Online Optimierung

Sven O. Krumke
&
Jörg Rambau

Entwurf vom 29. September 2003

Technische Universität Berlin

Dieses Skript ist aus dem Online-Teil der Vorlesung »Ausgewählte Kapitel aus der ganzzahligen Optimierung« (Wintersemester 1999/2000) und der Vorlesung »Online Optimierung« (Sommersemester 2000, Wintersemester 2001/2002) an der Technischen Universität Berlin entstanden.

Über Kritik, Verbesserungsvorschläge oder gefundene Tippfehler würden wir uns sehr freuen!

Sven O. Krumke & Jörg Rambau
{krumke,rambau}@zib.de

Inhaltsverzeichnis

I Theoretische Analyse von Online-Algorithmen 1

1 Einleitung 3
- 1.1 Was sind »Online Probleme«? 3
- 1.2 Literatur 5
- 1.3 Weitere Informationsquellen 5
- 1.4 WWW-Seiten zur Vorlesung 6

2 Grundbegriffe 7
- 2.1 Online und Offline Optimierung 7
- 2.2 Kompetitive Algorithmen 9
- 2.3 Randomisierte Algorithmen 10

3 Beispiele und elementare Techniken 17
- 3.1 Organisation von linearen Listen 17
- 3.2 Das Bahncard-Problem 21

4 Paging 29
- 4.1 Ein optimaler Offline Algorithmus 30
- 4.2 Deterministische Online-Algorithmen 32
- 4.3 Ein randomisierter Algorithmus für Paging 34

5 Ein Ausflug in die Spieltheorie 37
- 5.1 Zwei-Personen Nullsummenspiele 37
- 5.2 Yao's Prinzip und seine Anwendungen auf Online-Probleme 39
- 5.3 Beispiele für die Anwendung von Yao's Prinzip 40

6 Telekommunikation und Netzwerk Routing 43
- 6.1 Lastbalancierung 44
- 6.2 Durchsatzmaximierung 48
- 6.3 Routing in optischen Netzwerken 51

7 Metrische Tasksysteme **59**
7.1 Arbeitsfunktionen 60
7.2 Eine untere Schranke 61
7.3 Der Algorithmus WFA 61

8 Das k-Server Problem **65**
8.1 Faule Algorithmen 66
8.2 Eine untere Schranke für deterministische Algorithmen 66
8.3 Das k-Server Problem auf der Linie 67
8.4 Der Work Function Algorithmus 70

9 Transport- und Logistik-Probleme **83**
9.1 Problemstellung und Zeitstempel-Modell 83
9.2 Optimierung der Fertigstellungszeit 85
9.3 Optimierung der Fluß- und Wartezeiten 91

II Online-Algorithmen in der Praxis **99**

10 Online-Optimierung in flexiblen Fertigungsanlagen **101**
10.1 Herlitz PBS AG: das Versandlager in Falkensee (Stand 1999) 101

11 Diskrete ereignisbasierte Simulation **105**
11.1 Zielsetzung 105
11.2 Begriffe 105
11.3 Das Simulationssystem AMSEL 108
11.4 Praktische Probleme bei der Simulation 110

12 Hochregallagerbediengeräte **113**
12.1 Problembeschreibung 113
12.2 Mathematisches Modell 114
12.3 Online-Heuristiken 114
12.4 A-posteriori-Analyse 115
12.5 Fazit 116

13 Automatische Glückwunschkartenkommissionierung **119**
13.1 Problembeschreibung 119
13.2 Mathematisches Modell 120
13.3 Kompetitive Analyse 120
13.4 Online-Heuristiken 124
13.5 Fazit 125

A Abkürzungen und Symbole **127**

B Komplexität von Algorithmen **129**
B.1 Größenordnung von Funktionen 129
B.2 Berechnungsmodell 129
B.3 Komplexitätsklassen 130

C Lösungen zu den Übungsaufgaben **133**

Literaturverzeichnis **145**

Teil I

Theoretische Analyse von Online-Algorithmen

1

Einleitung

1.1 Was sind »Online Probleme«?

In der klassischen kombinatorischen Optimierung wird davon ausgegangen, daß die Daten jeder Probleminstanz vollständig gegeben sind. Aufbauend auf diesem vollständigen Wissen berechnet dann ein Algorithmus eine optimale (oder approximative) Lösung. In vielen Fällen modelliert diese *Offline-Optimierung* jedoch die Situationen aus Anwendungen nur ungenügend. Zahlreiche Problemstellungen in der Praxis sind in natürlicher Weise *online*: Sie erfordern Entscheidungen, die unmittelbar und ohne Wissen zukünftiger Ereignisse getroffen werden müssen. Die folgenden Fragestellungen sind Beispiele für *Online-Probleme*.

Offline-Optimierung

Online-Probleme

Skifahrerproblem Eine Sportlerin geht das erste Mal in ihrem Leben zum Skifahren. Sie fragt sich, ob sie Ski leihen oder kaufen soll. Man kann für 50,– DM pro Tag ein Paar Skier leihen, andererseits aber auch für 500,– DM ein Paar kaufen.

Frage: Wie ist die kostengünstigste Strategie?

Schwierigkeit: Die Dame weiß nicht, wie oft sie in der Zukunft zum Skifahren gehen wird.

Bahncardproblem Eine Bahncard kann zum Preis von 240,– DM bei der Deutschen Bahn erworben werden. Sie bleibt 12 Monate gültig und ermöglicht es, in diesem Zeitraum Fahrkarten zum halben Preis zu erhalten.

Frage: Wann kauft man eine Bahncard?

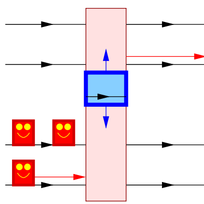
Schwierigkeit: Die Reisen in den nächsten 12 Monaten sind unbekannt.

Eismaschinenoptimierung Eisverkäufer Luigi verkauft an seiner Eisbude Eis in Portionsgrößen zu einem Liter. Er besitzt eine Eismaschine, welche zwei Typen von Eis produzieren kann: Vanille (V) und Schokolade (S). Die Maschine besitzt zwei Modi V und S und kann in einem Modus nur den entsprechen Typ Eis erzeugen. Das Umschalten der Modi erfordert eine Reinigung, die für Luigi Kostenaufwand von DM 1 bedeutet. Im Moduls V produziert die Maschine einen Liter Vanilleeis mit Kosten DM 1, im Modus S kann die Maschine einen Liter Schokoeis zu Kosten DM 2 erzeugen. Wenn Luigi einen Auftrag für einen Liter Eis erhält, kann er

entweder die Maschine benutzen, oder das Eis manuell produzieren, wobei ihm Kosten DM 2 für Vanilleeis und DM 4 für Schokoladeneis entstehen.

