max(|\alpha|, |\beta|).
\end{aligned}$$

□

Die Vereinigung synchroner Schedules ist kommutativ, d.h. es ist $\alpha \uplus \beta = \beta \uplus \alpha$.

Satz und Definition 4.3.10 (Sychrone Vereinigungen mehrerer Schedules)**.** *Sind α_i synchrone Schedules für R_i, $i \in I$, so ist für $R := \bigcup_{i \in I} R_i$*

$$\begin{aligned}
&\alpha : R \mapsto \mathbb{R}^{\geq 0}, \\
&r \to \alpha_i(r) \qquad \textit{für ein } i \textit{ mit } r \in R_i
\end{aligned}$$

ein Schedule für R und es gilt $\alpha_{|R_i} = \alpha_i$.

Der Schedule α heißt synchrone Vereinigung *von α_i, $i \in I$ und wird bezeichnet mit*

$$\biguplus_{i \in I} \alpha_i := \alpha.$$

Beweis.

- *Die Abbildung α ist wohldefiniert:* Zu jedem $r \in R$ ist $\alpha(r)$ eindeutig definiert. Zur Eindeutigkeit: Sei $r \in R_i$ und $r \in R_j$. Dann ist $r \in R_i \cap R_j$ und wegen der Synchronität von α_i und α_j folgt $\alpha := \alpha_i(r) = \alpha_j(r)$.
  Zur Existenz: Sei $r \in R = \bigcup_{i \in I} R_i$, dann gibt es ein $i \in I$ mit $r \in R_i$ und $\alpha(r) = \alpha_i(r)$ ist definiert.

- *α ist ein Schedule:* Seien $r, s \in R$ in Konflikt stehende Requests. Dann ist $r \in R_i$ und $s \in R_j$ und wegen der Synchronität von α_i und α_j folgt

$$\widehat{\alpha}(r) \cap \widehat{\alpha}(s) = \widehat{\alpha_i}(r) \cap \widehat{\alpha_j}(s) = \emptyset.$$

- *Es gilt $\alpha_{|R_i} = \alpha_i$:* Für $r \in R_i$ gilt $\alpha_{|R_i}(r) = \alpha(r) = \alpha_i(r)$.

□

Lemma 4.3.11. *Seien $\alpha_i : R_i \mapsto \mathbb{R}^{\geq 0}$ für $i = 1, \ldots, n$ Schedules. Dann ist äquivalent:*

(i) Die Menge $\{\alpha_i : i = 1, \ldots, n\}$ ist eine Menge synchroner Schedules.

(ii) Für $j = 1, \ldots, n$ ist α_j synchron zu $\biguplus_{i=1}^{j-1} \alpha_i$.

Beweis. Zur Hinrichtung $(i) \implies (ii)$: Sei

$$R = \bigcup_{i=1}^{j-1} R_i \qquad \text{und} \qquad \alpha = \biguplus_{i=1}^{j-1} \alpha_i.$$

Für $r \in R \cap R_j$ gibt es ein $k \in \{1, \ldots, j-1\}$ mit $r \in R_k$. Dann gilt nach Definition und Satz 4.3.10 $\alpha(r) = \alpha_k(r)$ und wegen der Synchronität von α_k und α_j die Gleichung $\alpha_k(r) = \alpha_j(r)$. Somit gilt:

$$\forall r \in R \cap R_j : \alpha(r) = \alpha_k(r) = \alpha_j(r) \implies \alpha_{|R \cap R_j} = \alpha_{j|R \cap R_j}.$$

Seien $r \in R$ und $s \in R_j$ in Konflikt stehende Requests. Dann gibt es ein $k \in \{1, \ldots, j-1\}$ mit $r \in R_k$ und es gilt

$$\begin{aligned} & r \leftrightsquigarrow s \\ \implies & \widehat{\alpha_k}(r) \cap \widehat{\alpha_j}(s) = \emptyset \\ \implies & \widehat{\alpha}(r) \cap \widehat{\alpha_j}(s) = \emptyset. \end{aligned}$$

Zur Rückrichtung $(ii) \implies (i)$: Angenommen, es gäbe zwei nicht synchrone Schedules α_j und α_l, o.B.d.A. $j > l$. Dann tritt mindestens einer der beiden folgenden Fälle ein:

- Für ein Request $r \in R_j \cap R_l \subset R_j \cap \bigcup_{i=1}^{k} R_i$ gilt

  $$\alpha_j(r) \neq \alpha_l(r) = \biguplus_{i=1}^{j-1} \alpha_i(r),$$

  also sind α_j und $\biguplus_{i=1}^{j-1} \alpha_i$ nicht synchron. Widerspruch.

- Für zwei in Konflikt stehende Requests $r \in R_j$, $s \in R_l \subset \bigcup_{i=1}^{k} R_i$ gilt

  $$\emptyset \neq \widehat{\alpha_j}(r) \cap \widehat{\alpha_l}(s) = \widehat{\alpha_j}(r) \cap \widehat{\biguplus_{i=1}^{j-1} \alpha_i(s)},$$

  also sind α_j und $\biguplus_{i=1}^{j-1} \alpha_i$ nicht synchron. Widerspruch.

□

4.3.2 Schedules zu Knotenmengen

Definition 4.3.12 (Schedule für Knotenmenge V)**.** *Sei $V \subset \mathbb{V}$ zusammenhängend. Ein Schedule für Γ_V heißt Schedule für die Knotenmenge V. Im Folgenden nennen wir einen Schedule für* $\mathbb{V}$ globalen Schedule *und einen Schedule für* $\{v\}$ lokalen Schedule für v.

Satz 4.3.13 (Globaler Schedule induziert synchrone lokale Schedules)**.** *Sei $\alpha : \mathbb{V} \mapsto \mathbb{R}^{\geq 0}$ ein globaler Schedule. Dann sind für alle $v \in \mathbb{V}$ die induzierten Schedules $\alpha_{|\Gamma_v}$ synchron.*

Beweis. Dieses folgt unmittelbar aus Satz 4.3.8. □

Satz 4.3.14 (Synchronität benachbarter Schedules äquivalent zu globaler Synchronität)**.** *Sei $V \subset \mathbb{V}$ zusammenhängend und für jedes $v \in V$ gebe es einen lokalen Schedule $\alpha_v : \Gamma_v \mapsto \mathbb{R}^{\geq 0}$. Dann sind die beiden folgenden Aussagen äquivalent:*

(i) Je zwei Schedules α_v und α_w für beliebige Knoten $v, w \in V$ sind synchron.

(ii) Je zwei Schedules α_v und α_w für benachbarte Knoten $v, w \in V$ sind synchron.

Gilt eine der beiden Aussagen, so gibt es einen Schedule β für V mit $\beta_{|\Gamma_v} = \alpha_v$.

Beweis. Die Richtung $(i) \implies (ii)$ ist trivial. Zu $(ii) \implies (i)$: Wir zeigen für beliebige v und w die beiden Kriterien der Synchronität.

- Für beliebige $v, w \in V$ und $r \in \Gamma_v \cap \Gamma_w$ ist $\alpha_v(r) = \alpha_w(r)$:
  Sei $P = (v = x_0, x_1, \ldots, x_{n-1}, x_n = w)$ der eindeutig bestimmte Weg von v nach w. Nach Lemma 4.2.23 gilt dann $r \in \Gamma_{x_i}$ für alle x_i. Somit folgt wegen der Synchronität benachbarter Schedules
  $$\alpha_v(r) = \alpha_{x_0}(r) = \alpha_{x_1}(r) = \cdots = \alpha_{x_n}(r) = \alpha_w(r).$$

- Für beliebige $v, w \in V$ und in Konflikt stehende $r \in \Gamma_v$ und $s \in \Gamma_w$ ist $\widehat{\alpha_v}(r) \cap \widehat{\alpha_w}(s) \neq \emptyset$:
  Haben $r \in \Gamma_v$ und $s \in \Gamma_w$ einen Konflikt in einem Knoten z, so gibt es einen eindeutig bestimmten Weg P von v nach z und Q von w nach z. Wegen $r \in \Gamma_v \cap \Gamma_z$ und $s \in \Gamma_w \cap \Gamma_z$ gilt nach der bereits bewiesenen Aussage $\alpha_v(r) = \alpha_z(r)$ und $\alpha_w(s) = \alpha_z(s)$ und somit
  $$\widehat{\alpha_v}(r) \cap \widehat{\alpha_w}(s) = \widehat{\alpha_z}(r) \cap \widehat{\alpha_z}(s) = \emptyset.$$

Gilt eine der beiden Aussagen, so lässt sich β definieren als

$$\beta := \biguplus_{v \in V} \alpha_v.$$

□

Satz 4.3.15. *Seien $V, W \subset \mathbb{V}$ zusammenhängende benachbarte Mengen. Dann sind Γ_V und Γ_W konfliktfrei.*

Beweis. Seien $r \in \Gamma_V$ und $s \in \Gamma_W$ in Konflikt stehende Requests. Nach Lemma 4.2.26 haben sie dann einen Konflikt in V oder in W. Somit ist auch $r \in \Gamma_W$ oder $s \in \Gamma_V$. □

Satz 4.3.16. *Sei $\mathcal{N}$ kein VD-MC-Netzwerk. Sei $V \subset \mathbb{V}$ und $\alpha : \Gamma_V \mapsto \mathbb{R}^{\geq 0}$ eine Abbildung. Dann ist α ein Schedule für V genau dann, wenn gilt*

$$r \leftrightsquigarrow_V s \implies \widehat{\alpha}(r) \cap \widehat{\alpha}(s) = \emptyset. \tag{4.3}$$

Beweis. Ist nun α ein Schedule für V, so ist nach Satz 4.2.24

$$r \leftrightsquigarrow_V s \implies r \leftrightsquigarrow s \implies \widehat{\alpha}(r) \cap \widehat{\alpha}(s) = \emptyset.$$

Ist umgekehrt die Aussage aus (4.3) erfüllt, so gilt für in Konflikt stehende r und s nach Satz 4.2.24

$$r \leftrightsquigarrow s \implies r \leftrightsquigarrow_V s \implies \widehat{\alpha}(r) \cap \widehat{\alpha}(s) = \emptyset.$$

und somit ist α ein Schedule. □

4.3.3 Schedules in Broadcast-Netzwerken

Neben Unicast- und Multicast-Netzwerken gibt es Broadcast-Netzwerke. Dort gibt es ausschließlich Requests von einem Device zu allen anderen. Das heißt, jedes Request r hat einen Kommunikationsbaum $KB(d, \mathbb{D} \setminus \{d\})$. Ein Broadcast-Netzwerk ist somit nur ein spezielles Multicast-Netzwerk. Abgesehen von trivialen Netzwerken mit nur zwei Devices haben dann je zwei Requests einen Konflikt.

Lemma 4.3.17. *Es gebe mindestens drei Devices. Dann haben je zwei Requests einen Konflikt.*

Beweis. Sei r das Request von d_1 zu $D \setminus \{d_1\}$ und s von d_2 zu $D \setminus \{d_2\}$. Dann gibt es ein weiteres Device d_3. Nun enthalten $KB(r)$ und $KB(s)$ die Kommunikationslinie $KL(d_1, d_3)$ bzw. $KL(d_2, d_3)$. Da d_3 ein Blatt ist, gibt es eine eindeutige zu d_3 inzidente Kante e. Es ist $e \in E_{KL(d_1,d_3)} \cap E_{KL(d_2,d_3)}$ und $e^{KL(d_1,d_3)+} = d_3 = e^{KL(d_2,d_3)+}$ und somit

$$KL(d_1, d_3) \leftrightsquigarrow KL(d_2, d_3) \implies KB(r) \leftrightsquigarrow KB(s) \implies r \leftrightsquigarrow s.$$

□

Für triviale Netzwerke mit zwei Devices hat $\mathcal{G}$ die Form eines Weges und die Requests entsprechen Unicast-Requests. Nach Korollar 4.2.5 haben in diesem Fall in Halbduplex-Netzwerken je zwei Requests einen Konflikt, in Vollduplex-Netzwerken haben zwei Requests genau dann einen Konflikt, wenn sie dieselbe Richtung haben.

Unmittelbar ergibt sich für Broadcast-Netzwerke der folgende Satz:

Satz 4.3.18. *Ist das Broadcast-Netzwerk $\mathcal{N}$ ein Vollduplex-Netzwerk mit mindestens 3 Knoten oder ein Halbduplex-Netzwerk, so ist für die Requests $R = \{r_i : i = 1, \ldots, n\}$ der rekursiv definierte Schedule*

$$\alpha : R \mapsto \mathbb{R}^{\geq 0}, \qquad r_k \to \sum_{i=1}^{k-1} \delta(r_i)$$

optimal und hat die Länge

$$\sum_{r \in R} \delta(r).$$

Beweis. Da je zwei Requests in Konflikt stehen, können keine zwei Requests gleichzeitig ausgeführt werden. Werden alle Requests unmittelbar hintereinander geschedulet, so entsteht obiger optimaler Schedule. □

Bemerkung 4.3.19 (Broadcast-Vollduplex-Netzwerke mit zwei Devices)**.** Ist das Netzwerk $\mathcal{N}$ ein Broadcast-Vollduplex-Netzwerk mit nur zwei Devices d_1, d_2, so definiert man R_1 als die Menge der Requests von d_1 nach d_2 und R_2 als die Menge der Requests von d_2 nach d_1. Nun definiert man α_1 bzw. α_2 als den in Satz 4.3.18 beschriebenen Schedule für R_1 bzw. R_2. Da R_1 und R_2 disjunkt sind und keine zwei Requests $r_1 \in R_1$ und $r_2 \in R_2$ einen Konflikt haben, sind α_1 und α_2 synchronisierbar und offenbar $\alpha := \alpha_1 \uplus \alpha_2$ ein Schedule der Länge $|\alpha| := \max(|\alpha_1|, |\alpha_2|)$. Dieser ist auf Grund der Optimalität von α_1 und α_2 optimal.

4.4 Konfliktgraphen

Die Konfliktstrukturen einer Requestmenge R können in Knoten- oder Kantenkonfliktgraphen gespeichert werden.

In Knotenkonfliktgraphen werden die Requests durch Knoten repräsentiert und zwei Knoten sind genau dann benachbart, wenn sie einen Konflikt haben. Wird der Konfliktgraph in einer Adjazenzliste gespeichert, lassen sich für einen Knoten $r \in R$ in $\mathcal{O}(d(r))$ alle zu r in Konflikt stehenden Requests auflisten. Das Prüfen, ob zwei Requests einen Konflikt haben, dauert ebenfalls $\mathcal{O}(d(r))$. Wird er in einer Adjazenzmatrix gespeichert, so lässt sich in konstanter Zeit prüfen, ob zwei Requests einen Konflikt haben. Das Auflisten aller mit r in Konflikt stehenden Requests dauert $\mathcal{O}(|R|)$.

In Kantenkonfliktgraphen werden die Requests durch Kanten repräsentiert und zwei Kanten sind genau dann adjazent, wenn sie einen Konflikt haben. Eine solche Darstellung ist nur dann für jede Requestmenge möglich, wenn auch Multihypergraphen zugelassen werden. Der Kantenkonfliktgraph für eine Menge Γ_v ist jedoch ein Multigraph.

Haben alle Requests dieselbe Übertragungslänge, so ist das Finden eines Schedules sogar äquivalent zur Färbung des Konfliktgraphen. Dieser Zusammenhang wird in Abschnitt 6.1 näher untersucht werden.

4.4.1 Knotenkonfliktgraphen

Zu einer Menge R von Requests definieren wir den Knotenkonfliktgraphen $C(R)$. Bei diesem entsprechen die Requests den Knoten und zwei Requests sind genau dann benachbart, wenn sie einen Konflikt haben.

Definition 4.4.1 (Knotenkonfliktgraph zu R)**.** *Sei $R \subset \Gamma$. Dann definiere den* Knotenkonfliktgraph $C(R) = (V, E)$ *durch*

$$\begin{aligned} V &:= R, \\ E &:= \left\{ \{r, s\} \in V^{\underline{2}} : r \leftrightsquigarrow s \right\}. \end{aligned}$$

Satz 4.4.2. *Für den Knotenkonfliktgraph $C(R)$ gilt:*

(i) Der Knotenkonfliktgraph $C(R)$ ist einfach und schlingenfrei.

(ii) Zwei Requests $r, s \in R$ haben genau dann einen Konflikt, wenn sie in $C(R)$ benachbart sind.

Beweis. Zu (i): Die Einfachheit ist offensichtlich. Des Weiteren steht kein Request mit sich selbst in Konflikt, daher ist $C(R)$ schlingenfrei.

Zu (ii): Nach Definition von $C(R)$ gilt $r \leftrightsquigarrow s \iff \{r, s\} \in E$.

□

4.4.2 Kantenkonfliktgraphen

Im Gegensatz zum Knotenkonfliktgraph, in dem die Requests den Knoten entsprechen und zwei Knoten genau im Falle eines Konflikts benachbart sind, werden die Requests im Kantenkonfliktgraph durch Kanten repräsentiert. Zwei Kanten sind genau dann adjazent, wenn die entsprechenden Requests einen Konflikt haben. Der Kantenkonfliktgraph ist der Liniengraph des Knotenkonfliktgraphen.

Wir definieren den Kantenkonfliktgraphen für Requestmengen Γ_v durch einen Knoten v nun für die verschiedenen Netzwerktypen. Dabei wird jedoch kein Kantenkonfliktgraph für VD-MC-Netzwerke definiert, da sich dort nicht lokal an dem Knoten v erkennen lässt, ob zwei Requests $r, s \in \Gamma_v$ einen Konflikt haben.

4.4.2.1 HD-UC-Kantenkonfliktgraph zu v

Satz und Definition 4.4.3 (HD-UC-Kantenkonfliktgraph zu v)**.** *Sei $\mathbb{V}$ ein HD-UC-Netzwerk und v kein Blatt. Dann definiere den ungerichteten Multigraphen $C = (V, E, g)$ zu v durch*

$$\begin{aligned} V &:= \mathbb{E}(v), \\ E &:= \Gamma_v, \\ g : E &\mapsto V^{\underline{2}}, \qquad r \to E_{KL(r)}(v). \end{aligned}$$

*Im Folgenden bezeichnen wir C als HD-UC-*Kantenkonfliktgraph zu v.

Dann haben zwei Requests $r, s \in \Gamma_v$ in einem HD-UC-Netzwerk genau dann einen Konflikt, wenn sie im HD-UC-Kantenkonfliktgraphen zu v adjazent sind.

Beweis. Damit der Graph wohldefiniert ist, muss für alle $r \in E$ auch $E_{KL(r)}(v) \in V^{\underline{2}}$ gelten. Dieses ist nach Definition 2.2.9 jedoch sichergestellt, da v kein Randknoten von $KL(r)$ ist und jeder innere Knoten eines Weges zwei inzidente Kanten hat.

Weiter gilt nach Satz 4.2.4

$$\begin{aligned} & r \leftrightsquigarrow s \\ \iff & r \leftrightsquigarrow_v s \\ \iff & \exists e \in E_{KL(r)}(v) \cap E_{KL(s)}(v) \\ \iff & \exists e \in g(r) \cap g(s) \\ \iff & r \text{ und } s \text{ sind adjazent in } C. \end{aligned}$$

□

Bemerkung 4.4.4 (HD-UC-Kantenkonfliktgraph für Blätter v)**.** Für ein Blatt v haben je zwei Requests $r, s \in \Gamma_v$ einen Konflikt. Daher definiert man den Konfliktgraph $C = (V, E, g)$ durch

$$\begin{aligned} V &:= \{v_1, v_2\}, \\ E &:= \Gamma_v, \\ g : E &\mapsto V^{\underline{2}}, \qquad r \to \{v_1, v_2\}. \end{aligned}$$

Offenbar haben auch hier zwei Requests genau dann einen Konflikt, wenn sie in C adjazent sind.

Dieser Graph lässt sich wie folgt konstruieren: Sei $e = vw$ die einzige zu v inzidente Kante. Dann wird in den Graphen $\mathcal{G}$ ein zusätzlicher Knoten x eingefügt und die Kante vw durch die Kanten vx und xw ersetzt. Anschließend wird der Konfliktgraph für x erzeugt. Dieser entspricht dem oben definierten Konfliktgraphen.

4.4.2.2 HD-MC-Kantenkonfliktgraph zu v

Für einen Kommunikationsbaum K ist $G_K = (V_K, E_K)$ ein Baum, so dass mit $E_K(v)$ die Menge der in G_K zu v inzidenten Kanten bezeichnet wird.

Satz und Definition 4.4.5 (HD-MC-Kantenkonfliktgraph zu v)**.** *Sei $\mathbb{V}$ ein HD-UC-Netzwerk. Dann definiere den ungerichteten Hypermultigraphen $C = (V, E, g)$ zu v durch*

$$\begin{aligned} V &:= \mathbb{E}(v), \\ E &:= \Gamma_v, \\ g &: E \mapsto \mathcal{P}(V), \qquad r \mapsto E_{KB(r)}(v). \end{aligned}$$

*Im Folgenden bezeichnen wir C als HD-MC-*Kantenkonfliktgraph zu v.

Dann haben zwei Requests $r, s \in \Gamma_v$ in einem HD-MC-Netzwerk genau dann einen Konflikt, wenn sie im HD-MC-Kantenkonfliktgraphen zu v adjazent sind.

Beweis. Offenbar ist für alle $r \in E$ auch $g(r) = E_{KB(r)}(v) \in \mathcal{P}(V)$, somit ist C wohldefiniert.

Weiter gilt nach Satz 4.2.13 und Satz 4.2.15

$$\begin{aligned} & r \leftrightsquigarrow s \\ \iff & r \leftrightsquigarrow_v s \\ \iff & \exists e \in E_{KB(r)}(v) \cap E_{KB(s)}(v) \\ \iff & \exists e \in g(r) \cap g(s) \\ \iff & r \text{ und } s \text{ sind adjazent in } C. \end{aligned}$$

□

4.4.2.3 VD-UC-Kantenkonfliktgraph zu v

Die Konstruktion der HD-Kantenkonfliktgraphen basiert wesentlich darauf, dass Requests in Halbduplex-Netzwerken genau dann einen Konflikt in v haben, wenn ihre Kommunikationslinien eine zu v adjazente Kante gemeinsam enthalten. Daher genügte es, jedem Request r (als Kante) die zu v adjazenten Kanten aus E_r als Randpunkte zuzuordnen.

Diese Analogie lässt sich nicht unmittelbar auf Vollduplex-Netzwerke übertragen, da dort die Richtung, in der die Kante durchlaufen wird, eine Rolle spielt. Daher wird jede Kante $e \in \mathbb{E}$ im VD-Kantenkonfliktgraphen durch Knoten e_{in} und e_{out} für die eingehende und die ausgehende Richtung repräsentiert.

Satz und Definition 4.4.6 (*VD-UC*-Kantenkonfliktgraph zu v)**.** *Sei $v \in \mathbb{V}$ kein Blatt. Dann definiere den ungerichteten Konflikt-Multigraphen $C = (V, E, g)$ zu v durch*

$$\begin{aligned} &V := V_{in} \cup V_{out} \qquad \textit{mit} \qquad V_{in} = \{e_{in} : e \in \mathbb{E}(v)\}, \qquad V_{out} = \{e_{out} : e \in \mathbb{E}(v)\}, \\ &E := \Gamma_v, \\ &g : E \mapsto V^{\underline{2}}, \qquad r \mapsto ((v^{KL(r)-})_{in}, (v^{KL(r)+})_{out}). \end{aligned}$$

Im Folgenden bezeichnen wir C als VD-UC-Kantenkonfliktgraph zu v.

Dann haben zwei Requests $r, s \in \Gamma_v$ in einem Vollduplex-Netzwerk genau dann einen Konflikt, wenn sie im VD-Kantenkonfliktgraphen zu v adjazent sind.

Beweis. Der Graph ist offenbar wohldefiniert, da $(v^{KL(r)-})_{in}$ und $(v^{KL(r)+})_{out}$ zu jedem Request $r \in \Gamma_v$ eindeutig definiert sind.

Weiter gilt nach Satz 4.2.4

$$\begin{aligned} & r \leftrightsquigarrow s \\ \iff & r \leftrightsquigarrow_v s \\ \iff & \exists e \in E_{KL(r)}(v) \cap E_{KL(s)}(v) \text{ mit } v = e^{KL(r)+} = e^{KL(s)+} \text{ oder } v = e^{KL(r)-} = e^{KL(s)-} \\ \iff & \exists e \in E_{KL(r)}(v) \cap E_{KL(s)}(v) \text{ mit } e = v^{KL(r)-} = v^{KL(s)-} \text{ oder } e = v^{KL(r)+} = v^{KL(s)+} \\ \iff & \exists e \in E_{KL(r)}(v) \cap E_{KL(s)}(v) \text{ mit } e_{in} = (v^{KL(r)-})_{in} = (v^{KL(s)-})_{in} \\ & \qquad\qquad \text{oder } e_{out} = (v^{KL(r)+})_{out} = (v^{KL(s)+})_{out} \\ \iff & \exists e \in \mathbb{E}(v) \text{ mit } e_{in} \in g(r) \cap g(s) \text{ oder } e_{out} \in g(r) \cap g(s) \\ \iff & g(r) \cap g(s) \neq \emptyset \\ \iff & r \text{ und } s \text{ sind adjazent in } C. \end{aligned}$$

□

Bemerkung 4.4.7 (*VD-UC*-Kantenkonfliktgraph für Blätter v)**.** Sei v ein Blatt und $e = vw$ die einzige zu v adjazente Kante. Dann haben zwei Requests $r, s \in \Gamma_v$ genau dann einen Konflikt, wenn $e^{r-} = e^{s-}$ ist.

Mit $R_v = \{r \in \Gamma_v : e^{r-} = v\}$ und $R_w = \{r \in \Gamma_v : e^{r-} = w\}$ haben je zwei Requests aus R_v und R_w einen Konflikt. Daher definiert man den Konfliktgraph $C = (V, E, g)$ durch

$$\begin{aligned} &V := \{e_{in}, e_{out}, f_{in}, f_{out}\}, \\ &E := \Gamma_v, \\ &g : E \mapsto V^{\underline{2}}, \qquad r \to \begin{cases} \{e_{out}, f_{in}\} & \text{falls } e^{KL(r)-} = v, \\ \{e_{in}, f_{out}\} & \text{falls } e^{KL(r)-} = w. \end{cases} \end{aligned}$$

Offenbar haben auch hier zwei Requests genau dann einen Konflikt, wenn sie in C adjazent sind.

Analog zum *HD-UC*-Fall lässt sich auch dieser Graph wie folgt konstruieren: In den Graphen $\mathcal{G}$ wird ein zusätzlicher Knoten x eingefügt und die Kante vw durch die Kanten vx und xw ersetzt. Anschließend wird der Konfliktgraph für den inneren Knoten x erzeugt. Dieser ist isomorph zu dem oben definierten Konfliktgraphen.

Satz 4.4.8 (Der VD-UC-Kantenkonfliktgraph ist bipartit)**.** *Sei $v \in \mathbb{V}$. Dann ist der VD-UC-Kantenkonfliktgraph $C(E, V, g)$ zu v bipartit.*

Beweis. Wir unterscheiden, ob v ein Blatt ist.

- Sei v kein Blatt. Zu jeder Kante $r \in E$ ist $g(r) = \{(v^{KL(r)-})_{in}, (v^{KL(r)+})_{out}\}$. Also verbindet jede Kante aus E einen Knoten aus V_{in} mit einem Knoten aus V_{out}. Daher sind keine zwei Knoten aus V_{in} bzw. V_{out} benachbart.

  Nun lässt sich leicht eine 2-Färbung für C definieren:

  $$c : V \mapsto \{1, 2\}, \qquad x \to= \begin{cases} 1 & \text{für } x = e_{in} \in V_{in}, \\ 2 & \text{für } x = e_{out} \in V_{out}. \end{cases}$$

  Dieses ist offenbar eine Färbung, denn seien x und y benachbart in C, dann ist o.B.d.A. $x \in V_{in}$ und $y \in V_{out}$ und somit $c(x) = 1 \neq 2 = c(y)$.

- Für ein Blatt v lässt sich eine 2-Färbung für C definieren durch

  $$c : V \mapsto \{1, 2\}, \qquad c(e_{in}) = c(f_{in}) = 1, \qquad c(e_{out}) = c(f_{out}) = 2.$$

□

4.5 Algorithmen

Bemerkung 4.5.1 (Annahme zur Komplexitätsberechnung)**.** Um bei der Angabe der Zeitkomplexität keine Sonderfälle betrachten zu müssen, nehmen wir an, dass durch jede Kante mindestens ein Request verläuft. Gibt es eine Kante, die von keinem Request benötigt wird, so kann diese im Modell weggelassen werden.

Des Weiteren nehmen wir an, dass in MC-Netzwerken die Anzahl $|Ziel(r)|$ der Zieldevices eines Requests r durch eine Konstante M beschränkt ist. Im Unicast-Fall definieren wir $M := 1$.

Lemma 4.5.2 (Hilfsmittel zur Komplexitätsberechnung)**.** *Es ist*

$$\mathcal{O}\left(\sum_{v \in \mathbb{V}} |\Gamma_v|\right) \subset \mathcal{O}(|\Gamma| \cdot l(\mathcal{G})).$$

Beweis. Addiert man für jeden Knoten die Anzahl der Requests durch diesen Knoten, so wird jedes Request r für jeden enthaltenen Knoten einmal gezählt, also $|V_r|$-fach. Entsprechend lässt sich der Wert berechnen, wenn für jedes Request r die Anzahl $|V_r|$ der Knoten addiert wird, d.h. es gilt

$$\sum_{v \in \mathbb{V}} |\Gamma_v| = \sum_{r \in \Gamma} |V_r|.$$

Im Unicast-Fall ist $|V_r| = |V_{KL(r)}| \leq l(\mathcal{G}) + 1 = M \cdot (l(\mathcal{G}) + 1)$, im Multicast-Fall ist $|V_r| = |V_{KB(r)}| \leq M \cdot (l(\mathcal{G}) + 1)$. Daher folgt

$$\sum_{v \in \mathbb{V}} |\Gamma_v| = \sum_{r \in \Gamma} |V_r| \leq \sum_{r \in \Gamma} M \cdot (l(\mathcal{G}) + 1) = M \cdot |\Gamma| \cdot (l(\mathcal{G}) + 1)$$

$$\Longrightarrow \mathcal{O}\left(\sum_{v \in \mathbb{V}} |\Gamma_v|\right) \subset \mathcal{O}(|\Gamma| \cdot l(\mathcal{G})).$$

□

4.5.1 Datenstrukturen und deren Erzeugung

Repräsentation einer partiellen Funktion Zu einer endlichen Menge $X \subset \mathbb{V}$ oder $X \subset \Gamma$ soll oft eine lokale Abbildung $\psi : X \mapsto A$ erstellt werden. Diese muss Lese- und Schreibzugriffe in konstanter Zeit ermöglichen. In der Praxis können solche Abbildungen in Hashtabellen gespeichert werden. Auf Grund von Hash-Konflikten liefern diese aus komplexitätstheoretischer Sicht jedoch nur eine lineare Zugriffszeit. Daher wird im Folgenden eine andere Struktur für partielle Funktionen vorgestellt.

Definition 4.5.3 (Lokale Indizierung einer Menge)**.** *Sei $X \subset \mathbb{N}$ eine endliche Menge. Wir bezeichnen die Elemente von X als globalen Index. Dann definieren wir die Menge $\underline{|X|}$ der lokalen Indizes.*

Dazu wird als Array eine bijektive Abbildung (lokale Indizierung von X) definiert

$$\varphi : \underline{|X|} \mapsto X, \qquad i \mapsto \varphi_i.$$

Zu gegebenem lokalen Index i lässt sich der globale Index φ_i nun in konstanter Zeit ermitteln.

Definition 4.5.4 (Partielle Abbildung)**.** *Eine partielle Abbildung $\psi : X \mapsto A$ lässt sich nun durch eine lokale Indizierung $\varphi : \underline{|X|} \mapsto X$ von X und einer Abbildung $\psi' : \underline{|X|} \mapsto A$ mit $\psi'(i) = \psi(\varphi_i)$ definieren. Wir nennen diese Darstellung ψ' von ψ* lokale Abbildung bezüglich der lokalen Indizierung φ.

Diese Datenstruktur ermöglicht:

- *Für lokale Variablen i Lesen und Schreiben des Funktionswertes $\psi'(i) = \psi(\varphi_i)$ in konstanter Zeit.*
- *Das Lesen und Schreiben aller Funktionswerte für globale Variablen in $\mathcal{O}(|X|)$.*

Repräsentation eines Pfades Ein Pfad P ist in einfachen Graphen eine doppelt verkettete Kantenliste von Knoten. Er enthält

- Die Indizes *Anfangsknoten(P)* und *Endknoten(P)*.
- Die Funktionen *EntferneAnfangsknoten(P)* und *EntferneEndknoten(P)* mit konstanter Laufzeit.
- Die Funktionen *FügeHinzuAnfangsknoten(P,v)* und *FügeHinzuEndknoten(P,v)*, die an den Anfang bzw. das Ende des Pfades in konstanter Laufzeit einen Knoten v hinzufügt.
- Eine Funktion *Invertiere(P)* zur Berechnung von P^{-1} in linearer Laufzeit.

Weiter gibt es eine Methode *Verkette(P,Q)*, die zwei Pfade in konstanter Zeit verkettet.

Repräsentation eines Baums Ein statischer Baum $G = (V, E)$ lässt sich in einer Array-Struktur speichern. Dazu nummeriert man die Knoten, d.h. es ist $V = \underline{|V|}$. Die Nummerierung sei so, dass 0 die Wurzel ist und für jeden Knoten v alle Nachfolger eine größere Nummer als v haben. Implizit erhält man eine Nummerierung der Kanten, indem man $e = 1, \ldots, |V| - 1$ mit der Kante zwischen e und dem Vater von e identifiziert. Somit ist $E = \{1, \ldots, |E| - 1\}$.

Wir bezeichnen nun die Anzahl der Nachfolger eines Knotens v mit

$$\delta'(v) := \begin{cases} \delta(v) & \text{falls } v = 0 \text{ die Baumwurzel ist,} \\ \delta(v) - 1 & \text{sonst.} \end{cases}$$

Zu jedem Knoten v des Baums sind in Arrays folgende Daten gespeichert:

- Eine lokale Indizierung $Nach_v : \{0, \ldots, \delta'(v)\} \mapsto N(v) \cup \{\infty\}$ aller Nachbarknoten von v. Dabei sei $Nach_0(0) = \infty$, da die Wurzel keinen Vater hat und ansonsten $Nach_v(0)$ der Vater von v.
- Wir definieren der Übersicht halber $Vater(v) = Nach_v(0)$ und $Nachfolger(v) = \{Nach_v(i) : i = 1, \ldots, \delta'(v)\}$.
- Implizit erhält man dadurch eine lokale Indizierung $Kante_v$ der zu v inzidenten Kanten gegeben durch

  $$Kante_v : \{0, \ldots, \delta'(v)\} \mapsto E(v) \cup \{\infty\}, \qquad Kante_v(i) = \begin{cases} v & \text{falls } i = 0 \\ Nach_v(i) & \text{sonst} \end{cases}$$

  Insbesondere hat – falls v nicht die Wurzel ist – die Kante von v zum Vater von v den lokalen Index 0.

Dann gibt es eine Funktion *Wurzelpfad(v)*, die in linearer Zeit der Weglänge einen Weg von der Baumwurzel zu v berechnet.

Repräsentation des Netzwerks Das Modell des Netzwerks soll so gespeichert werden, dass die Berechnungen effizient möglich sind. Dazu wird der Graph des Netzwerks in oben beschriebener Baumstruktur gespeichert. Des Weiteren werden neben den Knoten auch die Requests als $\Gamma = \underline{|\Gamma|}$ durchnummeriert.

Für das Netzwerk werden boolesche Variablen *Duplex* und *Cast* über die Art des Netzwerks gespeichert.

Zu jedem Request sind in Arrays folgende Daten gespeichert

- Die Übertragungsdauer $\delta(r)$ zu jedem Request r.
- Das Device $Quelle(r)$ zu jedem Request r.
- Im UC-Netzwerk
  - das Device $Ziel(r)$,
  - die Kommunikationslinie $KL(r)$ als Weg.

- Im MC-Netzwerk
  - eine lokale Indizierung $Ziel_r$ aller Elemente aus $Ziel(r)$,
  - den Kommunikationsbaum $KB(r)$ als Menge von Kommunikationslinien.

Neben diesen Datenstrukturen werden zur Effizienzsteigerung noch redundante Daten gespeichert. Diese sind statisch und können bei der Erstellung des Netzwerks bzw. dem Finden der Kommunikationslinien oder Kommunikationsbäume ohne zusätzlichen Aufwand erzeugt werden. Implementierungstechnische Details zu diesen Strukturen sollen in den dargestellten Algorithmen nicht betrachtet werden.

Zu jedem Knoten v werden die durch v verlaufenden Requests gespeichert. Diese werden nach einkommender und ausgehender Kante sortiert. Genauer:

- Es wird eine lokale Indizierung $\gamma_v : \underline{|\Gamma_v|} \mapsto \Gamma_v$ der Requests durch v definiert.
- In UC-Netzwerken: Zu jedem Request wird bezüglich des lokalen Index r des Requests der lokale Index der Eingangskante v^{r-} und der Ausgangskante v^{r+} gespeichert.
- In MC-Netzwerken: Zu jedem Request wird bezüglich des lokalen Index r des Requests der lokale Index der Eingangskante v^{r-} und die Menge der lokalen Indizes der Ausgangskanten v^{r+} gespeichert.
- Zu jeder zu v adjazenten Kante e (als lokaler Index) wird eine Liste $in_v(e)$ aller durch e in v hinein verlaufenden Requests (d.h. mit $v^{r-} = e$) und eine Liste $out_v(e)$ aller durch e aus v heraus verlaufenden Requests (d.h. in UC-Netzwerken mit $v^{r+} = e$ und in MC-Netzwerken mit $e \in v^{r+}$) gespeichert.

Algorithmus 11 : **Find-KL**
Berechnung der Kommunikationslinie zwischen zwei Devices

Daten : $\mathcal{G}$: Der baumförmige Graph des Netzwerks.
Parameter : d_1: Quelldevice der Kommunikationslinie.
d_2: Zieldevice der Kommunikationslinie.
Rückgabewert : Die Kommunikationslinie $KL(d_1, d_2)$.
Komplexität : $\mathcal{O}(l(\mathcal{G}))$.

```
1  Funktion Find-KL(d1: Device, d2: Device)
2      w := Wurzel(G);
3      P := Wurzelpfad(G,d1);
4      Q := Wurzelpfad(G,d2);
5      z := ∞ ;                    // Im Folgenden jeweils der entfernte Anfangsknoten
6      while Anfangsknoten(P)=Anfangsknoten(Q) do
7          z := Anfangsknoten(P);
8          EntferneAnfangsknoten(P);
9          EntferneAnfangsknoten(Q);
10     P := FügeHinzuAnfangsknoten(P,z);
11     P^-1 := Invertiere(P);
12     return Verkette(P^-1,Q);
13 end
```

Satz 4.5.5. *Der Algorithmus Find-KL (vgl. Algorithmus 11) berechnet die Kommunikationslinie $KL(d_1, d_2)$ zwischen zwei Devices in $\mathcal{O}(l(\mathcal{G}))$.*

Beweis. Der Weg von der Wurzel zu einem Knoten ist in $\mathcal{O}(l(\mathcal{G}))$ zu finden. Die While-Schleife wird maximal $\mathcal{O}(l(\mathcal{G}))$ mal durchlaufen. Daher ist die Komplexität $\mathcal{O}(l(G))$. Zu jeder Zeit des Algorithmus ist zu Beginn der While-Schleife $P = (x_j, \ldots, x_{n-1}, x_n = d_1)$ und $Q = (y_j, \ldots, y_{m-1}, y_m = d_2)$ mit $x_j = y_j$, dann wird $z := x_j$ definiert. Die Schleife terminiert, wenn das erste j erreicht wird, so dass $x_j \neq y_j$ ist. Dann ist $z = x_{j-1} = y_{j-1}$. Zurückgegeben wird $P^{-1}zQ = (d_1, x_{n-1}, \ldots, x_j, z, y_j, \ldots, y_{m-1}, d_2)$, also ein Weg von d_1 nach d_2. □

Das Finden der Kommunikationslinien aller Requests dauert $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}))$. Dabei können die redundanten Daten ohne Mehraufwand in der Datenstruktur gespeichert werden.

Algorithmus 12 : **Find-KB**
Finden der Kommunikationsbäume zwischen einem Device und einer Menge von Devices

Daten : $\mathcal{G}$: Der baumförmige Graph des Netzwerks.
Parameter : d: Quelldevice.
D: Eine Menge von Zieldevices.
Rückgabewert : Der Kommunikationsbaum $KB(d, D)$.
Komplexität : $\mathcal{O}(l(\mathcal{G}))$.

```
1 Funktion Find-KB(d: Device, D: Devicemenge)
2 |  K = ∅ ;                          // Die zu Beginn leere Kommunikationsmenge
3 |  foreach t ∈ D do K := K ∪ Find-KL(d, t);
4 |  return K;
5 end
```

Satz 4.5.6. *Der Algorithmus Find-KB (vgl. Algorithmus 12) berechnet den Kommunikationsbaum $KB(d, D)$ von dem Device d zur Devicemenge D in $\mathcal{O}(l(\mathcal{G}))$.*

Beweis. Offenbar wird die Schleife genau $|D| \leq M$ mal durchlaufen und die Komplexität eines Durchlaufs ist $\mathcal{O}(l(\mathcal{G}))$. Somit ergibt sich insgesamt eine Komplexität von

$$\mathcal{O}(|D| \cdot l(\mathcal{G})) \subset \mathcal{O}(M \cdot l(\mathcal{G})) = \mathcal{O}(l(\mathcal{G})).$$

□

Das Finden der Kommunikationsbäume aller Requests dauert daher $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}))$. Dabei können die redundanten Daten ohne zusätzlichen Aufwand in der Datenstruktur gespeichert werden.

4.5.2 Schedules

Bemerkung 4.5.7 (Repräsentation der Schedules im Computer)**.** Ein globaler Schedule $\alpha : \Gamma \mapsto \mathbb{R}^{\geq 0}$ wird in einem Array von Fließkommazahlen gespeichert.