Frage: Wie produziert Luigi am günstigsten die angeforderten Eismengen?

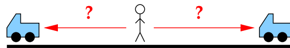
Schwierigkeit: Die zukünftigen Eisbestellungen sind nicht bekannt. Luigi erhält eine Bestellung (aus der Schlange, die vor seiner Bude ansteht) und muß diese bearbeiten, bevor er die nächste erhält.



Aufzugsteuerung Ein Aufzug kann eine Anzahl Personen zwischen den Stockwerken transportieren.

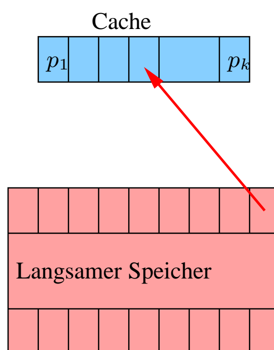
Frage: Wie steuert man den Aufzug optimal?

Schwierigkeit: Zukünftige Fahraufträge sind nicht bekannt. Was bedeutet eigentlich „optimal steuern" (maximale Wartezeit minimieren, Gesamtbearbeitungszeit minimieren, ...)?



Autosuche Ein Student hat sein Auto auf der Straße des 17. Juni geparkt, kann sich aber nicht mehr daran erinnern, ob rechts oder links von seinem aktuellen Standort.

Frage: Wie findet der Student das Auto und läuft dabei möglichst wenig?



Paging Das Pagingproblem ist ein fundamentales Problem der Online-Optimierung. Gegeben ist ein Speichersystem mit zwei Ebenen: dem schnellen Speicher (Cache) für k Speicherseiten und dem langsamen Speicher für N Speicherseiten. Diese N Seiten repräsentieren die virtuellen Speicherseiten. Ein *Paging Algorithmus* wird mit einer Folge von Anfragen für virtuelle Seiten konfrontiert. Wenn die angefragte Seite im Cache ist (»Cache Hit«), entstehen ihm keine Kosten. Wenn die Seite nicht im Cache ist (»Cache Miss«), dann muß der Algorithmus die Seite in den Cache bringen, was ihm Kosten 1 einträgt. Der Algorithmus muß dabei dann auch entscheiden, welche der aktuellen k Seiten im Cache er durch die neue Seite ersetzt.

Frage: Wie minimiert man die Anzahl der Seitenfehler?

Schwierigkeit: Zukünftige Seitenanfragen sind nicht bekannt.

Worst-Case Analyse

Average-Case Analyse

Wie soll man die Qualität eines Online Algorithmus bewerten? Eine „klassische" *Worst-Case Analyse* ist in der Regel sinnlos. Beispielsweise hat beim Paging *jeder* Algorithmus im Worst-Case bei *jeder* Anfrage einen Seitenfehler. Eine Alternative wäre die *Average-Case Analyse*. Das Problem hierbei ist aber, daß man ein statistisches Modell für den Input benötigt. Im allgemeinen kennt man für ein bestimmtes Problem kaum Verteilungen, welche die Realität zufriedenstellend widerspiegeln und auch in der Theorie zufriedenstellend analysierbar sind.

kompetitive Analyse

Motiviert durch diese Beobachtungen schlugen Sleator und Tarjan [30] die *kompetitive Analyse* vor. Bei der kompetitiven Analyse vergleicht man die Lösung

eines Online-Algorithmus relativ zu der eines optimalen Offline Algorithmus. Der maximale Quotient

$$\max_{\sigma}\left\{\frac{C_{\mathsf{ALG}}(\sigma)}{C_{\mathsf{OPT}}(\sigma)}, \frac{C_{\mathsf{OPT}}(\sigma)}{C_{\mathsf{ALG}}(\sigma)}\right\}$$

aus den Kosten des Online-Algorithmus ALG und denen des optimalen Offline-Algorithmus OPT (welcher die Sequenz σ „im Voraus kennt“) ist dann ein Maßstab zur Beurteilung der Qualität von ALG.

Teil I dieses Skripts beschäftigt sich mit der kompetitiven Analyse. Anhand des klassischen Paging-Problems werden grundlegende Techniken und Ergebnisse vorgestellt. Danach untersuchen wir mit Hilfe der kompetitiven Analyse Online-Probleme, die sich aus der Telekommunikation oder der Logistik motivieren.

Die kompetitive Analyse ist eine »Worst-Case Analyse«, die in vielen Fällen übermäßig pessimistisch ist. In Teil II behandeln wir die praktische Seite der Online-Optimierung. Wir zeigen auf, welche Schwierigkeiten bei realen Problemen entstehen und welche Lösungsmöglichkeiten sich anbieten.

1.2 Literatur

Bücher zum Thema sind:

Ref. Nr.	Buch	Preis
[10]	A. Borodin, R. El-Yaniv. *Online Computation and Competitive Analysis*.	90,– DM
[14]	A. Fiat and G. J. Woeginger (eds.). *Online Algorithms: The State of the Art*.	70,– DM
[23]	D. Hochbaum (ed.). *Approximation Algorithms for* NP*-hard Problems*.	80,– DM

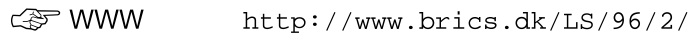
Bei [10] handelt es sich um ein Lehrbuch über kompetitive Analyse. In [14] findet man einen Überblick über die aktuellen theoretischen Ergebnisse in verschiedenen Bereichen der Online-Optimierung. Das Buch [23] beschäftigt sich eigentlich mit polynomialen Approximationsalgorithmen für NP-harte Optimierungsprobleme. In Kapitel 13 findet man aber eine kurze Einführung in die kompetitive Analyse von Online-Algorithmen.

1.3 Weitere Informationsquellen

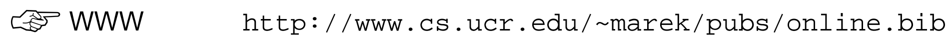
Zur kompetitiven Analyse gibt es Notizen zu einer Vorlesung von Yair Bartal in Berkeley (*Y. Bartal's course on online computation*) unter:

`http://www.icsi.berkeley.edu/~yairb/courses/on-line/on-line-course.html`

Weiterhin gibt es zu einem Kurs von Susanne Albers am BRICS in Dänemark ein kurzes Skript (*Susanne Albers' lecture notes on competitive online algorithms*):

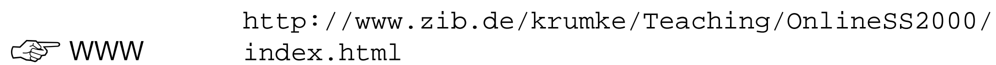


Eine Literaturliste zu Online-Algorithmen wird von Marek Chrobak in Riverside, CA, USA gewartet:



1.4 WWW-Seiten zur Vorlesung

Die WWW-Seiten zur Vorlesung sind zu erreichen unter



Wir werden uns bemühen, die Seiten möglichst aktuell zu halten.