Für einen Schedule $\alpha : R \mapsto \mathbb{R}^{\geq 0}$ für die Menge $R \subset \Gamma$ wird – falls nicht bereits vorgegeben – eine lokale Indizierung φ von R konstruiert und anschließend die lokale Abbildung bezüglich der lokalen Indizierung φ gespeichert.

Algorithmus 13 : **SchedUnion**
Vereinigung eines lokalen mit einem globalen Schedule

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
Parameter : α: Schedule für R, in einem Array als globaler Schedule gespeichert.
β: Schedule für die Menge S bezüglich der lokalen Indizierung φ von S.
φ: Lokale Indizierung der Menge S.
Rückgabewert : Die synchrone Vereinigung $\alpha \uplus \beta$ für $R \cup S$ als globalen Schedule.
Komplexität : $\mathcal{O}(|S|)$.

```
1 Funktion SchedUnion(α: Schedule, β: Schedule, φ: lokale Indizierung von S)
2 |   foreach j ∈ |S| do α(φ(j)) := β(j);
3 |   return α;
4 end
```

Satz 4.5.8. *Ist $\alpha : R \mapsto \mathbb{R}^{\geq 0}$ ein als Array gespeicherter globaler Schedule und $\beta : S \mapsto \mathbb{R}^{\geq 0}$ ein zu α synchroner Schedule bezüglich der lokalen Indizierung φ, so berechnet der Algorithmus SchedUnion in $\mathcal{O}(|S|)$ die Vereinigung $\alpha \uplus \beta$.*

Beweis. Wegen $\varphi(\underline{S}) = S$ durchläuft $\varphi(j)$ alle Requests aus S und für jedes Request $s \in S$ wird $\alpha(s)$ bezüglich des globalen Index auf den Wert $\beta(s)$ bezüglich des lokalen Index gesetzt. Da die Schleife $|S|$ mal durchlaufen wird, folgt die Laufzeit unmittelbar. □

Algorithmus 14 : **MultiSchedUnion**
Vereinigung synchroner Schedules

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
Parameter : $\mathcal{R}$: Eine Familie $(R_i)_{i=1,\ldots,n}$ von Requestmengen.
α: Ein Schedule für die Menge R, in einem Array als globaler Schedule gespeichert.
Rückgabewert : Der Schedule $\left(\biguplus_{i \in I} \beta_i\right) \uplus \alpha$.
Komplexität : Ist die Komplexität der Erzeugung von β_i in $\mathcal{O}(g(i))$, so ist die Gesamtkomplexität $\mathcal{O}(\sum_{i \in I}(g(i) + |R_i|))$.

```
1 Funktion MultiSchedUnion(α: Schedule, R: Familie von Requestmengen)
2 |   foreach R_i ∈ R do
3 |   |   β_i sei zu α synchroner Schedule für R_i bezüglich lokaler Indizierung φ_i;
4 |   |   α := SchedUnion(α, β_i, φ_i);
5 |   return α;
6 end
```

Satz 4.5.9. *Die Konstruktion des Schedules β_i habe eine Komplexität $\mathcal{O}(g(i))$. Der in Algorithmus 14 dargestellte Algorithmus MultiSchedUnion berechnet in $\mathcal{O}(\sum_{i \in I}(g(i) + |R_i|))$ die synchrone Vereinigung*

$$\alpha \uplus \left(\biguplus_{i \in I} \beta_i \right).$$

Beweis. Sei im i-ten Schritt α_i der Schedule $\alpha \uplus \left(\biguplus_{k=1}^{i-1} \beta_k\right)$. Dann ist β_i synchronisierbar zu α_i und es wird $\alpha_i \uplus \beta_i = \alpha \uplus \left(\biguplus_{k=1}^{i-1} \beta_k\right)$ berechnet.

Für jedes $R_i \in \mathcal{R}$ ist die Komplexität $\mathcal{O}(g(i)+|R_i|)$. Daraus ergibt sich die Gesamtkomplexität $\mathcal{O}(\sum_{i\in I}(g(i)+|R_i|))$. □

Korollar 4.5.10. *(i) Sind α und alle Schedules β_i synchron, so berechnet MultiSchedUnion die synchrone Vereinigung $\alpha \uplus \left(\biguplus_{i\in I} \beta_i\right)$.*

(ii) Ist α ein leerer Schedule und alle β_i synchron, so berechnet MultiSchedUnion die synchrone Vereinigung $\biguplus_{i\in I} \beta_i$.

Beweis. (i) Da α und alle β_i synchron sind, ist nach Lemma 4.2.25 in jedem Schritt b_j synchron zu $\biguplus_{i=1}^{j-1} \beta_i$ und die Aussage folgt aus Satz 4.5.9.

(ii) Dieses folgt unmittelbar aus (i).

□

Ein Spezialfall des Algorithmus MultiSchedUnion ist der Algorithmus GlobalSchedUnion. Dort wird für jeden Knoten $v \in \mathbb{V}$ ein Schedule β_v für Γ_v konstruiert. Dieser Spezialfall ist von besonderer Bedeutung und in Algorithmus 15 dargestellt.

Algorithmus 15 : **GlobalSchedUnion**
Vereinigung synchroner Schedules

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
Rückgabewert : Ein globaler Schedule.
Komplexität : $\mathcal{O}(\sum_{v\in\mathbb{V}} g(v) + |\Gamma| \cdot l(\mathcal{G}))$.

```
1 Funktion GlobalSchedUnion
2     α := leerer globaler Schedule;
3     foreach v ∈ 𝕍 do
4         β_v sei zu α synchroner Schedule für Γ_v bezüglich der lokalen Indizierung γ_v;
5         α := SchedUnion(α, β_v, γ_v);
6     return α;
7 end
```

Satz 4.5.11 (Konstruktion eines globalen Schedules aus synchronen lokalen Schedules)**.** *Die Konstruktion des Schedules β_v habe eine Komplexität $\mathcal{O}(g(v))$. Dann berechnet der in Algorithmus 15 dargestellte Algorithmus GlobalSchedUnion in $\mathcal{O}(\sum_{v\in\mathbb{V}} g(v) + |\Gamma| \cdot l(\mathcal{G}))$ den globalen Schedule $\biguplus_{v\in\mathbb{V}} \beta_v$.*

Beweis. Der Algorithmus ist eine spezielle Version von MultiSchedUnion (siehe Algorithmus 14). Nach Satz 4.5.9 und wegen $\bigcup_{v\in\mathbb{V}} \Gamma_v = \Gamma$ berechnet dieser den globalen Schedule $\biguplus_{v\in\mathbb{V}} \beta_v$.

Mit der Initialisierung des globalen Schedules α in $\mathcal{O}(|\Gamma|)$ ergibt sich nach Satz 4.5.9 und Lemma 4.5.2 eine Laufzeit von

$$\mathsf{GlobalSchedUnion} \in \mathcal{O}(|\Gamma|) + \mathcal{O}\left(\sum_{v \in \mathbb{V}} (g(v) + |\Gamma_v|)\right) = \mathcal{O}\left(\sum_{v \in \mathbb{V}} g(v) + |\Gamma| \cdot l(\mathcal{G})\right).$$

□

4.5.3 Erstellung der Konfliktgraphen

Im Folgenden sollen nun Algorithmen für die Erstellung von Konfliktgraphen vorgestellt werden.

4.5.3.1 Kantenkonfliktgraphen

Algorithmus 16 : *HD-UC*-**Kantenkonfliktgraph**
Erstellung des Kantenkonfliktgraphen für Γ_v

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
Parameter : v: Knoten, so dass der Konfliktgraph für Γ_v zu bestimmen ist.
Rückgabewert : Der Kantenkonfliktgraph als Adjazenzliste.
Komplexität : $\mathcal{O}(|V| + |E|)$ bezüglich des Konfliktgraphen, d.h. $\mathcal{O}(d(v) + |\Gamma_v|)$.

```
1 Funktion HD-UC-Kantenkonfliktgraph(v: Knoten)
2     C := Array [1, ..., |E(v)|] of Liste ;        // Konfliktgraph als Adjazenzliste
3     foreach r ∈ Γ_v do
          // Füge Kante {v^r−, v^r+} in den Konfliktgraphen ein.
4         C[v^r−] := C[v^r−] ∪ {v^r+};
5         C[v^r+] := C[v^r+] ∪ {v^r−};
6     return C;
7 end
```

Die Erstellung der *HD-UC*-, *HD-MC*- und *VD-UC*-Kantenkonfliktgraphen für die Menge Γ_v der Requests durch den Knoten v verläuft direkt nach der Definition. Es wird mit einem kantenleeren Graphen begonnen und für jedes Request werden Kanten hinzugefügt:

- Im *HD-UC*-Kantenkonfliktgraph die Kante mit den Randknoten $E_{KL(r)}(v)$.
- Im *HD-MC*-Kantenkonfliktgraph die Hyperkante mit den Randknoten $\{v^{r-}\} \cup v^{r+}$ (dieses ist eine Hyperkante, da die Menge v^{r+} mehrere Knoten enthalten kann).
- Im *VD-UC*-Kantenkonfliktgraph die Kante zwischen v_{in}^{r-} und v_{out}^{r+}.

Dieses dauert jeweils für ein Request konstante Zeit, insgesamt also $\mathcal{O}(|\Gamma_v|)$. Wegen der Erzeugung der Datenstruktur ergibt sich somit jeweils eine Komplexität von $\mathcal{O}(d(v) + |\Gamma_v|)$. Dieses entspricht einer Komplexität von $\mathcal{O}(|V| + |E|)$ bezüglich des Konfliktgraphen und ist somit optimal.

Algorithmus 17 : HD-MC-**Kantenkonfliktgraph**
Erstellung des Kantenkonfliktgraphen für Γ_v

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
Parameter : v: Knoten, so dass der Konfliktgraph für Γ_v zu bestimmen ist.
Rückgabewert : Der Kantenkonfliktgraph als Adjazenzliste.
Komplexität : $\mathcal{O}(|V| + |E|)$ bezüglich des Konfliktgraphen, d.h. $\mathcal{O}(d(v) + |\Gamma_v|)$.

```
1 Funktion HD-MC-Kantenkonfliktgraph(v: Knoten)
2     C := Array [1,...,|E(v)|] of Liste ;          // Konfliktgraph als Adjazenzliste
3     foreach r ∈ Γ_v do
          // Füge Hyperkante {v^r-} ∪ v^r+ in den Konfliktgraphen ein.
4         e := Hyperkante mit den Knoten {v^r-} ∪ v^r+;
5         C[v^r-] := C[v^r-] ∪ {e};
6         foreach w ∈ v^r+ do C[w] := C[w] ∪ {e};
7     return C;
8 end
```

Algorithmus 18 : VD-UC-**Kantenkonfliktgraph**
Erstellung des Kantenkonfliktgraphen für Γ_v

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
Eine Nummerierung der Knoten $\{e_{in}, e_{out} : e \in E(v)\}$ mit $1, \ldots, 2|E(v)|$.
Parameter : v: Knoten, so dass der Konfliktgraph für Γ_v zu bestimmen ist.
Rückgabewert : Der Kantenkonfliktgraph als Adjazenzliste.
Komplexität : $\mathcal{O}(|V| + |E|)$ bezüglich des Konfliktgraphen, d.h. $\mathcal{O}(d(v) + |\Gamma_v|)$.

```
1 Funktion VD-UC-Kantenkonfliktgraph(v: Knoten)
2     C := Array [1,...,2|E(v)|] of Liste ;          // Konfliktgraph als Adjazenzliste
3     foreach r ∈ Γ_v do
          // Füge Kante {v_in^r-, v_out^r+} in den Konfliktgraphen ein.
4         C[v_in^r-] := C[v_in^r-] ∪ v_out^r+;
5         C[v_out^r+] := C[v_out^r+] ∪ v_in^r-;
6     return C;
7 end
```

4.5.3.2 Knotenkonfliktgraphen

Der Algorithmus zur Erzeugung des Knotenkonfliktgraphen für einen Knoten v fügt im Wesentlichen für je zwei in Konflikt stehende Requests r, s die Kante $\{r, s\}$ in den Konfliktgraphen ein. Das Problem besteht darin, dass eine solche Kante nur einmal in die Adjazenzliste eingefügt werden darf. Während bei einer Adjazenzmatrix das Prüfen, ob eine Kante bereits vorhanden ist, in konstanter Zeit erfolgt, ist dieses in einer Adjazenzliste schwieriger.

Prinzipiell gibt es zwei mögliche Lösungen für dieses Problem:

- Kanten werden mehrfach eingefügt und anschließend doppelte Kanten gelöscht. Die Anzahl der Kanten pro Tupel ist durch die Konstante $M + 1$ beschränkt, da zwei Requests in maximal $M + 1$ zu v inzidenten Kanten einen Konflikt haben können.

  Nach der Erstellung des Konfliktgraphen G gibt es für jeden Knoten r eine Liste $C[r]$ der mit r adjazenten Kanten, repräsentiert durch den verbundenen Nachbarknoten. Dort können jedoch Kanten mehrfach vorkommen. Dann wird ein Array *used* der Länge $|\Gamma_v|$ mit booleschen Variablen angelegt und für jedes Request r die Liste $C[r]$ der mit r in Konflikt stehenden Requests s durchlaufen. Kommt eine Kante $\{r, s\}$ das erste mal, so wird der Nachbarknoten s im Array *used* als *true* markiert. Kommt sie ein weiteres mal, so ist *used*$[s]$ bereits auf *true* gesetzt und die Kante wird entfernt. Nachdem alle Kanten aus $C[r]$ durchlaufen – und doppelte Kanten entfernt – wurden, wird die Liste ein weiteres mal durchlaufen und für jeden Knoten $s \in C[r]$ wird *used*$[s]$ auf $false$ gesetzt.

  Das Array *used* wird in $\mathcal{O}(|\Gamma_v|) = \mathcal{O}(|V_G|)$ erstellt. Da zwei Requests im Knoten v einen Konflikt in maximal $M+1$ Kanten haben, ist $|C[r]| \leq (M+1)d_G(r)$ und für jedes Request $s \in C[r]$ wird konstante Zeit benötigt. Das Zurücksetzen nach dem Durchlauf einer Liste $C[r]$ dauert ebenfalls $\mathcal{O}(d_G(r))$. Daher ergibt sich nach Lemma 2.2.6 eine Gesamtlaufzeit von

  $$\begin{aligned}
  &\mathcal{O}(|\Gamma_v|) + \sum_{v \in V_G} \mathcal{O}(|C[r]|) + \sum_{v \in V_G} \mathcal{O}(d_G(r)) \\
  &= \mathcal{O}(|V_G|) + \mathcal{O}\left(\sum_{v \in V_G} d_G(r)\right) \\
  &= \mathcal{O}(|V_G|) + \mathcal{O}(|E_G|) = \mathcal{O}(|V_G| + |E_G|).
  \end{aligned}$$

  Somit erhöht das Entfernen der doppelten Kanten die Komplexität nicht.

- Für jedes Paar in Konflikt stehender Requests r und s wird sichergestellt, dass die Kante $\{r, s\}$ genau einmal in den Konfliktgraph eingefügt wird.

  Dazu wird jedes mal, wenn die Kante $\{r, s\}$ in den Konfliktgraph eingefügt werden soll, die (beschränkte) Menge K aller Kanten erzeugt, in denen r und s einen Konflikt haben. Die Kante $\{r, s\}$ wird nun genau dann hinzugefügt, wenn sie das Maximum von K ist.

  Da die Mächtigkeit der Menge K beschränkt ist, dauert sowohl die Konstruktion von K als auch das Finden des Maximums nur konstante Zeit. Daher wird die Komplexität des gesamten Algorithmus nicht vergrößert.

Obwohl die erste Methode in der Praxis effizienter ist, sind beide Methoden aus komplexitätstheoretischer Sicht gleichwertig. In dem in Algorithmus 19 dargestellten Algorithmus Knotenkonfliktgraph-Lokal wird die zweite Methode verwendet.

Satz 4.5.12. *Der in Algorithmus 19 dargestellte Algorithmus Knotenkonfliktgraph-Lokal erzeugt einen Knotenkonfliktgraph für Γ_v in $\mathcal{O}(\Theta(v) + |\Gamma_v|)$ bzw. $\mathcal{O}(|E| + |V|)$ bezüglich des Konfliktgraphen.*

Beweis.

- Zur Korrektheit: Jede Funktion *KonfliktMenge(r, s)* liefert für den entsprechenden Fall in konstanter Zeit die Menge der zu v adjazenten Kanten, in denen r und s einen Konflikt haben. Diese Menge ist von beschränkter Größe und lässt sich deshalb in konstanter Zeit berechnen. Daher kann auch in konstanter Zeit das Maximum der Menge (bezüglich beliebiger Ordnung) gefunden werden.

  Die Funktion wird für jede gemeinsame Kante e aufgerufen, d.h. für jede Kante, in denen r und s möglicherweise einen Konflikt haben könnten. Die Kante $\{r, s\}$ wird nun – falls r und s einen Konflikt haben – genau einmal zu den Kanten für r und einmal zu den Kanten für s hinzugefügt.

- Zur Laufzeitkomplexität: Die Schleife wird für jede Kante $e \in E(v)$ genau $\Psi(e)^2$ mal durchlaufen und benötigt in jedem Durchlauf konstante Zeit. Somit ist nach Lemma 4.2.32

$$\text{Knotenkonfliktgraph-Lokal} \in \mathcal{O}\left(\sum_{e \in E(v)} \Psi(e)^2\right) \in \mathcal{O}(|\Theta(v)| + |\Gamma_v|) = \mathcal{O}(|E| + |V|)$$

□

Ein globaler Konfliktgraph G für Γ lässt sich erzeugen, indem alle lokalen Knotenkonfliktgraphen erzeugt werden und anschließend zwei Requests in G einen Konflikt haben, wenn sie in einem lokalen Konfliktgraph einen Konflikt haben. Diese Methode ist in Algorithmus 20 dargestellt.

Satz 4.5.13. *Der in Algorithmus 20 dargestellte Algorithmus Knotenkonfliktgraph-Global erzeugt einen globalen Knotenkonfliktgraphen in $\mathcal{O}(|\Gamma|^2 + \Theta^\Sigma + |\Gamma| \cdot l(\mathcal{G})) \subset \mathcal{O}(|\Gamma|^2 \cdot l(\mathcal{G}))$.*

Beweis. Haben zwei Requests r, s einen Konflikt, so haben sie diesen in einem Knoten v. Dann ist $r \in C_v[s]$ und $s \in C_v[r]$, somit wird $C[r; s]$ und $C[s; r]$ auf *true* gesetzt. Umgekehrt haben zwei Requests r, s in keinem Knoten einen Konflikt, somit wird niemals $C[s; r]$ auf *true* gesetzt.

Algorithmus 19 : **Knotenkonfliktgraph-Lokal**
Erstellung des Knotenkonfliktgraphen für Γ_v

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
Parameter : v: Knoten, so dass der Konfliktgraph für Γ_v zu bestimmen ist.
Rückgabewert : Der Knotenkonfliktgraph als Adjazenzliste.
Komplexität : $\mathcal{O}(|V| + |E|)$ bezüglich des Konfliktgraphen, d.h. $\mathcal{O}(|\Gamma_v| + \Theta(v))$.

```
1  Funktion Knotenkonfliktgraph-Lokal(v: Knoten)
2      C := Array [1,...,|Γ_v|] of Liste ;          // Konfliktgraph als Adjazenzliste
3      for e := 0,...,δ'(v) do
           /* Durchsuche alle Requestpaare (r,s), die die Kante e gemeinsam
           haben. Haben diese in e einen Konflikt und ist e die optimale Kante
           der Menge aller Konfliktkanten, so füge den Konflikt von der Kante r
           aus gesehen in den Konfliktgraphen ein                              */
4          foreach (r,s) ∈ (in_v(e) ∪ out_v(e))^2 mit r ≠ s do
5              if e = max(KonfliktMenge(r,s,v)) then      // Fkt. entsprechend N
6                  C[r] := C[r] ∪ {rs};
7      return C;
8  end

   // Menge der Kanten e mit r ⇜⇝_e s in HD-MC-Netzwerken
9  Funktion KonfliktMenge-hd-mc(r: Request, s: Request, v: Knoten)
10     return ({v^KB(r)−} ∪ v^KB(r)+) ∩ ({v^KB(s)−} ∪ v^KB(s)+);
11 end

   // Menge der Kanten e mit r ⇜⇝_e s in HD-UC-Netzwerken
12 Funktion KonfliktMenge-hd-uc(r: Request, s: Request, v: Knoten)
13     return {v^KL(r)−, v^KL(r)+} ∩ {v^KL(s)−, v^KL(s)+};
14 end

   // Menge der Kanten e mit r ⇜⇝_e s in VD-MC-Netzwerken
15 Funktion KonfliktMenge-vd-mc(r: Request, s: Request, v: Knoten)
16     return ({v^KB(r)−} ∩ {v^KB(s)−}) ∪ (v^KB(r)+ ∩ v^KB(s)+);
17 end

   // Menge der Kanten e mit r ⇜⇝_e s in VD-UC-Netzwerken
18 Funktion KonfliktMenge-vd-uc(r: Request, s: Request, v: Knoten)
19     return ({v^KL(r)−} ∩ {v^KL(s)−}) ∪ ({v^KL(r)+} ∩ {v^KL(s)+});
20 end
```

Algorithmus 20 : **Knotenkonfliktgraph-Global**
Erstellung des globalen Knotenkonfliktgraphen für Γ

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
Rückgabewert : Der Knotenkonfliktgraph als Adjazenzmatrix.
Komplexität : $\mathcal{O}(|\Gamma|^2 + \Theta^\Sigma) \subset \mathcal{O}(|\Gamma|^2 \cdot l(\mathcal{G}))$.

```
1 Funktion Knotenkonfliktgraph-Global
2     C := Array [1,...,|Γ|, 1,...,|Γ|] of Boolean ;      // Konfliktgraph als Matrix
3     foreach v ∈ 𝕍 do
4         C_v = (V_v, E_v) := Knotenkonfliktgraph-Lokal(v);
5         foreach r ∈ V_v do                  // Füge jeden Konflikt aus C_v in C ein
6             foreach s ∈ C_v[r] do C[r; s] := true
7     return C;
8 end
```

Für die Laufzeit ergibt sich wegen Knotenkonfliktgraph-Lokal $\in \mathcal{O}(\Theta(v) + |\Gamma_v|)$:

$$\begin{aligned}
\text{Knotenkonfliktgraph-Global} &\in \mathcal{O}\left(|\Gamma|^2 + \sum_{v\in\mathbb{V}}(\Theta(v) + |\Gamma_v| + \sum_{r\in\Gamma_v} d_{C_v}(r))\right) \\
&= \mathcal{O}\left(|\Gamma|^2 + \sum_{v\in\mathbb{V}}(\Theta(v) + |\Gamma_v| + 2|E_{C_v}|)\right) \\
&= \mathcal{O}\left(|\Gamma|^2 + \sum_{v\in\mathbb{V}}(\Theta(v) + |\Gamma_v| + 2\Theta(v))\right) \\
&\subset \mathcal{O}(|\Gamma|^2 + \Theta^\Sigma + |\Gamma| \cdot l(\mathcal{G})) \\
&\subset \mathcal{O}(|\Gamma|^2 \cdot l(\mathcal{G}))
\end{aligned}$$

□

Es gibt einen weiteren, direkteren Ansatz: Zwei Requests haben genau dann einen Konflikt, wenn sie diesen in einer Kante e haben. Nun werden alle Kanten durchlaufen und für jede Kante alle Konflikte in dieser Kante in den Konfliktgraph eingefügt. Da Kanten in der Datenstruktur nicht direkt gespeichert sind, wird für jeden Knoten v – außer der Wurzel – die Kante zum Vaterknoten betrachtet. Dieser in Algorithmus 21 dargestellte Algorithmus Knotenkonfliktgraph-Global2 bringt aus komplexitätstheoretischer Sicht leichte Vorteile und ist in der Praxis effizienter.

Satz 4.5.14. *Der in Algorithmus 21 dargestellte Algorithmus Knotenkonfliktgraph-Global2 erzeugt einen globalen Knotenkonfliktgraphen in* $\mathcal{O}(|\Gamma|^2 + \Theta^\Sigma + |\Gamma| \cdot l(\mathcal{G}))$.

Beweis. Haben zwei Requests r, s einen Konflikt, so haben sie diesen in einer Kante e. Sei e die Kante, die v mit seinem Vater verbindet. Dann enthalten r und s die Kante genau dann, wenn $(r, s) \in (in_v(0) \cup out_v(0))^2$ und genau dann in dieselbe Richtung, wenn $(r, s) \in in_v(0)^2 \cup out_v(0)^2$ gilt.

Algorithmus 21 : **Knotenkonfliktgraph-Global2**
Erstellung des globalen Knotenkonfliktgraphen für Γ

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
Rückgabewert : Der Knotenkonfliktgraph als Adjazenzmatrix.
Komplexität : $\mathcal{O}(|\Gamma|^2 + \Theta^\Sigma + |\Gamma| \cdot l(\mathcal{G}))$.

```
1 Funktion Knotenkonfliktgraph-Global2
2     C := Array [1,...,|Γ|, 1,...,|Γ|] of Boolean ;      // Konfliktgraph als Matrix
3     for v := 1,...,|𝕍| − 1 do                          // Alle Knoten außer der Wurzel
4         if Cast = HD then
5             foreach (r, s) ∈ (in_v(0) ∪ out_v(0))² do if r ≠ s then C[r; s] := true;
6         else
7             foreach (r, s) ∈ (in_v(0)² ∪ out_v(0)²) do if r ≠ s then C[r; s] := true;
8     return C;
9 end
```

Da unterschiedliche Requests genau dann einen Konflikt in e haben, wenn sie

- in HD-Netzwerken beide die Kante e enthalten,
- in VD-Netzwerken beide die Kante e in dieselbe Richtung enthalten,

wird in einem Durchlauf für die Kante e genau dann $C[r; s] = true$ gesetzt, wenn r und s in e einen Konflikt haben.

Zur Laufzeitkomplexität: Die ForEach-Schleifen werden $\mathcal{O}(\Psi(e)^2)$ mal durchlaufen. Mit Lemma 4.2.30 und Lemma 4.2.31 ergibt sich die Abschätzung

$$
\begin{aligned}
&|\Gamma|^2 + \sum_{e\in\mathbb{E}} \Psi(e)^2 \\
&= |\Gamma|^2 + \sum_{e\in\mathbb{E}} \Psi(e)(\Psi(e)-1) + \sum_{e\in\mathbb{E}} \Psi(e) && \text{|Lemma 4.2.30(i)} \\
&\leq |\Gamma|^2 + \sum_{e\in\mathbb{E}} (4\Theta(e) + 2\Psi(e)) + \sum_{e\in\mathbb{E}} \Psi(e) \\
&= |\Gamma|^2 + 4\sum_{e\in\mathbb{E}} \Theta(e) + 3\sum_{e\in\mathbb{E}} \Psi(e) && \text{|Lemma 4.2.30(v), Lemma 4.2.31(v)} \\
&\leq |\Gamma|^2 + 4(\Theta^\Sigma - \Theta) + 2M \cdot |\Gamma| \cdot l(\mathcal{G}) + M \cdot |\Gamma| \cdot l(\mathcal{G}) \\
&\leq |\Gamma|^2 + 4\Theta^\Sigma + 3M \cdot |\Gamma| \cdot l(\mathcal{G})
\end{aligned}
$$

und somit

$$
\text{Knotenkonfliktgraph-Global2} \in \mathcal{O}\left(|\Gamma|^2 + \sum_{e\in\mathbb{E}} \Psi(e)^2\right) \subset \mathcal{O}(|\Gamma|^2 + \Theta^\Sigma + |\Gamma| \cdot l(\mathcal{G})).
$$

□

Kapitel 5

Beliebige Requestlängen

5.1 First-Fit-Algorithmus

Der Algorithmus First-Fit ist ein Spezialfall von dem in Algorithmus 14 dargestellten Algorithmus MultiSchedUnion, bei dem jede Requestmenge nur aus einem Request besteht. Er ist für Bin-Packing[6] ein etabliertes Verfahren und wurde in [JDU+74] analysiert.

Algorithmus 22 : **First-Fit**
Erstellung eines globalen Schedules

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
Parameter : C: Knotenkonfliktgraph für R als Adjazenzmatrix bezüglich lokaler Indizierung γ.
Rückgabewert : Ein Schedule für R bezüglich lokaler Indizierung γ.
Komplexität : $\mathcal{O}(|R|^2)$.

```
1  Funktion First-Fit(C: Knotenkonfliktgraph
2      α := leerer Schedule für R als Array;
3      L := ∅ als nach Anfangszeit sortierte Liste;
4      foreach r ∈ R do
           // Finde den ersten möglichen Zeitpunkt für r, so dass r mit keinem
           zeitgleich ausgeführten Request in Konflikt steht.
5          t := 0;
6          Setze L auf den Listenanfang;
7          while L hat nächstes Element do
8              s := nächstes Element von L;
9              if α(s) ≥ t + δ(r) then break ;          // Setzen von α(r) := t möglich
10             if r ⇜⇝ s then     // r kann erst nach Ende von s geschedulet werden
11                 t := max(t, α(s) + δ(s));
12         α(r) := t;
13         Sortiere r in L ein.;
14     return α;
15 end
```

[6] Beim Bin-Packing-Problem geht es darum, eine Menge von Stücken gegebener Länge in eine möglichst kleine Anzahl von Containern gleicher Länge einzufügen.

Satz 5.1.1. *Der in Algorithmus 22 dargestellte Algorithmus* First-Fit *berechnet in einer Komplexität von $\mathcal{O}(|R|^2)$ einen Schedule für R.*

Obwohl der Algorithmus ein Spezialfall von Algorithmus 14 ist, wird hier der direkte Beweis skizziert:

Beweis. Jedes Request wird so geschedulet, dass es mit keinem bisher gescheduleten in Konflikt stehenden Request zeitgleich ausgeführt wird.

- *Während der While-Schleife in Zeile 22 ist t der erste Zeitpunkt, für den auf Basis des bisherigen Wissens r geschedulet werden kann:* Dieses ist offenbar vor dem ersten Schleifendurchlauf erfüllt. Nun wird in Zeile 22 ein Request s gewählt. Steht dieses nicht mit r in Konflikt, so muss t nicht verändert werden. Steht dieses mit r in Konflikt, so kann r frühstens wieder geschedulet werden, wenn s beendet ist. Dieses wird in Zeile 22 sichergestellt. Somit ist
- *Nach Beendigung der Schleife kann α zum Zeitpunkt t geschedulet werden:* Die While-Schleife kann entweder in Zeile 22 abgebrochen werden oder terminieren, wenn das Ende von L erreicht ist.

  Wird in Zeile 22 ein Request s mit $\alpha(s) \geq t + \delta(r)$ erreicht, so kann r zum Zeitpunkt t geschedulet werden. Denn von den bisherigen Elementen in L gab es kein zu r in Konflikt stehendes Request, dass innerhalb des Intervalls $[t, t + \delta(r))$ aktiv war. Alle in L noch kommenden Requests hingegen sind nach $\alpha(s) \geq t + \delta(r)$ geschedulet, also ebenfalls nicht während diesem Intervall aktiv.

  Wird die Schleife beendet, weil L keine weiteren Elemente mehr enthält, so kann r zum Zeitpunkt t geschedulet werden. Denn kein Request, welches zu einer Zeit nach t aktiv ist, hat ein Konflikt mit r.

Somit erzeugt der Algorithmus einen Schedule.

Zur Laufzeit: Für jedes Request $r \in R$ wird die Liste L durchsucht und anschließend r in die Liste eingeordnet. Da maximal $|R|$ Requests in L gespeichert sind, hat jede dieser Operationen eine Laufzeit von $\mathcal{O}(|R|)$. Somit ergibt sich eine Gesamtkomplexität von

$$\text{First-Fit} \in \mathcal{O}\left(\sum_{r \in R} |R|\right) = \mathcal{O}(|R|^2).$$

□

Bemerkung 5.1.2 (Algorithmus Decreasing-First-Fit)**.** Eine mögliche gute Reihenfolge wäre, die Requests absteigend der Länge nach zu sortieren. Dieses benötigt einen Zeitaufwand von $\mathcal{O}(|R| \log |R|) \subset \mathcal{O}(|R|^2)$, erhöht die Komplexität also nicht. Dieser Algorithmus sei im Folgenden mit Decreasing-First-Fit bezeichnet.

Bemerkung 5.1.3. Eine weitere mögliche Reihenfolge für Nicht-VD-MC-Netzwerke wäre, Knoten für Knoten jeweils die Requests durch diesen Knoten zu schedulen. Der Vorteil dabei ist, dass kein globaler Konfliktgraph benötigt wird, sondern nur für jeden Knoten der lokalen Konfliktgraphen. Dadurch reduziert sich die Komplexität. Jedoch ist – verglichen mit Decreasing-First-Fit – ein schlechteres Ergebnis zu erwarten. Dieser Algorithmus sei mit First-Fit-Nodes bezeichnet.

5.2 List-Scheduling

Ein zu First-Fit ähnlicher Ansatz ist der Algorithmus List-Scheduling. Dieser wurde zuerst 1969 von Graham in [Gra69] vorgestellt.

Algorithmus 23 : **List-Scheduling**
Erstellung eines Schedules

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
Adjazenzliste und Adjazenzmatrix des Knotenkonfliktgraphen C für R bezüglich lokaler Indizierung γ.
Rückgabewert : Ein Schedule für R bezüglich lokaler Indizierung γ.
Komplexität : $\mathcal{O}(|R|^2)$.

```
1  Funktion List-Scheduling
2      α := leerer Schedule für R;
3      LR := Liste der Requests aus R;
4      LT := ∅ als sortierte Liste ganzer Zahlen;
5      foreach r ∈ LR do ft(r) := 0;
6      LT = LT ∪ {0};
7      while LR ≠ ∅ do                  // Es gibt noch nicht geschedulete Requests
8          t := kleinstes Element aus LT ;                  // Nächster Ereignispunkt
9          LT := LT \ {t};
10         foreach r ∈ LR do                          // Suche schedulebare Requests
11             if t ≥ ft(r) then                     // Schedulen von α(r) = t möglich
12                 α(r) := t;
13                 LT := LT ∪ {t + δ(r)} ;            // Füge neuen Ereignispunkt ein
                   /* Alle zu r in Konflikt stehenden Requests können erst nach
                   Beendigung von r geschedulet werden.                          */
14                 foreach s ∈ N_C(r) do ft(s) := max(ft(s), t + δ(r));
15                 LR := LR \ {r};
16     return α;
17 end
```

Satz 5.2.1. *Der in Algorithmus 23 dargestellte Algorithmus List-Scheduling berechnet in einer Komplexität von $\mathcal{O}(|R|^2)$ einen Schedule für R.*

Auch dieser Algorithmus ist ein Spezialfall von Algorithmus 14. Hier soll jedoch eine Skizze des direkten Beweises gegeben werden:

Beweis. Während des gesamten Algorithmus enthält Menge LT die Zeitpunkte, in denen ein derzeit geschedultes Request beendet wird. Dieses sind die Ereignispunkte, an denen neue Requests geschedulet werden können. Daher genügt es, von einem Ereignispunkt zum nächsten zu springen und keine Betrachtungen für die Zeiten zwischen zwei Ereignispunkten zu machen.

- *Für jedes Request s ist $ft(s)$ der frühste Zeitpunkt, zu dem s nach derzeitigem Wissen geschedulet werden kann:* Zu Beginn ist noch kein Request geschedulet, daher ist für jedes Request $ft(s) = 0$.

  Es ist nur dann notwendig, den Wert $ft(s)$ zu verändern, wenn ein mit s in Konflikt stehendes Request r geschedulet wird. Ist dieses in Zeile 23 der Fall, so wird in der Zeile 23 sichergestellt, dass $ft(s) \geq \alpha(r) + \delta(r)$ ist, dass also s erst wieder nach der Beendigung von r geschedulet werden kann.

- *Jedes Request wird zum frühstmöglichen Zeitpunkt geschedulet:* Wurde in Zeile 23 der nächste Ereignispunkt gewählt, werden nun in der in Zeile 23 beginnenden ForEach-Schleife der Reihe nach alle Requests geschedulet, die zu diesem Zeitpunkt möglich sind. Anschließend wird in Zeile 23 der neue Ereignispunkt hinzugefügt.

Auf diese Weise sind nach Terminierung des Algorithmus alle Requests geschedulet und keine zwei in Konflikt stehenden Requests werden zur selben Zeit ausgeführt.

Zur Laufzeit: Es gibt neben dem Zeitpunkt 0 für jedes Request maximal einen Ereignispunkt, daher wird die While-Schleife maximal $|R|+1$ mal durchlaufen. Da LR maximal $|R|$ Elemente enthält, wird die ForEach-Schleife in Zeile 23 maximal $|R|$ mal durchlaufen. Ohne Betrachtung der Komplexität des Inneren des If-Blocks ab Zeile 23 ergibt sich somit eine Komplexität von $\mathcal{O}(|R|^2)$.

Das Innere des If-Blocks ab Zeile 23 wird für jedes Request r genau einmal ausgeführt und hat daher insgesamt eine Komplexität von

$$\mathcal{O}\left(\sum_{r \in R} d_C(r)\right) = \mathcal{O}(2|E_C|) = \mathcal{O}(\Theta(R)) \subset \mathcal{O}(|R|^2).$$

Somit ergibt sich eine Gesamtkomplexität von $\mathcal{O}(|R|^2 + |R|^2) = \mathcal{O}(|R|^2)$. □

Der Algorithmus List-Scheduling hat gegenüber First-Fit den Vorteil, dass sich gute theoretische Abschätzungen treffen lassen.

Lemma 5.2.2. *Sei α ein mit List-Scheduling erzeugter Schedule für R und $r \in R$ ein Request. Dann war zum Zeitpunkt $t < \alpha(r)$ eine Kante $e \in E_r$ belegt – in VD-Netzwerken sogar in dieselbe Richtung wie in r. Formaler:*

$$\forall t < \alpha(r) \quad \exists s \in R: \qquad t \in \widehat{\alpha}(s) \wedge s \leftrightsquigarrow r.$$

Somit gilt für jeden Zeitpunkt $t < \alpha(r)$.

- *In HD-Netzwerken: Es gibt ein Request s mit $t \in \widehat{\alpha}(s)$ und $E_r \cap E_s \neq \emptyset$.*
- *In VD-Netzwerken: Es gibt ein Request s mit $t \in \widehat{\alpha}(s)$ und einer Kante $e \in E_r \cap E_s$ mit $e^{r-} = e^{s-}$.*

Beweis. Angenommen, es gäbe einen Zeitpunkt $t' < \alpha(r)$, zu dem r keinen Konflikt mit einem aktiven Request hatte. Sei $t_0 < t'$ die maximale Zeit kleiner als t', zu der ein aktives Request beendet wurde, d.h.

$$t_0 := \max\left\{\alpha(s) + \delta(s) : s \in R\right\} \cap [0, t')\big).$$

Dann ist die Menge der zum Zeitpunkt t_0 aktiven Requests gleich der Menge der zum Zeitpunkt t aktiven Requests. Insbesondere hat r keinen Konflikt mit einem zum Zeitpunkt t_0 aktiven Request. Daher würde beim Durchlauf der ForEach-Schleife $t_0 \geq ft(r)$ sein, also $\alpha(r) = t_0$ definiert werden. Widerspruch. □

Definition 5.2.3 (Stern). *Ein Baum $G = (V, E)$ heißt* Stern, *wenn er $|V| - 1$ Blätter hat.*

Coffman zeigte in [EGCGJL85], dass der Algorithmus List-Scheduling mit absteigender Requestlänge für sternförmige *HD-UC*-Netzwerke eine Approximationsgüte von 2 hat. Hier wird dieser Satz auf beliebige Netzwerke verallgemeinert.

Satz 5.2.4 (List-Scheduling hat die Approximationsgüte 2 für Sterne). *Sei $G = (V, E)$ ein Baum und R eine Menge von Requests zwischen zwei Blättern. Dann erzeugt der Algorithmus List-Scheduling einen Schedule mit maximal doppelter Länge des optimalen Schedules, d.h. für das Scheduling-Problem in Sternen hat List-Scheduling eine Approximationsgüte von 2.*

Der hier dargestellte Beweis ist eine Verallgemeinerung des in [Erl99, S. 170] dargestellten Beweises für Coffmans Satz.

Beweis. Sei R die Menge von Requests, γ ein optimaler Schedule für R und α der von List-Scheduling erzeugte Schedule für R. Sei $r \in R$ eines der Requests, das zuletzt beendet ist, d.h. ein Request mit $\alpha(r) + \delta(r) = |\alpha|$. Da r ein Request zwischen zwei Blättern des Sterns ist, durchläuft es genau zwei Kanten. Sei $E_r = \{e_1, e_2\}$.

- *Fall Duplex = HD:* Da die Lastdauer von e_1 und e_2 bezüglich R gerade $\delta(r)$ größer ist als bezüglich $R \setminus \{r\}$, gilt

  $$\Phi_{R\setminus\{r\}}(e_i) + \delta(r) = \Phi_R(e_i), \qquad i \in \{1, 2\}.$$

  Nach Lemma 5.2.2 war zu jedem Zeitpunkt $t < \alpha(r)$ mindestens eine der Kanten e_1 oder e_2 mit einem Request aus $R \setminus \{r\}$ belegt. Somit folgt nach Lemma 4.3.3

  $$\begin{aligned} &\alpha(r) \leq \Phi_{R\setminus\{r\}}(e_1) + \Phi_{R\setminus\{r\}}(e_2) = \Phi_R(e_1) + \Phi_R(e_2) - 2\delta(r) \leq 2\Phi_R - 2\delta(r) \\ \Longrightarrow\ &|\alpha| = \alpha(r) + \delta(r) < \alpha(r) + 2\delta(r) \leq 2\Phi_R \leq 2|\gamma| \end{aligned}$$

  und somit die Behauptung.