2

Grundbegriffe

2.1 Online und Offline Optimierung

Referenzwerke: [19, 28, 10, 18]

Um die Eingabe für ein bestimmtes »Problem« für einen Computer darzustellen, muß man sie geeignet »codieren«. Die Laufzeit eines Algorithmus mißt man dann in Abhängigkeit der Eingabegröße. Eine ausführliche Behandlung von Codierungsschemata, formalen »Problemen« und Komplexität findet man etwa in [19, 28, 18].

Im folgenden geben wir einen kurzen Einblick in die für dieses Skript benötigten Begriffe. Im Anhang B findet man weitere Informationen über die Komplexität von Algorithmen.

Die Codierung der Eingabe erfolg dadurch, daß man die Eingabedaten durch Zeichenketten über einem Alphabet darstellt. Ein *Alphabet* Σ ist eine endliche Menge von Symbolen. Mit Σ^* bezeichnen wir die Menge aller endlichen Zeichenketten über Σ. Oft ist $\Sigma = \{0, 1\}$ und damit Σ^* die Menge aller (Codierungen von) Binärzahlen.

Alphabet

Man geht davon aus, daß Eingaben redundanzfrei und »vernünftig« codiert werden. Insbesondere werden im Normalfall Zahlen binär codiert. Das bedeutet, daß die *Codierungslänge* (d.h. Anzahl der Zeichen aus Σ die zur Darstellung nötig sind) einer natürlichen Zahl n dann $\lceil \log_2 n \rceil + 1$ sind.

Definition 2.1 (Optimierungsproblem)
Ein ***Optimierungsproblem*** *(über einem Alphabet Σ) ist ein Quadrupel $(\mathcal{I}, F, C, M)$ mit folgenden Bedeutungen:*

Optimierungsproblem

- $\mathcal{I} \subseteq \Sigma^*$ *ist die Menge von Eingabe-Instanzen (Inputs);*
- $F(I) \subseteq \Sigma^*$ *ist die Menge zulässiger Lösungen (Outputs) für die Instanz* $I \in \mathcal{I}$*;*
- $C\colon \mathcal{I} \times \Sigma^* \to \mathbb{R}_{\geq 0}$ *ist die Zielfunktion, d.h.* $C(I, O)$ *bezeichnet die Kosten der Lösung* O *bei Eingabe von* I*. Der Wert* $C(I, O)$ *ist nur dann defi niert, wenn* $O \in F(I)$*;*
- $M \in \{\min, \max\}$ *gibt an, ob es sich bei dem Problem um ein Minimierungsproblem oder ein Maximierungsproblem handelt.*

Ein Algorithmus ALG für ein Optimierungsproblem berechnet bei Eingabe einer gültigen Instanz I eine Lösung $\text{ALG}[I] \in F(I)$. Die Kosten für diese Lösung bezeichnen wir mit $\text{ALG}(I) := C(I, \text{ALG}[I])$. Ein **optimaler (Offline-) Algorithmus** OPT ist ein Algorithmus mit

optimaler (Offline-) Algorithmus

$$\text{OPT}(I) = \underset{O \in F(I)}{M} C(I, O) \qquad \text{für alle } I \in \mathcal{I},$$

wobei in der obigen Definition $M = \min$ für ein Minimierungsproblem und $M = \max$ für ein Maximierungsproblem gilt.

$\langle n \rangle$: Binärcod. von $n \in \mathbb{N}$

Wir betrachten im Rest dieses Kapitels nur Minimierungsprobleme. Die Definitionen und Ergebnisse übertragen sich jedoch in naheliegender Weise auch auf Maximierungsprobleme (siehe Übung 2.1).

$\mathcal{I} = \{ \langle n \rangle : n \in \mathbb{N} \}$
$F(\langle n \rangle) = \{ \langle i \rangle : i \in \mathbb{N} \}$
$C(\langle n \rangle, \langle i \rangle) = \begin{cases} 50(i-1) + 500 & (i < n) \\ 50n & (i \geq n) \end{cases}$
$M = \min$

Beim *Offline-Skifahrerproblem* (der Offline-Variante des Skifahrerproblems aus Abschnitt 1.1) ist die Anzahl n der Tage, an denen Ski gefahren wird, bereits zu Anfang bekannt. Eine Eingabeinstanz besteht aus einer Codierung der Zahl n. Eine zulässige Lösung ist die Codierung der Tageszahl i, an dem Skier gekauft werden (hierbei sei $+\infty$ für »niemals kaufen« zugelassen). Die Kosten sind dann $50(i-1) + 500$ falls $i < n$ und $50n$ falls $i \geq n$.

Online-Problem

Online-Algorithmus

Informell ist ein **Online-Problem** ein Optimierungsproblem, bei dem jede Instanz als Folge $\sigma = r_1, \ldots, r_n$ von Anfragen (Requests) gegeben ist. Ein **Online-Algorithmus** ALG, der die Folge σ bearbeitet, muß nach jeder Anfrage eine Antwort generieren, so daß die von ihm generierte Lösung eine Folge $\text{ALG}[\sigma] = a_1, a_2, \ldots, a_n$ ist. Die Antwort a_i darf dabei nur von den Anfragen $r_1, \ldots, r_i$ abhängen.

Wir formalisieren die obige Definition mit Hilfe von *Frage-Antwort Spielen* (*Request-Answer Games*).

Frage-Antwort Spiel

Definition 2.2 (Frage-Antwort Spiel)
Ein ***Frage-Antwort Spiel*** $(R, \mathcal{A}, \mathcal{C})$ *besteht aus einer Anfragemenge* R*, einer Folge von nichtleeren Antwortmengen* $\mathcal{A} = A_1, A_2, \ldots$ *und einer Folge von Kostenfunktionen* $\mathcal{C} = C_1, C_2, \ldots$*, wobei* $C_j \colon R^j \times A_1 \times \cdots \times A_j \to \mathbb{R}_+ \cup \{+\infty\}$.

Definition 2.3 (Deterministischer Online Algorithmus)
Ein ***deterministischer Online Algorithmus*** *für das Frage-Antwort Spiel* $(R, \mathcal{A}, \mathcal{C})$ *ist eine Folge von Funktionen* $f_j \colon R^j \to A_j$, $j \in \mathbb{N}$. *Die* ***Ausgabe*** *von* ALG *bei der Eingabesequenz* $\sigma = r_1, \ldots, r_n$ *ist dann*

$$\text{ALG}[\sigma] := (a_1, \ldots, a_m) \in A_1 \times \cdots \times A_m, \qquad \text{wobei } a_j := f_j(r_1, \ldots, r_j).$$

Kosten

Die ***Kosten*** *von* ALG *auf* σ*, bezeichnet mit* $\text{ALG}(\sigma)$*, defi nieren wir als:*

$$\text{ALG}(\sigma) := C_m(\sigma, \text{ALG}[\sigma]).$$

Wir betrachten nun das Online-Skifahrerproblem aus Abschnitt 1.1 etwas allgemeiner mit Leihkosten 1 und Kaufkosten B. Jede Anfrage σ_i ist in diesem Fall einfach nur ein Symbol, welches „Skifahren" bedeutet. Die Länge n der Folge σ ist die (für den Online-Algorithmus unbekannte und für den Offline Algorithmus bekannte) Anzahl von Tagen, an denen Ski gefahren wird.

$\sigma = \overbrace{(1, 1, \ldots, 1)}^{n}$

Wie sieht ein Online-Algorithmus für das Skifahrerproblem aus? Ein Online-Algorithmus muß jeden Tag entscheiden, ob er an diesem Tag ein Paar Skier kauft oder nicht (sofern er noch nicht gekauft hat). Ein generischer (deterministischer) Online-Algorithmus ALG leiht i mal Skier und kauft dann am $(i+1)$-ten Tag ($i = +\infty$ ist hierbei mit der offensichtlichen Bedeutung, daß niemals gekauft wird, zugelassen). Im Fall des Skifahrerproblems kennen wir also *alle* Online Algorithmen. Die Kosten von ALG bei Eingabe von $\sigma = r_1, \dots, r_n$ sind:

$$\mathsf{ALG}(\sigma) = \begin{cases} n & \text{, falls } n \leq i, \\ i + B & \text{, falls } n > i. \end{cases}$$

2.2 Kompetitive Algorithmen

Referenzwerke: [10, 14]

Definition 2.4 (c-kompetitiver Algorithmus, Kompetitivität)
Ein (deterministischer) Online-Algorithmus ALG *ist* ***c-kompetitiv****, wenn es eine Konstante α gibt, so daß für alle (endlichen) Eingabefolgen σ gilt:*

c-kompetitiv

$$\mathsf{ALG}(\sigma) \leq c \cdot \mathsf{OPT}(\sigma) + \alpha.$$

Kann $\alpha = 0$ gewählt werden, so nennt man ALG ***strikt c-kompetitiv****.*

strikt c-kompetitiv

Die ***Kompetitivität (competitiveness)*** *oder der* ***kompetitive Faktor (competitive ratio)*** *des Algorithmus* ALG *ist das Infimum über alle c, so daß* ALG *c-kompetitiv ist.*

Kompetitivität

In der Literatur steht der Ausdruck *kompetitiver Algorithmus* oft für einen c-kompetitiven Algorithmus, bei dem c eine Konstante ist.

Man beachte, daß wir bei der kompetitiven Analyse keinerlei Beschränkungen über die Rechenzeit des Online-Algorithmus machen. Die »knappe Ressource« eines Online-Algorithmus bei der kompetitiven Analyse ist die Information über die zukünftgen Anfragen.

Die kompetitive Analyse von Online-Algorithmen kann man als »Spiel« zwischen einem *Online-Spieler* und einem böswilligen Gegner (Adversary) sehen. Der Online-Spieler arbeitet mit einem Online-Algorithmus auf einer Eingabe, die vom Adversary vorgegeben wird. Der Adversary konstruiert, basierend auf seinem Wissen über den Online-Algorithmus, eine Eingabe, die den Quotienten aus den Online- und den Offline-Kosten maximiert. Manchmal bezeichnet man den Adversary zusammen mit den optimalen Offline-Algorithmus auch als *Offline-Spieler*. Im Fall von deterministischen Online-Algorithmen hat der Adversary komplette Kenntnis über den Online-Spieler. Bei randomisierten Online-Algorithmen gibt es hier wichtige Unterschiede (siehe Abschnitt 2.3).

Online-Spieler

Gegner (Adversary)

Offline-Spieler

2.2.1 Ein Beispiel

Als Beispiel für die kompetitive Analyse betrachten wir wieder das Skifahrerproblem aus Abschnitt 1.1 mit Leihkosten 1 und Kaufkosten B. Die optimale

Offline-Lösung bei n Tagen Skilaufen besteht darin, Skier zu leihen, falls $n < B$ gilt, und ansonsten Skier für B DM zu kaufen. Somit ist

$$\mathsf{OPT}(r_1, \ldots, r_n) = \min\{n, B\}$$

Wir betrachten den Online-Algorithmus, der i mal Skier leiht und dann ein Paar kauft. Wenn wir $i = B - 1$ wählen, ist der entsprechende Algorithmus $(2 - 1/B)$-kompetitiv: Falls $n \leq B - 1$, dann ist $\mathsf{ALG}(\sigma) = n = \mathsf{OPT}(\sigma)$. Wenn $n > B - 1$, dann ist

$$\frac{\mathsf{ALG}(\sigma)}{\mathsf{OPT}(\sigma)} = \frac{B - 1 + B}{B} = 2 - \frac{1}{B}.$$

Kann man eine bessere Kompetitivität (durch einen deterministischen Algorithmus) erreichen? Wir zeigen, daß dies nicht der Fall ist.

Sei dazu ALG ein beliebiger deterministischer Online-Algorithmus für das Skifahrerproblem. Der Algorithmus kaufe am $i$ten Tag Skier. Wenn der Algorithmus kompetitiv sein will, dann muß er irgendwann Skier kaufen, d.h. es muß $i < +\infty$ gelten. Der Adversary wählt $n = i + 1$, d.h. er wartet, bis ALG ein Paar Skier kauft, und beendet dann das Skifahren. Der Online-Algorithmus hat also Kosten i für das Leihen und B für das Kaufen, also insgesamt $\mathsf{ALG}(\sigma) = i + B$. Der Offline Adversary hat Kosten $\mathsf{OPT}(\sigma) = \min\{i + 1, B\}$.