- *Fall Duplex = VD:* Wir definieren die Lastdauer in Richtung r durch

  $$\Phi'_R(e) = \sum_{s \in R \cap \Gamma_e, e^{s-} = e^{r-}} \delta(s), \qquad e \in \{e_1, e_2\}.$$

  Dann ist offenbar $\Phi'_R(e) \leq \overrightarrow{\Phi}_R(e)$. Analog zum *HD*-Fall die Lastdauer von e_1 und e_2 in Richtung r bezüglich R gerade $\delta(r)$ größer ist als bezüglich $R \setminus \{r\}$. Daher gilt

  $$\Phi'_{R\setminus\{r\}}(e_i) + \delta(r) = \Phi'_R(e_i), \qquad i \in \{1, 2\}.$$

Nach Lemma 5.2.2 war zu jedem Zeitpunkt $t < \alpha(r)$ mindestens eine der Kanten e_1 oder e_2 mit einem Request aus $R \setminus \{r\}$ in dieselbe Richtung wie in r belegt. Somit gilt nach Lemma 4.3.3

$$\begin{aligned}\alpha(r) &\leq \Phi'_{R\setminus\{r\}}(e_1) + \Phi'_{R\setminus\{r\}}(e_2) = \Phi'_R(e_1) + \Phi'_R(e_2) - 2\delta(r) \\ &\leq \overrightarrow{\Phi_R}(e_1) + \overrightarrow{\Phi_R}(e_2) - 2\delta(r) \leq 2\overrightarrow{\Phi_R} - 2\delta(r)\end{aligned}$$

und die Behauptung wegen

$$|\alpha| = \alpha(r) + \delta(r) < \alpha(r) + 2\delta(r) \leq 2\overrightarrow{\Phi_R} \leq 2|\gamma|.$$

□

5.3 List-Scheduling mit Levels

Auf dem Algorithmus List-Scheduling aufbauend entwickelte Erlebach in [Erl99, S. 193ff] einen polynomiellen Algorithmus List-Scheduling-Levels mit Approximationsgüte $\kappa = 5 \log |V|$. Dieser ist jedoch für die leicht veränderte Problemstellung von *HD-UC*-Netzwerken mit variablen Bandbreiten.

Dieser Algorithmus List-Scheduling-Levels lässt sich auf beliebige hier betrachtete Netzwerke verallgemeinern und hat in allen Fällen eine Approximationsgüte von $\kappa = 2\lfloor \log_2 |\mathbb{V}| \rfloor \leq 2 \log_2 |\mathbb{V}| \in \mathcal{O}(\log |\mathbb{V}|)$. Die selbst geführten Beweise sind an [Erl99, S. 193ff] angelehnt.

Definition 5.3.1 (Trennlevel)**.** *Sei $G = (V, E)$ ein Baum. Dann lässt sich rekursiv der* Trennlevel *definieren.*

- *Wähle einen Zerlegungsknoten $v \in V$, so dass jede der Zusammenhangskomponenten $G^1, \ldots, G^n$ aus $G[V \setminus \{v\}]$ maximal $\frac{|V|}{2}$ Knoten enthält. Dann bekommt v den Trennlevel $t(v) := 0$ und die Zusammenhangskomponenten haben die Ebene 1.*
- *Wähle in jeder Zusammenhangskomponente $G^i = (V^i, E^i)$ einen Zerlegungsknoten v^i, so dass die Zusammenhangskomponenten aus $G^i[V^i \setminus \{v^i\}]$ maximal $\frac{|V^i|}{2}$ Knoten haben. Alle diese Knoten v^i haben den Trennlevel $t(v^i) = 1$, die Zusammenhangskomponente die Ebene 2.*
- *Fahre rekursiv mit jeder dieser Zusammenhangskomponenten fort, bis jeder Knoten einen Trennlevel hat, d.h. bis jede Zusammenhangskomponente nur noch aus einem Knoten besteht.*

Eine so definierte Abbildung t heißt Trennlevelfunktion.

Man beachte, dass die Trennlevelfunktion nicht eindeutig ist.

Die Zusammenhangskomponenten der Ebene k seien $H_{k,1}, \ldots, H_{k,n_k}$. Dann gibt es in jeder Zusammenhangskomponente $H_{k,i}$ der Ebene k keine Knoten mit Trennlevel kleiner als k und genau einen Knoten – den Zerlegungsknoten $v_{k,i}$ – mit Trennlevel gleich k.

Lemma 5.3.2. *Für einen Baum $G = (V, E)$ und eine Trennlevelfunktion t gilt: Für jeden Knoten v ist $0 \leq t(v) \leq \lfloor \log_2 |V| \rfloor$.*

Beweis. Sei $v \in V$ ein Knoten mit $t(v) = k$. Die Aussage $t(v) \geq 0$ ist offensichtlich. Sei $G = G_0, G_1, \ldots, G_k$ mit $G_i = (V_i, E_i)$ die Folge der Zusammenhangskomponenten, in denen v bei der Konstruktion der Trennlevel liegt. Dann ist nach Definition $|V_{i+1}| \leq \frac{|V_i|}{2}$ und somit

$$\begin{aligned} 1 = |V_k| &\leq \frac{|V_{k-1}|}{2} \leq \frac{|V_{k-2}|}{4} \leq \cdots \leq \frac{|V_0|}{2^k} \\ \Longrightarrow 2^k &\leq |V_0| = |V| \\ \Longrightarrow t(v) = k &\leq \log_2 |V|. \end{aligned}$$

Wegen $t(v) \in \mathbb{N}_0$ folgt $t(v) \leq \lfloor \log_2 |V| \rfloor$. □

Korollar 5.3.3. *Die Zusammenhangskomponenten $H_{k,1}, \ldots, H_{k,n_k}$ bestehen für den Wert $k = \lfloor \log_2 |V| \rfloor$ nur aus einem Knoten.*

Beweis. Die Komponente $H_{k,i}$ enthält genau einen Zerlegungsknoten $v_{k,i}$ mit Trennlevel k. Alle anderen Knoten in $H_{k,i}$ haben einen Trennlevel größer als k. Solche Knoten existieren nach Lemma 5.3.2 jedoch nicht. □

Im Folgenden betrachten wir ausschließlich den Baum $\mathcal{G} = (\mathbb{V}, \mathbb{E})$ und nehmen eine Trennlevelfunktion t als fest gewählt an. Dann lässt sich der Level eines Requests definieren.

Definition 5.3.4 (Level eines Requests)**.** *Sei r ein Request. Dann ist der Level $l(r)$ der minimale Trennlevel eines Knotens von r, formaler*

$$l(r) := \min_{v \in V_r} t(v).$$

Lemma 5.3.5. *Sei r ein Request. Dann gibt es einen eindeutig bestimmten Knoten $v \in V_r$ mit $t(v) = l(r)$. Dieser Knoten wird als Wurzelknoten $w(r)$ bezeichnet.*

Beweis. Die Existenz ist durch die Definition von $l(r)$ klar.

Zur Eindeutigkeit. Angenommen, v und v' seien verschiedene Knoten in V_r mit minimalem Trennlevel $t(v) = t(v') = k$. Sei $G = G_0, G_1, \ldots, G_k$ bzw. $G = G'_0, G'_1, \ldots, G'_k$ die Folge der Zusammenhangskomponenten, in denen v bzw. v' bei der Trennlevelkonstruktion liegt. Dann ist $G_k \neq G'_k$, da sonst v und v' der eindeutige Zerlegungsknoten in $G_k = G'_k$ wären. Sei $j < k$ der größte Index mit $G_j = G'_j$ und w der Zerlegungsknoten in G_j. Dann ist $t(w) = j$. Da v und v' in $G_j[V_j \setminus \{w\}]$ in verschiedenen Zusammenhangskomponenten liegen, muss der eindeutige Weg P von v nach v' den Knoten w enthalten. Da $\mathcal{G}[V_r]$ zusammenhängend ist, enthält $\mathcal{G}[V_r]$ den Weg P und es folgt $w \in V_P \subset V_r$. Dieses ist wegen $t(w) = j < k = t(v)$ ein Widerspruch zur Wahl von v. Daher kann es keine zwei solchen Knoten v und v' geben. □

Definition 5.3.6. *Zu $0 \leq k \leq \log_2 \lfloor |\mathbb{V}| \rfloor$ ist die Menge R_k die Menge der Requests mit Level k. Formaler ist*

$$R_k := \{r \in \Gamma : l(r) = k\}.$$

Lemma 5.3.7. *Seien $r, s \in R^k$. Dann gilt*

(i) Das Request r ist komplett in einer Zusammenhangskomponente $H_{k,i}$ der Ebene k enthalten, d.h. es gibt ein $H_{k,i} = (V_{k,i}, E_{k,i})$ mit $V_r \subset V_{k,i}$.

(ii) *Ist r in $H_{k,i}$ enthalten, so enthält r den Zerlegungsknoten von $H_{k,i}$, d.h. es ist $v_{k,i} \in V_r$.*

(iii) *Ist r in $H_{k,i}$ und s in $H_{k,j}$ mit $i \neq j$, so sind r und s konfliktfrei.*

Beweis. Zu (i): Das Request r enthält nur Knoten mit Trennlevel $t \geq k$. Würde r Knoten aus $H_{k,i}$ und $H_{k,j}$ enthalten, so gäbe es einen Weg zwischen diesen mit Knoten mit Trennlevel größer oder gleich k, die Zusammenhangskomponenten sind also identisch und somit ist $H_{k,i} = H_{k,j}$.

Zu (ii): Für den Wurzelknoten $w(r)$ von r gilt $w(r) \in H_{k,i}$ und $t(w(r)) = l(r) = k$. Somit ist $w(r)$ der eindeutige Knoten $v_{k,i}$ mit Trennlevel k in $H_{k,i}$, also ist $v_{k,i} = w(r) \in V_r$.

Zu (iii): Die Knotenmengen $V_{k,i}$ und $V_{k,j}$ sind disjunkt. Wegen $V_r \subset V_{k,i}$ und $V_s \subset V_{k,j}$ folgt

$$V_r \cap V_s \subset V_{k,i} \cap V_{k,j} = \emptyset.$$

Also haben r und s keinen Knoten gemeinsam und sind somit konfliktfrei.

□

Nun lässt sich die Menge R^k zerlegen.

Definition 5.3.8. *Für $i = 1, \ldots, n_k$ definiere*

$$R^{k,i} = \{r \in R^k : r \textit{ ist komplett in } H_{k,i} \textit{ enthalten}\}.$$

Dann ist $(R^{k,i})_{i=1,\ldots,n_k}$ eine disjunkte Zerlegung von R^k.

Lemma 5.3.9. *Requests aus $R^{\lfloor \log_2 |\mathbb{V}| \rfloor}$ sind zu allen anderen Requests konfliktfrei.*

Beweis. Sei $l = \lfloor \log_2 |\mathbb{V}| \rfloor$ und $r \in R^l$, dann ist r nach Lemma 5.3.7 komplett in einer Komponente $H_{l,i}$ enthalten. Diese besteht nach Korollar 5.3.3 nur aus einem Knoten. Somit ist $E_r \subset E_{l,i} = \emptyset$. Da zwei Requests nur dann einen Konflikt haben können, wenn sie eine Kante gemeinsam haben, ist r zu jedem Request konfliktfrei. □

Lemma 5.3.10. *Für $r, s \in R^k$ gilt*

$$r \leftrightsquigarrow s \iff \exists i : r, s \in R^{k,i} \wedge r \leftrightsquigarrow_{v_{k,i}} s.$$

Beweis. Die Rückrichtung ist trivial, da $r \leftrightsquigarrow_{v_{k,i}} s \implies r \leftrightsquigarrow s$. Sei nun $r \leftrightsquigarrow s$.

Wäre $r \in R^{k,i}$ und $s \in R^{k,j}$ mit $i \neq j$, so wären sie nach Lemma 5.3.7(iii) konfliktfrei. Widerspruch. Somit gibt es ein i mit $r, s \in R^{k,i}$. Nach Lemma 5.3.7(ii) ist dann $v_{k,i} \in V_r \cap V_s$, also $r, s \in \Gamma_{v_{k,i}}$. Daher haben r und s nach Satz 4.2.24 einen Konflikt in $v_{k,i}$. □

Daraus ergibt sich das folgende Korollar:

Korollar 5.3.11. *Die Erstellung eines Schedules für $R^{k,i}$ ist äquivalent zur Erstellung eines Schedules für einen Stern.*

Beweis. Lemma 5.3.10 bedeutet, dass zwei Requests aus $R^{k,i}$ genau dann einen Konflikt haben, wenn sie einen Konflikt in $v_{k,i}$ haben. Es genügt somit, den Graph bestehend aus dem Knoten $v_{k,i}$ und seinen Nachbarknoten zu betrachten. Formaler: Definiere $H = (V_H, E_H)$ durch $v = \{v_{k,i}\}$ und

$$V_H = \{v\} \cup N(v), \qquad E_H = \{vw : w \in N(v)\},$$

wobei jedes Request $r \in R^{k,i}$ als Request von dem Vorgängerknoten von v zum Nachfolgerknoten von v in $KL(r)$ aufgefasst wird.

Dann ist H ein Stern und ein Schedule für H entspricht einem Schedule für $R^{k,i} \in \mathcal{G}$. □

Korollar 5.3.12. *Der Algorithmus List-Scheduling erzeugt für R^k einen Schedule mit maximal doppelter Länge des optimalen Schedules.*

Beweis. Wegen Satz 5.2.4 und Korollar 5.3.11 erzeugt List-Scheduling für $R^{k,i}$ einen Schedule $\alpha_{k,i}$ mit maximal doppelter Länge des optimalen Schedules. Da je zwei Requests $r \in R^{k,i}$, $s \in R^{k,j}$ mit $i \neq j$ konfliktfrei und die Mengen disjunkt sind, sind beliebige Schedules $\alpha_{k,i}$ für $R^{k,i}$, $i = 1, \ldots, n_k$ synchron. Insbesondere ist dann

$$\alpha_k := \biguplus_{i=1}^{n_k} \alpha_{k,i}$$

ein Schedule für R^k mit maximal doppelter Länge des optimalen Schedules. □

Algorithmus 24 : **List-Scheduling-Levels**
Erstellung eines globalen Schedules

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
Rückgabewert : Ein globaler Schedule.
Komplexität : $\mathcal{O}(|\mathbb{V}| \log |\mathbb{V}| + |\Gamma| \cdot l(\mathcal{G}) + |\Gamma|^2)$.

```
1  Funktion List-Scheduling-Levels
       // Zur Vereinfachung definiere l := ⌊log_2 |V|⌋
2      α := leerer globaler Schedule;
3      Bilde eine Zerlegung in die Klassen R^k, k = 0, ..., l;
4      t := 0 ;                                  // Länge des bisherigen Schedules α
5      for k = 0, ..., l − 1 do                  // Erzeuge alle nicht-trivialen Schedules
6          β := List-Scheduling(R^k) ;           // R^k ist disj. Vereinigung von Sternen
7          foreach r ∈ R^k do α(r) := t + β(r) ;  // Schedule hinten anhängen
8          t := t + |β| ;                        // Neue Länge des Schedules α
9      foreach r ∈ R^l do α(r) := 0 ;            // Requests aus R^l konfliktfrei
10     return α;
11 end
```

Satz 5.3.13. *Der Algorithmus List-Scheduling-Levels ist ein approximativer Algorithmus für einen globalen Schedule mit Approximationsgüte $\kappa = 2\lfloor \log_2 |\mathbb{V}| \rfloor$. Er hat eine Komplexität von $\mathcal{O}(|\mathbb{V}| \log |\mathbb{V}| + |\Gamma| \cdot l(\mathcal{G}) + |\Gamma|^2)$. Falls $|\Gamma| \geq |\mathbb{V}|$ ist diese $\mathcal{O}(|\Gamma|^2)$.*

Beweis. Im Folgenden sei $l := \lfloor \log_2 |\mathbb{V}| \rfloor$.

- Der Algorithmus List-Scheduling-Levels ist ein Spezialfall des in Algorithmus 14 dargestellten Algorithmus MultiSchedUnion, also korrekt. Die Variablen sind dabei wie folgt:
  - Es ist $\mathcal{R} = \{R^0, \ldots, R^l\}$.
  - Für $i < l$ wird der Schedule β_i' mittels List-Scheduling bezüglich der globalen Indizierung erzeugt. Wegen der Disjunktheit der R^k und der Verschiebung

    $$\beta_i(r) := \beta_i'(r) + |\alpha|$$

    um die Länge des bisherigen Schedules $|\alpha|$ ist die Synchronität von β_i und α gesichert.
  - Der Schedule β_l wird durch $\beta_l(r) := 0$ für alle $r \in R^l$ definiert und ist synchron zu α, da die Requests aus R^l zu keinem anderen Request in Konflikt stehen.
- Für die Länge $OPT(\Gamma)$ des optimalen Schedules gilt

  $$OPT(\Gamma) \geq \max_{0 \leq k \leq l} OPT(R^k).$$

  Nach Korollar 5.3.12 gilt für den von List-Scheduling erzeugten Schedule β_k für R^k die Abschätzung $|\beta_k| \leq 2 \cdot OPT(R^k)$. Für R^l ist der Schedule β_l, der jedes Request zum Zeitpunkt 0 schedulet, sogar optimal.

  Für die Länge $|\alpha|$ des erzeugten Schedules gilt daher

  $$\begin{aligned}
  |\alpha| &= \max\left(\sum_{k=0}^{l-1} |\beta_k|,\ \max\{\delta(r) : r \in R^l\}\right) \\
  &\leq \max\left(\sum_{k=0}^{l-1} 2 \cdot OPT(R^k), OPT(R^l)\right) \\
  &\leq \max\left(2l \cdot \max_{0 \leq k \leq l-1} OPT(R^k), OPT(R^l)\right) \\
  &\leq \max\left(2l \cdot OPT(\Gamma), OPT(\Gamma)\right) \\
  &\leq 2l \cdot OPT(\Gamma).
  \end{aligned}$$

- Das Finden eines Zerlegungsknotens eines Teilbaums ist wie folgt in linearer Zeit der Knotenanzahl möglich: Berechne für jeden Knoten v während der Post-Order-Traversierung, wie viele Knoten der von v aufgespannte Teilbaum enthält. Wähle als Zerlegungsknoten den ersten Knoten, für den dieser Teilbaum mindestens die Hälfte der Knoten der Zusammenhangskomponente enthält.
- Es gebe in der k-ten Ebene die n_k Zusammenhangskomponenten $H_{k,1}, \ldots, H_{k,n_k}$ mit $H_{k,i} = (V_{k,i}, E_{k,i})$. Da alle Komponenten disjunkt und Teilmenge von $\mathbb{V}$ sind, gilt offenbar

  $$\sum_{i=1}^{n_k} |V_{k,i}| \leq |\mathbb{V}|.$$

Nach Lemma 5.3.2 folgt für die Konstruktion der Trennlevelfunktion eine Komplexität von

$$\sum_{k=0}^{l}\sum_{i=1}^{n_k}\mathcal{O}(|V_{k,i}|) = \sum_{k=0}^{l}\mathcal{O}(|\mathbb{V}|) = \mathcal{O}(l\cdot|\mathbb{V}|) = \mathcal{O}(|\mathbb{V}|\log|\mathbb{V}|).$$

- Die Bestimmung des Levels eines Requests r hat eine Komplexität von $\mathcal{O}(|V_r|) \subset \mathcal{O}(M\cdot l(\mathcal{G})) = \mathcal{O}(l(\mathcal{G}))$, da der Knoten mit minimalem Trennlevel in V_r gefunden werden muss. Das Einteilen aller Requests in die Requestmengen $R^{k,i}$ hat somit eine Komplexität von $\mathcal{O}(|\Gamma|\cdot l(\mathcal{G}))$.
- Für die For-Schleife in Zeile 24 ergibt sich eine Komplexität von

$$\mathcal{O}\left(\sum_{k=0}^{l-1}|R^k|^2\right) \subset \mathcal{O}\left(\left(\sum_{k=0}^{l-1}|R^k|\right)^2\right) \subset \mathcal{O}(|\Gamma|^2).$$

- Für die ForEach-Schleife in Zeile 24 ergibt sich eine Komplexität von

$$\mathcal{O}(|R^l|) \subset \mathcal{O}(|\Gamma|) \subset \mathcal{O}(|\Gamma|^2).$$

- Somit folgt für List-Scheduling-Levels eine Komplexität von

$$\text{List-Scheduling-Levels} \in \mathcal{O}(|\mathbb{V}|\log|\mathbb{V}|) + \mathcal{O}(|\Gamma|\cdot l(\mathcal{G})) + \mathcal{O}(|\Gamma|^2).$$

Ist $|\Gamma| \geq |\mathbb{V}|$, so folgt $l(\mathcal{G}) \leq |\mathbb{V}| \leq |\Gamma|$ und somit

$$\text{List-Scheduling-Levels} \in \mathcal{O}(|\Gamma|\log|\Gamma|) + \mathcal{O}(|\Gamma|\cdot|\Gamma|) + \mathcal{O}(|\Gamma|^2) = \mathcal{O}(|\Gamma|^2).$$

□

Kapitel 6

Einheitliche Requestlängen

6.1 Natürliche Schedules

In diesem Kapitel wird vorausgesetzt, dass alle Requests dieselbe Dauer haben. Dann kann o.B.d.A. $\delta(r) = 1$ für jedes Request r angenommen werden. Wir zeigen, dass es sinnvoll ist, den Requests als Startzeit nur nicht-negative ganze Zahlen zuzuordnen. Insbesondere folgt dann für einen Schedule α und Requests r, s:

$$\widehat{\alpha(r)} \cap \widehat{\alpha(s)} = \emptyset \iff \alpha(r) \neq \alpha(s).$$

Satz und Definition 6.1.1 (Natürlicher Schedule)**.** *Sei $R \subset \Gamma$. Eine Abbildung $\alpha : R \mapsto \mathbb{N}_0$ ist genau dann ein Schedule für R, wenn für $r, s \in R$ gilt:*

$$r \leftrightsquigarrow s \implies \alpha(r) \neq \alpha(s). \tag{6.1}$$

In diesem Fall ist $|\alpha| = \max\{\alpha(r) + 1 : r \in R\}$ die Länge von α.

Ein solcher Schedule α heißt natürlicher Schedule.

Beweis. Zur Hinrichtung: Sei α ein Schedule. Dann gilt

$$r \leftrightsquigarrow s \implies \widehat{\alpha}(r) \cap \widehat{\alpha}(s) = \emptyset \implies \alpha(r) \neq \alpha(s)$$

und somit die Bedingung (6.1). Erfüllt umgekehrt α die Bedingung (6.1), dann gilt

$$r \leftrightsquigarrow s \implies \alpha(r) \neq \alpha(s) \implies \widehat{\alpha}(r) \cap \widehat{\alpha}(s) = \emptyset,$$

somit ist α ein Schedule. □

Satz 6.1.2 (Es gibt einen optimalen natürlichen Schedule)**.** *Sei R eine Menge von Requests mit Dauer 1 und $\alpha : R \mapsto \mathbb{R}^{\geq 0}$ ein Schedule. Dann ist*

$$\beta : R \mapsto \mathbb{N}_0, \qquad r \to \lfloor r \rfloor$$

ein natürlicher Schedule mit $|\beta| \leq |\alpha|$.

Insbesondere ist der optimale Schedule ein natürlicher Schedule.

Beweis. Seien $r, s \in R$ in Konflikt stehende Requests, o.B.d.A. mit $\alpha(r) \leq \alpha(s)$. Dann gilt

$$\begin{aligned} r \rightsquigarrow s &\implies \widehat{\alpha(r)} \cap \widehat{\alpha(s)} = \emptyset \\ &\implies [\alpha(r), \alpha(r)+1) \cap [\alpha(s), \alpha(s)+1) = \emptyset \\ &\implies \alpha(s) \notin [\alpha(r), \alpha(r)+1) && |\text{Es ist } \alpha(r) \leq \alpha(s) \\ &\implies \alpha(r) + 1 \leq \alpha(s) \\ &\implies \lfloor \alpha(r) \rfloor < \lfloor \alpha(s) \rfloor \\ &\implies \beta(r) < \beta(s). \end{aligned}$$

Für alle $r \in R$ ist $\beta(r) \leq \alpha(r)$, somit folgt

$$|\beta| = \max\{\beta(r) + 1 : r \in R\} \leq \max\{\alpha(r) + \delta(r) : r \in R\} = |\alpha|.$$

□

Damit lässt sich jeder Schedule für R immer als Abbildung $\alpha : R \mapsto \underline{|\alpha|}$ auffassen. Oft verwenden wir auch einen Schedule $\alpha : R \mapsto \underline{n}$, was aber nur $|\alpha| \leq n$ und nicht zwangsläufig $|\alpha| = n$ bedeutet.

In diesem Kapitel wird – soweit nicht explizit anders angegeben – mit dem Begriff *Schedule* immer ein *natürlicher Schedule* bezeichnet.

Definition 6.1.3 (Klassen eines natürlichen Schedules, induzierte Überdeckung)**.** *Durch einen natürlichen Schedule $\alpha : R \mapsto \underline{n}$ für $R \subset \Gamma$ wird nach Lemma 2.1.4 eine eindeutige disjunkte Überdeckung $\mathcal{T} = (\alpha^{-1})_{i \in \underline{n}}$ von R definiert. Wir nennen $\mathcal{T}$ die* von α induzierte Überdeckung *und die Klassen von $\mathcal{T}$ bezeichnen wir auch als die* Klassen von α*. Umgekehrt induziert eine disjunkte Überdeckung $\mathcal{T} = (T_i)_{i \in \underline{n}}$ von R einen eindeutigen Schedule $ind : R \mapsto \underline{n}$.*

Satz und Definition 6.1.4 (Umindizierung eines Schedules)**.** *Sei $R \subset \Gamma$ und $\alpha : R \mapsto \underline{n}$ ein Schedule für R. Sei $\varphi : \underline{n} \mapsto \underline{m}$ eine injektive Abbildung. Dann ist $\varphi \circ \alpha$ ein Schedule für R mit Länge $|\varphi \circ \alpha| \leq m$.*

Der so definierte Schedule $\varphi \circ \alpha$ entsteht aus α durch Umindizierung mit φ*.*

Beweis. Offensichtlich ist $\varphi \circ \alpha : R \mapsto \underline{m}$. Nun ist für $r, s \in R$ wegen der Injektivität von φ

$$r \rightsquigarrow s \implies \alpha(r) \neq \alpha(s) \implies \varphi \circ \alpha(r) \neq \varphi \circ \alpha(s),$$

also ist $\varphi \circ \alpha$ ein Schedule. Die Länge folgt, da für alle $r \in R$ gilt:

$$\varphi \circ \alpha(r) + 1 \leq (m-1) + 1 = m.$$

□

Definition 6.1.5 (Synchronisierbare Schedules)**.** *Seien $R, S \subset \Gamma$. Zwei Schedules $\alpha : R \mapsto \underline{n}$ und $\beta : S \mapsto \underline{m}$ heißen* synchronisierbar*, wenn es injektive Abbildungen $\varphi : \underline{n} \mapsto \mathbb{N}_0$ und $\psi : \underline{m} \mapsto \mathbb{N}_0$ gibt, so dass $\varphi \circ \alpha$ synchron zu $\psi \circ \beta$ ist.*

Eine Menge von Schedules $\{\alpha_i : R_i \mapsto \underline{n_i} \mid i \in I\}$ heißt synchronisierbar, wenn es injektive Abbildungen $\varphi_i : \underline{n_i} \mapsto \mathbb{N}_0$ gibt, so dass die Schedules $\varphi_i \circ \alpha_i$ synchron sind.

Bemerkung 6.1.6. Offenbar sind synchrone Schedules synchronisierbar. Für einen Schedule α und eine injektive Abbildung φ ist der Schedule $\varphi \circ \alpha$ zu α synchronisierbar. Weiter sind je zwei Schedules einer Menge synchronisierbarer Schedules synchronisierbar.

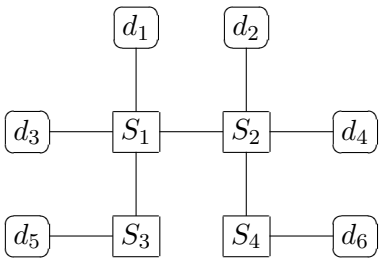


Abbildung 6.1: Ein Netzwerk mit Requests $r_1 : d_1 \to d_5$, $r_2 : d_2 \to d_6$, $r_3 : d_3 \to d_4$.

Man beachte jedoch, dass für eine Menge $\{\alpha_i : i \in I\}$ von Schedules paarweise Synchronisierbarkeit keine Synchronisierbarkeit impliziert. Man betrachte das Netzwerk aus Abbildung 6.1 mit den Requests $r_1 : d_1 \to d_5$, $r_2 : d_2 \to d_6$, $r_3 : d_3 \to d_4$. Sei $V = \{S_3, S_4\}$. Dann sind

$$\begin{aligned}
&\alpha_{S_1} : \{r_1, r_3\} \mapsto \{1\} && \alpha_{S_1}(r_1) = \alpha_{S_1}(r_3) = 1,\\
&\alpha_{S_2} : \{r_2, r_3\} \mapsto \{1\} && \alpha_{S_2}(r_2) = \alpha_{S_2}(r_3) = 1,\\
&\alpha_V : \{r_1, r_2\} \mapsto \{1, 2\} && \alpha_V(r_1) = 1, \qquad \alpha_V(r_2) = 2.
\end{aligned}$$

Schedules für S_1, S_2 und V. Offenbar lassen sich je zwei dieser Schedules synchronisieren.

Jedoch lassen sich nicht alle 3 Schedules synchronisieren. Denn gäbe es injektive $\varphi_{S_1}, \varphi_{S_2}, \varphi_V$ so, dass $\varphi_{S_1} \circ \alpha_{S_1}$, $\varphi_{S_2} \circ \alpha_{S_2}$ und $\varphi_V \circ \alpha_V$ synchron sind, so wäre

$$\begin{aligned}
\varphi_V(1) &= \varphi_V \circ \alpha_V(r_1) = \varphi_{S_1} \circ \alpha_{S_1}(r_1) = \varphi_{S_1}(1) = \varphi_{S_1} \circ \alpha_{S_1}(r_3)\\
&= \varphi_{S_2} \circ \alpha_{S_2}(r_3) = \varphi_{S_2}(1) = \varphi_{S_2} \circ \alpha_{S_2}(r_2) = \varphi_V \circ \alpha_V(r_2) = \varphi_V(2),
\end{aligned}$$

was ein Widerspruch zur Injektivität von φ_V ist.

Natürlich ist in diesem Fall der Schedule α_V nicht optimal und es könnte $\alpha_V(r_2) = 1$ definiert werden. Das ändert jedoch nichts an der Gültigkeit dieses Gegenbeispiels.

Satz 6.1.7. *Seien R und S konfliktfreie Mengen und $\alpha : R \mapsto \underline{n}$ und $\beta : S \mapsto \underline{m}$ zwei Schedules. Dann ist äquivalent:*

(i) *α und β sind synchronisierbar.*

(ii) *Für $r, s \in R \cap S$ gilt $\alpha(r) = \alpha(s) \iff \beta(r) = \beta(s)$.*

(iii) *Es gibt eine injektive Abbildung $\varrho : \underline{m} \mapsto \underline{\max(m, n)}$, so dass $\varrho \circ \beta$ synchron zu α ist.*

Beweis. Sei $\Lambda := R \cap S$. Wir führen den Ringschluss $(i) \implies (ii) \implies (iii) \implies (i)$.

Zu $(iii) \implies (i)$: Gibt es eine solche Abbildung, so ist mit $\varphi := id$ der Schedule $\varphi \circ \alpha = \alpha$ synchron zu $\varrho \circ \beta$, also sind α und β synchronisierbar.

Zu $(i) \implies (ii)$: Da α und β synchronisierbar sind, gibt es $\varphi : \underline{n} \mapsto \underline{\max(n,m)}$ und $\psi : \underline{m} \mapsto \underline{\max(n,m)}$ so, dass $\varphi \circ \alpha$ und $\psi \circ \beta$ synchron sind, d.h. es gilt: $\varphi \circ \alpha(r) = \psi \circ \beta(r)$ für alle $r \in \Lambda$.

Dann gilt für $r, s \in \Lambda$ und wegen der Injektivität von φ und ψ

$$\beta(r) = \beta(s) \iff \psi \circ \beta(r) = \psi \circ \beta(s) \iff \varphi \circ \alpha(r) = \varphi \circ \alpha(s) \iff \alpha(r) = \alpha(s).$$

Zu $(ii) \implies (iii)$: Definiere die Abbildung

$$\begin{aligned} &\varrho' : \beta(\Lambda) \mapsto \underline{\max(n,m)}, \\ &j \to \alpha(r) \qquad \text{für ein } r \in \Lambda \text{ mit } \beta(r) = j. \end{aligned}$$

- *Die Abbildung ϱ' ist wohldefiniert:* Für jedes $j \in \beta(\Lambda)$ gibt es ein $r \in \Lambda$ mit $\beta(r) = j$. Gibt es nun verschiedene $r, s \in \Lambda$ mit $\beta(r) = \beta(s) = j$, so gilt nach Voraussetzung auch $\alpha(r) = \alpha(s)$.
- *Die Abbildung ϱ' ist injektiv:* Seien $j, l \in \beta(\Lambda)$ und $r, s \in \Lambda$ mit $\beta(r) = j$ und $\beta(s) = l$. Dann gilt Injektivität wegen

$$\varrho'(j) = \varrho'(l) \iff \alpha(r) = \alpha(s) \iff \beta(r) = \beta(s) \iff j = l.$$

Nun ist $\beta(\Lambda) \subset \underline{m}$ und $|\underline{m}| = m \leq \max(m,n) = |\underline{\max(n,m)}|$, also existiert nach Satz 2.1.7 eine injektive Fortsetzung $\varrho : \underline{m} \mapsto \underline{\max(n,m)}$ von ϱ'.

Dann sind α und $\varrho \circ \beta$ nach Satz 4.3.6 synchron, denn für $r \in R \cap S = \Lambda$ gilt nach Definition

$$\varrho \circ \beta(r) = \varrho'(\beta(r)) = \alpha(r).$$

□

Korollar 6.1.8 (Vereinigung von Schedules)**.** *Seien R und S konfliktfreie Mengen und $\alpha : R \mapsto \underline{n}$ und $\beta : S \mapsto \underline{m}$ synchronisierbare Schedules mit $n \geq m$. Dann gibt es eine injektive Abbildung $\varrho : \underline{m} \mapsto \underline{n}$ so, dass $\alpha \uplus (\varrho \circ \beta)$ ein Schedule für $R \cup S$ ist.*

Beweis. Nach Satz 6.1.7 gibt es ein $\varrho : \underline{m} \mapsto \underline{\max(n,m)} = \underline{n}$ so, dass $\varrho \circ \beta$ synchron zu α ist. Nach Definition und Satz 4.3.9 ist $\alpha \uplus (\varrho \circ \beta)$ dann ein Schedule für $R \cup S$. □

Die in Satz 6.1.7(iii) gegebene Funktion ϱ und die Umindizierung $\varrho \circ \beta$ lässt sich effizient berechnen. Dieser Algorithmus Sync-Schedules ist in Algorithmus 25 dargestellt.

Satz 6.1.9. *Seien R und S konfliktfreie Mengen und $\alpha : R \mapsto \underline{n}$ und $\beta : S \mapsto \underline{m}$ synchronisierbare Schedules. Der in Algorithmus 25 dargestellte Algorithmus Sync-Schedules berechnet in $\mathcal{O}(\max(|S|, m))$ eine zu α synchrone Umindizierung von β.*

Für einen formalen Beweis des Algorithmus siehe Anhang A.1. Hier ist der Beweis skizziert:

Beweis. Im Wesentlichen wird eine Funktion ϱ mit den in Beweis zu Satz 6.1.7 verwendeten Eigenschaften konstruiert. Mit $\Lambda = R \cup S$ konstruieren wir eine Funktion $\varrho : \underline{m} \mapsto \underline{\max(m,n)}$, so dass gilt:

Algorithmus 25 : **Sync-Schedules**
Synchronisation zweier synchronisierbarer Schedules

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
R,S: Konfliktfreie Mengen.
Parameter : α: Ein Schedule für die Menge R, in einem Array als globaler Schedule gespeichert.
β: Ein zu α synchronisierbarer Schedule $\beta : S \mapsto \underline{m}$ bezüglich der lokalen Indizierung φ.
φ: Eine lokale Indizierung der Menge S.
Rückgabewert : Eine zu α synchrone Umindizierung γ von β mit $|\gamma| \leq \max(|\alpha|, |\beta|)$.
Komplexität : $\mathcal{O}(\max(|S|, m))$.

```
1 Funktion Sync-Schedules(α: Schedule, β: Schedule, φ: Indizierung von S)
2     ϱ : m ↦ ℕ0 ∪ {∞};
3     foreach j ∈ m do ϱ(j) := ∞ ;            // Zu Beginn ist die Abbildung ϱ leer
4     foreach r ∈ S do if r ∈ R then ϱ(β(r)) := α(r) ;          // Definiere ϱ|{R∩S}
5     foreach j ∈ m mit ϱ(j) = ∞ do ϱ(j) := min(ℕ0 \ ϱ(m)) ;     // Injekt. Forts.
6     γ =leerer lokaler Schedule;
7     foreach r ∈ S do γ(r) = ϱ(β(r)) ;            // Umindizierung von β(r) mit ϱ
8     return γ;
9 end
```

1. Für alle $r \in \Lambda$ gilt $\varrho(\beta(r)) = \alpha(r)$.
2. Die Abbildung $\varrho_{|\underline{m} \setminus \beta(\Lambda)} : \underline{m} \setminus \beta(\Lambda) \mapsto \underline{\max(m,n)} \setminus \alpha(\Lambda)$ ist injektiv.

Die erste Aussage gilt nach der Schleife in Zeile 4. Im Weiteren wird ein so gesetztes $\varrho(\beta(r))$ nicht mehr verändert. Denn sollte ein weiteres $s \in \Lambda$ mit $\beta(r) = \beta(s)$ existieren, so wird in der Schleife nach Satz 6.1.7 definiert:

$$\varrho(\beta(s)) = \alpha(s) = \alpha(r) = \varrho(\beta(r)).$$

In der zweiten Schleife in Zeile 5 wird für alle $j \notin \beta(\Lambda)$ gerade $\varrho(j)$ auf den kleinsten bis dahin noch nicht getroffenen Wert abgebildet, also ist die zweite Bedingung erfüllt.

In Zeile 7 wird γ als $\varrho \circ \beta$ definiert. Da ϱ nur Werte aus $\underline{\max(m,n)}$ annimmt, ist $|\gamma| \leq \max(m,n)$.

Zur Laufzeitkomplexität: Die Initialisierung von ϱ hat eine Komplexität von $\mathcal{O}(m)$, die Schleife in Zeile 4 von $\mathcal{O}(|S|)$. Die Schleife in Zeile 5 wird maximal m mal durchlaufen. Durch geschickte Implementierung der Berechnung des zugewiesenen Funktionswertes $\min(\mathbb{N}_0 \setminus \varrho(\underline{m}))$ kann sichergestellt werden, dass die Komplexität $\mathcal{O}(m)$ erhalten bleibt (siehe Anhang A.1). Die letzte Schleife in Zeile 7 hat eine Komplexität von $\mathcal{O}(|S|)$. Zusammen ergibt sich eine Komplexität von

$$\mathsf{SchedUnion} \in \mathcal{O}(m) + \mathcal{O}(|S|) + \mathcal{O}(m) + \mathcal{O}(|S|) = \mathcal{O}(\max(|S|, m)).$$

□

6.2 Natürliche Schedules zu Knotenmengen

Satz 6.2.1 (In Halbduplex-Netzwerken sind je zwei Schedules benachbarter Mengen synchronisierbar)**.** *Sei $\mathcal{N}$ ein Halbduplex-Netzwerk und $V, W \subset \mathbb{V}$ zusammenhängende benachbarte Mengen. Dann sind zwei Schedules $\alpha : \Gamma_V \mapsto \mathbb{N}_0$ für V und $\beta : \Gamma_W \mapsto \mathbb{N}_0$ für W synchronisierbar.*

Beweis. Nach Satz 4.3.15 sind Γ_V und Γ_W konfliktfrei. Je zwei verschiedene Requests in $\Gamma_V \cap \Gamma_W$ haben nach Lemma 4.2.25 einen Konflikt in V und in W, d.h. für $r, s \in \Gamma_V \cap \Gamma_W$ gilt

$$\alpha(r) = \alpha(s) \iff r = s \iff \beta(r) = \beta(s).$$

Nach Satz 6.1.7 sind α und β synchronisierbar. □

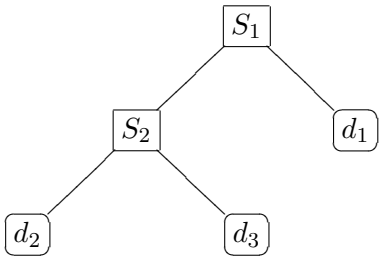


Abbildung 6.2: Vollduplex-Netzwerk mit Requests $r_1 : d_1 \to d_2$, $r_2 : d_2 \to d_1$, $r_3 : d_3 \to d_1$.

Bemerkung 6.2.2 (Die Aussage ist für Vollduplex-Netzwerke i.A. falsch)**.** Die Aussage aus Satz 6.2.1 gilt im Allgemeinen nicht für Vollduplex-Netzwerke. Denn im Vollduplex-Netzwerk aus Abbildung 6.2 mit Requests $r_1 : d_1 \to d_2$, $r_2 : d_2 \to d_1$, $r_3 : d_3 \to d_1$ sind

$$\begin{aligned} \alpha : \Gamma_{S_2} \mapsto \{1,2\}, \qquad & \alpha(r_1) = \alpha(r_3) = 1, \qquad \alpha(r_2) = 2 \\ \beta : \Gamma_{S_1} \mapsto \{1\}, \qquad & \beta(r_1) = \beta(r_2) = 1 \end{aligned}$$

Schedules für S_1 bzw. S_2. Jedoch lassen sich diese nicht geeignet synchronisieren. Denn würde ein solches $\varphi : \{1\} \mapsto \{1,2\}$ existieren, so dass $\varphi \circ \beta$ synchron zu α wäre, so wäre

$$1 = \alpha(r_1) = \varphi \circ \beta(r_1) = \varphi(1) = \varphi \circ \beta(r_2) = \alpha(r_2) = 2.$$

6.3 Konstruktion der Schedules

Im Folgenden versuchen wir nun, Schedules zu erstellen. Die wesentliche Idee ist, die Konstruktion eines Schedules auf Färbung von Konfliktgraphen zurückzuführen.