Wenn $i \leq B - 1$ ist, dann gilt:

$$\frac{\mathsf{ALG}(\sigma)}{\mathsf{OPT}(\sigma)} = \frac{i + B}{\min\{i + 1, B\}} = \frac{i + B}{i + 1} = 1 + \frac{B - 1}{i + 1} \geq 1 + \frac{B - 1}{B} = 2 - \frac{1}{B}.$$

Wenn $i \geq B - 1$ ist, dann haben wir

$$\frac{\mathsf{ALG}(\sigma)}{\mathsf{OPT}(\sigma)} = \frac{i + B}{\min\{i + 1, B\}} \geq \frac{2B - 1}{B} = 2 - \frac{1}{B}.$$

2.3 Randomisierte Algorithmen

Referenzwerke: [10, 14]

Für randomisierte Algorithmen benutzen wir folgende einfache »Definition«. Eine formalere Definition von randomisierten Algorithmen findet man in [27], speziell für Online-Algorithmen sind die Unterschiede zwischen einzelnen (spieltheoretischen) Modellen für Algorithmen in [10] dargelegt.

Definition 2.5 (Randomisierter Online-Algorithmus)
Ein ***randomisierter Online-Algorithmus*** *ist ein Algorithmus, welcher Zufallsentscheidungen benutzt, während er auf Anfragen reagiert.*

Bei deterministischen Online-Algorithmen besitzt der Gegner vollständige Information über den Online-Spieler. Er kann dieses Wissen benutzen, um eine für den Online-Spieler möglichst schlechte und für ihn möglichst gute Eingabefolge zu konstruieren. Bei randomisierten Algorithmen müssen wir entscheiden, wieviel Information der Gegner über den Online-Algorithmus hat.

oblivious Adversary

Blinder Gegner Der blinde Gegner (oblivious Adversary) muß die komplette Eingabefolge im Voraus wählen. Dabei hat er kein Wissen über den Ausgang der Zufallsentscheidungen des Online-Algorithmus. Er kennt allerdings den Online-Algorithmus und die von ihm benutzten Wahrscheinlichkeitsverteilungen.

adaptive Adversary

Adaptiver Gegner Der adaptive Gegner (adaptive Adversary) kann jede Anfrage mit Wissen aller bisherigen Aktionen des Online-Algorithmus einschließlich des Ausgangs aller Zufallsentscheidungen treffen.

Bei den adaptiven Gegnern unterscheidet man zwischen zwei Typen, je nachdem, wie der Gegner die Eingabefolge selbst bearbeiten muß.

Adaptiver Offline Gegner Der *adaptive Offline-Gegner* bearbeitet die von ihm generierte Folge im Nachhinein mit dem optimalen Offline Algorithmus. Seine Kosten für die Sequenz σ sind also $\mathsf{OPT}(\sigma)$.

Adaptiver Online Gegner Der *adaptive Online-Gegner* muß die Folge selbst online bearbeiten.

Beim blinden Gegner macht die obige Unterscheidung keinen Sinn, da er sowieso die komplette Folge im Voraus wählen muß. Seine Kosten bei der von ihm generierten Folge σ entsprechen den optimalen Offline Kosten $\mathsf{OPT}(\sigma)$.

Wir haben also insgesamt drei Typen von Gegnern: den blinden Gegner OBL, den adaptiven online Gegner ADON und den adaptiven Offline Gegner ADOFF.

Definition 2.6 (Randomisierter c-kompetitiver Algorithmus)
Ein randomisierter Online-Algorithmus ALG *ist c-**kompetitiv** gegen einen Gegner vom Typ* ADV *wenn es eine Konstante α gibt, so daß für alle (endlichen) Eingabefolgen σ gilt:*

$$\mathbb{E}[\mathsf{ALG}(\sigma) - c \cdot \mathsf{ADV}(\sigma)] \leq \alpha.$$

Hierbei wird der Erwartungswert über alle Zufallsentscheidungen des Online-Algorithmus ALG *genommen.*

Man beachte, daß $\mathsf{OBL}(\sigma)$ und $\mathsf{ADOFF}(\sigma)$ beide gleich $\mathsf{OPT}(\sigma)$ sind, es aber dennoch einen wichtigen Unterschied gibt. Während $\mathsf{OBL}(\sigma)$ ein fester Wert ist (weil die vom blinden Gegner gewählte Sequenz nicht von den Zufallsentscheidungen des Online-Algorithmus abhängt), ist $\mathsf{ADOFF}(\sigma)$ eine *Zufallsvariable*. Beim adaptiven Gegner hängt σ von den Zufallsentscheidungen des Online-Algorithmus ab. Beide Werte, $\mathsf{ADOFF}(\sigma)$ und $\mathsf{ADON}(\sigma)$, sind Zufallsvariablen.

Da beim blinden Gegner $\mathsf{OBL}(\sigma)$ nicht zufällig ist, kann man $\mathsf{OBL}(\sigma)$ in der letzten Definition aus dem Erwartungswert herausziehen. Ein Online-Algorithmus ist also c-kompetitiv gegen einen blinden Gegner, wenn gilt:

$$\mathbb{E}[\mathsf{ALG}(\sigma)] \leq c \cdot \mathsf{OPT}(\sigma) + \alpha.$$

2.3.1 Ein Beispiel

Wir betrachten wieder das Skifahrerproblem aus Abschnitt 1.1 mit Leihkosten 1 und Kaufkosten B. Wir definieren für $i = 0, \dots, B-1$ den Algorithmus ALG_i als denjenigen deterministischen Algorithmus, der i mal Skier leiht und dann am $(i+1)$ten Tag Skier kauft. Seien

$$\rho = \frac{B}{B-1} \qquad \text{und} \qquad \alpha = \frac{\rho - 1}{\rho^B - 1}.$$

Unser randomisierter Algorithmus $\mathsf{RANDSKI}$ wählt nun zufällig eine Zahl $i \in \{0, 1, \dots, B-1\}$. Dabei ist die Wahrscheinlichkeit p_i, daß die Zahl i gewählt wird wie folgt gegeben:

$$p_i = \alpha \rho^i, \quad \text{für } i = 0, \dots, B-1.$$

Man rechnet leicht nach, daß $\sum_{i=0}^{B-1} p_i = 1$ gilt. $\mathsf{RANDSKI}$ benutzt ALG_i, um die vom Adversary vorgegebene Sequenz σ zu bearbeiten. $\mathsf{RANDSKI}$ benötigt also nur eine einzige Zufallsentscheidung. Nach dieser arbeitet er deterministisch. Seine »Stärke« zieht $\mathsf{RANDSKI}$ daraus, daß der blinde Gegner nicht weiß, wie die anfängliche Zufallsentscheidung ausgegangen ist. Man beachte, daß ALG definitiv *vor* der Bten Anfrage ein Paar Skier gekauft hat.