Der Knotenkonfliktgraph C (vgl. Abschnitt 4.4.2) ist so konstruiert, dass zwei Knoten in C genau dann benachbart sind, wenn die repräsentierten Requests einen Konflikt haben. Haben wir nun eine Knotenfärbung des Konfliktgraphen (d.h. jedem Knoten eine Farbe so geordnet, dass zwei benachbarte Knoten niemals dieselbe Farbe haben), so haben niemals zwei in Konflikt stehende Requests dieselbe Farbe. Daher impliziert eine Knotenfärbung unmittelbar einen Schedule.

Satz 6.3.1 (Knotenfärbung des Knotenkonfliktgraphen impliziert Schedule)**.**
Sei $C = (V, E, g)$ der Knotenkonfliktgraph zu R und $c : R \mapsto \underline{n}$. Dann ist c eine Knotenfärbung von C genau dann, wenn c ein Schedule für R ist. Insbesondere hat ein optimaler Schedule für R gerade die Länge $\chi(C)$.

Beweis. Wegen $V := R$ gilt $c : V \mapsto \underline{n} \iff c : R \mapsto \underline{n}$. Weiter gilt für $r \neq s$ nach Definition des Konfliktgraphen

$$r \text{ und } s \text{ sind benachbart in } C \iff r \text{ und } s \text{ haben einen Konflikt.}$$

Somit folgt

$$\begin{aligned} & c \text{ ist Knotenfärbung für } C \\ \iff & \text{Sind } r \text{ und } s \text{ benachbart in } C, \text{ so ist } c(r) \neq c(s) \\ \iff & \text{Haben } r \text{ und } s \text{ einen Konflikt, so ist } c(r) \neq c(s) \\ \iff & c \text{ ist Schedule für } R. \end{aligned}$$

□

Analog ist der Kantenkonfliktgraph C (vgl. Abschnitt 4.4.2) für eine Requestmenge Γ_v für Nicht-VD-MC-Netzwerke so konstruiert, dass zwei Kanten in C genau dann adjazent sind, wenn die repräsentierten Requests einen Konflikt haben. Somit impliziert Kantenfärbung des Kantenkonfliktgraphen ebenfalls einen Schedule.

Satz 6.3.2 (Kantenfärbung des Kantenkonfliktgraphen impliziert Schedule)**.** *Sei $\mathcal{N}$ kein VD-MC-Netzwerk, $C = (V, E, g)$ der Kantenkonfliktgraph zu v und $c : E \mapsto \underline{n}$. Dann ist c eine Kantenfärbung von C genau dann, wenn c ein lokaler Schedule für v ist. Insbesondere hat ein optimaler Schedule für v gerade die Länge $\chi'(C)$.*

Beweis. Wegen $E := \Gamma_v$ gilt $c : E \mapsto \underline{n} \iff c : \Gamma_v \mapsto \underline{n}$. Weiter gilt für $r \neq s$ nach Definition des Konfliktgraphen

$$r \text{ und } s \text{ sind adjazent in } C \iff r \text{ und } s \text{ haben einen Konflikt.}$$

Somit folgt

$$\begin{aligned} & c \text{ ist Kantenfärbung für } C \\ \iff & \text{Sind } r \text{ und } s \text{ adjazent in } C, \text{ so ist } c(r) \neq c(s) \\ \iff & \text{Haben } r \text{ und } s \text{ einen Konflikt, so ist } c(r) \neq c(s) \\ \iff & c \text{ ist Schedule für } v. \end{aligned}$$

□

Algorithmus 26 : **GlobalNatSchedule-BC**
Erstellung eines natürlichen Schedules durch Färbung des globalen Konfliktgraphen

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
color: Methode zum Färben des Konfliktgraphen C.
Rückgabewert : Ein natürlicher Schedule für R.
Komplexität : $\mathcal{O}(|\Gamma|^2 + \Theta^{\Sigma}) + \mathcal{O}(color)$.

```
1 Funktion GlobalNatSchedule-BC
2 |  C := Knotenkonfliktgraph-Global;
3 |  α := color(C);
4 |  return α;
5 end
```

6.3.1 Globaler Ansatz

Ein für alle Netzwerke funktionierender Ansatz ist das Färben des globalen Knotenkonfliktgraphen. Dieser Algorithmus GlobalNatSchedule-BC ist in Algorithmus 26 dargestellt.

Satz 6.3.3. *Der in Algorithmus 26 dargestellte Algorithmus GlobalNatSchedule-BC erstellt einen globalen Schedule in $\mathcal{O}(|\Gamma|^2 + \Theta^{\Sigma}) + \mathcal{O}(color)$. Die Approximationsgüte κ des Algorithmus ist gleich der Approximationsgüte κ_c des verwendeten Färbungsalgorithmus.*

Beweis. Nach Satz 6.3.1 ist die Färbung des globalen Knotenkonfliktgraphen C ein Schedule für Γ, also ist der Algorithmus korrekt. Da ein optimaler Schedule für Γ die Länge $\chi(C)$ hat, folgt $\kappa = \kappa_c$. □

Unmittelbar folgt das folgende Korollar:

Korollar 6.3.4. *Für beliebige Netzwerke gibt es einen Algorithmus zur Bestimmung eines optimalen globalen Schedules mit einer Laufzeit von*

$$\mathcal{O}\left(\left(\frac{4}{3} + \frac{3^{\frac{4}{3}}}{4}\right)^{|\Gamma|}\right) \approx \mathcal{O}(2,4150^{|\Gamma|}).$$

Des Weiteren gibt es einen polynomiellen Algorithmus mit Approximationsgüte

$$\kappa \in \mathcal{O}\left(\frac{|\Gamma|(\log\log|\Gamma|)^2}{(\log|\Gamma|)^3}\right).$$

Beweis. Dieses folgt durch Verwendung der in Abschnitt 3.2.2 beschriebenen Algorithmen Eppstein-Color bzw. Halldorson-Color. □

6.3.2 Lokaler Ansatz

Bei dem lokalen Ansatz werden der Reihe nach synchrone Schedules für die Knoten erstellt und diese vereinigt. Im Wesentlichen sind daher alle hier vorgestellten Algorithmen spezielle Versionen des in Algorithmus 15 dargestellten GlobalSchedUnion.

In Algorithmus 27 ist der Prototyp GlobalNatSchedule der hier verwendeten Algorithmen dargestellt. Die Korrektheit dieses Algorithmus lässt sich erst für die speziellen Fälle und Implementierungen beweisen. Jedoch lässt sich die Laufzeit berechnen.

Algorithmus 27 : **GlobalNatSchedule**
Prototyp der Erstellung eines globalen natürlichen Schedules durch Erstellung synchroner lokaler Schedules

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
color: Methode zum Färben des Konfliktgraphen C_v in $\mathcal{O}(h(v))$.
Rückgabewert : Ein globaler natürlicher Schedule.
Komplexität : Bei Kantenfärbung: $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} h(v))$.
Bei Knotenfärbung: $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma} + \sum_{v \in \mathbb{V}} h(v))$.

```
1  Funktion GlobalNatSchedule
2      α := leerer globaler Schedule;
3      foreach v ∈ 𝕍 do                                  // Breiten- oder Tiefensuche
           // Erzeugen eines β_v für Γ_v bezüglich lokaler Indizierung γ_v.
4          C_v := Konfliktgraph(v) ;     // Erzeuge Knoten- oder Kantenkonfliktgraph
5          Evtl.: Definiere Vorfärbung α_v ;              // Erzwingung der Synchronität
6          α_v := color(C_v) ;                    // Färbung mit den Farben 0,...,m−1
7          Erzeuge synchronen Schedule β_v aus α_v;
8          α := SchedUnion(α, β_v, γ_v);
9      return α;
10 end
```

Bemerkung 6.3.5. Sei W jederzeit die Menge, für die die ForEach-Schleife bereits durchlaufen wurde, d.h. Γ_W die Menge der bereits gescheduleten Requests. Dann sind in jedem Durchlauf die Mengen v und W zusammenhängend und benachbart, nach Satz 4.3.15 somit konfliktfrei. Daher ist der in Zeile 27 erzeugte Schedule β_v genau dann synchron zu α, wenn $\beta_{v|\Gamma_v \cap \Gamma_W} = \alpha_{|\Gamma_v \cap \Gamma_W}$ ist.

Satz 6.3.6 (Der Algorithmus GlobalNatSchedule)**.** *Ist jeder in Zeile 27 erzeugte Schedule β_v synchron zu α, d.h. gilt*

$$\beta_{v|\Gamma_v \cap \Gamma_W} = \alpha_{|\Gamma_v \cap \Gamma_W},$$

so erzeugt der Algorithmus einen globalen Schedule.

Für die einzelnen Schritte gelte:

- *Die Erzeugung des Konfliktgraphen $C_v = (V, E)$ hat eine Komplexität von $\mathcal{O}(|V|+|E|)$, d.h. $\mathcal{O}(d_{\mathcal{G}}(v) + |\Gamma_v|)$ bei Kantenkonfliktgraphen und $\mathcal{O}(|\Gamma_v| + \Theta(v))$ bei Knotenkonfliktgraphen.*
- *Die Erzeugung der Vorfärbung dauert maximal $\mathcal{O}(|\Gamma_v|)$.*

- *Das Färben des Konfliktgraphen hat eine Komplexität von* $\mathcal{O}(h(v))$.
- *Die Erzeugung des synchronen Schedules dauert* $\mathcal{O}(|\Gamma_v|)$.

Dann hat der Algorithmus GlobalNatSchedule *eine Laufzeit von*

$$\begin{aligned} &\textit{bei Kantenfärbung:} \quad \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} h(v)\right), \\ &\textit{bei Knotenfärbung:} \quad \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma} + \sum_{v \in \mathbb{V}} h(v)\right). \end{aligned}$$

Hat das Färben des Konfliktgraphen eine Komplexität $h(v) \in \mathcal{O}(|V| + |E|)$*, so folgt für die Gesamtkomplexität des Algorithmus*

$$\begin{aligned} &\textit{bei Kantenfärbung:} \quad \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G})\right), \\ &\textit{bei Knotenfärbung:} \quad \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma}\right). \end{aligned}$$

Beweis. Da in jedem Schritt der Schedule β_v zu α synchron ist, folgt die Korrektheit aus Satz 4.5.11.

Zur Laufzeit: Der Algorithmus GlobalSchedUnion hat nach Satz 4.5.11 eine Komplexität von $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} g(v))$, wobei $g(v)$ die Komplexität der Erstellung des Schedules für Γ_v ist.

- Bei der Verwendung von Kantenkonfliktgraphen und Kantenfärbung ergibt sich nach Voraussetzung

  $$g(v) \in \mathcal{O}(d_{\mathcal{G}}(v) + |\Gamma_v|) + \mathcal{O}(|\Gamma_v|) + \mathcal{O}(h(v)) + \mathcal{O}(|\Gamma_v|) = \mathcal{O}(d_{\mathcal{G}}(v) + |\Gamma_v| + h(v)).$$

  Somit folgt für die Gesamtkomplexität

  GlobalNatSchedule

  $$\begin{aligned} &\in \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} g(v)\right) \\ &\subset \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} (d_{\mathcal{G}}(v) + |\Gamma_v| + h(v))\right) \\ &= \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} d_{\mathcal{G}}(v) + \sum_{v \in \mathbb{V}} |\Gamma_v| + \sum_{v \in \mathbb{V}} h(v)\right) \quad \text{|Lemma 2.2.6 und Lemma 4.5.2} \\ &\subset \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + 2|\mathbb{E}| + |\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} h(v)\right) \qquad \text{|Es ist } 2|\mathbb{E}| = 2|\mathbb{V}| - 2 < 2 \sum_{v \in \mathbb{V}} h(v) \\ &= \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} h(v)\right). \end{aligned}$$

Ist nun $h(v) \in \mathcal{O}(d_{\mathcal{G}}(v) + |\Gamma_v|)$, so folgt daraus analog

$$\begin{aligned}\mathsf{GlobalNatSchedule} &\in \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} h(v)\right) \subset \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} (d_{\mathcal{G}}(v) + |\Gamma_v|)\right) \\ &\subset \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + 2|\mathbb{E}| + |\Gamma| \cdot l(\mathcal{G})\right) = \mathcal{O}(|\Gamma| \cdot l(\mathcal{G})).\end{aligned}$$

- Bei der Verwendung von Knotenkonfliktgraphen und Knotenfärbung ergibt sich nach Voraussetzung

$$g(v) \in \mathcal{O}(|\Gamma_v| + \Theta(v)) + \mathcal{O}(|\Gamma_v|) + \mathcal{O}(h(v)) + \mathcal{O}(|\Gamma_v|) = \mathcal{O}(|\Gamma_v| + \Theta(v) + h(v)).$$

Somit folgt für die Gesamtkomplexität analog zu oben

$$\begin{aligned}&\mathsf{GlobalNatSchedule} \\ &\in \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} g(v)\right) \subset \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} (|\Gamma_v| + \Theta(v) + h(v))\right) \\ &= \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} |\Gamma_v| + \sum_{v \in \mathbb{V}} \Theta(v) + \sum_{v \in \mathbb{V}} h(v)\right) \subset \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma} + \sum_{v \in \mathbb{V}} h(v)\right).\end{aligned}$$

Ist nun $h(v) \in \mathcal{O}(|\Gamma_v| + \Theta(v))$, so folgt daraus analog

$$\begin{aligned}\mathsf{GlobalNatSchedule} &\in \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} h(v)\right) \subset \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} (|\Gamma_v| + \Theta(v))\right) \\ &\subset \mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma}).\end{aligned}$$

□

6.3.2.1 Halbduplex-Netzwerke

Der folgende Algorithmus löst das Scheduling-Problem für Halbduplex-Netzwerke mit konstanter Request-Länge. Wesentlich ist dabei, dass in einem *HD*-Netzwerk je zwei Schedules für zusammenhängende benachbarte Mengen synchronisierbar sind.

Es wird der in Algorithmus 27 dargestellte Algorithmenprototyp GlobalNatSchedule nun konkretisiert.

(i) Der Konfliktgraph C_v wird mit den in Abschnitt 4.5.3 dargestellten Algorithmen bezüglich der lokalen Indizierung γ_v erzeugt.

(ii) Eine Vorfärbung wird nicht benötigt.

(iii) Die Färbung von C_v kann mit einem beliebigen Algorithmus in $\mathcal{O}(h(v))$ erfolgen.

(iv) Der Algorithmus β_v wird erzeugt durch $\beta_v := \mathsf{Sync\text{-}Schedules}(\alpha, \alpha_v, \gamma_v)$.

Der so modifizierte Algorithmus wird im Folgenden mit GlobalNatSchedule-HD bezeichnet und ist in Algorithmus 28 dargestellt.

Algorithmus 28 : **GlobalNatSchedule-HD**
Erstellung eines globalen natürlichen Schedules für HD-Netzwerke

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
color: Methode zum Färben des Konfliktgraphen C_v in $\mathcal{O}(h(v))$.
Rückgabewert : Ein globaler natürlicher Schedule.
Komplexität : Bei Kantenfärbung: $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v\in\mathbb{V}} h(v))$.
Bei Knotenfärbung: $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma} + \sum_{v\in\mathbb{V}} h(v))$.

```
1 Funktion GlobalNatSchedule-HD
2     α := leerer globaler Schedule;
3     foreach v ∈ 𝕍 do                                  // Breiten- oder Tiefensuche
          // Erzeugen eines β_v für Γ_v bezüglich lokaler Indizierung γ_v.
4         C_v := Konfliktgraph(v) ;                     // Knoten- oder Kantenkonfliktgraph
5         α_v := color(C_v) ;                           // Färbung mit den Farben 0,...,m−1
6         β_v := Sync-Schedules(α, α_v, γ_v);
7         α := SchedUnion(α, β_v, γ_v);
8     return α;
9 end
```

Satz 6.3.7. *Für ein HD-Netzwerk erzeugt der Algorithmus GlobalNatSchedule-HD einen globalen Schedule.*

Hat das Färben des Konfliktgraphen C_v eine Komplexität von $\mathcal{O}(h(v))$, so gilt für die Komplexität

$$\textit{bei Kantenfärbung:}\quad \textsf{GlobalNatSchedule-HD} \in \mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v\in\mathbb{V}} h(v)),$$
$$\textit{bei Knotenfärbung:}\quad \textsf{GlobalNatSchedule-HD} \in \mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma} + \sum_{v\in\mathbb{V}} h(v)).$$

Die Approximationsgüte κ von GlobalNatSchedule-HD ist gleich der Approximationsgüte κ_c des Färbungsalgorithmus.

Beweis.

- Zur Korrektheit: Der Algorithmus GlobalNatSchedule-HD ist eine konkrete Version von GlobalNatSchedule. Nach Satz 6.3.6 genügt es zu zeigen, dass in jedem Schritt β_v zu α synchron ist.

  Wird der Konfliktgraph C_v mit den in Abschnitt 4.5.3 beschriebenen Algorithmen erzeugt, so ist C_v ein Konfliktgraph bezüglich der lokalen Indizierung γ_v. Wegen Satz 6.3.1 bzw. Satz 6.3.2 ist dann nach der Färbung α_v ein lokaler Schedule für v bezüglich der Indizierung γ_v. Nach Bemerkung 6.3.5 und Satz 6.2.1 ist α_v zu α synchronisierbar. Daher erzeugt Sync-Schedules einen zu α synchronen Schedule β_v.

- Zur Laufzeit: Die Erzeugung des Konfliktgraphen hat eine Komplexität von $\mathcal{O}(|V| + |E|)$. Ein Vorfärben ist nicht nötig, die Färbung selbst dauert nach Voraussetzung $\mathcal{O}(h(v))$. Die Erzeugung des Schedules β_v aus α_v durch Synchronisation hat eine Komplexität von $\mathcal{O}(|\Gamma_v|)$. Damit folgt die Laufzeit aus Satz 6.3.6.

- Zur Approximationsgüte: Sei $OPT(R)$ die Länge eines optimalen Schedules für R. Dann gilt für jeden Knoten $|\beta_v| \leq \kappa_c \cdot OPT(\Gamma_v)$ und somit für den vom Algorithmus erzeugten Schedule α

$$|\alpha| = \left| \biguplus_{v \in \mathbb{V}} \beta_v \right| = \max_{v \in \mathbb{V}} |\beta_v| \leq \max_{v \in \mathbb{V}} \kappa_c OPT(\Gamma_v) = \kappa_c OPT(\Gamma),$$

  somit folgt die Behauptung.

□

Wird nun in GlobalNatSchedule-HD eine $\mathcal{O}(|V| + |E|)$-Heuristik zur Knoten- bzw. Kantenfärbung verwendet, so erzeugt der Algorithmus nach Satz 6.3.6 einen globalen Schedule in $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma})$ bei Knotenfärbung bzw. in $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}))$ bei Kantenfärbung.

Aus Abschnitt 3.4 können konkretere Schlussfolgerungen gezogen werden:

Korollar 6.3.8 (Approximatives Scheduling mit Approximationsgüte $\frac{4}{3}$)**.** *Sei $\mathcal{N}$ eine HD-UC-Netzwerk. Dann gibt es einen $\mathcal{O}(|\Gamma|^2 \cdot l(\mathcal{G}))$-Algorithmus mit Approximationsgüte $\kappa = \frac{4}{3}$ zur Bestimmung eines globalen Schedules.*

Beweis. Wird zur Kantenfärbung des Kantenkonfliktgraphen C_v der in Abschnitt 3.4 vorgestellte $\mathcal{O}(|E|(\Delta(G) + |V|))$-Algorithmus NK-Edgecolor verwendet, so hat der oben beschriebene Algorithmus eine Laufzeit von

$$\begin{aligned}
&\mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} |\Gamma_v|(\Delta(C_v) + d_{\mathcal{G}}(v))\right) \\
\subset &\mathcal{O}(|\Gamma| \cdot l(\mathcal{G})) + \mathcal{O}\left(\sum_{v \in \mathbb{V}} |\Gamma_v|(\Psi + \Delta(\mathcal{G}))\right) \\
\subset &\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \mathcal{O}((\Psi + \Delta(\mathcal{G}))|\Gamma| \cdot l(\mathcal{G})) \\
= &\mathcal{O}((\Psi + \Delta(\mathcal{G}))|\Gamma| \cdot l(\mathcal{G})) \\
\subset &\mathcal{O}(|\Gamma|^2 \cdot l(\mathcal{G})).
\end{aligned}$$

Die Approximationsgüte κ des Algorithmus GlobalNatSchedule-HD ist gleich der Approximationsgüte $\kappa_c = \frac{4}{3}$ des Färbungsalgorithmus. □

Korollar 6.3.9 (Bei beschränktem Grad gibt es einen exakten Algorithmus)**.** *Sei $\mathcal{N}$ ein Halbduplex-Netzwerk und der maximale Grad $d_{\mathcal{G}}(v)$ beschränkt. Dann gibt es einen Algorithmus mit Laufzeit $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}))$ zur Bestimmung eines optimalen globalen Schedules.*

Beweis. Die Anzahl der Knoten des Kantenkonfliktgraphen C_v ist gerade $|V_{C_v}| = |\mathbb{E}(v)| = d_{\mathcal{G}}(v)$, also beschränkt. Verwendet man nun zur Kantenfärbung den in Abschnitt 3.4 vorgestellten $\mathcal{O}(|E|)$-Algorithmus Bounded-Edgecolor, so hat der oben beschriebene Algorithmus wegen $h(v) = \mathcal{O}(|E|) \subset \mathcal{O}(|V| + |E|)$ nach Satz 6.3.6 eine Laufzeit von $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}))$.

Die Approximationsgüte κ des Algorithmus ist gleich der Approximationsgüte $\kappa_c = 1$ des exakten Färbungsalgorithmus. Also ist der Algorithmus GlobalNatSchedule-HD in diesem Fall exakt. □

Dieser Algorithmus ist jedoch praktisch nicht verwendbar, da die multiplikative Konstante von Bounded-Edgecolor zu groß ist.

6.3.2.2 Vollduplex-Netzwerke

Für Halbduplex-Netzwerke ist die Erstellung der lokalen Schedules die Hauptschwierigkeit. Denn dieses entspricht für Bäume beliebigen Grades der schwierigen Färbung der Konfliktgraphen. Bei VD-UC-Netzwerken hingegen ist die Kantenfärbung der Konfliktgraphen einfach, da diese bipartit sind. Die Konzepte dieses Abschnittes lassen sich nicht auf VD-MC-Netzwerke übertragen, da dort Requests in einem Knoten konfliktfrei sein aber global einen Konflikt haben können. Daher sollen hier ausschließlich VD-UC-Netzwerke betrachtet werden.

Satz 6.3.10 (Optimaler lokaler Schedule zu v in $\mathcal{O}(|\Gamma_v| \log \Psi)$)**.** *In VD-UC-Netzwerken lässt sich zu jedem Knoten $v \in \mathbb{V}$ in $\mathcal{O}(|\Gamma_v| \log \Psi)$ ein optimaler Schedule erstellen.*

Beweis. Nach Satz 4.4.8 ist der Kantenkonfliktgraph zu Γ_v bipartit. Wendet man den in Abschnitt 3.4 dargestellten $\mathcal{O}(|E| \log \Delta(G))$-Algorithmus zur exakten Kantenfärbung bipartiter Graphen auf den VD-Konfliktgraph C_v an, so kann ein optimaler lokaler Schedule konstruiert werden in

$$\mathcal{O}(|\Gamma_v| \log \Delta(C_v)) \subset \mathcal{O}(|\Gamma_v| \log \Psi).$$

□

Das Problem bei Vollduplex-Netzwerken ist, dass diese Schedules nicht zwangsläufig synchronisierbar sind. Daher ist es nicht möglich, für jeden Knoten – unabhängig von den bereits erzeugten Schedules für die anderen Knoten – optimale Schedules zu erstellen und diese anschließend zu synchronisieren.

Synchrone Schedules durch Vorfärben Eine Möglichkeit ist, vor der Färbung des Konfliktgraphen den bereits geschedulten Requests eine nicht mehr zu ändernde Farbe zuzuweisen. Der so erzeugte Schedule wäre synchron zu dem vorherigen Schedule.

Für VD-UC-Netzwerke wird der in Algorithmus 27 dargestellte Prototyp GlobalNatSchedule nun konkretisiert.

(i) Der Konfliktgraph C_v wird mit den in Abschnitt 4.5.3 dargestellten Algorithmen bezüglich der lokalen Indizierung γ_v erzeugt.

(ii) Bereits geschedulete Requests werden im Graphen so vorgefärbt, dass die lokale Farbe mit der globalen Zeit ihrer Ausführung übereinstimmt.

(iii) Die Färbung von C_v kann mit einem beliebigen Algorithmus in $\mathcal{O}(h(v))$ erfolgen, wobei die Farben der vorgefärbten Requests beibehalten werden.

(iv) Der Algorithmus β_v ist nun gleich der Färbung α_v.

Der so modifizierte Algorithmus GlobalNatSchedule-VD-UC-Prec (für Precolor) ist in Algorithmus 29 dargestellt.

Algorithmus 29 : **GlobalNatSchedule-VD-UC-Prec**
Erstellung eines globalen natürlichen Schedules für $VD\text{-}UC$-Netzwerke

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
color: Methode zum Färben des Konfliktgraphen C_v in $\mathcal{O}(h(v))$ unter Beibehaltung der Farben bereits gefärbter Knoten.
Rückgabewert : Ein globaler natürlicher Schedule.
Komplexität : Bei Kantenfärbung: $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} h(v))$.
Bei Knotenfärbung: $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma} + \sum_{v \in \mathbb{V}} h(v))$.

```
1  Funktion GlobalNatSchedule-VD-UC-Prec
2      α := leerer globaler Schedule;
3      foreach v ∈ 𝕍 do                                  // Breiten- oder Tiefensuche
4          C_v := Konfliktgraph(v) ;                  // Knoten- oder Kantenkonfliktgraph
           // Färbe bereits geschedulete Requests vor
5          β_v := Färbung für C_v;
6          foreach j ∈ |Γ_v| do                        // Erzwingen der Synchronität
7              if α(γ_v(j)) ≠ ∞ then β_v(j) := α(γ_v(j))
8          β_v := color(C_v) ;          // Färbung unter Beibehaltung bestehender Farben
9          α := SchedUnion(α, β_v, γ_v);
10     return α;
11 end
```

Satz 6.3.11. *Für ein $VD\text{-}UC$-Netzwerk erzeugt der in Algorithmus 29 dargestellte Algorithmus GlobalNatSchedule-VD-UC-Prec einen globalen Schedule.*

Hat das Färben des Konfliktgraphen C_v eine Komplexität von $\mathcal{O}(h(v))$, so gilt für die Komplexität

$$\textit{bei Kantenfärbung:} \quad \textsf{GlobalNatSchedule-VD-UC} \in \mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \sum_{v \in \mathbb{V}} h(v)),$$

$$\textit{bei Knotenfärbung:} \quad \textsf{GlobalNatSchedule-VD-UC} \in \mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma} + \sum_{v \in \mathbb{V}} h(v)).$$

Beweis. • Zur Korrektheit: Der Algorithmus GlobalNatSchedule-VD-UC ist eine konkrete Version von GlobalNatSchedule. Sei Γ_W die Menge der bereits gescheduleten Requests. Nach Satz 6.3.6 genügt es zu zeigen, dass in jedem Schritt $\beta_{v|\Gamma_v \cap \Gamma_W} = \alpha_{|\Gamma_v \cap \Gamma_W}$ ist.

Wird der Konfliktgraph C_v mit den in Abschnitt 4.5.3 beschriebenen Algorithmen erzeugt, so ist C_v ein Konfliktgraph bezüglich der lokalen Indizierung γ_v.

In Zeile 29 wird sichergestellt, dass

$$\forall j \in \underline{|\Gamma_v|} : \beta_v(j) := \alpha(\gamma_v(j))$$

gilt, d.h. dass für alle $r \in \Gamma_W \cap \Gamma_v$ die $\alpha(r)$ bezüglich des globalen Index gleich $\beta_v(r)$ bezüglich des lokalen Index gilt. Dann ist $\alpha_{|\Gamma_W \cap \Gamma_v} = \beta_{v|\Gamma_W \cap \Gamma_v}$.

- Zur Laufzeit: Die Erstellung der Konfliktgraphen dauert $\mathcal{O}(|V| + |E|)$, das Vorfärben hat eine Komplexität von $\mathcal{O}(|\Gamma_v|)$. Da $\beta_v = \alpha_v$ ist, folgt für den Algorithmus nach Satz 6.3.6 die Komplexität.

□

Wird in GlobalNatSchedule-VD-UC-Prec eine $\mathcal{O}(|V| + |E|)$-Heuristik zur Knoten- bzw. Kantenfärbung verwendet, so erzeugt der Algorithmus nach Satz 6.3.6 einen globalen Schedule in $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma})$ bei Knotenfärbung bzw. in $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}))$ bei Kantenfärbung.

Ein Problem dieses Algorithmus ist jedoch, dass viele Färbungsalgorithmen es nicht ermöglichen, vorgefärbte Knoten beizubehalten. Offensichtlich kann die Methode des Umfärbens bei sequentieller Färbung nicht verwendet werden. Außerdem wird bei Färbung mittels unabhängigen Mengen oder Matchings die Wahl des Algorithmus zum Finden unabhängiger Mengen eingeschränkt. Daher lassen sich viele gute Heuristiken zur Färbung nicht anwenden.

Synchronisierbare Schedules durch modifizierten Konfliktgraphen Ziel ist es nun, den Knotenkonfliktgraphen so zu modifizieren, dass auch in Vollduplex-Netzwerken eine Färbung des Konfliktgraphen einen zu α synchronisierbaren Schedule impliziert. Seien dazu $R, S \subset \Gamma$ konfliktfreie Mengen und $\alpha : R \mapsto \underline{n}$ ein Schedule. Wir konstruieren nun einen Konfliktgraphen C, so dass eine Knotenfärbung von C einen zu α synchronisierbaren Schedule β für S impliziert.

Satz und Definition 6.3.12. *Auf S wird eine Relation $\sim_\alpha$ definiert durch*

$$r \sim_\alpha s \iff r = s \vee (r, s \in R \wedge \alpha(r) = \alpha(s)).$$

Die Relation $\sim_\alpha \subset S^2$ ist eine Äquivalenzrelation.

Falls offensichtlich ist, auf welchen Schedule sich die Relation bezieht, lassen wir den Index α weg.

Beweis. Reflexivität und Symmetrie sind offensichtlich.

Zur Transitivität: Sei $r \sim s$ und $s \sim t$. Falls $r \in R$, dann ist wegen $r \sim s$ auch $s \in R$ und $\alpha(r) = \alpha(s)$. Somit ist wegen $s \sim t$ auch $t \in R$ und $\alpha(s) = \alpha(t)$. Es folgt $r, t \in R$ und $\alpha(r) = \alpha(t)$, also ist $r \sim t$.
Falls $r \notin R$, dann ist wegen $r \sim s$ auch $r = s$. Somit ist wegen $s \sim t$ auch $s = t$. Es folgt $r = t$ und somit $r \sim t$. □

Die Äquivalenzklasse zu einem Request r sei mit $[r]_\alpha$ bezeichnet und es sei $\mathcal{R}_\alpha = \{[r]_\alpha : r \in R\}$ die Menge aller Äquivalenzklassen. Auch hier wird der Index α ggf. weggelassen.

Definition 6.3.13 (Knotenkonfliktgraph zu S bezüglich α)**.** *Der Knotenkonfliktgraph $C = (V, E)$ zu S bezüglich α ist gegeben durch*

$$\begin{aligned}
&V := \mathcal{R},\\
&E := E_{konf} \cup E_{sched} \text{ mit}\\
&E_{konf} = \left\{\{[r],[s]\} \in V^{\underline{2}} : [r] \neq [s], r \rightsquigarrow s\right\},\\
&E_{sync} = \left\{\{[r],[s]\} \in V^{\underline{2}} : [r] \neq [s], r,s \in R \wedge \alpha(r) \neq \alpha(s)\right\}.
\end{aligned}$$

Der Graph C heißt Knotenkonfliktgraph zu S bezüglich α.

Der Knotenkonfliktgraph C zu S bezüglich α ist einfach und schlingenfrei.

Satz 6.3.14. *Sei $\alpha : R \mapsto \mathbb{N}_0$ ein Schedule, $C = (V, E)$ der Knotenkonfliktgraph zu S bezüglich α und $c : V \mapsto \underline{n}$. Definiere*

$$\beta : S \mapsto \underline{n}, \qquad \beta(r) = c([r]).$$

Dann ist c eine Knotenfärbung von C genau dann, wenn β ein zu α synchronisierbarer Schedule für S ist.

Beweis. Sei $c : V \mapsto \underline{n}$ eine Knotenfärbung von C. Dann ist β ein zu α synchronisierbarer Schedule für S, denn es gilt $\beta : S \mapsto \underline{n}$ und

$$\begin{aligned}
&\beta(r) = \beta(s)\\
\Longrightarrow\ &c([r]) = c([s])\\
\Longrightarrow\ &\{[r],[s]\} \notin E\\
\Longrightarrow\ &\{[r],[s]\} \notin E_{konf} \wedge \{[r],[s]\} \notin E_{sync}\\
\Longrightarrow\ &r,s \text{ haben keinen Konflikt} \wedge (r \notin R \vee s \notin R \vee \alpha(r) = \alpha(s)).
\end{aligned}$$

Insbesondere ist β ein Schedule und für $r, s \in R \cap S$ gilt

$$\beta(r) = \beta(s) \implies \alpha(r) = \alpha(s).$$

Weiter ist

$$\alpha(r) = \alpha(s) \implies r \sim s \implies [r] = [s] \implies c([r]) = c([s]) \implies \beta(r) = \beta(s).$$

Daher ist $\beta(r) = \beta(s) \iff \alpha(r) = \alpha(s)$, also ist nach Satz 6.1.7 β zu α synchronisierbar.

Sei umgekehrt β ein zu α synchronisierbarer Schedule für S. Dann ist $c : V \mapsto \underline{n}$ und es gilt

$$\begin{aligned}
&c([r]) = c([s])\\
\Longrightarrow\ &\beta(r) = \beta(s)\\
\Longrightarrow\ &\beta(r) = \beta(s) \wedge (r \notin R \vee s \notin R \vee \alpha(r) = \alpha(s))\\
\Longrightarrow\ &r \text{ und } s \text{ sind konfliktfrei} \wedge (r \notin R \vee s \notin R \vee \alpha(r) = \alpha(s))\\
\Longrightarrow\ &\{[r],[s]\} \notin E_{konf} \wedge \{[r],[s]\} \notin E_{sync}\\
\Longrightarrow\ &\{[r],[s]\} \notin E.
\end{aligned}$$

Also ist c eine Färbung für C. □

Nun soll die Anzahl der Kanten und Knoten des so definierten Konfliktgraphen abgeschätzt werden.

Lemma 6.3.15. *Seien $\{v\}, W \subset \mathbb{V}$ benachbarte zusammenhängende Mengen. Sei α ein Schedule für W. Wird die Relation $\sim_\alpha$ auf der Menge Γ_v definiert, so enthält jede Äquivalenzklasse maximal zwei Elemente.*

Beweis. Seien $r, s, t \in \Gamma_v$ paarweise verschiedene Requests mit $r \sim s$ und $r \sim t$. Dann gilt wegen $r \neq s$ und $s \neq t$: $r, s, t \in \Gamma_W$ und $\alpha(s) = \alpha(r) = \alpha(t)$, insbesondere stehen keine zwei der Requests r, s, t in Konflikt. Jedoch haben nach Lemma 4.2.25(ii) mindestens zwei der Requests r, s, t einen Konflikt. Widerspruch. □

Lemma 6.3.16. *Seien $\{v\}, W \subset \mathbb{V}$ benachbarte zusammenhängende Mengen. Sei $\Lambda := \Gamma_v \cap \Gamma_W$ Dann ist $|\Lambda|(|\Lambda| - 1) \leq 4\Theta(v) + 2|\Gamma_v|$.*

Beweis. Jedes Request $r \in \Lambda$ enthält Knoten aus v und W, somit die v-W-Kante $e = \{v, w\}$. Daher ist $r \in \Gamma_e$. Daraus folgt $\Lambda \subset \Gamma_e$ und nach Lemma 4.2.30(i)

$$|\Lambda|(|\Lambda| - 1) \leq |\Gamma_e|(|\Gamma_e| - 1) =\leq 4\Theta(e) + 2\Psi(e) \leq 4\Theta(v) + 2|\Gamma_v|.$$

□

Lemma 6.3.17. *Seien $\{v\}, W \subset \mathbb{V}$ benachbarte zusammenhängende Mengen und α ein Schedule für W. Dann gilt für den Knotenkonfliktgraph $C = (V, E)$ zu Γ_v bezüglich α:*

$$|V| \leq |\Gamma_v|, \qquad |E| \leq 3\Theta(v) + |\Gamma_v|$$

Insbesondere ist $\mathcal{O}(|V| + |E|) \subset \mathcal{O}(|\Gamma_v| + \Theta(v))$.

Beweis. Die Menge V ist die Menge der auf Γ_v definierten Äquivalenzklassen bezüglich $\sim$. Da keine Äquivalenzklasse leer ist, folgt $|V| \leq |\Gamma_v|$.

Weiter ist $|E_{konf}| \leq |\{(r, s) : r \rightsquigarrow_v s\}| = \Theta(v)$.

Dann ist

$$|E_{sync}| \leq |\{\{r, s\} : r, s \in \Gamma_v \cap \Gamma_W, r \neq s\}| = \frac{1}{2}|\Gamma_v \cap \Gamma_W|(|\Gamma_v \cap \Gamma_W| - 1) \leq 2\Theta(v) + |\Gamma_v|.$$

Somit ist $|E| \leq |E_{konf}| + |E_{sync}| \leq 3\Theta(v) + |\Gamma_v|$. □

Satz 6.3.18 (Der Algorithmus Knotenkonfliktgraph-Sync)**.** *Sei $\mathcal{N}$ ein VD-UC-Netzwerk. Der in Algorithmus 30 dargestellte Algorithmus Knotenkonfliktgraph-Sync bestimmt den Konfliktgraphen $C = (V, E)$ zu S bezüglich α in $\mathcal{O}(|\Gamma_v| + \Theta(v))$.*

Für einen formalen Beweis des Algorithmus siehe Anhang A.2. Dort wird auch die Bedeutung der globalen Variablen *class* sichtbar. Hier ist der Beweis skizziert.

Algorithmus 30 : **Knotenkonfliktgraph-Sync**
Erstellung des Knotenkonfliktgraph für Γ_v bezüglich α

Daten : $\mathcal{N}$: Die Struktur des VD-UC-Netzwerks.
class: Eine zu Beginn und am Ende leere Abbildung $class : \underline{|\Gamma|} \to \mathbb{N}_0 \cup \{\infty\}$.
Parameter : v: Der Knoten, für dessen Requestmenge Γ_v der Schedule erzeugt wird.
α: Der Schedule für die zusammenhängende zu v benachbarte Menge W, zu dem der konstruierte Schedule synchron sein soll.
Rückgabewert : C: Der Knotenkonfliktgraph zu Γ_v bezüglich α als Adjazenzliste.
cl: Abbildung, die jedem Request $r \in \Gamma_v$ in lokaler Indizierung γ_v die Nummer der Klasse $[r]$ zuordnet.
Komplexität : $\mathcal{O}(|\Gamma| + \Theta(v)) = \mathcal{O}(|V| + |E|)$ bezüglich des Konfliktgraphen.

```
1  Funktion Knotenkonfliktgraph-Sync(v: Knoten, α: Schedule)
2      cl := leere Abbildung als Array;
3      foreach r ∈ Γ_v do                                  // Bildung der Klassen [r]_α
4          if r ∉ R then lege neue Klasse [r] an und ergänze cl(r);
5          else if ∃s : α(r) = α(s) then cl(r) := cl(s) ;   // Füge r in Klasse [s] ein
6          else lege neue Klasse [r] an und ergänze cl(r);
7      Es gebe clnum Klassen {0, ..., clnum − 1};
8      C := leerer Graph mit clnum Knoten;
9      foreach r, s ∈ Γ_v ∩ Γ_W mit α(r) ≠ α(s) do        // Kanten aus E_sync einfügen
10         Füge Kante {cl(r), cl(s)} in C ein;
11     foreach e ∈ E(v) do                                 // Kanten aus E_konf einfügen
12         foreach r, s mit r ⇜⇝_e s do füge Kante {cl(r), cl(s)} in C ein;
13     Entferne mehrfach vorkommende Kanten aus C;
14     return (C, cl);
15 end
```

Beweis. Zur Korrektheit:

- *Es werden die korrekten Äquivalenzklassen gebildet:* In der ForEach-Schleife ab Zeile 30 werden die Klassen der Requests erstellt. Dabei ist jedes Request r in einer Klasse j und es wird $cl(r) := j$ definiert. Des Weiteren wird in Zeile 30 sichergestellt, dass zwei verschiedene Requests r, s in derselben Klassen sind, wenn $r, s \in \Gamma_W$ ist und sie in α zur selben Zeit ausgeführt werden.

- *Es wird der Konfliktgraph aus Definition 6.3.13 gebildet:* In der ForEach-Schleife ab Zeile 30 wird genau dann eine Kante zwischen zwei Klassen eingefügt, wenn diese zu unterschiedlichen Zeiten geschedulete Requests aus Γ_W enthalten. Somit sind nach dieser Schleife alle Kanten aus E_{sync} im Graphen eingefügt.

  In der ForEach-Schleife ab Zeile 30 wird genau dann eine Kante zwischen zwei Klassen eingefügt, wenn zwei Requests dieser Klassen einen Konflikt in einer zu v adjazenten Kante, also in v haben. Somit enthält C nach dieser Schleife alle Kanten aus E_{konf}.

  Nach dem Entfernen mehrfach vorkommender Kanten enthält der Konfliktgraph daher jede Kante aus $E_{konf} \cup E_{sync}$ genau einmal.

Zur Laufzeit: Es gibt $|V| \in \mathcal{O}(|\Gamma_v|)$ Knoten und $|E| \in \mathcal{O}(\Theta(v))$ Kanten.