Wir analysieren die Kompetitivität von ALG gegen den blinden Gegner OBL. Sei dazu $\sigma = r_1, \dots, r_n$ eine beliebige Anfragefolge. Falls $n > B$, so sind die Kosten von ALG auf σ die gleichen, wie auf der Teilfolge σ', die nur aus den ersten B Anfragen besteht (da ALG spätestens am Bten Tag Skier kauft). Da $\mathsf{OPT}(\sigma') \leq \mathsf{OPT}(\sigma)$ können wir also annehmen, daß $n \leq B$ gilt. In diesem Fall haben wir $\mathsf{OPT}(\sigma) = n$.

Wie groß sind die erwarteten Kosten von ALG? Für $0 \leq i < n$ sind die Kosten von Algorithmus ALG_i genau $\mathsf{ALG}_i(\sigma) = i + B$. Für $i \geq n$ gilt $\mathsf{ALG}_i(\sigma) = n$. Somit ergibt sich

$$\begin{aligned}
\mathbb{E}[\mathsf{RANDSKI}(\sigma)] &= \sum_{i=0}^{n-1} p_i(i+B) + \sum_{i=n}^{B-1} p_i n \\
&= \sum_{i=0}^{n-1} \alpha\rho^i(i+B) + \sum_{i=n}^{B-1} \alpha\rho^i n \\
&= \alpha \sum_{i=0}^{n-1} \rho^i i + \alpha B \sum_{i=0}^{n-1} \rho^i + \alpha n \sum_{i=n}^{B-1} \rho^i \\
&= \alpha \frac{(n-1)\rho^{n+1} - n\rho^n + \rho}{(\rho-1)^2} + \alpha B \frac{\rho^n - 1}{\rho - 1} + \alpha n \frac{\rho^B - \rho^n}{\rho - 1}
\end{aligned}$$

(benutze nun $\rho - 1 = 1/B$)

$$
\begin{aligned}
&= \frac{\alpha}{(\rho-1)^2}\left((n-1)\rho^{n+1} - n\rho^n + \rho + \rho^n - 1 + n(\rho-1)(\rho^B - \rho^n)\right) \\
&= \frac{\alpha}{(\rho-1)^2}\left(n\rho^{B+1} - n\rho^B - \rho^{n+1} + \rho^n + \rho - 1\right) \\
&= \frac{\alpha}{(\rho-1)^2}\left((\rho-1)n\rho^B - (\rho-1)(-\rho^n + 1)\right) \\
&= \frac{\alpha}{\rho-1}\left(n\rho^B - \rho^n + 1\right) \\
&\le \frac{\alpha}{\rho-1} \cdot n\rho^B \qquad \text{(da } \rho \ge 1\text{)} \\
&= \frac{\rho^B}{\rho^B - 1} \cdot n \\
&= \frac{\rho^B}{\rho^B - 1} \cdot \mathsf{OPT}(\sigma).
\end{aligned}
$$

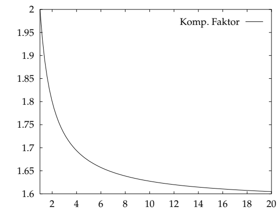
Somit haben wir gezeigt, daß RANDSKI gegen OBL c_B-kompetitiv ist mit

$$c_B = \frac{\rho^B}{\rho^B - 1} = \frac{\left(\frac{B+1}{B}\right)^B}{\left(\frac{B+1}{B}\right)^B - 1}.$$

Man beachte, daß c_B streng monoton fällt, wenn B wächst, und ferner

$$\lim_{B\to\infty} c_B = \frac{e}{e-1} \approx 1.58$$

gilt. Für vernünftig großes B ist der Faktor c_B strikt kleiner als der beste kompetitive Faktor von $2 - 1/B$, den wir durch einen deterministischen Algorithmus erreichen können.



Kompetitivität c_B von RANDSKI in Abhängigkeit von B.

Wir haben eben gesehen, daß wir durch Randomisieren die untere Schranke für deterministische Algorithmen zumindest gegen den blinden Gegner schlagen können. Wie sieht die Kompetitivität von RANDSKI gegenüber dem adaptiven Offline Gegner aus? Hier zeigt die Konstruktion aus Abschnitt 2.2.1, daß RANDSKI auch nur $(2 - 1/B)$-kompetitiv ist: Der adaptive Gegner kann sehen, welchen Algorithmus ALG_i RANDSKI am Anfang wählt. Da ALG_i deterministisch ist, kann der Gegner gemäß Abschnitt 2.2.1 dann eine Sequenz wählen, so daß die Kosten von ALG_i mindestens $(2 - 1/B)$-mal so groß wie die von OPT sind.

Die obigen Argumente zeigen auch, daß *kein* randomisierter Algorithmus ALG für das Skifahrerproblem besser als $(2 - 1/B)$-kompetitiv sein kann: Der adaptive Gegner wartet einfach, bis der Algorithmus ALG kauft. Dann beendet er das Skifahren. Die Rechnung aus Abschnitt 2.2.1 zeigt dann wieder, daß ALG mindestens Kosten von $(2 - 1/B)$-mal die optimalen Offline Kosten hat.

Im Falle des Skifahrerproblems hilft Randomisieren gegen den adaptiven Gegner nichts. Der folgende Satz zeigt, daß dies kein Einzelfall ist.

Satz 2.7 *Sei Π ein Online Problem und* ALG *ein randomisierter Algorithmus für Π mit Kompetitivität c gegen den adaptiven Offline-Gegner. Dann gibt es auch einen deterministischen c-kompetitiven Algorithmus für Π.*

Beweis: Siehe [10, Kapitel 7]. □

Wir schließen das Kapitel mit einem Satz, der die verschiedenen Gegner in Relation setzt.

Satz 2.8 *Sei* Π *ein Online Problem und* ALG *ein Online Algorithmus für* Π. *Mit* $\bar{R}_{\mathsf{ADV}}(\mathsf{ALG})$ *bezeichnen wir die Kompetitivität von* ALG *gegen einen Gegner vom Typ* ADV. *Dann gilt:*

$$\bar{R}_{\mathsf{OBL}}(\mathsf{ALG}) \leq \bar{R}_{\mathsf{ADON}}(\mathsf{ALG}) \leq \bar{R}_{\mathsf{ADOFF}}(\mathsf{ALG}).$$

Beweis: Siehe [10, Kapitel 7]. □

Übungsaufgaben

Übung 2.1 (Kompetitive Algorithmen bei Maximierungsproblemen)
Geben Sie geeignete Definitionen der Kompetitivität eines Algorithmus für Maximierungsprobleme. Übertragen Sie Definition 2.4 auf Seite 9 für deterministische und Definition 2.6 auf Seite 11 für randomisierte Algorithmen. Bei Ihrer Definition sollte analog zu den Minimierungsproblemen ein c-kompetitiver Algorithmus umso »besser« sein, je kleiner c ist.