- Die Schleife in Zeile 30 hat eine Komplexität von $\mathcal{O}(|\Gamma_v|)$.
- Die Erzeugung des leeren Graphen dauert $\mathcal{O}(|\Gamma_v|)$.
- Die Schleife ab Zeile 30 lässt sich in $\mathcal{O}(|\Gamma_v|)$ implementieren (siehe Anhang A.2).
- Die Schleife ab Zeile 30 benötigt eine Laufzeit von $\mathcal{O}(\Theta(v))$.
- Das Entfernen doppelter Kanten lässt sich ebenfalls in $\mathcal{O}(\Theta(v))$ implementieren.

Insgesamt ergibt sich eine Laufzeit von

$$\mathcal{O}(|\Gamma_v| + |\Gamma_v| + |\Gamma_v| + \Theta(v) + \Theta(v)) = \mathcal{O}(|\Gamma_v| + \Theta(v)).$$

□

Mit diesem Konfliktgraphen lässt sich nun ein Algorithmus zur Erstellung eines globalen Schedules für *VD-UC*-Netzwerke erzeugen. Dazu wird der in Algorithmus 27 dargestellte Algorithmenprototyp GlobalNatSchedule konkretisiert.

(i) Der Knotenkonfliktgraph C_v für Γ_v bezüglich α wird mit Knotenkonfliktgraph-Sync bezüglich der lokalen Indizierung γ_v erzeugt.

(ii) Es ist keine Vorfärbung notwendig.

(iii) Die Färbung von C_v kann mit einem beliebigen Algorithmus in $\mathcal{O}(h(v))$ erfolgen.

(iv) Der Algorithmus β_v wird nun erzeugt, indem jedes Request r als Startzeit die Farbe $\alpha_v([r])$ der Klasse $[r]$ erhält.

Der so modifizierte Algorithmus wird mit GlobalNatSchedule-VD-UC-Classes bezeichnet und ist in Algorithmus 31 dargestellt.

Algorithmus 31 : **GlobalNatSchedule-VD-UC-Classes**
Erstellung eines globalen natürlichen Schedules für $VD\text{-}UC$-Netzwerke

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
color: Methode zum Färben des Konfliktgraphen C_v in $\mathcal{O}(h(v))$.
Rückgabewert : Ein globaler natürlicher Schedule.
Komplexität : $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma} + \sum_{v \in \mathbb{V}} h(v))$.

```
1  Funktion GlobalNatSchedule-VD-UC-Classes
2      class := leerer Abbildung |Γ| → ℕ_0 ∪ ∞;
3      α := leerer globaler Schedule;
4      foreach v ∈ 𝕍 do                                  // Breiten- oder Tiefensuche
5          (C_v, cl) := Knotenkonfliktgraph-Sync(v, α) ;  // Konfliktgraph zu Γ_v bzgl. α
6          c := color(C_v) ;                              // Knotenfärbung der Klassen
7          foreach r ∈ Γ do α_v(r) := c(cl(r)) ;          // Erzeuge Schedule
8          β_v := Sync-Schedules(α, α_v, γ_v) ;           // Synchronisiere α_v zu α
9          α := SchedUnion(α, β_v, γ_v);
10     return α;
11 end
```

Satz 6.3.19. *Für ein $VD\text{-}UC$-Netzwerk erzeugt der in Algorithmus 31 dargestellte Algorithmus GlobalNatSchedule-VD-UC-Classes einen globalen Schedule.*

Hat das Färben des Konfliktgraphen C_v eine Komplexität von $\mathcal{O}(h(v))$, so gilt für die Komplexität

$$\textit{GlobalNatSchedule-VD-UC-Classes} \in \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma} + \sum_{v \in \mathbb{V}} h(v)\right).$$

Beweis. • Zur Korrektheit: Der Algorithmus GlobalNatSchedule-VD-UC-Classes ist eine konkrete Version von GlobalNatSchedule. Nach Satz 6.3.6 genügt es zu zeigen, dass in jedem Schritt β_v zu α synchron ist.

Nach Bemerkung 6.3.5 sind W und $\{v\}$ benachbarte zusammenhängende Mengen und somit Γ_W und Γ_v konfliktfrei. Der Konfliktgraph C_v ist ein Knotenkonfliktgraph für Γ_v bezüglich α, die Abbildung cl ordnet jedem Request r in lokaler Indizierung γ_v die r enthaltende Klasse $[r]$ zu. In Zeile 31 wird c eine Färbung für C_v, in der darauf folgenden Zeile wird für jedes Request $\alpha_v(r) := c([r])$ gesetzt. Nach Satz 6.3.14 ist dann α_v ein zu α synchronisierbarer Schedule. Somit wird in Zeile 31 β_v zu einem zu α synchroner Schedule für Γ_v.

- Zur Laufzeit: Die Erstellung der Konfliktgraphen dauert $\mathcal{O}(|\Gamma| + \Theta(v))$. Das Färben hat eine Komplexität von $h(v)$. Das anschließende Erzeugen von α_v und die Synchronisation dauern jeweils $\mathcal{O}(|\Gamma_v|)$. Somit ergibt sich die Komplexität aus Satz 6.3.6.

□

Auch dieser Algorithmus GlobalNatSchedule-VD-UC-Classes erzeugt bei der Verwendung einer $\mathcal{O}(|V|+|E|)$-Heuristik zur Knotenfärbung wegen $\mathcal{O}(|V|+|E|) \subset \mathcal{O}(|\Gamma|+\Theta(v))$ nach Satz 6.3.6 einen globalen Schedule in $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma})$.

Fazit für Scheduling in VD-UC-Netzwerken:

Bemerkung 6.3.20. Für keiner der beiden Algorithmen für VD-UC-Netzwerke entspricht die Approximationsgüte des Färbungsalgorithmus der Approximationsgüte des Algorithmus. Denn selbst wenn ein exakter Färbungsalgorithmus verwendet wird ist es möglich, dass durch bereits geschedulete Requests Bedingungen einfließen, die eine Optimalität des lokalen Schedules und insbesondere des globalen Schedules unmöglich machen.

Vergleicht man die beiden Algorithmen, so haben beide Vor- und Nachteile. Der erste Algorithmus benötigt das Färben unter Beibehaltung vorgefärbter Requests. Dadurch werden einige Heuristiken ausgeschlossen. Jedoch ist dafür nur der bipartite Kantenkonfliktgraph zu färben. Im zweiten Algorithmus hingegen wird das Beibehalten vorgefärbter Requests nicht benötigt, d.h. es können beliebige Färbungsalgorithmen angewandt werden. Dafür ist jedoch eine Knotenfärbung zu finden.

Bemerkung 6.3.21 (Bekannte Resultate zu VD-UC-Netzwerken)**.** In [Erl99, S.53f] zeigt Erlebach, dass es – falls P $\neq$ NP – für VD-UC-Netzwerke keinen polynomiellen Algorithmus mit Approximationsgüte $\kappa \leq \frac{4}{3}$ geben kann. Des Weiteren entwickelt er in [Erl99, Kapitel 4.2] einen polynomiellen Algorithmus für das Scheduling-Problem in VD-UC-Netzwerken, der einen Schedule mit maximaler Länge $\kappa = \lceil \frac{5}{3} \overrightarrow{\Psi} \rceil$ erzeugt. Da $\overrightarrow{\Psi}$ eine untere Schranke für die Länge des optimalen Schedules ist, hat dieser Algorithmus die Approximationsgüte 2.

Kapitel 7

Gestaffelte Requestlängen

7.1 Darstellung der Requestzeiten in gestaffelten Schedules

In diesem Kapitel nehmen wir an, dass die Requests exponentiell gestaffelte Übertragungslängen haben, d.h. für jedes Request $r \in \Gamma$ ist $\delta(r) \in \{2^k : k = 0, \ldots, N\}$, wobei $N \in \mathbb{N}_0$ als feste Konstante angenommen wird. Dabei sollen die Schedules so entworfen werden, dass ein Request der Länge 2^k nur zu einer Zeit $j \cdot 2^k$ mit $j \in \mathbb{N}_0$ ausgeführt werden kann.

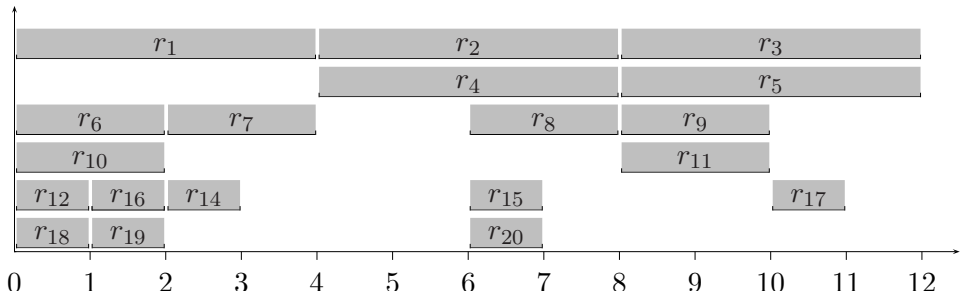


Abbildung 7.1: Gestaffelter Schedule für 20 Requests der Längen $1, 2$ und 4

Definition 7.1.1 (Gestaffelter Schedule)**.** *Sei $R \subset \Gamma$. Ein Schedule $\alpha : R \mapsto \mathbb{N}_0$ heißt* gestaffelter Schedule*, wenn für jedes $r \in R$ gilt: $\alpha(r)$ ist durch $\delta(r)$ teilbar.*

Bei konstanten Requestlängen gibt es nach Satz 6.1.2 immer einen optimalen natürlichen Schedule. Dieses ist bei gestaffelten Schedules nicht der Fall.

Beispiel 7.1.2 (Es gibt nicht immer einen optimalen gestaffelten Schedule)**.** In dem in Abbildung 7.2 dargestellten Netzwerk gilt $r_1 \leftrightsquigarrow r_2$, $r_2 \leftrightsquigarrow r_3$ und $r_3 \leftrightsquigarrow r_4$.

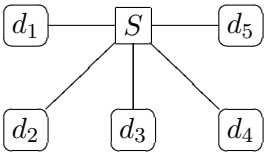
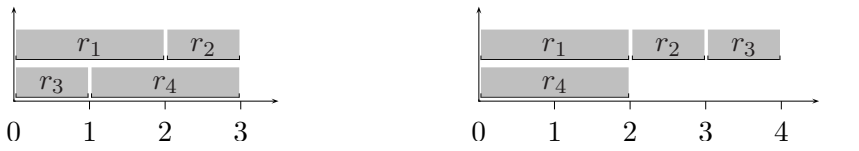


Abbildung 7.2: Ein HD-Netzwerk mit Requests $r_1 : d_1 \to d_2$, $r_2 : d_2 \to d_3$, $r_3 : d_3 \to d_4$, $r_4 : d_4 \to d_5$ und Dauer $\delta(r_1) = \delta(r_4) = 2$, $\delta(r_2) = \delta(r_3) = 1$.



(a) Schedule α mit Länge 3 (b) Gestaffelter Schedule β mit Länge 4

Abbildung 7.3: Optimaler Schedule und optimaler gestaffelter Schedule verschiedener Länge.

Ein optimaler Schedule mit Länge 3 ist gegeben durch $\alpha(r_1) = \alpha(r_3) = 0$, $\alpha(r_4) = 1$ und $\alpha(r_2) = 2$ (siehe Abbildung 7.3(a)).

Dieser ist jedoch nicht gestaffelt. In einem gestaffelten Schedule der Länge kleiner oder gleich 3 müssen die Requests r_1 und r_4 zur Zeit 0 ausgeführt werden. Da r_2 und r_3 einen Konflikt entweder mit r_1 oder r_4 haben, können sie dann frühstens zur Zeit 2 ausgeführt werden. Sie können aber nicht zeitgleich ausgeführt werden, insofern wird ein Request zum Zeitpunkt 2, das andere zum Zeitpunkt 3 ausgeführt. Der so entstehende Schedule β mit $\beta(r_1) = \beta(r_4) = 0$, $\beta(r_2) = 2$ und $\beta(r_3) = 3$ ist ein optimaler gestaffelter Schedule mit Länge 4 (siehe Abbildung 7.3(b)).

Gestaffelte Schedules lassen sich in binären Wäldern speichern. Dazu wird jedoch eine andere Darstellung der Ausführungszeiten der Requests benötigt.

Lemma 7.1.3. *Jede Zahl $k \cdot 2^n \in \mathbb{N}_0$ hat eine Darstellung*

$$k \cdot 2^n = \sum_{i=n}^{N} z_i \cdot 2^i, \qquad z_n \in \mathbb{N}_0, \qquad z_i \in \{0,1\}, \qquad i = 1, \ldots, N-1.$$

Somit können wir jede Zahl $k \cdot 2^n$ eindeutig als $(N+1)$-Tupel $(z_N, \ldots, z_0) \in \mathbb{N}_0 \times \{0,1\}^N$ mit $z_0 = \ldots = z_{n-1} = 0$ auffassen.

Beweis. Mit der eindeutigen Binärdarstellung

$$k = \sum_{i=0}^{j} k_i \cdot 2^i, \qquad k_i \in \{0,1\}$$

für k (wobei o.B.d.A. $j \geq N$ angenommen wird und ggf. die führenden Koeffizienten $k_i = 0$ sind) gilt

$$\begin{aligned} k \cdot 2^n &= \left(\sum_{i=0}^{j} k_i \cdot 2^i \right) \cdot 2^n = \sum_{i=0}^{j} k_i \cdot 2^{i+n} = \sum_{i=n}^{j+n} k_{i-n} \cdot 2^i \\ &= \sum_{i=n}^{N-1} k_{i-n} \cdot 2^i + \sum_{i=N}^{j+n} k_{i-n} \cdot 2^i = \sum_{i=n}^{N-1} k_{i-n} \cdot 2^i + 2^N \sum_{i=0}^{j+n-N} k_{N+i-n} \cdot 2^i = \sum_{i=n}^{N} z_i \cdot 2^i \end{aligned}$$

mit

$$z_i = k_{i-n} \in \{0,1\}, \qquad z_N = \sum_{i=0}^{j+n-N} k_{N+i-n} \cdot 2^i \in \mathbb{N}_0.$$

□

7.2 Slots

Mit dieser Darstellung lassen sich gestaffelte Schedules effizient in der Struktur eines binären Waldes bzw. eines Baumes speichern, bei dem jeder Knoten außer der Wurzel einen Grad kleiner oder gleich zwei hat. In jedem Knoten der Ebene k werden die Requests der Länge 2^{N-k} gespeichert, die zu diesem Zeitpunkt geschedulet sind. Zur Formalisierung soll im Folgenden der Begriff *Slot* definiert werden.

Eine Menge R von Requests lässt sich nun aufteilen in die Requests der verschiedenen Längen. Wir definieren

$$\begin{aligned} R^{=k} &= \{r \in R : d(r) = 2^k\}, \qquad k = 0, \ldots, N, \\ R^{\geq k} &= \{r \in R : d(r) \geq 2^k\}, \qquad k = 0, \ldots, N, \\ R^{\leq k} &= \{r \in R : d(r) \leq 2^k\}, \qquad k = 0, \ldots, N \end{aligned}$$

und entsprechend $\Gamma^{=k}$, $\Gamma_V^{=k}$ usw.

Definition 7.2.1 (Slot). *Sei α ein gestaffelter Schedule für R. Der Slot $S := Sl\,\langle z_N, \ldots, z_n\rangle$ ist formal definiert als das Tupel $(z_N, \ldots, z_n)$. Er hat die* Länge $\delta(S) = 2^n$, *die* Startzeit $start(S) = (z_N, \ldots, z_n, 0, \ldots, 0)$, *die* Endzeit $ende(S) = start(S) + \delta(S)$ *und das* Aktivitätsintervall

$$\widehat{S} = [start(S), ende(S))\,.$$

Des Weiteren ist

$$\Gamma_\alpha(S) := \left\{\{r \in R^{\leq n}\} : \alpha(r) = (z_N, \ldots, z_n, y_{n-1}, \ldots, y_0)\right\}$$

und $\Gamma_\alpha^=(S) := (\Gamma_\alpha(S))^{=n}$ die Menge der Requests mit Länge 2^n, die innerhalb von S ausgeführt werden.

In Lemma 7.2.4(i) wird gezeigt, dass $\Gamma_\alpha(S)$ die Menge der Requests ist, die komplett innerhalb des Aktivitätsintervalls $\widehat{S}$ ausgeführt wird.

Definition 7.2.2 (Eigenschaften von Slots). *Für Slots werden einige Begriffe definiert. Sei dazu α ein Schedule.*

- *Ein Slot S ist* leer *in α, wenn $\Gamma_\alpha^=(S) = \emptyset$ und* rekursiv leer, *wenn $\Gamma_\alpha(S) = \emptyset$. Ein nicht leerer Slot heißt* gefüllt, *ein nicht rekursiv leerer Slot heißt* rekursiv gefüllt.
- *Ein Slot S liegt* innerhalb *eines Slots oder* unter *einem Slot T, wenn $\widehat{S} \subset \widehat{T}$ ist. Dann liegt T* oberhalb *von S. Ist S innerhalb von T oder umgekehrt, so heißen diese Slots* abhängig. *Zwei nicht abhängige Slots heißen* unabhängig.
- *Ein Slot S heißt* frei *in α, wenn er nicht innerhalb eines gefüllten Slots liegt, d.h. wenn es keinen Slot oberhalb von S gibt, der ein Request mit größerer Länge als $\delta(S)$ enthält.*
- *Für $n < N$ heißen die Slots $S := Sl\,\langle z_N, \ldots, z_{n+1}, 0\rangle$ und $T := Sl\,\langle z_N, \ldots, z_{n+1}, 1\rangle$* benachbart. *Entsprechend heißt S* Nachbar *von T und umgekehrt.*
- *Ein Slot S hat in α einen Konflikt mit einem Request r, wenn ein Request $s \in \Gamma_\alpha(S)$ ein Konflikt mit r hat.*

Ist offensichtlich, auf welchen Schedule sich die Aussagen beziehen, so lässt man die Angabe des Schedules im Index weg.

Lemma 7.2.3. *Seien $S = Sl\langle s_N, \ldots, s_{n_s}\rangle$ und $T = Sl\langle t_N, \ldots, t_{n_t}\rangle$ zwei Slots. Dann ist äquivalent*

(i) $\widehat{S}$ *und* $\widehat{T}$ *sind nicht disjunkt, d.h.* $\widehat{S} \cap \widehat{T} \neq \emptyset$.

(ii) $s_i = t_i$ *für* $i = \max(n_s, n_t), \ldots, N$.

(iii) S *und* T *sind abhänging, d.h.* $\widehat{S} \subset \widehat{T}$ *oder* $\widehat{T} \subset \widehat{S}$.

Beweis. Sei o.B.d.A $\delta(S) \geq \delta(T)$, d.h. $n_s = j + n_t$ mit $j \geq 0$. Dann ist

$$start(S) = (s_N, \ldots, s_{n_s}, 0, \ldots, 0) = \sum_{i=n_s}^{N} s_i 2^i = \left(\sum_{i=n_s}^{N} s_i 2^{i-n_s}\right) \cdot 2^{n_s} = k \cdot 2^{n_s} = (k2^j) \cdot 2^{n_t},$$

$$start(T) = (t_N, \ldots, t_{n_t}, 0, \ldots, 0) = \sum_{i=n_t}^{N} t_i 2^i = \left(\sum_{i=n_t}^{N} t_i 2^{i-n_t}\right) \cdot 2^{n_t} = l \cdot 2^{n_t}.$$

Wir zeigen nun die Kette $(i) \iff (ii) \implies (iii) \implies (i)$.

Zu $(i) \iff (ii)$: Es ist

$$\begin{aligned}
&\widehat{S} \cap \widehat{T} \neq \emptyset \\
\iff &[start(S), start(S) + \delta(S)) \cap [start(T), start(T) + \delta(T)) \neq \emptyset \\
\iff &[k \cdot 2^{n_s}, (k+1) \cdot 2^{n_s}) \cap [l \cdot 2^{n_t}, (l+1) \cdot 2^{n_t}) \neq \emptyset \\
\iff &[(k2^j) \cdot 2^{n_t}, (k+1)2^j \cdot 2^{n_t}) \cap [l \cdot 2^{n_t}, (l+1) \cdot 2^{n_t}) \neq \emptyset \\
\iff &[k2^j, (k+1)2^j) \cap [l, l+1) \neq \emptyset \qquad \text{|es ist } l \in \mathbb{N}_0 \\
\iff &l \in [k2^j, (k+1)2^j) \\
\iff &\exists k_i \in \{0,1\} : \quad l = k2^j + \sum_{i=0}^{j-1} k_{i+n_t} 2^i \\
\iff &\exists k_i \in \{0,1\} : \quad l2^{n_t} = k2^{n_s} + \sum_{i=n_t}^{n_s-1} k_i 2^i \\
\iff &\exists k_i \in \{0,1\} : \quad start(T) = start(S) + \sum_{i=n_t}^{n_s-1} k_i 2^i \\
\iff &\exists k_i \in \{0,1\} : \quad \sum_{i=n_s}^{N} t_i 2^i + \sum_{i=n_t}^{n_s-1} t_i 2^i = \sum_{i=n_s}^{N} s_i 2^i + \sum_{i=n_t}^{n_s-1} k_i 2^i \\
\iff &\exists k_i \in \{0,1\} : \quad t_i = s_i \text{ für } i = n_s, \ldots, N \quad \wedge \quad t_i = k_i \text{ für } i = n_t, \ldots, n_s - 1 \\
\iff &t_i = s_i \text{ für } i = n_s, \ldots, N.
\end{aligned}$$

Wegen $n_s = \max(n_s, n_t)$ gilt somit die Behauptung.

Zu $(ii) \implies (iii)$: Sei $s_i = t_i$ für $i = n_s, \ldots, N$. Somit ist mit $s := start(S)$ und $t := start(S)$

$$t = \sum_{i=n_t}^{N} t_i 2^i = \sum_{i=n_s}^{N} t_i 2^i + \sum_{i=n_t}^{n_s-1} t_i 2^i = s + \sum_{i=n_t}^{n_s-1} t_i 2^i \geq s,$$

und weiter

$$\begin{aligned} t + \delta(T) &= s + \sum_{i=n_t}^{n_s-1} t_i 2^i + 2^{n_t} \leq s + \sum_{i=n_t}^{n_s-1} 2^i + 2^{n_t} \\ &= s + \frac{1-2^{n_s}}{1-2} - \frac{1-2^{n_t}}{1-2} + 2^{n_t} \\ &= s + (2^{n_s} - 1) + (1 - 2^{n_t}) + 2^{n_t} = s + 2^{n_s} = s + \delta(S). \end{aligned}$$

Es folgt

$$\widehat{T} = [t, t + \delta(T)) \subset [s, s + \delta(S)) = \widehat{S}.$$

Zu $(iii) \implies (i)$: Sei $\widehat{S} \subset \widehat{T}$ oder $\widehat{T} \subset \widehat{S}$. Wegen $\delta(S) \geq \delta(T)$ ist dann $\widehat{T} \subset \widehat{S}$ und es folgt $\widehat{S} \cap \widehat{T} = \widehat{T} \neq \emptyset$.

□

Lemma 7.2.4. *Sei α ein Schedule.*

(i) Ist S ein Slot und $r \in \Gamma_\alpha(S)$, so gilt $\widehat{\alpha}(r) \subset \widehat{S}$.
(ii) Sind Requests s und t in unabhängigen Slots, so ist $\widehat{\alpha}(s) \cap \widehat{\alpha}(t) \neq \emptyset$.

Beweis. Zu (i): Sei $S = Sl\,\langle s_N, \ldots, s_n \rangle$ und $\alpha(r) = (s_N, \ldots, s_n, r_{n-1}, \ldots, r_{n_r}, 0, \ldots, 0)$ mit $\delta(r) = 2^{n_r}$. Mit $T = Sl\,\langle s_N, \ldots, s_n, r_{n-1}, \ldots, r_{n_r} \rangle$ ist dann $r \in \Gamma^{=}(T)$ und nach Lemma 7.2.3 folgt $\widehat{\alpha}(r) = \widehat{T} \subset \widehat{S}$.

Zu (ii): Seien S und T unabhängig mit $s \in \Gamma(S)$ und $t \in \Gamma(T)$, so ist nach (i) und Lemma 7.2.3

$$\widehat{\alpha}(s) \cap \widehat{\alpha}(t) \subset \widehat{S} \cap \widehat{T} = \emptyset.$$

□

Lemma 7.2.5. *Sei $S = Sl\,\langle s_N, \ldots, s_n \rangle$ ein Slot und α ein Schedule für R. Dann sind die folgenden Aussagen äquivalent.*

(i) S ist frei.

(ii) Der Nachbarslot von S ist frei.

(iii) Für alle $r \notin \Gamma(S)$ gilt $\widehat{S} \cap \widehat{\alpha}(r) = \emptyset$.

(iv) Es gibt kein $r \in R^{=k}$, $k > n$ gibt mit $\alpha(r) = (s_N, \ldots, s_k, 0, \ldots, 0)$.

Beweis. Zu $(i) \iff (ii)$: Es gilt die Äquivalenz

$$\begin{aligned} & Sl\,\langle s_N, \ldots, s_{n+1}, 0 \rangle \text{ ist frei} \\ \iff & \forall k \geq n+1 : Sl\,\langle s_N, \ldots, s_k \rangle \text{ ist leer} \\ \iff & Sl\,\langle s_N, \ldots, s_{n+1}, 1 \rangle \text{ ist frei.} \end{aligned}$$

Zu $(i) \iff (iii)$: Sei S frei und $r \notin \Gamma(S)$. Sei T der Slot mit $r \in \Gamma^{=}(T)$. Ist T oberhalb von S, dann ist S nicht frei. Ist T in S, so wäre $r \in \Gamma^{=}(T) \subset \Gamma(S)$. Dieses steht beides im Widerspruch zur Annahme, daher sind S und T unabhängig. Nach Lemma 7.2.3 ist dann

$$\widehat{S} \cap \widehat{\alpha}(r) = \widehat{S} \cap \widehat{T} = \emptyset.$$

Ist hingegen S nicht frei, so gibt es einen nicht-leeren Slot T oberhalb von S. Dann ist $r \in \Gamma^{=}(T)$ und es folgt

$$\widehat{S} \cap \widehat{\alpha}(r) = \widehat{S} \cap \widehat{T} = \widehat{S} \neq \emptyset.$$

Zu $(i) \iff (iv)$: Es gilt die Äquivalenz

$$\begin{aligned}
& S = Sl\,\langle s_N, \ldots, s_n \rangle \text{ ist frei} \\
\iff & \forall k \geq n+1 : Sl\,\langle s_N, \ldots, s_k \rangle \text{ ist leer} \\
\iff & \nexists k \geq n+1 : Sl\,\langle s_N, \ldots, s_k \rangle \text{ ist nicht leer} \\
\iff & \nexists k \geq n+1 : \exists r \in R^{=k} \text{ mit } \alpha(r) = (s_N, \ldots, s_k, 0, \ldots, 0)
\end{aligned}$$

□

Lemma 7.2.6. *Sei α ein gestaffelter Schedule für R. Dann gilt für einen im Slot T:*

$$\{s \in R : \widehat{T} \cap \widehat{\alpha}(s) \neq \emptyset\} = \Gamma(T) \cup \{\Gamma^{=}(S) : S \textit{ oberhalb von } T\}.$$

Beweis. Sei $s \in R$ und $\widehat{\alpha}(r) \cap \widehat{\alpha}(s) \neq \emptyset$. Falls s in keinem Slot oberhalb von S liegt und nicht in $\Gamma(T)$, so liegt es in einem zu T unabhängigen Slot. Nach Lemma 7.2.4(ii) folgt $\widehat{r} \cap \widehat{s} = \emptyset$.

Falls $s \in \Gamma(T) \cup \{\Gamma^{=}(S) : S \text{ oberhalb von } T\}$. Ist $s \in \Gamma(T)$, so ist nach Lemma 7.2.4(i)

$$\widehat{\alpha}(s) \subset \widehat{T} \implies \widehat{\alpha}(r) \cap \widehat{\alpha}(s) = \widehat{T} \cap \widehat{\alpha}(s) = \widehat{\alpha}(s) \neq \emptyset.$$

Ist s in einem Slot S oberhalb von T geschedulet, so ist $r \in \Gamma(S)$ und nach Lemma 7.2.4(i) gilt analog

$$\widehat{\alpha}(r) \subset \widehat{S} \implies \widehat{\alpha}(r) \cap \widehat{\alpha}(s) = \widehat{\alpha}(r) \cap \widehat{S} = \widehat{\alpha}(r) \neq \emptyset.$$

□

Satz und Definition 7.2.7 (Vertauschung freier Slots)**.** *Sei α ein gestaffelter Schedule für R. Seien $S = Sl\,\langle s_N, \ldots, s_n \rangle$ und $T = Sl\,\langle t_N, \ldots, t_n \rangle$ zwei freie Slots gleicher Länge. Dann ist $\beta : R \mapsto \mathbb{N}_0$ definiert durch*

$$r \to \begin{cases} \alpha(r) & \textit{falls } r \notin \Gamma_\alpha(S) \cup \Gamma_\alpha(T) \\ (t_N, \ldots, t_n, r_{n-1}, \ldots, r_0) & \textit{falls } r \in \Gamma_\alpha(S) \textit{ mit } \alpha(r) = (s_N, \ldots, s_n, r_{n-1}, \ldots, r_0) \\ (s_N, \ldots, s_n, r_{n-1}, \ldots, r_0) & \textit{falls } r \in \Gamma_\alpha(T) \textit{ mit } \alpha(r) = (t_N, \ldots, t_n, r_{n-1}, \ldots, r_0) \end{cases}$$

ein gestaffelter Schedule für R. Dieser entsteht durch Vertauschen der Slots S und T *aus α.*

Beweis. Die Abbildung β ist offenbar wohldefiniert. Nun bleibt zu zeigen, dass β ein Schedule ist, also dass in Konflikt stehende Requests r und s nicht zeitgleich ausgeführt werden. Ist $S = T$, so ist $\beta = \alpha$ und die Aussage erfüllt. Sei nun $S \neq T$.

Dann gilt offenbar

$$\begin{aligned} \forall r \in \Gamma_\alpha(S): &\quad \widehat{\beta}(r) \subset \widehat{T}, \\ \forall r \in \Gamma_\alpha(T): &\quad \widehat{\beta}(r) \subset \widehat{S}. \end{aligned}$$

Seien $r, s \in R$ mit $r \leftrightsquigarrow s$.

- Sind $r, s \notin \Gamma_\alpha(S) \cup \Gamma_\alpha(T)$, so ist $\widehat{\beta}(r) \cap \widehat{\beta}(s) = \widehat{\alpha}(r) \cap \widehat{\alpha}(s) \neq \emptyset$.
- Sind $r, s \in \Gamma_\alpha(S)$, so wird durch die Vertauschung ihre relative Position zueinander nicht verändert. Da α ein Schedule ist, werden r und s also nicht gleichzeitig ausgeführt. Dasselbe gilt für $r, s \in \Gamma_\alpha(T)$.
- Ist $r \in \Gamma_\alpha(S)$ und $s \notin \Gamma_\alpha(S) \cup \Gamma_\alpha(T)$. Dann gilt $\widehat{\beta}(r) \subset \widehat{T}$ und $\widehat{\alpha}(s) = \widehat{\beta}(s)$. Da T frei ist, folgt
$$\widehat{\beta}(r) \cap \widehat{\beta}(s) \subset \widehat{T} \cap \widehat{\alpha}(s) = \emptyset.$$
Ist $r \in \Gamma_\alpha(T)$ und $s \notin \Gamma_\alpha(S) \cup \Gamma_\alpha(T)$, so folgt die Aussage analog.
- Ist $r \in \Gamma_\alpha(S)$ und $s \in \Gamma_\alpha(T)$. Dann gilt $\widehat{\beta}(r) \subset \widehat{T}$ und $\widehat{\beta}(s) \subset \widehat{S}$. Da S und T frei sind, gilt
$$\widehat{\beta}(r) \cap \widehat{\beta}(s) \subset \widehat{T} \cap \widehat{S} = \emptyset.$$
Ist $r \in \Gamma_\alpha(T)$ und $s \in \Gamma_\alpha(S)$, so folgt die Aussage analog.

□

7.3 Begriff Synchronisierbarkeit nicht übertragbar

Während bei natürlichen Schedules in Halbduplex-Netzwerken je zwei Schedules zu Bäumen synchronisierbar waren, fehlt bisher für gestaffelte Schedules der Begriff der Synchronisierbarkeit.

Um den Zeitaufwand der Synchronisation gering zu halten, müssen hinreichend große Blöcke synchronisierbarer Schedules betrachtet werden. So ergibt es keinen Sinn, Synchronisierbarkeit auf Basis der einzelnen Requests zu definieren. Somit gibt es zwei mögliche plausible Definitionen für die Synchronisierbarkeit gestaffelter Schedules. Man könnte Schedules synchronisierbar nennen, wenn sie durch eine der folgenden möglichen Umordnungen synchron würden.

- Die weniger restriktive Definition würde das Umschedulen aller Requests $\Gamma^{=}(S)$ eines Slots S zulassen. Allerdings ist dabei das Problem, dass für den neuen Zeitpunkt sichergestellt werden muss, dass kein Request ein Konflikt mit einem zu diesem Zeitpunkt ausgeführten anderen Request hat. Dafür muss jedes umgeschedulete Request mit jedem zu diesem Zeitpunkt laufenden Request überprüft werden. Der Aufwand hierfür ist zu groß, um effiziente Synchronisation zu ermöglichen.

- Die restriktivere Definition erlaubt, dass nur freie Slots gleicher Länge sowie benachbarte Slots vertauscht werden dürfen. Für diesen Begriff der Synchronisierbarkeit ließen sich synchronisierbare Schedules leicht synchronisieren. Jedoch ist dieser Begriff zu restriktiv. Seien $R, S \subset \Gamma$ Mengen mit

$$R = S = \{r_1, r_2, r_3\}$$
$$\delta(r_1) = \delta(r_2) = 2, \qquad \delta(r_3) = 1,$$

wobei nur r_1 und r_2 einen Konflikt haben. Dann sind die Schedules α und β für R bzw. S gegeben durch

$$\alpha(r_1) = 0, \qquad \alpha(r_2) = 2, \qquad \alpha(r_3) = 0,$$
$$\beta(r_1) = 0, \qquad \beta(r_2) = 2, \qquad \beta(r_3) = 2$$

nicht synchronisierbar.

(a) Der Schedule α (b) Der Schedule β

Abbildung 7.4: Zwei nicht synchronisierbare Schedules.

Denn durch die beschriebene Umindizierung könnte nicht geändert werden, dass im Schedule β immer r_3 parallel zu r_2 ausgeführt wird, in Schedule α hingegen nie.

Somit lässt sich ein Begriff von synchronisierbaren gestaffelten Schedules nicht sinnvoll einführen. Dennoch wird das Konzept der Vertauschung freier Slots bei der Erzeugung von Schedules nützlich sein.

7.4 Überblick über die Konstruktion der Schedules

Die gestaffelten Schedules sollen nun ähnlich zu den First-Fit-Algorithmus bzw. der sequentiellen Färbung erzeugt werden, d.h. in jedem Schritt soll einem Request die kleinstmögliche Zeit zugewiesen werden, wann dieser ausgeführt werden kann. Gegebenenfalls wird jedoch zugelassen, dass bereits geschedulete Requests zu einer anderen Zeit neu geschedulet werden. Prinzipiell gibt es zwei Möglichkeiten:

- Als eine Konkretisierung von GlobalSchedUnion wird der Schedule sequentiell auf jeden Knoten ausgeweitet. Dann genügt es, lokale Konfliktgraphen zu erzeugen.
- Basierend auf einem globalen Konfliktgraphen kann der Schedule in beliebiger Reihenfolge erzeugt werden. Da die Requests einer Menge $\Gamma^{=k}$ mit den in Abschnitt 6.1 dargestellten Algorithmen effizient geschedulet werden können, ist eine sinnvolle Möglichkeit, zuerst einen Schedule für $\Gamma^{=k}$ zu erstellen und diesen dann auf Requests anderer Längen auszuweiten. Dabei könnte begonnen werden mit einem Schedule für:

- Die Menge $\Gamma^{=N}$ der längsten Requests. Denn werden kürzere Requests suboptimal geschedulet, so verlängert dieses den Schedule weniger, als wenn längere Requests schlecht geschedulet werden.
- Die Menge $\Gamma^{=k}$, die die meisten Requests enthält. Bei diesem Ansatz würde die Verwendung effizienter Algorithmen aus Abschnitt 6.1 den größten Vorteil bieten.
- Die Menge $\Gamma^{=k}$, deren Gesamtdauer aller Requests am längsten ist, d.h. für die gilt
$$2^k \cdot |\Gamma^{=k}| = \max_{j=0,\dots,N} 2^j \cdot |\Gamma^{=j}|.$$
Dieses stellt einen Kompromiss der beiden vorherigen Möglichkeiten dar, in dem sowohl die Länge der Requests als auch die Anzahl berücksichtigt wird.

Für die Erweiterung auf eine Menge $\Gamma^{=j}$ gibt es ebenfalls zwei Möglichkeiten:

- Die Requests aus $\Gamma^{=j}$ können einzeln geschedulet werden.
- Existiert ein Schedule für $\Gamma^{=j}$, so können ganze Klassen von Requests geschedulet werden.

Datenstruktur für gestaffelte Schedules

Für gestaffelte Schedules lässt sich die folgendende Datenstruktur *GSchedule* verwenden. Der *GSchedule* für einen Schedule $\alpha : R \mapsto \mathbb{N}_0$ enthält:

- Ein Array *Slots* mit den Feldern $0, \dots, M$, das ein Wald von Binärbäumen enthält. Jeder Knoten repräsentiert einen Slot. Dabei ist M die maximale Anzahl benötigter Slots der Länge 2^N. Wird an α keine weiteren Bedingungen gestellt, so ist die Wahl $M = |R| - 1$ möglich.
- Jeder Knoten repräsentiert einen Slot S und enthält Verweise *lc(S)* und *rc(S)* auf den linken bzw. rechten Sohn (falls vorhanden, sonst sind diese Zeiger ∞). Haben die Felder aus dem Array *Slots* die Ebene 0, so wird der Slot $Sl\langle z_N, \dots, z_n\rangle$ als der Knoten der $N-n$-ten Ebene gespeichert, der von der Wurzel aus den Pfad $(z_N, z_{N-1}, \dots, z_n)$ hat. Dabei bezeichnet die z_N die Nummer des Binärbaums, und für $k = 1, \dots, N-n$ wird dort in der k-ten Ebene der linke Sohn gewählt, wenn $r_{N-k} = 0$ ist und der rechte, wenn $r_{N-k} = 1$ ist.

  Jeder Slot enthält einen Verweis *Vater(S)* auf den Vater und eine Liste *SlotRequests(S)* mit den Elemente aus $\Gamma^{=}(S)$. Des Weiteren enthält ein Slot die Informationen, der wievielte Sohn des Vaters der Slot ist. Für die Slots maximaler Länge ist dieses eine Nummer $0, \dots, M$, für alle anderen 0 oder 1.
- Für ein Request r wird in einem Array *Requests* bezüglich der (lokalen oder globalen) Indizierung eine Referenz auf den r enthaltenden Slot S gespeichert.

Zu Beginn werden nur die beiden leeren Arrays für Slots und Requests erzeugt, die einzelnen Knoten werden dann bei Bedarf eingefügt. Insbesondere existiert ein Slot nur, wenn er rekursiv gefüllt ist.

Diese Datenstruktur beinhaltet jedoch ein Problem. Sie ermöglicht nicht, effizient einen Schedule β für eine kleine Requestmenge $|R|$ zu erzeugen, der zu einem längeren Schedule α

synchron ist. Denn die maximale Länge des Schedules ist durch das Festlegen der Größe des Array *Slots* beschränkt. Wird dieses Array jedoch zu lang gewählt, ist die Erzeugung ineffizient.

Dieses Problem lässt sich durch eine Modifikation der Datenstruktur umgehen. Die Datenstruktur *GGSchedule* (globaler GSchedule) ist ein GSchedule für Γ. Dieser soll nun zu Beginn des Algorithmus einmal initialisiert werden (mit $|\Gamma|$ Slots maximaler Länge) und eine effiziente Methode bereitstellen, seinen Inhalt zu löschen. Dann ist es möglich, in jedem Teilschritt immer denselben, zu Beginn leeren globalen GGSchedule zu verwenden und am Ende seinen Inhalt zu löschen.

Dazu wird eine Liste *used* gespeichert, die die Nummern der verwendeten Slots maximaler Länge enthält.

Auf Grund der beschränkten Höhe des Baums existieren für *GGSchedule* die in Tabelle 7.1 dargestellten Methoden. Dabei ist R die Menge der tatsächlich im Schedule gespeicherten Requests.

7.5 Sequentielle Erweiterung des Schedules auf Knoten

Für Nicht-VD-MC-Netzwerke ist der folgende Algorithmus Exp-First-Fit-Nodes eine besonders effiziente mögliche Implementierung von First-Fit-Nodes und als solches ein Spezialfall von GlobalSchedUnion. Der Vorteil ist dabei die gute Laufzeitkomplexität, da kein globaler Konfliktgraph benötigt wird und das Schedulen der einzelnen Requests effizient möglich ist.

Algorithmus 32 : **Exp-First-Fit-Nodes**
Sequentielle Generierung eines globalen gestaffelten Schedules

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
Rückgabewert : Einen globalen Schedule als GGSchedule.
Komplexität : $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma})$.

```
1  Funktion Exp-First-Fit-Nodes
2      α := leerer GGSchedule mit |Γ| Slots maximaler Länge;
3      β := leerer GGSchedule mit |Γ| Slots maximaler Länge;
4      foreach v ∈ 𝕍 do                                  // Breiten- oder Tiefensuche
5          C := Knotenkonfliktgraph zu v als Adjazenzliste;
6          foreach r ∈ Γ_v mit α(r) = ∞ do              // Ungeschedulete Requests aus Γ_v
7              foreach w ∈ N_C(r) mit α(w) ≠ ∞ do füge w in β ein;
8              T := erster rekursiv leerer und freier Slot der Länge δ(r) in β;
9              Schedule r in Slot T;
10             Leere β;
11     return α;
12 end
```

Satz 7.5.1. *Sei $\mathcal{N}$ kein VD-MC-Netzwerk. Der Algorithmus Exp-First-Fit-Nodes berechnet in $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma})$ einen globalen Schedule.*

Operation	Komp.	Beschreibung
Erzeugen des Schedules	$\mathcal{O}(\|\Gamma\|)$	Es werden die leeres Arrays *Slots* und *Requests* sowie die leere Liste *used* angelegt.
Einfügen eines neuen Request	$\mathcal{O}(1)$	Der Slot $S = Sl\,\langle z_N, \ldots, z_n\rangle$ lässt sich in konstanter Zeit aufsuchen. Das Einfügen des Requests in *SlotRequests(S)* und der Referenz in *Requests* benötigt ebenfalls konstante Zeit. War der Slot $Sl\,\langle s_N\rangle$ bisher rekursiv leer, so ist er in *used* einzufügen.
Lesen von $\alpha(r)$	$\mathcal{O}(1)$	Im Array *Requests* ist eine Referenz auf den r enthaltenden Slot gespeichert. Von diesem lässt sich der Pfad zur Wurzel in konstanter Zeit bestimmen. Dieser entspricht der Zeit $\alpha(r)$.
Test, ob ein Slot rekursiv gefüllt ist	$\mathcal{O}(1)$	Ein Slot existiert genau dann, wenn er rekursiv gefüllt ist. Dieses ist in konstanter Zeit zu testen.
Test, ob ein Slot frei ist	$\mathcal{O}(1)$	Ein Slot S ist genau dann frei, wenn für alle Slots T auf dem Weg von der Wurzel zu S das Feld *SlotRequests(T)* leer ist.
Vertauschen zweier freier Slots	$\mathcal{O}(1)$	Die Zeiger der Elternelemente der Slots sind umzudefinieren.
Finden des ersten rekursiv freien Slots bestimmter Länge	$\mathcal{O}(\|R\|)$	Dieses entspricht einer Traversierung des Baums bis zur Höhe des Slots. Da es maximal $\|R\|$ rekursiv gefüllte Slots maximaler Länge gibt und die Anzahl der Slots innerhalb eines maximalen Slots beschränkt ist, ergibt sich die Komplexität von $\mathcal{O}(\|R\|)$.
Leeren des Schedules	$\mathcal{O}(\|R\|)$	Die maximal $\|R\|$ in *used* enthaltenden Slots maximaler Länge werden traversiert. Für jeden traversierten Slot S wird für jedes *SlotRequests(S)* gespeicherte Request die Referenz aus *Requests* entfernt. Anschließend müssen alle in *used* gespeicherten Slots entfernt und *used* gelöscht werden. Jede Operation benötigt für die maximal $\|R\|$ in *used* gespeicherten Slots nur eine beschränkte Anzahl an Operationen, daher ergibt sich die Komplexität $\mathcal{O}(\|R\|)$.

Tabelle 7.1: Methoden und Komplexität eines *GGSchedules* für die Requestmenge R

Beweis.

- Zur Korrektheit: Es genügt zu zeigen, dass jedes Request r zu einem Zeitpunkt geschedulet wird, zu dem kein bisher gescheduletes mit r in Konflikt stehendes Request aktiv ist.

  Sei $W \subset \mathbb{V}$ die Menge der Knoten, für die die in Zeile 32 beginnende Schleife bereits durchlaufen wurde. Das Request r werde im aktuellen Durchlauf für v in Zeile 32 geschedulet. In Zeile 32 wird

  Dann ist vorher $\alpha(r) \neq \infty$, also ist $r \notin \Gamma_W$. Sei nun s ein zu r in Konflikt stehendes, bereits gescheduletes Request. Dann ist $s \in \Gamma_W \cup \Gamma_v$.

  - Fall $s \in \Gamma_v$: Per Konstruktion enthält β alle bereits gescheduleten Requests t mit $t \leftrightsquigarrow_v r$. Da r in einen in β freien Slot geschedulet wird, wird es zu keinem solchen Request zeitgleich ausgeführt. Daher ist $\widehat{\alpha}(r) \cap \widehat{\alpha}(s) = \emptyset$.

  - Fall $s \notin \Gamma_v$: Dann ist $s \in \Gamma_W$. Nach Lemma 4.2.26 haben r und s dann einen Konflikt in v oder in W. Jedoch haben r und s wegen $s \notin \Gamma_v$ keinen Konflikt in v und wegen $r \notin \Gamma_W$ keinen Konflikt in W. Widerspruch zur Annahme $r \leftrightsquigarrow s$.

- Zur Laufzeit: Sei jeweils C der Konfliktgraph für v. Für einen Durchlauf der in Zeile 32 beginnenden ForEach-Schleife für ein Request r ergibt sich die folgende Komplexität:

  - Es müssen maximal $d_C(r)$ zu r in Konflikt stehende Requests jeweils in konstanter Zeit in β eingefügt werden. Dieses hat zusammen eine Komplexität von $\mathcal{O}(d_C(r))$, Anschließend enthält β maximal $d_C(r)$ Requests.
  - Das Finden des ersten rekursiv freien Slots der Länge $\delta(r)$ in β dauert ebenfalls $\mathcal{O}(d_C(r))$.
  - Das Einfügen des Requests in α ist in konstanter Zeit möglich.
  - Das Leeren des Schedules β benötigt $\mathcal{O}(d_C(r))$ Operationen.

  Somit ergibt sich für einen Schleifendurchlauf der in Zeile 32 beginnenden ForEach-Schleife eine Komplexität von $\mathcal{O}(1 + d_C(r))$. Für die gesamte Schleife folgt eine Komplexität von

  $$\sum_{r \in \Gamma_v} \mathcal{O}(1 + d_C(r)) = \mathcal{O}(|\Gamma_v| + 2|E_C|) = \mathcal{O}(|\Gamma_v| + \Theta(v)).$$

  Das Erzeugen des Knotenkonfliktgraphen für einen Knoten v benötigt $\mathcal{O}(|\Gamma_v| + \Theta(v))$. Somit folgt für den gesamten Algorithmus einschließlich der Initialisierung der Schedules α und β in $\mathcal{O}(|\Gamma|)$ eine Komplexität von

  $$\begin{aligned} \textsf{Exp-First-Fit-Nodes} &\in \mathcal{O}(|\Gamma|) + \sum_{v \in \mathbb{V}} (\mathcal{O}(|\Gamma_v| + \Theta(v)) + \mathcal{O}(|\Gamma_v| + \Theta(v))) \\ &= \mathcal{O}\left(|\Gamma| + |\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma}\right) = \mathcal{O}\left(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma}\right). \end{aligned}$$

□

7.6 Globales Scheduling

Wird zuerst ein globaler Konfliktgraph erzeugt, so sind mehrere Modifikationen von First-Fit denkbar, die im Folgenden vorgestellt werden. Alle Varianten benötigen einen globalen Konfliktgraphen als Adjazenzliste und Datenstruktur *GGSchedule* für das Finden des ersten möglichen Zeitpunktes eines Request r in $\mathcal{O}(d_C(r))$ bezüglich des globalen Konfliktgraphen C.

Bemerkung 7.6.1. Während des gesamten restlichen Kapitels wird die Existenz eines globalen Konfliktgraphen vorausgesetzt, der im Folgenden immer mit $C = (V_C, E_C)$ bezeichnet ist. Dann ist $V_C = \Gamma$, $|E_C| = \Theta$ und $d_C(r)$ die Anzahl der mit r in Konflikt stehenden Requests.

Definition 7.6.2. *Im Folgenden bezeichnen wir für eine Menge S von Requests mit*

$$N_C(S) := \bigcup_{s \in S} N_C(s)$$

die Menge der mit einem Requests aus S in Konflikt stehenden Requests.

Satz 7.6.3 (Erweiterung eines Schedules)**.** *Sei C der Konfliktgraph und α ein partieller gestaffelter Schedule für R.*

(i) Sei $s \notin R$. Dann dann entsteht durch Schedulen des Requests s in den Slot T der Länge $\delta(T) = \delta(s)$ genau dann ein Schedule für $R \cup \{s\}$, wenn T rekursiv leer und frei in $\alpha_{|N_C(s)}$ ist.

(ii) Sei S eine zu R disjunkte Menge paarweise konfliktfreier Requests der Länge 2^k. Dann dann entsteht durch Schedulen aller Requests $s \in S$ in den Slot T der Länge $\delta(T) = 2^k$ genau dann ein Schedule für $R \cup S$, wenn T rekursiv leer und frei in $\alpha_{|N_C(S)}$ ist.

Beweis. Zu (ii): Es gilt die mit $\beta := \alpha_{|N_C(S)}$ wegen der paarweisen Konfliktfreiheit der $s \in S$ die Äquivalenz

$$\begin{aligned}
&\text{Durch Schedulen aller } s \in S \text{ in } T \text{ entsteht ein Schedule für } R \cup S \\
\iff &\text{Nach Schedulen aller } s \in S \text{ in } T \text{ gilt für alle mit einem } s \in S \text{ in Konflikt stehenden} \\
&\text{Request } r\text{: } \widehat{T} \cap \widehat{\alpha}(r) = \emptyset \\
\iff &\{s \in N_C(S) : \widehat{T} \cap \widehat{\alpha}(r) \neq \emptyset\} = \emptyset \qquad \text{|Anwendung von Lemma 7.2.6} \\
\iff &\Gamma\beta(T) \cup \{\Gamma^{=}_{\beta}(S) : S \text{ oberhalb von } T\} = \emptyset \\
\iff &T \text{ ist rekursiv leer in } \beta \text{ und } T \text{ ist frei in } \beta.
\end{aligned}$$

Zu (i): Die Aussage folgt aus (ii) mit $S := \{s\}$.

□

7.6.1 Exp-First-Fit

Werden die Requests in beliebiger Reihenfolge geschedulet, so entspricht dieses dem Algorithmus First-Fit (siehe Algorithmus 22). Jedoch kann durch die Datenstruktur *GGSchedule* jedes Requests in $\mathcal{O}(1 + d_C(r))$ geschedulet werden, daher verringert sich die Laufzeit von $\mathcal{O}(|\Gamma|^2)$ auf

$$\mathcal{O}(\sum_{r\in\Gamma}(1 + d_C(r))) = \mathcal{O}(|\Gamma| + \Theta).$$

Dieser Algorithmus wird mit Exp-First-Fit bezeichnet.

Analog zum Algorithmus Decreasing-First-Fit (siehe Bemerkung 5.1.2) können die Requests nach absteigender Reihenfolge sortiert werden. Auf Grund der beschränkten Werte $\delta(r) \in \{2^k : k = 0, \ldots, N\}$ der Requestlängen lassen sich die Requests in linearer Zeit $\mathcal{O}(|\Gamma|)$ z.B. mit Bucketsort oder Radixsort nach der Länge sortieren. Dadurch vergrößert sich die Laufzeit nicht, d.h. es ist auch Decreasing-First-Fit $\in \mathcal{O}(|\Gamma| + \Theta)$.

7.6.2 Erweiterung eines Schedule einer Länge

In diesem Abschnitt soll die in Abschnitt 7.4 angedeutete Idee der Erweiterung eines Schedules für $\Gamma^{=k}$ konkretisiert werden. Dabei werden zwei Methoden benötigt.

- Die *Top-Down-Erweiterung* erweitert einen Schedule für $R \subset \Gamma^{>k}$ um eine Requestmenge $S \subset \Gamma^{=k}$ mit kürzerer Dauer k.
- Die *Bottom-Up-Erweiterung* erweitert einen Schedule für $R \subset \Gamma^{<k}$ um eine Requestmenge $S \subset \Gamma^{=k}$ mit längerer Dauer k.

Bei den dargestellten Algorithmen werden jeweils von einer Zerlegung der Menge S in Mengen S_i paarweise konfliktfreier Requests gleichzeitig geschedulet. In der Praxis gibt es zwei wesentliche Spezialfälle:

- Im ersten Fall enthält jede Menge S_i nur ein Request, d.h. jedes Request wird einzeln geschedulet.
- Im zweiten Fall wird mittels den Algorithmen für konstante Schedulelänge ein Schedule für $\Gamma^{=k}$ erzeugt und die S_i sind die Klassen dieses Schedules.

7.6.2.1 Top-Down-Erweiterung

Bei der *Top-Down-Erweiterung* wird ein Schedule α für eine Menge $R \subset \Gamma^{>k}$ auf eine Menge $R \cup S$ mit $S \subset \Gamma^{=k}$ erweitert. Das heißt, Requests der Länge 2^k werden in den Schedule eingefügt, in dem bisher nur Requests größerer Länge enthalten sind. Aus Komplexitätsgründen ergibt es keinen Sinn, die einzelnen bereits gescheduleten Requests umzuschedulen. Die Umordnung ganzer Slots der Länge $2^l, l \geq k$ hingegen ist nutzlos, da jedes neue Request der Länge 2^k komplett innerhalb eines solchen Slots ist.

Daher wird bei der Top-Down-Erweiterung ausschließlich jeder Block von Requests zum erstmöglichen Zeitpunkt geschedulet.

Algorithmus 33 : **Top-Down**
Top-Down-Erweiterung eines Schedules

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
β: Ein leerer GGSchedule. Dieser wird am Funktionsende wieder leer sein.
C: Ein globaler Knotenkonfliktgraph als Adjazenzliste.
Parameter : $\mathcal{S}$: Disjunkte Überdeckung $(S_i)_{i=1,\dots,n}$ der Menge $S \subset \Gamma^{=k}$. Die Requests einer Menge S_i sind paarweise konfliktfrei.
α: Ein GGSchedule für eine Menge R von Requests mit einer Länge größer als 2^k.
Rückgabewert : Einen GGSchedule α für $R \cup S$, der auf R unverändert ist.
Komplexität : $\mathcal{O}(|S| + \sum_{r \in S} d_C(r))$.

```
1  Funktion Top-Down(α: GGSchedule, 𝒮: Klassen von Requestmengen)
2      foreach S_i ∈ 𝒮 do
           /* Suche Slot T als ersten Slot der Länge 2^k, in den alle Requests
           r ∈ S_i geschedulet werden können.                                  */
3          foreach r ∈ S_i do
4              foreach s ∈ N_C(r) do
5                  if α(s) ≠ ∞ und β(s) = ∞ then füge s zur Zeit α(s) in β ein;
6          T := erster freier und leerer Slot der Länge 2^k in β;
7          foreach r ∈ S_i do                          // Schedule alle r ∈ S_i in Slot T
8              Füge r in α zu dem Slot T hinzu;
9          Leere β;
10     return α;
11 end
```

Satz 7.6.4. *Sei α ein Schedule für $R \subset \Gamma^{>k}$ und $(S_i)_{i=1,\dots,n}$ eine disjunkte Überdeckung von $S \subset \Gamma^{=k}$. Der in Algorithmus 33 dargestellte Algorithmus Top-Down erweitert den Schedule α auf $R \cup S$ mit $S \subset \Gamma^{=k}$ in einer Laufzeit von $\mathcal{O}(\sum_{r \in S} d_C(r))$.*

Beweis.

- Zur Korrektheit: Für jeden Block $S_i \in \mathcal{S}$ ist nach Durchlauf der in Zeile 33 beginnenden Schleife jedes Requests aus S_i geschedulet und α ein gültiger Schedule.

  Nach Ende der in Zeile 33 beginnenden Schleife ist $\beta = \alpha_{|N_C(S_i)}$. Dann wird T als in β freier und leerer Slot gewählt. Da bisher nur Requests aus $\Gamma^{\geq k}$ geschedulet wurden, ist T dann sogar rekursiv leer in β. Nach Satz 7.6.3 kann somit in Zeile 33 jedes Request aus S_i in T geschedulet werden. Anschließend wird β geleert.

- Zur Laufzeit: Für jedes $r \in S_i$ dauert das Innere der in Zeile 33 beginnenden Schleife $\mathcal{O}(d_C(r))$, die Gesamtkomplexität der in Zeile 33 beginnenden Schleife ist daher

$$\mathcal{O}\left(\sum_{r \in S_i} (1 + d_C(r))\right).$$

  Anschließend enthält der Schedule maximal $\sum_{r \in S_i} d_C(r)$ Requests. Daher hat sowohl Finden des ersten leeren und freien Slots sowie das Leeren von β eine Komplexität von

$$\mathcal{O}\left(\sum_{r \in S_i} d_C(r)\right).$$

  Mit der Komplexität $\mathcal{O}(|S_i|)$ des Einfügens aller Requests r in α ergibt sich für die Schleife eine Komplexität von

$$\mathcal{O}\left(\sum_{r \in S_i} (1 + d_C(r)) + \sum_{r \in S_i} d_C(r) + |S_i|\right) = \mathcal{O}\left(|S_i| + \sum_{r \in S_i} d_C(r)\right).$$

  Für den gesamten Algorithmus ergibt sich wegen der Disjunktheit der S_i

$$\text{Block-Top-Down} \in \sum_{S_i \in \mathcal{S}} \mathcal{O}\left(|S_i| + \sum_{r \in S_i} d_C(r)\right) = \mathcal{O}\left(|S| + \sum_{r \in S} d_C(r)\right)$$

□

Bemerkung 7.6.5. Bestehen die Mengen S_i nur aus einem Element, so werden die Requests einzeln geschedulet. Dieser Algorithmus wird als Single-Top-Down bezeichnet.

7.6.2.2 Bottom-Up-Erweiterung

Bei der *Bottom-Up-Erweiterung* wird ein Schedule α für eine Menge $R \subset \Gamma^{<k}$ auf eine Menge $R \cup S$ mit $S \subset \Gamma^{=k}$ zu erweitern. Das heißt, Requests der Länge 2^k werden in den Schedule eingefügt, in dem bisher nur Requests kleinerer Länge enthalten sind. In diesem Fall kann es sinnvoll sein, freie Slots der Länge 2^{k-1} umzuordnen.

(a) Vor dem Umordnen von α müsste r_5 zur Zeit 4 geschedulet werden.

(b) Nach dem Umordnen der Slots $Sl\langle 0,1\rangle$ und $Sl\langle 1,0\rangle$ kann r_5 geschedulet werden.

Abbildung 7.5: Beispiel für die Nützlichkeit der Umordnung gestaffelter Schedules.

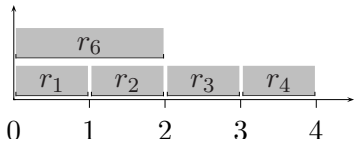
Beispiel 7.6.6 (Umordnen der Slots nützlich)**.** Sei $R = \{r_1, \ldots, r_5\}$ eine Menge von Requests mit $\delta(r_1) = \cdots = \delta(r_4) = 1$ und $\delta(r_5) = 2$. Nun haben $r_1, \ldots, r_4$ paarweise Konflikte und r_5 steht in Konflikt mit r_1 und r_3. Dann ist $\alpha(r_1) = 0$, $\alpha(r_2) = 1$, $\alpha(r_3) = 2$, $\alpha(r_4) = 3$ ein partieller Schedule (siehe Abbildung 7.5(a))

Nun kann r_5 weder zum Zeitpunkt 0 (wegen des Konflikts mit r_1) noch zum Zeitpunkt 2 (wegen des Konflikts mit r_3) geschedulet werden. Daher wird $\alpha(r_5) := 4$ definiert. Durch Vertauschen der freien Slots $Sl\langle 0,1\rangle$ und $Sl\langle 1,0\rangle$ entsteht ein Schedule mit $\alpha(r_1) = 0$, $\alpha(r_2) = 2$, $\alpha(r_3) = 1$, $\alpha(r_4) = 3$. Nun kann $\alpha(r_5) = 2$ geschedulet werden.

Soll ein Request r der Länge 2^k geschedulet werden, so genügt es, zwei freie Slots der Länge 2^{k-1} zu finden, die nicht mit r in Konflikt stehen. Diese können dann per Vertauschung so nebeneinander geschedulet werden, dass eine Slot der Länge 2^k entsteht, in dem kein Request einen Konflikt zu r hat.

Dieses Vertauschen ist jedoch nur für freie Slots uneingeschränkt möglich.

Beispiel 7.6.7 (Umordnen der Slots nicht immer möglich)**.** Sei $R = \{r_1, \ldots, r_5, r_6\}$ eine Menge von Requests mit $\delta(r_1) = \cdots = \delta(r_4) = 1$ und $\delta(r_5) = \delta(r_6) = 2$. Nun haben $r_1, \ldots, r_4$ paarweise Konflikte, r_5 mit r_1 und r_3 sowie r_6 mit r_3 und r_4. Dann ist $\alpha(r_1) = 0$, $\alpha(r_2) = 1$, $\alpha(r_3) = 2$, $\alpha(r_4) = 3$, $\alpha(r_6) = 0$ ein partieller Schedule.



Nun kann r_5 weder zum Zeitpunkt 0 (wegen des Konflikts mit r_1) noch zum Zeitpunkt 2 (wegen des Konflikts mit r_3) geschedulet werden. Daher wird $\alpha(r_5) := 4$ definiert. Im Gegensatz zu Beispiel 7.6.6 ist es jedoch nicht möglich, die Slots $Sl\langle 0,1\rangle$ und $Sl\langle 1,0\rangle$ zu vertauschen. Denn jetzt ist $Sl\langle 0,1\rangle$ nicht mehr frei und eine Vertauschung würde dazu führen, dass r_3 und r_6 trotz ihres Konflikts parallel ausgeführt würden.

Nun wird analog zum Top-Down-Algorithmus ein Schedule für $R \subset \Gamma^{<k}$ auf $R \cup S$ mit $S \subset \Gamma^{=k}$ erweitert, indem ganze Blöcke S_i paarweise konfliktfreier Requests einer disjunkten Überdeckung $(S_i)_{i=1,\ldots,n}$ von S gleichzeitig geschedulet werden.

- Es wird zuerst versucht, die neuen Requests neben bereits geschedulete Requests der Länge 2^k zu schedulen. Denn dadurch werden keine neuen freien Slots der Länge 2^{k-1} gebunden.
- Ist dieses nicht möglich, so werden freie Slots der Länge 2^{k-1} so getauscht, dass ein Slot der Länge 2^k entsteht, in den die neuen Requests geschedulet werden können. Dabei wird durch die Wahl der freien Slots die Länge des Schedules nicht vergrößert.
- Ist dieses nicht möglich, so werden die Requests an das Ende des Schedules angefügt.

Bemerkung 7.6.8. Auch hier gibt es die beiden wichtigen Spezialfälle, dass alle S_i einelementig sind (also jedes Request einzeln geschedulet wird) oder dass die S_i die Klassen eines Schedules für S sind. Beide Verfahren haben Schwächen: Beim einzelnen Schedulen wird es vermeintlich nur für die ersten Requests der Länge 2^k kleinere freie Slots geben. Die restlichen Requests werden mittels einfachen First-Fit geschedulet. Beim Schedulen der Klassen eines Schedules für S wird dieser ohne das Wissen der möglichen Konflikte mit Requests aus R erzeugt. Daher gibt es evtl. Konflikte, die bei der Berücksichtigung dieser Informationen vermeidbar gewesen wären.

Satz 7.6.9. *Sei α ein Schedule für $R \subset \Gamma^{<k}$ und $(S_i)_{i=1,\dots,n}$ eine disjunkte Überdeckung von $S \subset \Gamma^{=k}$. Der in Algorithmus 34 dargestellte Algorithmus* Bottom-Up *erweitert den Schedule α auf $R \cup S$ mit $S \subset \Gamma^{=k}$ in einer Laufzeit von $\mathcal{O}(|S| + \sum_{r \in S} d_C(r))$.*

Beweis. Da bisher nur Requests aus $\Gamma^{\leq k}$ geschedulet wurden, ist offenbar jeder Slot der Länge 2^k frei. Insbesondere können je zwei Slots der Länge 2^k vertauscht werden. Dieses wird im Algorithmus verwendet, um dich gefüllten Slots der Länge 2^k an den Anfang des Schedules zu tauschen.

Für ein festes k sei für $j \in \mathbb{N}_0$ der Slot T_j der Slot, mit $\delta(T_j) = 2^k$ und der Anfangszeit $j \cdot 2^k$. Während des gesamten Algorithmus gibt es eine Variable l so, dass gilt:

$$\text{Alle Slots } T_i,\ i < l \text{ sind gefüllt und alle Slots } T_i,\ i \geq l \text{ sind leer.} \tag{7.1}$$

Es werde die in Zeile 34 beginnende ForEach-Schleife für ein S_i durchlaufen.

- Nach der in Zeile 34 beginnenden ForEach-Schleife ist $\beta := \alpha_{|N_C(S_i)}$. Denn vor der Schleife war β leer und in der Schleife wurden für alle $r \in S_i$ alle Requests aus $N_C(r)$, die in α bereits geschedulet waren, in β eingefügt. Somit ist $\beta = \alpha_{|N_C(S_i)}$.
- Fall: Die Bedingung in Zeile 34 ist erfüllt: Dann wird T als rekursiv leerer Slot in β gewählt. T ist wegen $\delta(T) = 2^k$ frei. Nach Satz 7.6.3 kann jedes Request aus S_i in T geschedulet werden. Da kein neuer gefüllter Slot der Länge 2^k entsteht, ist kein vertauschen dieser Slots notwendig. Es wird in dem Schleifendurchlauf nur noch β geleert und anschließend mit dem nächsten Request fortgefahren.
- Ansonsten wird der Else-Zweig ab Zeile 34 durchlaufen. Da genau die Slots $T_0, \dots, T_{l-1}$ der Länge 2^k in α gefüllt sind, sind die in α freien Slots der Länge 2^{k-1} gerade die Slots T mit $\mathit{start}(T) \geq \mathit{ende}(T_{l-1}) = l \cdot 2^k$. Daher werden in Zeile 34 und Zeile 34 X und Y als die ersten beiden in α freien und in β rekursiv leeren Slots der Länge 2^{k-1} gewählt. Es ist gesichert, dass solche Slots existieren, da es $|\Gamma|$ Slots maximaler Länge gibt und diese noch nicht alle rekursiv gefüllt sind.

Algorithmus 34 : **Bottom-Up**
Bottom-Up-Erweiterung eines Schedules

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
β: Ein leerer GGSchedule. Dieser wird am Funktionsende wieder leer sein.
C: Ein globaler Knotenkonfliktgraph als Adjazenzliste.
Parameter : $\mathcal{S}$: Disjunkte Überdeckung $(S_i)_{i=1,\dots,n}$ der Menge $S \subset \Gamma^{=k}$. Die Requests einer Menge S_i sind paarweise konfliktfrei.
α: Ein GGSchedule für eine Menge $R \subset \Gamma^{<k}$.
Rückgabewert : Ein GGSchedule α für $R \cup S$.
Komplexität : $\mathcal{O}(|S| + \sum_{r \in S} d_C(r))$.

```
1  Funktion Bottom-Up(α: GGSchedule, 𝒮: Klassen von Requestmengen)
2      l := 0 ;            // Zur Speicherung des ersten leeren Slots der Länge 2^k
3      foreach S_i ∈ 𝒮 do
4          foreach r ∈ S_i do
5              foreach s ∈ N_C(r) do
6                  if α(s) ≠ ∞ und β(s) = ∞ then füge s zur Zeit α(s) in β ein;
7          T := erster in β rekursiv leerer Slot der Länge 2^k;
8          if ende(T) ≤ l · 2^k then
9              Füge alle r ∈ S_i in α zu dem Slot T hinzu;
10             Leere β;
11         else
12             X:=in β rekursiv leerer Slot der Länge 2^(k-1) mit start(X) ≥ l · 2^k;
13             Y:=in β rekursiv leerer Slot der Länge 2^(k-1) mit start(Y) ≥ ende(X);
14             X' := Nachbar von X;
15             Vertausche in α die Slots X' und Y;
16             T := Slot der Länge 2^k oberhalb von X;
17             Füge jedes r ∈ S_i in α zu dem Slot T hinzu;
18             Vertausche die freien Slots T und S_l in α;
19             l := l + 1;
20         Leere β;
21 end
```

- Dann können die beiden in α freien und in β rekursiv leeren Slots X und Y der Länge 2^{k-1} nebeneinander getauscht werden: Da X frei in α ist, ist nach Lemma 7.2.5 auch der Nachbar X' frei. Nach Definition und Satz 7.2.7 können daher X' und Y vertauscht werden. Sei $\alpha' := \alpha$ der Schedule vor der Vertauschung in Zeile 34. Nach Wahl von X und Y gibt es in α' keinen zu r in Konflikt stehendes Request in X und in Y. Somit gibt es nach der Vertauschung von Y und X' in α kein zu einem Requests aus S_i in Konflikt stehendes Request in X und X', d.h. X und X' sind rekursiv leer in $\alpha_{|N_C(S_i)}$. Dann wird T als der Slot der Länge 2^k unmittelbar oberhalb von X und X' gewählt. Da X in α frei ist, ist T leer in α. Somit ist T sogar rekursiv leer in $\alpha_{|N_C(S_i)}$ und wegen $\delta(T) = 2^k$ ist T frei. Nach Satz 7.6.3 kann jedes Request aus S_i in T geschedulet werden.

  Nach dem Schedulen der Requests ist T gefüllt. Anschließend werden die Slots T und T_l in α vertauscht, nach dem vertauschen ist somit T_l gefüllt. Die Slots $T_i, i < l$ waren nach Voraussetzung bereits gefüllt, die Slots T_i, $i > l$ hingegen leer. Somit sind nun genau alle T_i, $i \leq l$ gefüllt. Durch Definition von $l := l + 1$ ergibt sich die Eigenschaft (7.1)

Nach Beendigung der Schleife wurden daher alle Requests aus S gültig geschedulet, d.h. α wurde zu einem Schedule auf $R \cup S$.

Zur Laufzeit: Sei $b := \sum_{r \in S_i} d_C(r)$. Dann gilt

- Die ForEach-Schleife ab Zeile 34 hat eine Komplexität vom $\mathcal{O}(\sum_{r \in S_i} 1 + d_C(r))$ und anschließend sind im Schedule β maximal b Requests gespeichert.
- Die Suche des rekursiv leeren Slots T in Zeile 34 hat eine Komplexität vom $\mathcal{O}(b)$.
- Ist die Bedingung in Zeile 34 erfüllt, so wird r in konstanter Zeit geschedulet, in $\mathcal{O}(b)$ der Schedule β geleert und anschließend der Schleifendurchlauf beendet.
- Ist die Bedingung in Zeile 34 nicht erfüllt, so verläuft das Suchen der beiden ersten in β rekursiv leeren Slots der Länge 2^{k-1} in $\mathcal{O}(b)$.
- Die Slots können in konstanter Zeit vertauscht werden und T in konstanter Zeit festgelegt werden.
- Das Schedulen von r in T, dass vertauschen der Slots und das Inkrementieren von l in Zeile 34 bis 34 dauert konstante Zeit, dass anschließende leeren von β ist in $\mathcal{O}(b)$.

Insgesamt sind alle Operationen in $\mathcal{O}\left(\sum_{r \in S_i}(1 + d_C(r))\right)$, somit auch der Schleifendurchlauf für S_i. Da die Schleife für jedes $S_i \mathcal{S}$ durchlaufen wird folgt für den Algorithmus

$$\textsf{Bottom-Up} \in \sum_{S_i \in \mathcal{S}} \sum_{r \in S_i} \mathcal{O}(1 + d_C(r)) = \sum_{r \in S} \mathcal{O}(1 + d_C(r)) = \mathcal{O}\left(|S| + \sum_{r \in S} d_C(r)\right).$$

□

Bemerkung 7.6.10. Bestehen die Mengen S_i nur aus einem Element, so werden die Requests einzeln geschedulet. Dieser Algorithmus wird als Single-Bottom-Up bezeichnet.

7.6.2.3 Multidimensionaler Greedy

Somit ergibt sich der gesamte Algorithmus dadurch, dass erst ein guter Schedule für die Requests einer Länge erzeugt wird, dieser dann mittels der Bottom-Up-Algorithmen auf längere Requests ausgedehnt wird und anschließend mittels Top-Down-Algorithmen auf die kürzeren erweitert wird.

Algorithmus 35 : **Multidim-Greedy**
Konstruktion eines globalen gestaffelten Schedules

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
C: Ein globaler Knotenkonfliktgraph als Adjazenzliste.
Methode zum Erzeugen eines Schedules für einheitliche Requestlängen in $\mathcal{O}(g)$.
Parameter : k: Zahl, so dass zu Beginn ein Schedule für $\Gamma^{k=}$ erzeugt und anschließend erweitert wird.
Rückgabewert : Einen globalen GGSchedule.
Komplexität : $\mathcal{O}(g + |\Gamma| + \Theta)$.

```
1 Funktion Multidim-Greedy(k: Integer)
2 |  α: Schedule für Γ^{=k} als GGSchedule;
3 |  β: leerer GGSchedule;
4 |  for j = k+1,...,N do Bottom-Up(Γ^{=j}) ; // Erweiterung auf längere Requests
5 |  for j = k-1,...,0 do Top-Down(Γ^{=j}) ;  // Erweiterung auf kürzere Requests
6 |  return α.
7 end
```

Satz 7.6.11. *Die Erzeugung des Schedules für $\Gamma^{=k}$ erfolge in $\mathcal{O}(g)$. Dann erzeugt der in Algorithmus 35 dargestellte Algorithmus Multidim-Greedy einen globalen gestaffelten Schedule in einer Laufzeit von $\mathcal{O}(g + |\Gamma| + \Theta)$.*

Beweis.

- Zur Korrektheit: Offenbar ist nach Zeile 35 α ein Schedule für $\Gamma^{=k}$. Dieser wird in Zeile 35 auf einen Schedule für $\Gamma^{\geq k}$ und schließlich in Zeile 35 auf einen Schedule für Γ erweitert. Anschließend wird der globale Schedule α zurückgegeben.
- Zur Laufzeit: Die Erzeugung von α in Zeile 35 benötigt $\mathcal{O}(g)$, die Erzeugung von β erfolgt in $\mathcal{O}(|\Gamma|)$. Mit den zuvor bewiesenen Laufzeiten für die Bottom-Up-Erweiterung und für die Top-Down-Erweiterung folgt eine Gesamtkomplexität von

$$
\begin{aligned}
&\mathcal{O}(g) + \mathcal{O}(|\Gamma|) + \sum_{i=k+1}^{N} \mathcal{O}\left(|\Gamma^{=i}| + \sum_{r\in\Gamma^{=i}} d_C(r)\right) + \sum_{i=0}^{k-1} \mathcal{O}\left(|\Gamma^{=i}| + \sum_{r\in\Gamma^{=i}} d_C(r)\right) \\
&\subset \mathcal{O}\left(g + |\Gamma| + \sum_{i=0}^{N}\left(|\Gamma^{=i}| + \sum_{r\in\Gamma^{=i}} d_C(r)\right)\right) \\
&= \mathcal{O}\left(g + |\Gamma| + |\Gamma| + \sum_{r\in\Gamma} d_C(r)\right) \\
&= \mathcal{O}(g + |\Gamma| + 2|E_C|) \\
&= \mathcal{O}(g + |\Gamma| + \Theta).
\end{aligned}
$$

□

7.6.3 Reihenfolge der Requests einer Länge

Werden die Requests einzeln geschedulet, so sind die beschriebenen Algorithmen vergleichbar mit denen der sequentiellen Färbung (ggf. mit Umfärben). Auch dort ist die Reihenfolge, in der die Requests geschedulet werden, von besonderer Bedeutung (vgl. Abschnitt 3.3.1). Insbesondere die Degree-Saturation-Largest-First Reihenfolge bewährt sich in der Praxis. Diese lässt sich auf gestaffelte Schedules verallgemeinern.

Bei der Graphenfärbung ist der Sättigungsgrad eines Knotens v die Anzahl der in der Nachbarschaft von v verwendeten Farben. Es gibt zwei Möglichkeiten, den Begriff des Sättigungsgrads zu verallgemeinern.

Definition 7.6.12 (Sättigungszeit, Sättigungsgrad)**.** *Sei α ein partieller Schedule für R.*

- *Die Sättigungszeit $d_{st}(r)$ eines Requests $r \in R$ ist die Gesamtdauer der Zeit, zu der mit r in Konflikt stehende Requests ausgeführt werden. Dabei werden Zeiten, zu denen mehrere Requests ausgeführt werden, einfach gewertet. Formaler*

$$M := \bigcup_{\substack{r \rightsquigarrow s \\ \alpha(s) \neq \infty}} \widehat{\alpha}(s),$$

$$d_{st}(r) := \textit{Gesamtdauer von } M = \int_M 1 \mathrm{d}x.$$

- *Der Sättigungsgrad $d_{sd}(r)$ eines Requests $r \in R$ der Länge 2^n ist die Anzahl der Slots der Länge $\delta(r)$, die parallel zu einem mit r in Konflikt stehenden Request verlaufen, d.h. die Anzahl der in β nicht freien oder rekursiv gefüllten Slots. Formaler*

$$d_{sp}(r) := \left| \left\{ S : \delta(S) = 2^n \wedge \exists s \in \Gamma \textit{ mit } (r \rightsquigarrow s) \wedge (\alpha(s) \neq \infty) \wedge \left(\widehat{\alpha}(s) \cap \widehat{S} \neq \emptyset \right) \right\} \right|.$$

Ist ein partieller gestaffelter Schedule und der globale Konfliktgraph als Adjazenzliste gegeben, so lassen sich Sättigungszeit und Sättigungsgrad effizient bestimmen.

Bemerkung 7.6.13 (Berechnung des Sättigungsgrads und der Sättigungszeit)**.** Sei r ein Request der Länge 2^{n_r}. Ist β ein GGSchedule für $N_C(r)$, so lassen sich Sättigungszeit und Sättigungsgrad von r wie folgt bestimmen:

Zur Sättigungszeit:

- Setze die Variable *stime* auf Null.
- Traversiere alle existierenden Slots von β in Tiefensuche. Wird ein nicht-leerer Slot S erreicht, so erhöhe *stime* um $\delta(S)$. Alle innerhalb von S liegenden Slots tragen nichts mehr zur Sättigungszeit bei, daher wird die Traversierung für diesen Teilbaum abgebrochen.
- Zum Schluss enthält *stime* die Sättigungszeit von r.

Zum Sättigungsgrad:

- Setze die Variable *sdeg* auf Null.
- Traversiere alle existierenden Slots von β in Tiefensuche.
  - Wird ein existierender Slot der Länge $\delta(r)$ erreicht, so ist dieser nicht rekursiv leer. Daher erhöhe *sdeg* um 1 und beende die Traversierung für diesen Teilbaum.
  - Wird ein nicht-leerer Slot S erreicht, so enthält dieser $\frac{\delta(S)}{\delta(r)}$ nicht-freie Slots der Länge $\delta(r)$. Daher erhöhe *sdeg* um $\frac{\delta(S)}{\delta(r)}$ und beende die Traversierung für diesen Teilbaum.
- Zum Schluss enthält *sdeg* den Sättigungsgrad von r.

Es müssen nur die Slots maximaler Länge aus *used* traversiert werden. Da es maximal $d_C(r)$ solche Slots gibt und die Traversierung eines solchen Slots konstante Zeit benötigt, ergibt sich für die Laufzeit der obigen Berechnungen $\mathcal{O}(1 + d_C(r))$.

Soll beim Schedulen das Request mit maximalem Sättigungsgrad berechnet werden, so muss dieser für jedes Request bestimmt werden und es ergibt sich mit Erzeugung des Schedules für $N_C(r)$ in $\mathcal{O}(1 + d_C(r))$ und der Suche des Maximums in $\mathcal{O}(|\Gamma|)$ eine Komplexität von

$$\mathcal{O}(|\Gamma|) + \sum_{r \in \Gamma} \mathcal{O}(1 + d_C(r)) = \mathcal{O}(|\Gamma| + 2|E_C|) = \mathcal{O}(|\Gamma| + \Theta).$$

Da dieses für das Schedulen jedes Requests notwendig ist, ergibt sich eine Gesamtkomplexität von

$$\sum_{r \in \Gamma} \mathcal{O}(|\Gamma| + \Theta) = \mathcal{O}(|\Gamma|^2 + |\Gamma|\Theta) \subset \mathcal{O}(|\Gamma|^3).$$

Entsprechend ergibt sich die Komplexität, falls jeweils der Knoten mit maximaler Sättigungszeit geschedulet werden soll.

Zur Effizienzsteigerung sollen Datenstrukturen *Saturation-Time-Counter* und *Saturation-Degree-Counter* entwickelt werden, die das Auslesen der Sättigungszeit und des Sättigungsgrads eines Knotens in konstanter Zeit ermöglicht.

7.6.3.1 Saturation-Time-Counter

Die Struktur *Saturation-Time-Counter* ist ähnlich zu *GGSchedule*. In dieser sollen in konstanter Zeit sukzessive geschedulete Requests eingefügt werden können. Enthält die Struktur alle Requests aus $N_C(r)$, so soll in konstanter Zeit die Sättigungszeit ausgelesen werden können. Insbesondere soll die Datenstruktur das Umschedulen freier Blöcke ermöglichen.

Die Datenstruktur enthält ein Array *Slots* mit Feldern $0, \ldots, |\Gamma| - 1$. Dabei ist in jedem Feld ein Binärbaum gespeichert. Jeder Knoten dieses Baums repräsentiert einen Slot S und enthält Verweise $lc(S)$ bzw. $rc(S)$ auf die beiden Nachkommen und einen booleschen Wert *full*, der genau dann wahr ist, wenn der repräsentierte Slot gefüllt ist. Des Weiteren gibt es eine Variable *stime*, die die Sättigungszeit speichert und zu Beginn Null ist.

Das Vertauschen freier Slots S und T lässt sich nun analog zu *GGSchedule* in konstanter Zeit implementieren. Dazu sind die Zeiger der Väter von S und T zu vertauschenden. Eventuell sind dabei neue Slots anzulegen oder aber rekursiv leer gewordene Slots zu löschen, falls Slots

nicht vorhanden waren. Auf Grund der beschränkten Baumhöhe ist dieses in konstanter Zeit möglich, die Sättigungszeit ändert sich dabei nicht.

Das Einfügen eines neuen Slots ist in konstanter Zeit möglich und in Algorithmus 36 dargestellt.