Übung 2.2 (Eigenes Online-Problem)
Formulieren Sie ein eigenes Online-Problem aus Ihrem Interessengebiet oder aus dem täglichen Leben. Geben Sie mindestens zwei Online-Algorithmen für das Problem an. Versuchen Sie, diese Algorithmen mit Hilfe der kompetitiven Analyse zu bewerten.

Übung 2.3 (Dynamische Arrayverwaltung)
Zur Speicherung eines dynamisch wachsenden Arrays soll Speicherplatz alloziiert werden. Speicherplatz steht in Form von Blöcken zur Verfügung. Das Array muß in aufeinanderfolgenden Speicheradressen untergebracht werden. Eine Anfrage r_i bedeutet, daß das Array auf r_i Speicherplätze angewachsen ist. Wenn der aktuelle Speicherblock weniger als r_i Speicheradressen enthält, muß ein komplett neuer Speicherblock mit mindestens r_i Adressen belegt werden. Der alte Speicherblock wird dann vollständig verworfen. Das Alloziieren eines Speicherblocks mit Größe s kostet s Einheiten.

Wir nehmen an, daß in der Eingabefolge $\sigma = r_1, \ldots, r_n$ eine Ordnung $r_1 \leq \cdots \leq r_n$ vorliegt. Die optimalen Offline-Kosten für σ sind dann $\mathsf{OPT}(\sigma) = r_n$.

(a) Geben Sie einen kompetitiven Algorithmus für die dynamische Arrayverwaltung mit möglichst guter Kompetitivität ($c \leq 4$) an.

(b) Beweisen Sie, daß jeder c-kompetitive deterministische Algorithmus $c \geq 2$ erfüllt.

Übung 2.4 (Bin Packing)
Beim *Bin Packing* sollen Gegenstände in möglichst wenige Kisten verpackt werden. Jeder Gegenstand i hat eine Größe $r_i \in [0, 1]$. Eine Kiste kann Gegenstände mit Gesamtvolumen höchstens 1 aufnehmen. Beim Online Bin Packing muß der Gegenstand r_i verpackt werden, bevor der nächste Gegenstand r_{i+1} bekannt wird. Einmal verpackte Gegenstände dürfen nicht umverpackt werden.

(a) Zeigen Sie, daß jeder c-kompetitive deterministische Algorithmus für das Online Bin Packing $c \geq 4/3$ erfüllt.

(b) Der FIRSTFIT-Algorithmus verpackt den Gegenstand r_i immer in die erste Kiste, die noch genügend Platz hat. Wenn keine Kiste mehr genügend Freiraum hat, öffnet er eine neue Kiste. Beweisen Sie, daß FIRSTFIT 2-kompetitiv ist.

Übung 2.5 (Online Graph-Matching)
In dieser Aufgabe soll eine Online-Variante des *Matching Problems* vorgestellt und analysiert werden. Sei dazu $G = (H \cup D, R)$ ein bipartiter Graph, d.h. jeder Pfeil $r \in R$ ist von der Form $r = (h, d)$ mit $h \in H$ und $d \in D$ und $H \cap D = \emptyset$.

Sei $G = (V, E)$ ein ungerichteter Graph. Ein *Matching* ist dann eine Teilmenge $M \subseteq E$, so daß keine zwei Kanten aus M miteinander inzidieren. Ein Matching heißt *maximal*, wenn man keinen Pfeil mehr hinzufügen kann, ohne die Matchingeigenschaft zu zerstören. Ein Matching M, bei dem jede Ecke aus V zu einer Kante aus M inzident ist, nennt man ein *perfektes* Matching.

Die Heiratsvermittlung *Online-Matching* hat noch n unverheiratete Herren $H = \{h_1, \ldots, h_n\}$ aus guten Verhältnissen im Angebot. Diese sollen im Rahmen einer Tanzveranstaltung »an die Frau« gebracht werden. Dazu hat die Heiratsvermittlung an n unverheiratete Damen $D = \{d_1, \ldots, d_n\}$ Einladungen verschickt und auf den Einladungen Portraits der n Herren abgebildet. Jede Dame soll nun diejenigen Herren ankreuzen, die ihr gefallen. Am Abend des Balls bringt dann jede Dame die ausgefüllte Karte mit und zeigt diese am Eingang vor.[1] Der Herr am Empfang an der Tür ordnet sie dann einem noch verfügbaren Herren, der auf Ihrer Präferenzliste angekreuzt ist, als Tanzpartnerin für den Abend zu. Ist keiner der entsprechenden Herren mehr frei, so wird die Dame wieder heimgeschickt, und zur Entschädigung für ihre Mühen erhält sie einen finanziellen Ausgleich. Natürlich sollen möglichst viele Paare gebildet werden, damit die Aussichten, einen ledigen Herren zu verheiraten, möglichst groß sind.

Das obige Problem der Partnerzuordnung läßt sich als *Online* Version des Heiratsproblems (perfektes Matching) modellieren. Gegeben sei ein bipartiter Graph $G = (H \cup D, E)$ mit $2n$ Ecken. Wir setzen voraus, daß G ein perfektes Matching besitzt, d.h. daß das Offline Heiratsproblem auf G lösbar ist.

Ein Online-Algorithmus kennt zu Anfang alle Herren H. Eine Anfrage r_i besteht aus der Nachbarschaft $N(d_i) = \{h \in H \; : \; (h, d_i) \in E\}$ des »Damenknotens« d_i. Der Online-Algorithmus muß bei Erhalt der Information zur Dame d_i entscheiden, ob und wenn ja welchem Herren aus $N(d_i)$ er die neu angekommene Dame als Partnerin zuordnet (sofern dies überhaupt noch möglich ist).

[1] Wir nehmen der Einfachheit halber an, daß jede der eingeladenen Damen zur Veranstaltung erscheint.

Die Anfragefolge $\sigma = r_1, \ldots, r_n$ besteht aus einer Permutation der Damen. Die Aufgabe des Algorithmus ist es, möglichst viele Paare zu bilden.

Wir betrachten nun folgenden *sehr simplen* Algorithmus: Wenn eine Dame eintrifft, wird sie *irgendeinem* Herren, der Ihr gefällt und noch frei ist, zugeordnet. Wenn keine Zuordnung möglich ist, wird die Dame heimgeschickt.