Algorithmus 36 : **STC-Insert**
Einfügen eines Slots in *Saturation-Time-Counter*

Daten : *STC*: Die beschriebene Datenstruktur *Saturation-Time-Counter*.
Globale Variablen : *stime*: Eine in *STC* gespeicherte Variable für die Sättigungszeit.
Parameter : S: Der zu belegende Slot.
Komplexität : $\mathcal{O}(1)$.

```
1  Funktion STC-Insert(S: Slot)
2      Sei S := Sl⟨s_N, ..., s_n⟩;
3      for k := N, ..., n do          // Durchlaufe aller Slots von der Wurzel zu S
4          S_k := Sl⟨s_n, ..., s_k⟩;
5          if S_k existiert nicht then lege Slot S_k an;
6          else if S_k ist nicht leer then return ;
       // Der Slot S existiert nun und ist leer
7      S := existierende Slots des von S aufgespannten Teilbaums;
8      foreach T ∈ S do
9          if T ist nicht leer then stime := stime − δ(T);
10         Entferne T aus der Struktur STC;
11     Markiere S als gefüllt;
12     stime := stime + δ(S);
13 end
```

Satz 7.6.14. *Sei r ein Request. Wurde in die Datenstruktur* Saturation-Time-Counter *für alle gescheduleten Requests $s \in N_C(r)$ der s enthaltende Slot eingefügt, so ist $stime = d_{st}(r)$.*

Die Struktur kann in $\mathcal{O}(|\Gamma|)$ erzeugt werden und jede Operation benötigt konstante Laufzeit.

Beweis. Wir beweisen über Induktion:

- Bevor das erste Request $s \in N_C(r)$ geschedulet wurde, ist $stime = 0 = d_{st}(r)$.
- Sei die Bedingung bisher erfüllt und nun werde das Request s in den Slot S geschedulet.
  - Fall: Der Slot S ist nicht frei:
    Dann ist S in einem nicht-leeren Slot S' enthalten, d.h. es ist $\widehat{S} \subset \widehat{S'}$. Daher gibt es keinen Zeitpunkt, zu dem vor dem Schedulen von s kein Request aus $N_C(r)$ aktiv war, nach dem Schedulen von s jedoch schon. Somit bleibt d_{st} unverändert. Entsprechend wird in Zeile 36 für S' die Funktion beendet, ohne das *stime* verändert wurde.
  - Fall: Der Slot S ist nicht leer:
    Analog zum vorherigen Fall ändert sich d_{st} nicht, und in Zeile 36 wird für S die Funktion beendet, ohne dass *stime* verändert wurde.

– Fall: Der Slot S ist frei und leer:
In diesem Fall verändert sich die Sättigungszeit. Nach Schedulen von s ist während der gesamten Zeit $\widehat{S}$ ein zu r in Konflikt stehendes Request aktiv. Sei t die Dauer, zu der vor dem Schedulen ein innerhalb von S liegendes zu r in Konflikt stehendes Request aktiv war. Dann wird d_{st} zu $d_{st} + \delta(S) - t$. Diese Dauer t entspricht der Dauer aller nicht-leerer Slots innerhalb von S. Daher wird in Zeile 36 gerade $stime - t$ berechnet und in Zeile 36 wird $stime$ zu $stime - t + \delta(S)$.

Umschedulen von Blöcken verändert weder $stime$ noch d_{st}, daher bleibt bei diesen Operationen $stime = d_{st}(r)$.

Zur Laufzeit: Bei der Erzeugung muss die Variable $stime$ und ein leeres Array der Länge $|\Gamma|$ angelegt werden. Die konstante Laufzeit für die Operationen folgt daher, dass N eine Konstante ist und daher die Zahl der Schleifendurchläufe der einzelnen Schleifen beschränkt ist. □

7.6.3.2 Saturation-Degree-Counter

Die Struktur des *Saturation-Degree-Counter* ist sehr ähnlich zum *Saturation-Time-Counter*. Anstelle der Variablen $stime$ gibt es eine Variable $sdeg$ für den Sättigungsgrad. Außerdem muss sowohl das Einfügen eines Slots als auch das Vertauschen zweier freier Slots leicht abgewandelt werden. Diese Operationen sind prinzipiell nicht schwierig, jedoch etwas technisch. Daher sollen sie hier nur skizziert werden.

Der Sättigungsgrad $d_{sd}(r)$ eines Requests entspricht der Anzahl der rekursiv gefüllten oder nicht freien Slots der Länge $\delta(r)$.

Wird nun ein Request in einen Slot S eingefügt, so können folgende Fälle auftreten:

- Der Slot S liegt innerhalb eines gefüllten Slots, dann ist nichts zu verändern.
- Ansonsten wird der Slot S eingefügt.
  – Fall $\delta(S) > \delta(r)$: Es gibt $k = \frac{\delta(S)}{\delta(r)}$ Slots der Länge $\delta(r)$ in S.

    Durch Traversierung lässt sich die Anzahl l der rekursiv gefüllten oder nicht-freien Slots der Länge $\delta(r)$ innerhalb von S wie folgt bestimmen: Ein Slot der Länge $\delta(r)$ ist genau dann rekursiv gefüllt, wenn er existiert. Des Weiteren enthält jeder gefüllte Slot T gerade $\frac{\delta(T)}{\delta(r)}$ nicht-freie Slots der Länge $\delta(r)$.

    Durch das Füllen von S erhöht sich $sdeg$ um $k-l$, d.h. definiere $sdeg := sdeg+k-l$. Markiere den Slot S als gefüllt (lege ggf. die dafür nötigen Slots an) und entferne anschließend alle Slots des von S aufgespannten Teilbaums.

  – Fall $\delta(S) \leq \delta(r)$: Sei T der Slot der Länge $\delta(r)$, der S enthält. Markiere den Slot S als gefüllt (lege ggf. die dafür nötigen Slots an) und entferne alle Requests des von S aufgespannten Teilbaums. Existierte T vor dem Einfügen nicht – d.h. war T bisher rekursiv leer – so setze $sdeg := sdeg + 1$.

(a) Vor der Umordnung ist $\delta_C(r) = 2$.

(b) Nach Umordnen der Slots $Sl\langle 0, 1\rangle$ und $Sl\langle 1, 0\rangle$ ist $\delta_C(r) = 1$.

Abbildung 7.6: Durch Umordnen freier Slots kann sich der Sättigungsgrad verändern: Zwei Schedules für $N_C(r)$ eines Request r der Länge 2.

Sollen zwei freie Slots S und T vertauscht werden, so sind prinzipiell nur die Zeiger der Elternelemente (falls existent) umzudefinieren und ggf. neue Slots anzulegen oder rekursiv leer gewordene Slots zu löschen. Jedoch kann sich – falls $\delta(S) = \delta(T) < \delta(r)$ – der Sättigungsgrad verändern (siehe Abbildung 7.6). Falls $\delta(S) = \delta(T) < \delta(r)$ ist, werden die folgenden Schritte ausgeführt.

1. Wähle Slots S' und T' der Länge $\delta(r)$, die S bzw. T enthalten.
2. Prüfe, ob das Entfernen von S oder T den Sättigungsgrad verändert, d.h. wenn S rekursiv gefüllt ist und S' frei ist und keinen weiteren rekursiv gefüllten Slot enthält, so setze $sdeg := sdeg - 1$. Analog für T.
3. Prüfe, ob das Hinzufügen an die neue Position den Sättigungsgrad verändert, d.h. wenn S nach der Umordnung rekursiv gefüllt ist und S' frei ist und keinen weiteren rekursiv gefüllten Slot enthält, so setze $sdeg := sdeg + 1$. Analog für T.
4. Vertausche die Slots, d.h. ändere – falls existent – die Zeiger der Elternelemente. Lösche ggf. rekursiv leer gewordene Slots.

Auch diese Operationen sind alle in konstanter Zeit möglich. Daher ergibt sich für die einzelnen Operationen *Saturation-Degree-Counter* dieselbe Komplexität wie bei *Saturation-Time-Counter*.

7.6.3.3 Die Algorithmen Multidim-DSATUR und Multidim-TSATUR

In Anlehnung an den DSATUR-Algorithmus gibt es für das Schedulen der Requests gleicher Dauer in Multidim-Greedy nun die folgenden möglichen Heuristiken:

Saturation-Degree-Largest-First: In jedem Schritt wird das noch nicht geschedulete Request mit maximalem Sättigungsgrad gewählt. Dieser Algorithmus heißt Multidim-DSATUR.

Saturation-Time-Largest-First: In jedem Schritt wird das noch nicht geschedulete Request mit maximaler Sättigungszeit gewählt. Dieser Algorithmus heißt Multidim-TSATUR.

Dabei wird ein Array *stc* bzw, *sdc* der Länge $|\Gamma|$ angelegt, welches für jedes Request einen *Saturation-Time-Counter* bzw. *Saturation-Degree-Counter* enthält. Anschließend wird analog zu Multidim-Greedy ein optimaler Schedule für die Requests der Länge 2^k erstellt. Dieser wird anschließend mittels Bottom-Up-Single bzw. Top-Down-Single auf die längeren bzw. kürzeren Requests ausgeweitet. Jedes geschedulete Request wird in die Saturation-Counter in Konflikt stehenden Requests eingefügt und es wird unter gleichlangen Requests jeweils das Request

mit maximaler Sättigungszeit bzw. maximalem Sättigungsgrad gewählt. Dabei ergeben sich folgende Modifikationen der Laufzeit:

- Zu Beginn muss ein Array der Länge $|\Gamma|$ angelegt werden, wobei für jedes Feld eine Laufzeit von $\mathcal{O}(|\Gamma|)$ zur Initialisierung benötigt wird. Insgesamt benötigt dieser Schritt eine Laufzeit von $\mathcal{O}(|\Gamma|^2)$.
- Die Wahl des Requests mit maximalem Sättigungsgrad bzw. maximaler Sättigungszeit hat eine Komplexität von $\mathcal{O}(|\Gamma|)$, da aus einer Menge mit maximal $|\Gamma|$ Requests das Maximum gesucht wird. Dieses muss für jedes Request nur einmal geschehen, daher benötigt dieser Schritt insgesamt eine Laufzeit von $\mathcal{O}(|\Gamma|^2)$.
- Für jedes geschedulete Request $r \in \Gamma$ müssen die Strukturen der $d_C(r)$ Nachbarrequests in konstanter Zeit aktualisiert werden. Dieses hat insgesamt eine Komplexität von

$$\mathcal{O}\left(\sum_{r\in|\Gamma|} d_C(r)\right) = \mathcal{O}(2|E_C|) = \mathcal{O}(\Theta).$$

- Beim Umschedulen zweier Slots müssen $|\Gamma|$ Strukturen in konstanter Zeit modifiziert werden. Dieses ist maximal $|\Gamma|$ mal erforderlich, daher hat dieses insgesamt eine Komplexität von $\mathcal{O}(|\Gamma|^2)$

Somit ergibt sich für alle zusätzlichen Schritte insgesamt eine Laufzeit von

$$\mathcal{O}(|\Gamma|^2) + \mathcal{O}(|\Gamma|^2) + \mathcal{O}(\Theta) + \mathcal{O}(|\Gamma|^2) = \mathcal{O}(|\Gamma|^2).$$

Die Komplexität von Multidim-DSATUR und Multidim-TSATUR entsprechen nun der Komplexität $\mathcal{O}(g+|\Gamma|+\Theta)$ von Multidim-Greedy und der durch die Wahl der Reihenfolge zusätzlichen entstandenen Komplexität $Order(|\Gamma|^2)$, d.h. es ist

$$\text{Multidim-DSATUR, Multidim-TSATUR} \in \mathcal{O}(|\Gamma|^2) + \mathcal{O}(g + |\Gamma| + \Theta) = \mathcal{O}(g + |\Gamma|^2),$$

wobei $\mathcal{O}(g)$ die Komplexität der Scheduleerstellung für $\Gamma^{=k}$ ist.

Falls einer der folgenden Fälle eintritt, so ist $g \in \mathcal{O}(|\Gamma|^2)$.

- Der Schedule für $\Gamma^{=k}$ wird mittels einer Greedy-Färbung, z.B. DSATUR, für den globalen Konfliktgraphen für $\Gamma^{=k}$ in $\mathcal{O}(|\Gamma^{=k}|^2) \subset \mathcal{O}(|\Gamma|^2)$ erstellt.
- Es ist $l(\mathcal{G}) \leq |\Gamma|$ und der Schedule für $\Gamma^{=k}$ wird für Nicht-VD-MC-Netzwerke mit einem der in Abschnitt 6.3.2 beschriebenen Algorithmen GlobalNatSchedule-HD oder GlobalNatSchedule-VD-UC-Prec in $\mathcal{O}(|\Gamma^{=k}| \cdot l(\mathcal{G})) \subset \mathcal{O}(|\Gamma|^2)$ erstellt.

In diesen Fällen ergibt sich unmittelbar

$$\text{Multidim-DSATUR, Multidim-TSATUR} \in \mathcal{O}(|\Gamma|^2).$$

Es ist zu erwarten, dass die Saturation-Degree-First-Methode bessere Schedules erzeugt.

Kapitel 8

Fazit

8.1 Ergebnisse

In dieser Arbeit wurden verschiedene Ansätze und Möglichkeiten zum Schedulen von Requests in baumförmigen Netzwerken analysiert und entwickelt. Die wesentlichen Ergebnisse sind im Folgenden dargestellt.

Dabei beachte man, dass Algorithmen für beliebige Requestdauer auch auf die anderen Fälle übertragbar sind. Jedoch wird dann nicht zwangsläufig ein natürlicher bzw. gestaffelter Schedule erstellt. Entsprechend sind die Ergebnisse für gestaffelte Requestdauer auf natürliche Schedules übertragbar, bringen dort jedoch keine neuen Erkenntnisse.

Einheitlicher Requestdauer

Für Requests mit einheitlicher Dauer wird die Erstellung des Schedules auf das Färben von Konfliktgraphen zurückgeführt.

- In Halbduplex-Netzwerken sind je zwei Schedules für einen Knoten synchronisierbar. Die Schedules werden dann mit dem Algorithmus GlobalNatSchedule-HD erzeugt. Daraus folgt:
  - In Unicast-Netzwerken kann in $\mathcal{O}(|\Gamma|^2 \cdot l(\mathcal{G}))$ ein Schedule mit maximal $\frac{4}{3}$-facher Länge des optimalen Schedules erstellt werden (vgl. Korollar 6.3.8).
  - Ist der Knotengrad des Netzwerks beschränkt, so kann – zumindest theoretisch – in $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}))$ ein optimaler Schedule erstellt werden. Jedoch ist der Algorithmus auf Grund einer großen multiplikativen Konstanten in der Laufzeit praktisch nicht verwendbar (vgl. Korollar 6.3.9).
- In Vollduplex-Netzwerken für Unicast-Requests sind die Schedules für einzelne Knoten durch Färbung des bipartiten Kantenkonfliktgraphen leicht zu konstruieren. Diese sind jedoch nicht notwendig synchronisierbar. Daher gibt es verschiedene Lösungen, sequentiell für jeden Knoten den Schedule zu erweitern:
  - Unter Beibehaltung der bisherigen Farben können bei Traversierung des Baums für jeden Knoten die ungescheduleten Requests dieses Knotens geschedulet werden. Dieses Algorithmus GlobalNatSchedule-VD-UC-Prec hat bei der Verwendung von Kantenkonfliktgraphen und einem $\mathcal{O}(|V| + |E|)$-Färbungsalgorithmus eine Komplexität von $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}))$ (vgl. Satz 6.3.11).

  - Für jeden Knoten können anhand des Konfliktgraphen bezüglich des bisher erstellten Schedules α (Definition 6.3.13) zu α synchronisierbare Schedules erzeugt werden. Diese können dann zu α synchronisiert und synchron vereinigt werden. Dieser Algorithmus GlobalNatSchedule-VD-UC-Classes hat bei Verwendung einer $\mathcal{O}(|V|+|E|)$-Färbung eine Komplexität von $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma} + \sum_{v \in \mathbb{V}} h(v))$ (vgl. Satz 6.3.19).

- In beliebigen Netzwerken lässt sich mit GlobalNatSchedule-BC immer ein Schedule durch Färben des globalen Konfliktgraphen in $\mathcal{O}(|\Gamma|^2 + \Theta^{\Sigma}) + \mathcal{O}(color)$ erzeugen. Durch Ergebnisse über Knotenfärbung folgen die in Korollar 6.3.4 dargestellten Ergebnisse: Es gibt einen Algorithmus zur Erzeugung eines optimalen Schedules mit einer Laufzeit von

$$\mathcal{O}\left(\left(\frac{4}{3} + \frac{3^{\frac{4}{3}}}{4}\right)^{|\Gamma|}\right) \approx \mathcal{O}(2,4150^{|\Gamma|})$$

  und einen polynomiellen Algorithmus zur Erstellung eines Schedules mit einer Approximationsgüte $\kappa \in \mathcal{O}\left(\frac{|\Gamma|(\log\log|\Gamma|)^2}{(\log|\Gamma|)^3}\right)$.

Gestaffelte Requestdauer

Haben die Requests exponentiell gestaffelte Übertragungslängen, so werden die Requests in Slots gespeichert, so dass jedes Request als Startzeit ein ganzzahliges Vielfaches seiner Dauer zugewiesen bekommt.

- Falls das Netzwerk kein VD-MC-Netzwerk ist, so können sequentiell für jeden Knoten alle Requests des Knotens mit Exp-First-Fit-Nodes zum erstmöglichen Zeitpunkt geschedulet werden. Dieser Algorithmus hat eine Komplexität von $\mathcal{O}(|\Gamma| \cdot l(\mathcal{G}) + \Theta^{\Sigma})$ (vgl. Satz 7.5.1).

Existiert ein in $\mathcal{O}(|\Gamma|^2 \cdot l(\mathcal{G}))$ erstellbarer globaler Konfliktgraph, so gibt es für beliebige Netzwerke folgende Möglichkeiten:

- Werden (in beliebiger Reihenfolge) mit Exp-First-Fit die Requests nacheinander auf den erstmöglichen Zeitpunkt geschedulet, so wird in $\mathcal{O}(|\Gamma| + \Theta)$ ein globaler Schedule erstellt. Ohne Vergrößerung der Komplexität können die Requests in Decreasing-First-Fit absteigend nach der Länge sortiert werden (vgl. Abschnitt 7.6.1).

- Der Algorithmus Multidim-Greedy beginnt mit einem Schedule für die Requests einer Länge und erweitert diesen Anschließend auf längere und kürzere Requests. Dabei wird bei der Erweiterung auf längere Requests versucht, bestehende Requests umzuschedulen. Dieses Verfahren hat – falls der initiale Schedule in einer Dauer von $\mathcal{O}(g)$ erstellt wird – eine Komplexität von $\mathcal{O}(g + |\Gamma| + \Theta)$ (vgl. Satz 7.6.11).

  Dabei gibt bei der Erweiterung auf Requests einer Länge 2^j die Möglichkeit, dass die Requests der Länge 2^j einzeln geschedulet werden oder dass ein Schedule für die Requests erstellt wird und die Klassen dieses Schedules geschedulet werden.

- In Anlehnung an den DSATUR-Algorithmus zur Knotenfärbung können im Algorithmus Multidim-Greedy die Requests einer Länge nach absteigendem Sättigungsgrad oder absteigender Sättigungszeit sortiert werden. Die so entstehenden Algorithmen heißen Multidim-DSATUR bzw. Multidim-TSATUR und haben eine Komplexität von $\mathcal{O}(g+|\Gamma|^2)$.

Wird der Initiale Schedule für die Requests einer Länge in $\mathcal{O}(g) \subset \mathcal{O}(|\Gamma|^2)$ erstellt, so ist die Komplexität $\mathcal{O}(|\Gamma|^2)$ (vgl. Abschnitt 7.6.3.3).

Beliebige Requestdauer
Haben die Requests beliebige Übertragungslängen, so sind Ansätze wie Synchronisation oder Umschedulen von Requests nicht mehr möglich. Im Wesentlichen werden die Requests sequentiell geschedulet. Basierend auf einem in $\mathcal{O}(|\Gamma|^2 \cdot l(\mathcal{G}))$ erstellbaren globalen Konfliktgraphen gibt es folgende Verfahren:

- Mit First-Fit wird in $\mathcal{O}(|\Gamma|^2)$ jedes Request zur erstmöglichen Zeit geschedulet. Dabei können die Requests ohne Vergrößerung der Komplexität in Decreasing-First-Fit absteigend nach ihrer Dauer sortiert werden (vgl. Abschnitt 5.1).
- Der Algorithmus List-Scheduling bestimmt ebenfalls in $\mathcal{O}(|\Gamma|^2)$ einen globalen Schedule (vgl. Abschnitt 5.2).
- Der Algorithmus List-Scheduling-Levels zerlegt die Requests in disjunkte Mengen, deren Scheduling äquivalent zur Erstellung des Schedules für einen Stern ist. Aus der Approximationsgüte 2 von List-Scheduling für Sterne lässt sich folgern, dass die Approximationsgüte des Algorithmus $\kappa = 2\lfloor \log_2 |\mathbb{V}| \rfloor$ ist. Falls $|\Gamma| \geq |\mathbb{V}|$ ist, so hat der Algorithmus eine Komplexität von $\mathcal{O}(|\Gamma|^2)$ (vgl. Abschnitt 5.3).

8.2 Ausblick

Gerade in der diskreten Optimierung sind empirische Untersuchungen oft ebenso wichtig wie theoretische Ergebnisse. Daher sind die beschriebenen und bewiesenen Algorithmen noch zu implementieren und ihre Laufzeit und Güte zu analysieren. Dieses ist gerade bei den Algorithmen wichtig, deren Approximationsgüte nicht bekannt ist. Insbesondere existiert für die Algorithmen zu gestaffelten Requestlängen keine theoretische Aussage über die Approximationsgüte.

A Beweise ausgewählter Algorithmen

A.1 Der Algorithmus Sync-Schedules

Algorithmus : Konkretere Implementierung des Algorithmus Sync-Schedules zur Synchronisation zweier synchronisierbarer natürlicher Schedules (vgl. Algorithmus 25)

Daten : $\mathcal{N}$: Die Struktur des Netzwerks.
R,S: Konfliktfreie Mengen.
Parameter : α: Ein Schedule für R als partieller globaler Schedule gespeichert.
β: Ein zu α synchronisierbarer Schedule $\beta : S \mapsto \underline{m}$ bezüglich der lokalen Indizierung φ (mit $\beta[i]$ ist der Wert bzgl. des lokalen Index i gemeint).
φ: Eine lokale Indizierung der Menge S.
Rückgabewert : Eine zu α synchrone Umindizierung γ von β mit $|\gamma| \leq \max(|\alpha|, |\beta|)$.
Komplexität : $\mathcal{O}(\max(|S|, m))$.

```
1  Funktion Sync-Schedules(α: Schedule, β: Schedule, φ: Indizierung von S)
2      m := |β|;
3      ϱ : m ↦ ℕ0 ∪ {∞} ;                    // Als Array [0,...,m−1] of Integer
4      for j := 0,...,m−1 do ϱ(j) := ∞ ;              // Abbildung zu Beginn leer
5      u : Array [0,...,m−1] of Boolean ;        // Merke bereits getroffene Werte
6      for j := 0,...,m−1 do u(j) := false ;          // Noch kein Wert getroffen
       // Definiere ϱ so, dass ϱ ∘ β_{|R∩S} = α_{|R∩S}
7      foreach k = 0,...,|S|−1 do
           // Wurde r = φ(k) in α bereits geschedulet, dann ϱ(β(r)) := α(r)
8          if α(φ(k)) ≠ ∞ then
9              i := α(φ(k));
10             ϱ(β[k]) := i;
11             if i < m then u[i] := true ;     // Markiere Wert ϱ(β[k]) als getroffen
       // Setze die Funktion ϱ injektiv auf m fort
12     i := 0;
13     for j := 0,...,m−1 do
           // Ist ϱ(j) undefiniert, so definiere es als ϱ(j) := min(ℕ0 \ ϱ(m))
14         if ϱ(j) = ∞ then
15             while u[i] do i := i + 1 ;    // Finde nächsten nicht getroffenen Wert
16             ϱ(j) := i;
17             if i < m then u[i] := true ;      // Markiere Wert ϱ(j) als getroffen
       // Definiere zu α synchrone Umindizierung γ := ϱ ∘ β
18     γ : m ↦ ℕ0 ∪ {∞};
19     for k := 0,...,|S|−1 do γ[k] := ϱ(β[k]);
20     return γ;
21 end
```

Das Symbol ∞ lässt sich in der konkreten Implementierung durch einen nicht angenommenen Wert (z.B. -1) realisieren. Um Probleme auf Grund der unterschiedlichen Indizierung der Requests in α und β zu vermeiden, seien alle Requests r immer bezüglich der globalen Indizierung gemeint. Soll der Wert $\beta(r)$ für einen lokalen Index k des Requests r ausgewertet werden, so schreiben wir $\beta[k]$ (in Anlehnung an die Speicherung in einem Array). Somit ist

$$\forall k \in \underline{|S|}: \qquad \beta[k] = \beta(\varphi(k)).$$

Die entsprechende Schreibweise verwenden wir für γ.

Weiter seien zur Vereinfachung der Notation $\Lambda = R \cap S$ und $U := \{i : u[i] = true\}$ sowie $K := \{i \in \underline{m} : \varrho(i) \neq \infty\}$ die Elemente, für die ϱ bereits definiert wurde.

Die ForEach-Schleife ab Zeile 7 sei mit Schleife A bezeichnet, die For-Schleife ab Zeile 13 mit Schleife B.

Wir zeigen nun die Aussagen

(i) Nach Beendigung von Schleife A gilt für alle $r \in \Lambda$: $\varrho \circ \beta(r) = \alpha(r)$.

(ii) Nach Beendigung von Schleife B wurde ϱ injektiv auf $\underline{m}$ fortgesetzt.

(iii) Am Ende des Algorithmus ist γ eine zu α synchronisierbare Umindizierung von β mit $|\gamma| \leq \max(m, n)$.

Vor dem Aufruf der Funktion gilt nach Voraussetzung:

- $\alpha_{|R}$ ist ein Schedule für R und $\alpha(r) = \infty$ für $r \notin R$.
- β ist ein zu α synchronisierbarer Schedule für S bezüglich der lokalen Indizierung φ.

Dann gilt nach Zeile 6 offenbar

$$\begin{aligned} &m = |\beta|, \\ &\forall j =\in \underline{m} : \varrho(j) = \infty, \\ &U = \emptyset. \end{aligned}$$

Zu Aussage (i)

Behauptung 1: Das Innere des If-Blocks von Schleife A wird genau für alle $\varphi(k) \in \Lambda$ ausgeführt.

Begründung: Die Schleife wird für alle $k \in \underline{|S|}$ durchlaufen, d.h. $\varphi(k)$ durchläuft alle Werte aus S. Die Bedingung in Zeile 8 ist für alle k mit $\varphi(k) \in R$ erfüllt. Das Innere des If-Blocks ab Zeile 8 wird somit genau für alle $\varphi(k) \in S$ mit $\varphi(k) \in R$ durchlaufen, also für alle $\varphi(k) \in R \cap S = \Lambda$.

Behauptung 2: Ein einmal definierter Wert $\varrho(j)$ wird in Schleife A nicht mehr verändert.

Begründung: In Schleife A gilt: Die Funktion ϱ wird nur in Zeile 10 verändert. Wird die Schleife für ein $r_k = \varphi(k)$ mit $\varrho(\beta(r_k)) \neq \infty$ ausgeführt, so wurde sie bereits für ein $r_l = \varphi(l)$ mit $\beta(r_l) = \beta(r_k)$ ausgeführt. Dort wurde $\varrho(\beta[l])) = \alpha(r_l)$ definiert. Aus Satz 6.1.7 folgt

$$\varrho(\beta[l])) = \alpha(r_l) = \alpha(r_k) = \varrho(\beta[k])).$$

Somit bleibt ein bereits definierter Wert unverändert.

Behauptung 3: Es gilt Aussage (i).

Begründung: Nach Behauptung 1 wird in Schleife A genau für jedes $r_k = \varphi(k) \in \Lambda$ der Wert $\varrho(\beta[k]) = \alpha(r_k)$ definiert und nach Behauptung 2 anschließend nicht mehr verändert, d.h. es gilt

$$\forall r_k \in \Lambda : \qquad \varrho \circ \beta(r_k) = \varrho(\beta[k]) = \alpha(r_k)$$

und somit Aussage (i).

Zu Aussage (ii)

Behauptung 4: Außer unmittelbar vor Zeile 11 und Zeile 17 gilt

$$U = \varrho(\underline{m}) \cap \underline{m}.$$

Begründung: Unmittelbar nach der Initialisierung ist

$$\varrho(\underline{m}) \cap \underline{m} = \{\infty\} \cap \underline{m} = \emptyset = U.$$

Die Funktion ϱ wird ausschließlich in Zeile 10 und Zeile 16 in der Art verändert, dass $\varrho(j) = i$ für ein i definiert wird. Unmittelbar in den darauf folgenden Zeilen wird dann $u[i] := \mathit{true}$ gesetzt, falls $i < m$ bzw. $i \in \underline{m}$ ist. Da bestehende Werte $\varrho(j)$ nach Behauptung 2 in Schleife A und offenbar auch in Schleife B nicht mehr verändert werden, genügt dieses.

Behauptung 5: In Schleife B gilt jederzeit $i \in \underline{m}$ sowie $\underline{i} \subset U$. Nach Zeile 15 ist $i \notin U$

Begründung: Vor dem ersten Schleifendurchlauf ist $i = 0 \in \underline{m}$ und $\underline{i} = \underline{0} = \emptyset \subset U$. Sei nun die Aussage vor einem Schleifendurchlauf für j erfüllt. Nun sei i' der Wert von i zu Beginn der Schleife.

Dann gibt es ein $t \in \{i', \ldots, m-1\}$ mit $u[t] = \mathit{false}$. Denn ansonsten wäre

$$\underline{i'} \subset U \wedge \{i', \ldots, m-1\} \subset U \implies \underline{m} = \underline{i'} \cup \{i', \ldots, m-1\} \subset U \implies m \leq |U|$$

Bisher wurde ϱ weder in Schleife A noch Schleife B für Werte j definiert. Somit ist $\varrho(j) = \infty$ und

$$\begin{aligned} & U = \varrho(\underline{m}) \cap \underline{m} \subset \varrho\{\underline{m} \setminus \{j\}\} \\ \implies & |U| \leq |\varrho(\underline{m} \setminus \{j\})| \leq |\underline{m} \setminus \{j\}| = m - 1 < m \leq |U|, \end{aligned}$$

Widerspruch.

Somit existiert ein $t \in \{i', \ldots, m-1\}$ mit $u[t] = \mathit{false}$. Die While-Schleife in Zeile 15 terminiert, wenn i den ersten solchen Wert t annimmt. Dann ist $i \in \underline{m}$, $i \notin U$ und es gilt für alle t' mit $i' \leq t' < i$ ist $u[t'] = \mathit{true}$. Daher gilt $\{i', \ldots, i-1\} \subset U$. Nach Voraussetzung ist jedoch $\underline{i'} \subset U$. Somit gilt $\underline{i} = \underline{i'} \cup \{i', \ldots, i-1\} \subset U$.

Behauptung 6: Die Funktion ϱ ist nach jedem Durchlauf von Schleife B injektiv.

Begründung: Vor dem ersten Durchlauf ist dieses nach Aussage (i) und wegen

$$\varrho \circ \beta(r) = \varrho \circ \beta(s) \implies \alpha(r) = \alpha(s) \implies \beta(r) = \beta(s)$$

erfüllt.

Sei die Aussage nun vor einem Schleifendurchlauf erfüllt und die Schleife B werde für j durchlaufen. Dann wird ϱ nur in Zeile 16 verändert, dort wird $\varrho(j) := i$ gesetzt. Nach Behauptung 5 ist $i \in \underline{m}$ und $i \notin U$, nach Behauptung 4 ist $U = \varrho(\underline{m}) \cap \underline{m}$. Somit gilt

$$\begin{aligned} & i \in \underline{m} \wedge i \notin U \\ \implies & i \in \underline{m} \wedge i \notin \varrho(\underline{m}) \cap \underline{m} \\ \implies & i \in \underline{m} \setminus (\varrho(\underline{m}) \cap \underline{m}) = \underline{m} \setminus \varrho(\underline{m}) \\ \implies & i \notin \varrho(\underline{m}). \end{aligned}$$

Sei nun $K' = \varrho(\underline{m})$ vor Zeile 16. Nach Setzen von $\varrho(j) := i$ ist dann $\varrho(\underline{m}) = K' \cup \{j\}$. Dann gilt für $t \neq u \in \varrho(\underline{m})$: Ist $t \neq j$ und $u \neq j$ so ist $\varrho(t) \neq \varrho(u)$ wegen der Injektivität von ϱ auf K' nach Voraussetzung. Ansonsten sei o.B.d.A. $u = j$ und es gilt $\varrho(t) \neq \varrho(j) = i$ wegen $i \notin \varrho(\underline{m})$.

Behauptung 7: Es gilt Aussage (ii) und nach Ende von Schleife B $\varrho(\underline{m}) \in \underline{\max(m,n)}$.

Begründung: Nach Behauptung 6 ist ϱ nach jedem Durchlauf von Schleife B injektiv, also auch nach dem letzten Durchlauf. Somit gilt (ii).

Die Werte von ϱ wurden nur in Zeile 10 und Zeile 16 definiert. In Zeile 10 wird $\varrho(\beta[k]) := \alpha(\varphi(k)) \in \underline{\max(m,n)}$ gesetzt, in Zeile 16 wird $\varrho(j) := i \in \underline{m} \subset \underline{\max(m,n)}$ definiert.

Umindizierung des Schedules

Behauptung 8: Es gilt Aussage (iii).

Begründung: Nach Aussage (i) ist $\forall r \in \Lambda$: $\varrho \circ \beta(r) = \alpha(r)$, nach Aussage (ii) ist ϱ injektiv. Nach der Schleife in Zeile 19 gilt

$$\begin{aligned} & \forall k \in \underline{|S|} : \gamma[k] = \varrho(\beta[k]) \\ \implies & \forall r_k = \varphi(k) \in S : \gamma(k) = \varrho(\beta(r_k)). \end{aligned}$$

Somit ist γ die Umindizierung von β mit ϱ bezüglich der lokalen Indizierung φ. Wegen Behauptung 7 ist $|\gamma| \leq \max(m,n)$.

Nun ist γ nach Satz 4.3.6 synchron zu α, denn für alle $r \in \Lambda$ gilt

$$\gamma(r) = \varrho \circ \beta(r) = \alpha(r).$$

Laufzeitkomplexität
Der Algorithmus Sync-Schedules hat eine Komplexität von $\mathcal{O}(\max(|S|, m))$.

Zur Begründung überlegen wir uns die Laufzeitkomplexität der einzelnen Abschnitte:

- *Initialisierung bis Zeile 6 in $\mathcal{O}(m)$:* Es werden Variablen und Arrays der Länge $|m|$ angelegt und mit konstanten Werten initialisiert. Dieses hat eine Komplexität von $\mathcal{O}(m)$.
- *Schleife A in $\mathcal{O}(|S|)$:* Diese Schleife wird für $|S|$ Elemente ausgeführt und jede Zeile innerhalb der Schleife benötigt konstante Zeit. Somit hat die Schleife eine Komplexität von $\mathcal{O}(|S|)$.
- *While-Schleife in Zeile 15 insgesamt in $\mathcal{O}(m)$:* Die Werte von i wachsen monoton und nach Behauptung 5 terminiert die Schleife in jedem Durchlauf für einen Wert $i \in \underline{m}$. Somit kann i maximal m mal inkrementiert werden, die Schleife also maximal m mal durchlaufen werden. Daher hat sie eine Komplexität von $\mathcal{O}(m)$.
- *Schleife B in $\mathcal{O}(m)$:* Die Schleife wird für m Elemente durchlaufen, die While-Schleife ist insgesamt in $\mathcal{O}(m)$ und jede andere Zeile wird in konstanter Zeit ausgeführt. Somit hat die Schleife insgesamt eine Komplexität von $\mathcal{O}(m)$.
- *Definition von γ in $\mathcal{O}(|S|)$:* Es wird ein Array der Länge m angelegt. Die Definition der Werte in Zeile 19 hat eine Komplexität von $\mathcal{O}(|S|)$. Somit hat die Definition von γ eine Komplexität von $\mathcal{O}(\max(|S|, m))$.

Die übrigen Zeilen werden in konstanter Zeit ausgeführt. Insgesamt erhalten wir für den Algorithmus somit eine Komplexität von

$$\text{Sync-Schedules} \in \mathcal{O}(1) + \mathcal{O}(m) + \mathcal{O}(|S|) + \mathcal{O}(\max(|S|, m)) = \mathcal{O}(\max(|S|, m)).$$

A.2 Der Algorithmus Knotenkonfliktgraph-Sync

Algorithmus : Konkretere Implementierung des Algorithmus Knotenkonfliktgraph-Sync zur Konstruktion eines VD-UC-Konfliktgraphen zu Γ_v (vgl. Algorithmus 30)

Daten : $\mathcal{N}$: Die Struktur des VD-UC-Netzwerks.
$class$: Eine zu Beginn und am Ende leere Abbildung $class : \underline{|\Gamma|} \to \mathbb{N}_0 \cup \{\infty\}$.
Parameter : v: Der Knoten, für dessen Requestmenge Γ_v der Schedule erzeugt wird.
α: Der Schedule für die zusammenhängende zu v benachbarte Menge W.
Rückgabewert : C: Der Knotenkonfliktgraph zu Γ_v bezüglich α als Adjazenzliste.
cl: Abbildung, die jedem Request $r \in \Gamma_v$ in lokaler Indizierung γ_v die Nummer der Klasse $[r]$ zuordnet.
Komplexität : $\mathcal{O}(|\Gamma| + \Theta(v))$.

```
1 Funktion Knotenkonfliktgraph-Sync(v: Knoten, α: Schedule)
2 |  (cl, clnum, clc) := Bilde-Klassen;
3 |  C := Array [0, ..., clnum − 1] of Liste ;      // Konfliktgraph als Adjazenzliste
4 |  C := Add-Kanten(cl, clc);
5 |  C := Remove-Kanten(cl);
6 |  return (C, cl);
7 end
```

Analog zu Anhang A.1 wird für ein Request mit lokalem Index i die Schreibweise $r_i = \gamma_v(i)$ bezüglich der in Abschnitt 4.5.1 beschriebenen lokalen Indizierung γ_v verwendet. Für die Abbildung $cl : \Gamma_v \mapsto \mathbb{N}_0 \cup \{\infty\}$ wird die Notation $cl[i]$ verwendet, wenn i ein lokaler Index in ist und $cl(r_i) = cl[i]$, falls der Wert bezüglich des globalen Index r_i gemeint ist.

Wir zeigen die Korrektheit der einzelnen Funktionen und anschließend die Korrektheit des Algorithmus.

Es gelten die Aussagen:

(A) Nach Aufruf der Funktion Bilde-Klassen ist $cl : \Gamma_v \mapsto \underline{clnum}$ eine Abbildung, so dass bezüglich der in Definition und Satz 6.3.12 definierten Äquivalenzrelation gilt

$$\forall j, k \in |\Gamma_v| : cl[j] = cl[k] \iff r_j \sim r_k.$$

Dann ist $cl : \Gamma_v \to \underline{clnum}$ surjektiv und $clc = cl(R \cap \Gamma_v)$.

(B) Nach Aufruf der Funktion Bilde-Klassen ist für alle $j \in \Gamma_v$: $class(j) = \infty$.

(C) Nach Aufruf der Funktion Add-Kanten enthält C die in Kanten E_{konf} und E_{sync} aus Definition 6.3.13. Dabei ist jede Kante maximal 9 mal gespeichert.

(D) Nach Aufruf der Funktion Remove-Kanten enthält C jede Kante aus $E_{konf} \cup E_{sync}$ genau einmal, d.h. C ist der Konfliktgraph zu Γ_v bezüglich α.

Funktion : Bilde-Klassen zur Erzeugung der Klassen

Daten : $\mathcal{N}$, *class*.
Globale Variablen : v, α.
Rückgabewert : cl: Abbildung, die jedem Request $r \in \Gamma_v$ in lokaler Indizierung γ_v die Nummer der Klasse $[r]$ zuordnet.
clnum: Anzahl der verwendeten Klassen.
clc: Liste der Klassen $[r]$ für ein $r \in \Gamma_W$.
Komplexität : $\mathcal{O}(|\Gamma|)$.

```
1  Funktion Bilde-Klassen
2      clnum := 0 ;                                  // Anzahl der verwendeten Klassen
3      cl := Array [0,...,|Γ_v| − 1] of Integer ;     // Ordnet jedem r die Klasse [r] zu
4      for i = 0,...,|Γ_v| − 1 do cl[i] := ∞;
5      clc := ∅;
6      for i := 0,...,|Γ_v| − 1 do
7          if α(γ_v(i)) = ∞ then                     // r_i = γ_v(i) ∉ Γ_W, somit neue Klasse
8              cl[i] := clnum;
9              clnum := clnum + 1;
10         else if class(α(γ_v(i))) = ∞ then                  // Neue Klasse benötigt
11             class(α(γ_v(i))) := clnum;
12             clc := clc ∪ {clnum};
13             cl[i] := clnum;
14             clnum := clnum + 1;
15         else
16             cl[i] := class(α(γ_v(i)));
17     for i := 0,...,|Γ_v| − 1 do
18         if α(γ_v(i)) ≠ ∞ then class(α(γ_v(i))) := ∞;
19     return (cl, clnum, clc);
20 end
```

Bildung der Klassen

Soweit nicht anders angegeben beziehen sich alle verwendeten Zeilenangaben auf die Funktion Bilde-Klassen. Die For-Schleife ab Zeile 1 sei mit Schleife A bezeichnet.

Offenbar wird ein Wert $cl[i]$ nach der Initialisierung nur dann verändert, wenn die Schleife A für i durchlaufen wird.

Behauptung 1: Sei $\sim$ die in Definition und Satz 6.3.12 definierte Äquivalenzrelation bezüglich α. Dann gilt vor und nach jedem Schleifendurchlauf für i:

(i) Für alle $k \leq i$ ist $cl[k] \in \underline{clnum}$.

(ii) Für alle $j \in \underline{clnum}$ gibt es ein $k \leq i$ mit $cl[k] = j$.

(iii) Für alle $k \leq i$ mit $r_k \in \Gamma_W$ gilt $class(\alpha(r_k)) = cl[k]$.

(iv) Für alle Requests $s \notin \{\alpha(r_k) : k \leq i, r_k \in \Gamma_W\}$ gilt $class(s) = \infty$.

(v) Es ist $clc = \{cl(r_k) : k \leq i, r_k \in \Gamma_W\}$.

(vi) Für alle $j, k \in \{0, \ldots, i\}$ gilt $cl[j] = cl[k] \iff r_j \sim r_k$.

Begründung: Vor dem ersten Schleifendurchlauf sind die Aussagen erfüllt.

Seien $clnum'$, clc', $class'$ usw. die Werte vor dem Schleifendurchlauf und obige Bedingungen als Induktionsvoraussetzung (IV) bis $i-1$ erfüllt.

- Fall $r_i \notin \Gamma_W$: Dann ist nach Voraussetzung $\alpha(r_i) = \infty$, also die Bedingung in Zeile 1 erfüllt. Somit gilt nach dem Durchlauf der Schleife $cl(r_i) = cl[i] = clnum'$, $clnum = clnum' + 1$ und $class = class'$.

  Zu (i): Sei $k \leq i$. Falls $k \leq i-1$ so gilt nach IV $cl[k] \in \underline{clnum'} \subset \underline{clnum}$. Falls $k = i$, so ist $cl[k] = cl[i] = clnum' = clnum - 1 \in \underline{clnum}$.

  Zu (ii): Sei $j \in \underline{clnum}$. Falls $j \in \underline{clnum'}$ so gibt es nach IV ein $k \leq i-1 < i$ mit $cl[k] = j$. Falls $j = clnum'$, so ist $cl[i] = j$.

  Zu (iii): Wegen $r_i \notin \Gamma_W$ und $class = class'$ folgt dieses nach IV.

  Zu (iv): Wegen $r_i \notin \Gamma_W$ und $class = class'$ folgt dieses nach IV

  Zu (v): Wegen $r_i \notin \Gamma_W$ gilt

  $$clc = clc' = \{cl(r_k) : k \leq i-1, r_k \in \Gamma_W\} = \{cl(r_k) : k \leq i, r_k \in \Gamma_W\}.$$

  Zu (vi): Seien $j, k \in \{0, \ldots, i\}$. Falls $j, k \in \{0, \ldots, i-1\}$, so gilt die Aussage nach IV.

  Sonst sei o.B.d.A. $j = i$. Dann gilt $cl[j] = cl[i] = clnum' \notin \underline{clnum'}$ und nach IV (i) ist für alle $l \leq i-1$: $cl[k] \in \underline{clnum'}$. Somit ist $cl[j] = cl[k] \iff j = k = i$. Nun ist $r_j = r_i \notin \Gamma_W$, also $r_j \sim r_k \iff r_j = r_k$. Daraus folgt

  $$cl[j] = cl[k] \iff j = k \iff r_j = r_k \iff r_j = r_k.$$

– Fall $r_i \in \Gamma_W$ und $class(\alpha(r_i)) = \infty$: Dann wird der If-Zweig aus Zeile 1 durchlaufen. Somit gilt nach dem Durchlauf $class(\alpha(r_i)) = clnum'$, $clc = clc' \cup \{clnum'\}$, $cl(r_i) = cl[i] = clnum'$ und $clnum = clnum' + 1$.

  Zu (i): Analog zu Fall $r_i \notin \Gamma_W$.

  Zu (ii): Analog zu Fall $r_i \notin \Gamma_W$.

  Zu (iii): Sei $k \leq i$ mit $r_k \in \Gamma_W$. Falls $k \leq i-1$ so ist nach IV $class(\alpha(r_k)) = cl[k]$. Falls $k = i$ so ist $class(\alpha(r_k)) = clnum' = cl(r_k) = cl[k]$.

  Zu (iv): Nach Voraussetzung galt für alle $s \notin \{\alpha(r_k) : k \leq i-1, r_k \in \Gamma_W\}$ die Aussage. Es wurde ausschließlich $class(\alpha(r_i))$ umdefiniert, daher gilt die Aussage weiter für alle $s \notin \{\alpha(r_k) : k \leq i, r_k \in \Gamma_W\}$.

  Zu (v): Es ist

$$\begin{aligned} clc &= clc' \cup \{clnum'\} = \{cl(r_k) : k \leq i-1, r_k \in \Gamma_W\} \cup \{cl(r_i)\} \\ &= \{cl(r_k) : k \leq i, r_k \in \Gamma_W\}. \end{aligned}$$

  Zu (vi): Seien $j, k \in \{0, \ldots, i\}$. Falls $j, k \in \{0, \ldots, i-1\}$, so gilt die Aussage nach IV.

  Sonst sei o.B.d.A. $j = i$. Dann gilt $cl[j] = cl[i] = clnum' \notin \underline{clnum'}$ und nach IV (i) ist für alle $l \leq i-1$: $cl[k] \in \underline{clnum'}$. Somit ist

$$cl[j] = cl[k] \iff j = k \iff r_j = r_k$$

  Nun ist $class(\alpha(r_i)) = \infty$. Somit kann es kein $l < i$ mit $\alpha(r_l) = \alpha(r_i)$ geben, denn sonst wäre nach IV (iii)

$$class(\alpha(r_i)) = class(\alpha(r_l)) = c[l] \in \underline{clnum'} \implies class(\alpha(r_i)) \neq \infty.$$

  Widerspruch. Daher gilt für alle $l < i$ mit $r_l \in \Gamma_W$: $\alpha(r_l) \neq \alpha(r_i) = \alpha(r_j)$. Daraus folgt $\alpha(r_j) = \alpha(r_k) \iff r_j = r_k$ und somit die Äquivalenz

$$\begin{aligned} r_j \sim r_k &\iff r_j = r_k \vee (r_j, r_k \in \Gamma_W \wedge \alpha(r_j) = \alpha(r_k)) \\ &\iff r_j = r_k \iff cl[j] = cl[k]. \end{aligned}$$

– Fall $r_i \in \Gamma_W$ und $class(\alpha(r_i)) \neq \infty$: Dann gibt es wegen IV (iv) ein $l < i$ mit $r_l \in \Gamma_W$ und $\alpha(r_i) = \alpha(r_l)$. Es wird der Else-Zweig aus Zeile 1 durchlaufen und nach dem Durchlauf gilt $cl[i] = class(\alpha(r_i))$ und alle andere Variablen bleiben unverändert.

  Zu (i): Sei $k \leq i$. Falls $k \leq i-1$ so gilt nach IV $cl[k] \in \underline{clnum'} = \underline{clnum}$. Falls $k = i$, so ist nach IV (iii) und IV (i)

$$cl[k] = cl[i] = class(\alpha(r_i)) = class(\alpha(r_l)) = cl[l] \in \underline{clnum'} = \underline{clnum}.$$

  Zu (ii): Wegen $clnum = clnum'$ gibt es nach IV für jedes $j \in \underline{clnum}$ ein $k \leq i-1 < i$ mit $cl[k] = j$.

Zu (iii): Sei $k \leq i$ mit $r_k \in \Gamma_W$. Falls $k \leq i-1$ so ist nach IV $class(\alpha(r_k)) = cl[k]$. Falls $k = i$ so ist $cl[k] = class(\alpha(r_k))$.

Zu (iv): Nach IV gilt für alle $s \notin \{\alpha(r_k) : k \leq i-1, r_k \in \Gamma_W\}$ die Aussage, insbesondere gilt sie nun für alle $s \notin \{\alpha(r_k) : k \leq i, r_k \in \Gamma_W\}$.

Zu (v): Wegen

$$cl(r_i) = class(\alpha(r_i)) = class(\alpha(r_l)) = c(r_l) \in \{cl(r_k) : k \leq i-1, r_k \in \Gamma_W\}$$

gilt

$$\begin{aligned} clc = clc' &= \{cl(r_k) : k \leq i-1, r_k \in \Gamma_W\} \\ &= \{cl(r_k) : k \leq i-1, r_k \in \Gamma_W\} \cup \{cl(r_i)\} = \{cl(r_k) : k \leq i, r_k \in \Gamma_W\}. \end{aligned}$$

Zu (vi): Seien $j, k \in \{0, \ldots, i\}$. Falls $j, k \in \{0, \ldots, i-1\}$, so gilt die Aussage nach Voraussetzung.

Sonst sei o.B.d.A. $j = i$. Ist auch $k = i$, so gilt die Aussage. Sei nun $k < i$. Dann ist wegen $\alpha(r_i) = \alpha(r_l)$ auch $cl[i] = class(\alpha(r_i)) = class(\alpha(r_l)) = cl[l]$ und nach Definition $r_i \sim r_l$. Es folgt

$$\begin{aligned} & cl[i] = cl[k] && |\text{es ist } cl[i] = cl[l] \\ \iff & cl[l] = cl[k] && |\text{Wegen } k, l < i \text{ Anwendung von IV (vi)} \\ \iff & r_l \sim r_k && |\text{Transitivität der Äquivalenzrelation } \sim \text{ und } r_i \sim r_l \\ \iff & r_i \sim r_k. \end{aligned}$$

Behauptung 2: Es gilt Aussage (A).

Begründung: Aus Behauptung 1 folgt nach dem letzten Schleifendurchlauf: Nach (vi) gilt

$$\forall j, k \in \underline{|\Gamma_v|} : \qquad cl[j] = cl[k] \iff r_j \sim r_k.$$

Nach (i) gilt für alle $k \in \underline{|\Gamma_v|}$: $cl[k] \in \underline{clnum}$, d.h. für alle $r_k \in \Gamma_v$: $cl(r_k) \in \underline{clnum}$. Somit ist cl eine Abbildung $\Gamma_v \mapsto \underline{clnum}$. Nach (ii) existiert für alle $j \in \underline{clnum}$ ein $k \in \underline{|\Gamma_v|}$ mit $cl[k] = j$, d.h. ein $r_k \in \Gamma_v$ mit $cl(r_k) = j$. Somit ist $cl : \Gamma_v \mapsto \underline{clnum}$ surjektiv.

Nach (v) gilt

$$\begin{aligned} clc &= \{cl(r_k) : k \in \underline{|\Gamma_v|}, r_k \in \Gamma_W\} \\ &= \{cl(r_k) : r_k \in \Gamma_v, r_k \in \Gamma_W\} \\ &= \{cl(r_k) : r_k \in \Gamma_v \cap \Gamma_W\} \\ &= cl(\Gamma_v \cap \Gamma_W). \end{aligned}$$

Behauptung 3: Es gilt Aussage (B).

Begründung: Nach Behauptung 1(iv) gilt nach Beendigung der Schleife A: Für alle Requests $s \notin \{\alpha(r_k) : k \in \underline{|\Gamma_v|}, r_k \in \Gamma_W\}$ ist $class(s) = \infty$. Das heißt, für alle Requests $s \notin \alpha(\Gamma_v \cap \Gamma_W)$ ist $class(s) = \infty$.

In der For-Schleife ab Zeile 1 wird für alle $i \in \underline{|\Gamma_v|}$ mit $r_i \in \Gamma_W$ der Wert $class(\alpha(r_i)) := \infty$ gesetzt, d.h. nach der Schleife gilt: Für alle $s \in \Gamma_v \cap \Gamma_W$ ist $class(\alpha(s)) = \infty$.

Somit folgt: Für alle s ist $class(\alpha(s)) = \infty$.

Zur Komplexität: Es gilt Bilde-Klassen $\in \mathcal{O}(|\Gamma|)$.

- Die Initialisierung bis Schleife A hat eine Laufzeit von $\mathcal{O}(|\Gamma_v|)$.
- Jeder der $|\Gamma_v|$ Schleifendurchläufe von Schleife A benötigt konstante Zeit, daher hat Schleife A eine Gesamtkomplexität von $\mathcal{O}(|\Gamma_v|)$.
- Die Schleife in Zeile 1 hat eine Komplexität von $\mathcal{O}(|\Gamma|)$.

Insgesamt ergibt sich für die Komplexität Bilde-Klassen $\in \mathcal{O}(|\Gamma|)$.

Hinzufügen der Kanten

Funktion : Add-Kanten zum Einfügen der Kanten

Daten : $\mathcal{N}$, $class$.
Globale Variablen : v, C.
Parameter : cl: Abbildung, die jedem Request $r \in \Gamma_v$ in lokaler Indizierung γ_v die Nummer der Klasse $[r]$ zuordnet.
clc: Liste der Klassen $[r]$ für ein $r \in \Gamma_W$.
Rückgabewert : Konfliktgraph C, ggf. mit mehrfach vorkommende Kanten.
Komplexität : $\mathcal{O}(|\Gamma| + \Theta(v))$.

```
1  Funktion Add-Kanten(cl: Klassen als Abbildung, clc: Klassen der Requests aus Γ_W)
       // Kanten aus E_konf einfügen
2      for e := 0, ..., δ'(v) do
3          foreach i ∈ in_v(e) do  foreach j ∈ in_v(e) do if cl[i] ≠ cl[j] then
4              C[cl[i]] := C[cl[i]] ∪ {cl[j]};
5          foreach i ∈ out_v(e) do  foreach j ∈ out_v(e) do if cl[i] ≠ cl[j] then
6              C[cl(i)] := C[cl(i)] ∪ {cl(j)};

       // Kanten aus E_sync einfügen
7      foreach y ∈ clc do  foreach z ∈ clc do if y ≠ z then
8          C[y] := C[y] ∪ {z};

9      return C;
10 end
```

Im Folgenden beziehen sich alle Zeilenangaben auf die Funktion Add-Kanten. Die For-Schleife ab Zeile 2 wird mit Schleife B bezeichnet, die For-Schleife ab Zeile 2 mit Schleife C.

Zur Vereinfachung identifizieren wird die Liste $C[i]$ mit der Menge der in $C[i]$ enthaltenen Elemente. Das ist insofern ein Unterschied, als dass in einer Liste Elemente mehrfach gespeichert werden, in der Menge jedoch nicht. Dieses verhindert einen formalen Beweis von Behauptung 7, so dass diese nur anschaulich bewiesen werden soll.

Dann sagen wir, der Graph C enthält eine Kante $e = \{i, j\}$ (i.Z. $e \in E_C$), falls $i \in C[j]$ und $j \in C[i]$ ist. Für ein Request r ist $[r] = \{s \in \Gamma_v : r \sim s\} = \{s \in \Gamma_v : cl(r) = cl(s)\}$.

Behauptung 4: Nach jedem Durchlauf der Schleife B für eine Kante e gilt für alle k

$$C[cl[k]] = \{cl[l] : r_l \leftsquigarrow_f r_k \text{ für eine Kante } f \leq e\}.$$

Begründung: Vor dem ersten Durchlauf ist nichts zu zeigen. Sei die Aussage nun vor einem Durchlauf erfüllt und C' das Array C vor dem Durchlauf.

Sei $k \in \underline{|\Gamma_v|}$ fest gewählt.

- Fall $e \notin E_{r_k}$: Dann wird $C[cl[k]]$ nicht verändert. Da es kein r_l mit $r_l \leftsquigarrow_e r_k$ gibt folgt

$$\begin{aligned} C[cl(k)] = C'[cl(k)] &= \{cl[l] : r_l \leftsquigarrow_f r_k \text{ für eine Kante } f \leq e-1\} \\ &= \{cl[l] : r_l \leftsquigarrow_f r_k \text{ für eine Kante } f \leq e\}. \end{aligned}$$

- Fall $e \in E_{r_k}$ und $e^{r_k+} = v$: Dann ist $r_k \in in_v(e)$ und in der ForEach-Schleife ab Zeile 2 folgt

$$\begin{aligned} C[cl[k]] &= C'[cl[k]] \cup \{cl[j] : r_j \in in_v(e), cl[j] \neq cl[k]\} \\ &= C'[cl[k]] \cup \{cl[j] : e^{r_j+} = v, cl[j] \neq cl[k]\} \\ &= \{cl[l] : r_l \leftsquigarrow_f r_k \text{ für eine Kante } f \leq e-1\} \cup \{cl[j] : r_j \leftsquigarrow_e r_k\} \\ &= \{cl[l] : r_l \leftsquigarrow_f r_k \text{ für eine Kante } f \leq e\}. \end{aligned}$$

- Fall $e \in E_{r_k}$ und $e^{r_k-} = v$: Dann ist $r_k \in out_v(e)$ und die Überlegung ist Analog zum vorherigen Fall mit Zeile 2.

Für jeden Fall folgt nach dem Durchlauf die Behauptung.

Behauptung 5: Nach dem Ende der Schleife B gilt ist $E_C = E_{konf}$, d.h. für $k, l \in \underline{|\Gamma_v|}$:

$$\exists r_{k'} \in [r_k] \wedge r_{l'} \in [r_l] \text{ mit } r_{k'} \leftsquigarrow r_{l'} \iff \{cl[l], cl[k]\} \in E_C.$$

Begründung: Nach Behauptung 4 ist nach Ende der Schleife B für jedes k:

$$\begin{aligned} C[cl[k]] &= \{cl[l] : r_l \leftsquigarrow_f r_k \text{ für eine Kante } f \in E(v)\} \\ &= \{cl[l] : r_l \leftsquigarrow_v r_k\} = \{cl[l] : r_l \leftsquigarrow r_k\} \end{aligned}$$

wobei die letzten Gleichheit wegen $r_k, r_l \in \Gamma_v$ nach Satz 4.2.24 gilt.

Dann folgt

$$\begin{aligned}
&\exists r_{k'} \in [r_k] \wedge r_{l'} \in [r_l] \text{ mit } r_{k'} \leftrightsquigarrow r_{l'} \\
\iff &\exists k', l' \text{ mit } cl[k'] = cl[k] \wedge cl[l'] = cl[l] \text{ und } r_{k'} \leftrightsquigarrow r_{l'} \\
\iff &\exists k', l' \text{ mit } cl[k'] = cl[k] \wedge cl[l'] = cl[l] \text{ und } cl[k'] \in C[cl[l']] \wedge cl[l'] \in C[cl[k']] \\
\iff &cl[k] \in C[cl[l]] \wedge cl[l] \in C[cl[k]] \\
\iff &\{cl[l], cl[k]\} \in E_C.
\end{aligned}$$

Behauptung 6: Es gilt der erste Teil der Aussage (C): Nach dem Ende der Schleife C gilt ist $E_C = E_{konf} \cup E_{sync}$.

Begründung: Vor dem Durchlauf der Schleife C ist $E_{C'} = E_C = E_{konf}$.

Nach Aussage (A) ist $clc = cl(\Gamma_v \cap \Gamma_W)$. Ist nun $i \neq j \in clc$, so wird in Schleife C für $y = i$ und $z = j$ dann $C[i] := C[i] \cup \{j\}$ und für $y = j$ und $z = i$ entsprechend $C[j] := C[j] \cup \{i\}$ gesetzt, also die Kante $ij = \{i, j\}$ zu C hinzugefügt. Da offenbar keine weiteren Kanten zu C hinzugefügt werden, folgt nach Schleife C:

$$\begin{aligned}
E_C &= E_{C'} \cup \{ij : i, j \in clc, i \neq j\} && |clc = cl(\Gamma_v \cap \Gamma_W) \\
&= E_{konf} \cup \{ij : i = cl(r_k), j = cl(r_l), r_k, r_l \in \Gamma_v \cap \Gamma_W, cl(r_k) \neq cl(r_l)\}
\end{aligned}$$

Für $r_k, r_l \in \Gamma_W$ gilt $\alpha(r_k) = \alpha(r_l) \iff r_k \sim r_l \iff cl(r_k) = cl(r_l)$. Somit folgt

$$\begin{aligned}
E_C &= E_{konf} \cup \{ij : i = cl(r_k), j = cl(r_l), r_k, r_l \in \Gamma_v \cap \Gamma_W, \alpha(r_k) \neq \alpha(r_l)\} \\
&= E_{konf} \cup \{\{cl(r_k), cl(r_l)\} : r_k, r_l \in \Gamma_v \cap \Gamma_W, \alpha(r_k) \neq \alpha(r_l)\} && |r_k, r_l \in \Gamma_v \\
&= E_{konf} \cup E_{sync}
\end{aligned}$$

Behauptung 7: Es gilt der zweite Teil der Aussage (C): Jede Kante $e := \{cl(r), cl(s)\}$ ist maximal 9 mal in C gespeichert.

Begründung: (Anschaulich:) Seien r_k, r_l Requests mit $cl(r_k) = cl(r)$ und $cl(r_l) = cl(s)$. Dann haben r_k und r_s in maximal zwei zu v inzidenten Kanten einen Konflikt. In Schleife B wird somit für das Tupel (r_k, r_l) die Kante e maximal doppelt in C eingefügt. Nach Lemma 6.3.15 enthalten $[r]$ und $[s]$ maximal zwei Elemente, daher kann es maximal 4 solche Tupel (r_k, r_l) geben und die Kante e wird maximal 8 mal eingefügt. Da clc jedes Klasse $[t]$ für $t \in \Gamma_W$ genau einmal enthält, wird eine Kante in der Schleife C maximal 1 mal eingefügt. Somit ist eine Kante e maximal 9 mal in C gespeichert.

Zur Komplexität: Es gilt Add-Kanten $\in \mathcal{O}(\Theta(v) + |\Gamma_v|)$.

- Die Schleife Schleife B hat nach Lemma 4.2.32 eine Komplexität von

$$\begin{aligned}
\mathcal{O}\left(\sum_{e \in E(v)} (|in_v(e)|^2 + |out_v(e)|^2)\right) &\subset \mathcal{O}\left(\sum_{e \in E(v)} (|in_v(e)| + |out_v(e)|)^2\right) \\
&= \mathcal{O}\left(\sum_{e \in E(v)} \Psi(e)^2\right) \subset \mathcal{O}(\Theta(v) + |\Gamma_v|).
\end{aligned}$$

– Sei $\Lambda = \Gamma_v \cap \Gamma_W$. Dann hat die Schleife C nach Lemma 6.3.16 eine Komplexität von

$$\begin{aligned}\mathcal{O}(|clc|^2) &= \mathcal{O}\left(|cl(\Lambda)|^2\right) \subset \mathcal{O}(|\Lambda|^2) \\ &\subset \mathcal{O}(|\Lambda|(|\Lambda|-1)+|\Lambda|) \subset \mathcal{O}(4\Theta(v)+2|\Gamma_v|+|\Gamma_v|) \\ &= \mathcal{O}(\Theta(v)+|\Gamma_v|)\end{aligned}$$

Zusammen folgt für die Gesamtkomplexität Add-Kanten $\in \mathcal{O}(\Theta(v)+|\Gamma_v|)$.

Entfernen mehrfacher Kanten

Funktion : Remove-Kanten zum Entfernen mehrfacher Kanten

Daten : $\mathcal{N}$, *class*.
Globale Variablen : v, C.
Parameter : *cl*: Abbildung, die jedem Request $r \in \Gamma_v$ die Klasse $[r]$ zuordnet.
clnum: Anzahl der Klassen.
Rückgabewert : Konfliktgraph C als Adjazenzliste, in dem jede Kante einfach vorkommt.
Komplexität : $\mathcal{O}(|\Gamma| + \Theta(v))$.

```
1 Funktion Remove-Kanten(cl: Klassen als Abbildung, clnum: Anzahl der Klassen)
2     U := Array [0,...,clnum − 1] of Boolean, mit false initialisiert;
3     for i = 0,...,clnum − 1 do
4         foreach j ∈ C[i] do
5             if U[j] then entferne das aktuelle Element j aus C[i];
6             else U[j] := true;
7         foreach j ∈ C[i] do U[j] := false;
8     return C;
9 end
```

Im Folgenden beziehen sich alle Zeilenangaben auf die Funktion Remove-Kanten. Die For-Schleife ab Zeile 3 wird mit Schleife D bezeichnet, die ForEach-Schleife ab Zeile 3 mit Schleife E.

Der Zustand $C[i]$ wird durch den Vektor $(c_{i1},\ldots,c_{in})$ repräsentiert. Sei p die Position des aktuellen Elements in der Liste. Das Entfernen des aktuellen Elements bedeutet dann das aus dem Zustand $C'[i] = (c'_{i1},\ldots,c'_{in'})$ vor dem Entfernen der Zustand $C[i] = (c_{i1},\ldots,c_{in}) := (c'_{i1},\ldots,c'_{i,p-1},c'_{i,p+1},c'_{in'})$ wird. Das bedeutet, dass die Variablen sich wie folgt verändern:

$$\begin{aligned}n &:= n' - 1 \\ p &:= p' - 1 \\ c_{ik} &= c'_{ik} \text{ für } i = 1,\ldots,p'-1 \\ c_{ik} &= c'_{i,k+1} \text{ für } i = p',\ldots,n'-1\end{aligned}$$

Behauptung 8: Vor dem ersten Durchlauf der Schleife E sei $U[k] = false$ für alle k und $C[i] := (d_{i1}, \ldots, d_{im})$ der Zustand von C. Sei p die Position in der Liste $C[i]$ und q die Zahl der bereits durchlaufenden Elemente. Dann ist $c_{ip} = d_{iq}$ und es gilt:

(i) Es ist $\{c_{i1}, \ldots, c_{ip}\} = \{d_{i1}, \ldots, d_{iq}\}$.

(ii) Für alle $k, l \in \{0, \ldots, p\}$ ist $c_{ik} \neq c_{il}$.

(iii) Es ist $U[j] = true \iff j \in \{c_{i1}, \ldots, c_{ip}\}$.

Begründung: Vor dem ersten Durchlauf ist $p = q = 0$ und alle Aussagen sind offensichtlich erfüllt. Seien die Aussagen nun als Induktionsvoraussetzung IV vor dem Durchlauf q für j erfüllt. Seien $C'[i] = (c'_{i1}, \ldots, c'_{in'})$, p', q', n' und U' die Variablen vor dem Durchlauf. Dann gilt nach der Wahl des nächsten Elements: $q = q' + 1$, $p = p' + 1$ und $c_{ip} = j$.

– Fall $U[j] = true$: Das aktuelle Element $c_{ip} = j$ wird entfernt, d.h. es ist

$$\begin{aligned} &n := n' - 1 \\ &p := (p' + 1) - 1 = p' \\ &c_{ik} = c'_{ik} \text{ für } i = 1, \ldots, (p' + 1) - 1 = p' \\ &c_{ik} = c'_{i,k+1} \text{ für } i = p' + 1, \ldots n' - 1. \end{aligned}$$

Zu (i): Wegen $U[j] = true$ ist nach IV (iii)

$$d_{iq} = c_{ip} = j \in \{c'_{i1}, \ldots, c'_{ip'}\} = \{d'_{i1}, \ldots, d'_{iq'}\}$$

und somit folgt

$$\begin{aligned} \{c_{i1}, \ldots, c_{ip}\} = \{c_{i1}, \ldots, c_{ip'}\} &= \{c'_{i1}, \ldots, c'_{ip'}\} = \{d_{i1}, \ldots, d_{iq'}\} \\ &= \{d_{i1}, \ldots, d_{iq'}\} \cup \{d_{iq}\} = \{d_{i1}, \ldots, d_{iq}\}. \end{aligned}$$

Zu (ii): Seien $k, l \in \{0, \ldots, p\} = \{0, \ldots, p'\}$. Dann folgt die Aussage aus IV (ii) wegen $\{c_{i1}, \ldots, c_{ip}\} = \{c'_{i1}, \ldots, c'_{ip'}\}$

Zu (iii): Es ist

$$U[k] = true \iff U'[k] = true \iff k \in \{c'_{i1}, \ldots, c'_{ip'}\} = \{c_{i1}, \ldots, c_{ip}\}.$$

– Fall $U[j] = false$, dann ist nach IV (iii) $d_{iq} = c_{ip} = j \notin \{c'_{i1}, \ldots, c'_{ip'}\}$.

Zu (i): Es ist

$$\begin{aligned} \{c_{i1}, \ldots, c_{ip}\} &= \{c_{i1}, \ldots, c_{i,p-1}\} \cup \{c_{ip}\} = \{c'_{i1}, \ldots, c'_{ip'}\} \cup \{d_{iq}\} \\ &= \{d_{i1}, \ldots, d_{iq'}\} \cup \{d_{iq}\} = \{d_{i1}, \ldots, d_{iq}\}. \end{aligned}$$

Zu (ii): Seien $k, l \in \{0, \ldots, p\}$. Falls $k, l \in \{0, \ldots, p'\}$, so folgt die Aussage wegen $\{c_{i1}, \ldots, c_{ip'}\} = \{c'_{i1}, \ldots, c'_{ip'}\}$ aus IV (ii). Ansonsten sei o.B.d.A. $l = p$ und $k \leq p'$. Dann ist $c_{ik} = c'_{ik} \in \{c'_{i1}, \ldots, c'_{ip'}\}$ und wegen $U'[c_{ip}] = U[j] = false$ ist $c_{ip} \notin \{c'_{i1}, \ldots, c'_{ip'}\}$. Somit ist $c_{ip} \neq c_{ik}$.

Zu (iii): Es wird $U[j] := true$ definiert. Dann ist nach IV

$$U[k] = true \iff U'[k] = true \vee k = j \iff k \in \{c'_{i1}, \ldots, c'_{ip'}\} \vee k = c_{ip}$$
$$\iff k \in \{c_{i1}, \ldots, c_{i,p-1}\} \vee k = c_{ip} \iff k \in \{c_{i1}, \ldots, c_{ip}\}.$$

Behauptung 9: Sei vor dem ersten Durchlauf der Schleife E $U[k] = false$ für alle k. Dann gilt nach Ende der Schleife:

(i) Es ist $\{c_{i1}, \ldots, c_{in}\} = \{d_{i1}, \ldots, d_{im}\}$.

(ii) Für alle $k, l \in \{0, \ldots, n\}$ ist $c_{ik} \neq c_{il}$.

(iii) Es ist $U[j] = true \iff j \in \{c_{i1}, \ldots, c_{in}\}$.

Begründung: Dieses folgt unmittelbar aus Behauptung 8 nach dem letzten Schleifendurchlauf, denn dann ist $p = n$ und $q = m$.

Behauptung 10: Vor und nach jedem Schleifendurchlauf von Schleife D ist $U[j] = false$ für alle j.

Begründung: Vor dem ersten Schleifendurchlauf ist dieses wegen der Initialisierung von U erfüllt. Sei es vor einem Durchlauf der Schleife D für i erfüllt. Dann gilt nach Behauptung 9 nach Ende der Schleife E, also vor Zeile 3, $U[j] = true \iff j \in \{c_{i1}, \ldots, c_{ip}\}$, insbesondere also für alle $j \notin \{c_{i1}, \ldots, c_{ip}\}$ bereits $U[j] = false$. Nach Zeile 3 gilt für alle $j \in \{c_{i1}, \ldots, c_{ip}\}$ ebenfalls $U[j] = false$. Somit ist $U[j] = false$ für jedes j.

Behauptung 11: Es gilt Aussage (D).

Begründung: Nach Behauptung 10 gilt zu Beginn jedes Schleifendurchlaufs $U[j] = false$ für jedes j, somit ist die Bedingung von Behauptung 9 in jedem Durchlauf erfüllt. Darum gilt nach Ende der Schleife D für jedes i die Eigenschaften aus Behauptung 9. Insbesondere besagt (i), dass – als Menge aufgefasst – E_C nicht verändert wurde, also wegen Aussage (C) weiter $E_C = E_{konf} \cup E_{sync}$ ist. Weiter besagt (ii), dass in $C[i]$ jeder zu i benachbarte Knoten nur einmal gespeichert ist, insgesamt also jede Kante nur einmal gespeichert ist.

Zur Komplexität: Es gilt Remove-Kanten $\in \mathcal{O}(|V_C|+|E_C|)$: Im Folgenden ist mit $d_C(i)$ der Grad des Knoten im Konfliktgraphen i gemeint, während $|C[i]|$ die tatsächliche Länge der gespeicherten Liste (mit mehrfachen Kanten) ist. Dann gilt

- Die Initialisierung von U hat eine Komplexität von $\mathcal{O}(clnum) = \mathcal{O}(|V_C|)$.
- Die Liste $C[i]$ enthält nach Aussage (C) jeden zu i benachbarten Knoten maximal 9 mal, d.h. es ist für die Länge $|C[i]| \in \mathcal{O}(d_C(i))$. Die Schleife E hat somit eine eine Komplexität von $\mathcal{O}(1 + |C[i]|) = \mathcal{O}(1 + d_C(i))$.
- Die Schleife in Zeile 3 hat eine Komplexität von $\mathcal{O}(d_C(i))$.
- Die Schleife D hat daher eine Komplexität von

$$\sum_{i \in \underline{clnum}} (1 + \mathcal{O}(d_C(i)) = \mathcal{O}(clnum + 2|E_C|) = \mathcal{O}(|V_C| + |E_C|).$$

Insgesamt ergibt sich mit Lemma 6.3.17

$$\text{Remove-Kanten} \in \mathcal{O}(|V_C| + |E_C|) \subset \mathcal{O}(\Theta(v) + |\Gamma_v|).$$

Zusammenfassung Sei C der Konfliktgraph für Γ_v bezüglich α.

- Nach dem Aufruf von Bilde-Klassen wurden die Klassen korrekt gebildet.
- In der nächsten Zeile wird ein leerer Graph mit den Knoten V_C erzeugt.
- Nach dem Aufruf von Add-Kanten enthält dieser alle Kanten von E_C.
- Nach dem Aufruf von Remove-Kanten enthält dieser alle Kanten von E_C genau einmal.

Somit wird die Adjazenzliste für C zurückgegeben, die jede Kante genau einmal enthält.

Die Laufzeitkomplexität ergibt sich aus der Summe der Einzelkomplexitäten, es ist

$$\text{Knotenkonfliktgraph-Sync} \in \mathcal{O}(|\Gamma_v| + |\Gamma_v| + (\Theta(v) + |\Gamma_v|) + (\Theta(v) + |\Gamma_v|)) = \mathcal{O}(\Theta(v) + |\Gamma_v|).$$

Literaturverzeichnis

[Aho88] Aho, Alfred V.: *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1988. – ISBN 0–201–00029–6

[Alo03] Alon, Noga: A Simple Algorithm for Edge-Coloring Bipartite Multigraphs. In: *Inf. Process. Lett.* 85 (2003), Nr. 6, S. 301–302. http://dx.doi.org/10.1016/S0020-0190(02)00446-5. – ISSN 0020–0190

[Big90] Biggs, N. L.: Some Heuristics for Graph Colouring. In: Nelson, R. (Hrsg.) ; Wilson, R. J. (Hrsg.): *Graph Colourings.* Pitman Research Notes in Mathematics 218, 1990, S. 87–96

[Bré79] Brélaz, Daniel: New Methods to Color the Vertices of a Graph. In: *Communications of the ACM* 22 (1979), Nr. 4, S. 251–256. http://dx.doi.org/10.1145/359094.359101. – ISSN 0001–0782

[Bro72] Brown, J. R.: Chromatic Scheduling and the Chromatic Number Problem. In: *Management Science* 19 (1972), Nr. 4, S. 456–463

[CH94] Clark, John ; Holton, Derrek A.: *Graphentheorie.* Spektrum-Verlag, 1994. – ISBN 3–86025–331–X

[Chr75] Christofides, Nicos: *Graph Theory: An Algorithmic Approach.* Academic Press, 1975

[COS01] Cole, R. ; Ost, K. ; Schirra, S.: Edge-Coloring Bipartite Multigraphs in $O(E \log D)$ Time. In: *Combinatorica* 21 (2001), S. 5–12

[Die06] Diestel, Reinhard: *Graph Theory.* 3. Auflage. Berlin : Springer-Verlag, 2006. – ISBN 3–540–26183–4

[DW06a] Dopatka, Frank ; Wismüller, Roland: Achieving Realtime Capabilities in Ethernet Networks by Edge-Coloring of Communication Conflict-Multigraphs. In: *Parallel and Distributed Computing and Networks*, 2006, S. 180–185

[DW06b] Dopatka, Frank ; Wismüller, Roland: A Top-Down Approach for Realtime Industrial-Ethernet Networks using Edge-Colouring of Conflict-Multigraphs. In: *International Symposium on Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM)*, 2006. – ISBN 1–4244–0193–3, S. 883–890

[EGCGJL85] E. G. Coffman, Jr. ; Garey, M. R. ; Johnson, D. S. ; LaPaugh, A. S.: Scheduling File Transfers. In: *SIAM Journal on Computing* 14 (1985), Nr. 3, S. 744–780. http://dx.doi.org/10.1137/0214054

[Eis03] Eisenbrand, Friedrich: *Fast Integer Programming in Fixed Dimension.* Bd. 2832. 2003. – 196–207 S.

[Epp03] Eppstein, David: Small Maximal Independet Sets and Faster Exact Graph Colorin. In: *Journal of Graph Algorithms and Applications* 7 (2003), Nr. 2, S. 131–140

[Erl99] Erlebach, Thomas: *Scheduling Connections in Fast Networks.* `citeseer.ist.psu.edu/erlebach99scheduling.html`. Version: 1999

[GJ76] Garey, Michael R. ; Johnson, David S.: *Computers and Intractability; A Guide to the Theory of NP-Completeness.* New York, NY, USA : W. H. Freeman & Co., 1976

[Gra69] Graham, Ronald L.: Bounds on Multiprocessing Timing Anomalies. In: *SIAM Journal of Applied Mathematics* 17 (1969), Nr. 2, S. 416–429

[Hal93] Halldórsson, Magnús M.: A Still Better Performance Guarantee for Approximate Graph Coloring. In: *Information Processing Letters* 45 (1993), Nr. 1, S. 19–23. `http://dx.doi.org/10.1016/0020-0190(93)90246-6`. – ISSN 0020–0190

[Has96] Hastad, J.: Clique is Hard to Approximate within $n^{1-\varepsilon}$. In: *focs* 00 (1996), S. 627–636. `http://dx.doi.org/10.1109/SFCS.1996.548522`. ISBN 0–8186–7594–2

[HDD03] Hilgemeier, Mario ; Drechsler, Nicole ; Drechsler, Rolf: Fast Heuristics for the Edge Coloring of Large Graphs. In: *dsd* 00 (2003), S. 230. `http://dx.doi.org/10.1109/DSD.2003.1231932`. ISBN 0–7695–2003–0

[IC94] Ignizio, James P. ; Cavalier, Tom M.: *Linear Programming.* Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1994. – ISBN 0–13–183757–5

[JDU+74] Johnson, David S. ; Demers, Alan J. ; Ullman, Jeffrey D. ; Garey, M. R. ; Graham, Ronald L.: Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms. In: *SIAM Journal on Computing* 3 (1974), Nr. 4, S. 299–325

[Joh74] Johnsonn, D. S.: Worst Case Behaviour of Graph Coloring Algorithms. In: *Proceedings of the 5th South-Eastern Conference on Combinatorics, Graph Theory and Computing*, Utilitas Mathematica Publishing, 1974, S. 513–527

[Jun99] Jungnickel, Dieter: *Graphs, Networks and Algorithms.* Springer-Verlag, 1999. – ISBN 3–540–63760–5

[KJ85] Kubale, Marek ; Jackowski, Boguslaw: A Generalized Implicit Enumeration Algorithm for Graph Coloring. In: *Communications of the ACM* 28 (1985), Nr. 4, S. 412–418. `http://dx.doi.org/10.1145/3341.3350`. – ISSN 0001–0782

[LY93] Lund, Carsten ; Yannakakis, Mihalis: On the Hardness of Approximating Minimization Problems. In: *STOC '93: Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing.* New York, NY, USA : ACM Press, 1993. – ISBN 0–89791–591–7, S. 286–293

[MB83] Matula, David W. ; Beck, Leland L.: Smallest-Last Ordering and Clustering and Graph Coloring Algorithms. In: *Journal of the Association for Computing Machinery* 30 (1983), Nr. 3, S. 417–427

[MMI72] Matula, David W. ; Marble, George ; Isaacson, Joel D.: Graph Coloring Algorithms. In: Read, Ronald C. (Hrsg.): *Graph Theory and Computing*. Academic Press, 1972, S. 109–122

[MT96] Mehrotra, Anuj ; Trick, Michael A.: A Column Generation Approach for Graph Coloring. In: *INFORMS Journal on Computing* 8 (1996), Nr. 4, 344–354. citeseer.ifi.unizh.ch/mehrotra95column.html

[NK90] Nishizeki, Takao ; Kashiwagi, Kenichi: On the 1.1 Edge-Coloring of Multigraphs. In: *SIAM Journal on Discrete Mathematics* 3 (1990), Nr. 3, S. 391–410. http://dx.doi.org/10.1137/0403035. – ISSN 0895–4801

[Pee83] Peemöler, Jürgen: A Correction to Brelaz's Modification of Brown's Coloring Algorithm. In: *Communications of the ACM* 26 (1983), Nr. 8, S. 595–597. http://dx.doi.org/10.1145/358161.358171. – ISSN 0001–0782

[Sch83] Schmitz, Lothar: An Improved Transitive Closure Algorithm. In: *Computing* 30 (1983), S. 359–371

[SDK83] Syslo, Maciej ; Deo, Narsingh ; Kowalik, Janusz S.: *Discrete Optimization Algorithms with Pascal Programs*. Prentice Hall Professional Technical Reference, 1983. – ISBN 0–13–215509–5

[SV85] Spinrad, Jeremy P. ; Vijayan, Gopalakrishnan: Worst Case Analysis of a Graph Coloring Algorithm. In: *Diescrete Applied Mathematics and Combinatorial Operations Research* 12 (1985), Nr. 1, S. 89–92

[Tom05] Tomazic, Alessandro: *Graphenfärbung mit Hilfe Linearer Programmierung*. Augsburg, Universität Augsburg, Deutschland, Diplomarbeit, 2005

[Tur04] Turau, Volker: *Algorithmische Graphentheorie*. 2. Auflage. Addison-Wesley, 2004. – ISBN 3–486–20038–0

[Vol91] Volkmann, Lutz: *Eine Einführung in die Graphentheorie*. Wien, New York : Springer-Verlag, 1991. – ISBN 3–211–82267–4

[Wig83] Wigderson, Avi: Improving the Performance Guarantee for Approximate Graph Coloring. In: *Journal of the ACM* 30 (1983), Nr. 4, S. 729–735. http://dx.doi.org/10.1145/2157.2158. – ISSN 0004–5411

[WP67] Welsch, David J. A. ; Powell, Marianne B.: An upper Bound for the Chromatic Number of a Graph and its Application to Timetabling. In: *The Computer Journal*. Oxford Journals, 1967, S. 85–86

Ich erkläre, diese Diplomarbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt zu haben.

_____ _____