(a) Beweisen Sie, daß der simple Algorithmus 2-kompetitiv ist.

(Hinweis: Nehmen Sie an, daß M ein maximales Matching mit $|M| < n/2$ ist. Sei H' die Menge der Herren, die mittels M eine Partnerin bekommen haben. Dann ist $|H'| < n/2$. Benutzen Sie, daß G ein perfektes Matching besitzt.)

(c) Zeigen Sie, daß *jeder* deterministische Online-Algorithmus (und auch jeder randomisierte Online-Algorithmus gegen einen adaptiven Offline-Gegenspieler) für das obige Problem höchstens 2-kompetitiv sein kann.

Übung 2.6 (Autosuche)
Der Student aus dem Beispiel in Abschnitt 1.1 hat noch nicht mit der Suche nach seinem Auto begonnen. Wir wollen ihm dabei helfen. Dazu modellieren wir die Autosuche als Suche nach einem unbekannten Punkt $a \in \mathbb{R}$. Der Startpunkt unseres Studenten ist der Nullpunkt 0.

Die optimale Offline-Strategie kennt den Parkplatz des Autos a und hat Kosten $|a|$ (Länge des direkten Weges von 0 nach a). Wir nennen eine Suchstrategie für den unbekannten Autostandort a c-kompetitiv, wenn sie den Studenten ausgehend von Nullpunkt so bewegt, daß er dabei eine Wegstrecke von höchstens $c \cdot |a|$ zurücklegt.

Baeza-Yates et al. [9] haben gezeigt, daß kein Algorithmus besser als 9-kompetitiv sein kann. Ziel der Aufgabe ist es nun, einen 9-kompetitiven (und somit bestmöglichen) Algorithmus zu konstruieren.

Hinweis: Betrachten Sie den Algorithmus, der zunächst $\alpha > 1$ Einheiten nach rechts wandert, dann wieder bis zum Ursprung zurückläuft, danach α^2 Einheiten nach links läuft. Der i-te Wendepunkt des Algorithmus ist dann also $(-1)^{i+1}\alpha^i$. Wie sieht die Kompetitivität des Algorithmus aus? Wie soll man α wählen?

3

Beispiele und elementare Techniken

Referenzwerke: [10, 12, 15]

In diesem Kapitel werden wir uns mit einfachen Online-Problemen, unter anderem mit Erweiterungen des Skifahrerproblems aus Kapitel 2 beschäftigen. Anhand dieser (hoffentlich motivierenden) Beispiele stellen wir elementare Handgriffe der Analyse von Online-Algorithmen vor:

Potentialfunktionen Diese Technik ist eng mit der *amortisierten Analyse* amortisierte Analyse verwandt und ermöglicht es, gute Abschätzungen für komplette Anfragefolgen zu erhalten.

amortisierte Analyse

Grausamer Adversary Um eine untere Schranke für die Kompetitivität von (deterministischen) Online-Algorithmen zu beweisen, konstruiert man eine Anfragefolge, die in jedem Schritt für den Online-Algorithmus »schlechtestmöglich« ist.

Durchschnittsargument bei unteren Schranken Man vergleicht die Kosten eines Online-Algorithmus mit den durchschnittlichen Kosten einer Menge von (Offline-)Algorithmen.

3.1 Organisation von linearen Listen

Gegeben sei eine lineare Liste L mit endlich vielen Elementen. Diese lineare Liste wird benutzt, um Zugriffsanfragen zu beantworten. Die Kosten, um eine Zugriffsanfrage auf das Element x ist die Position des Elements in der Liste. Dies motiviert sich daraus, daß man eine lineare Liste (bei Nichtvorhandensein von weiteren Hilfstrukturen) von vorne ausgehend durchsuchen muß, um x zu finden. Dazu sind im wesentlichen so viele Operationen nötig, wie die Position, an der x steht. Zu Details über lineare Listen verweisen wir auf das Buch [12].

Ein Algorithmus, der die lineare Liste L organisiert, darf die Liste zu jeder Zeit durch Vertauschen von benachbarten Elementen umordnen. Pro Tausch entstehen ihm Kosten von 1. Direkt nach der Zugriffsanfrage auf das Element x darf der Algorithmus das Element x kostenfrei beliebig weit an den Anfang der Liste

verschieben. Die Motivation hierfür ist die, daß man sich während der sequentiellen Suche einen Pointer zu jeder gewünschten Position weiter vorne »kostenfrei« speichern kann, mit Hilfe dessen man x dann an die neue Position in der Liste einhängen kann.

List-Accessing Problem

Ziel beim *List-Accessing Problem* ist es, die Gesamtkosten bei der Bearbeitung einer Sequenz $\sigma = r_1, \dots, r_n$ von Anfragen $r_i \in L$ zu minimieren. Ein Online-Algorithmus muß die Anfrage r_i bearbeiten, bevor er die Anfrage r_{i+1} erhält.

Wir betrachten folgende Strategien:

Algorithmus MTF (Move to the Front) Nach der Anfrage r_i bewege das Element r_i (durch kostenlose Vertauschungen) an den Anfang der Liste.

Algorithmus TRANS (Transpose) Nach der Anfrage r_i vertausche das Element r_i (durch kostenlose Vertauschung) mit seinem Vorgänger in der Liste (sofern r_i nicht schon am Anfang steht).

Algorithmus FC (Frequency Count) Halte für jedes der m Elemente in der Liste einen Zähler count$[x]$, welcher angibt, wie oft x bereits angefragt wurde. Bei der Anfrage r_i wird der Zähler count$[r_i]$ aktualisiert und die Liste dann sofort gemäß absteigender Zählerwerte umsortiert.

Satz 3.1 MTF *ist* $(2 - 1/m)$*-kompetitiv für das List-Accessing-Problem.*

Potentialfunktion

amortisierte Kosten

Beweis: Zum Beweis des Satzes verwenden wir eine *Potentialfunktion* Φ, welche der aktuellen Listenkonfiguration von MTF und OPT eine nichtnegative Zahl $\Phi_i \geq 0$ zuordnet. Die *amortisierten Kosten*amortisierte Kosten a_i bei der $i$ten Operation sind

$$a_i = \underbrace{c_i}_{\text{Kosten von MTF für } r_i} + \underbrace{\Phi_i - \Phi_{i-1}}_{\text{Potentialänderung}} .$$

Die Intuition hinter eine Potentialfunktion und den amortisierten Kosten ist die folgende: Φ_i mißt, »wie gut« die aktuelle Konfiguration ist. Man kann Φ_i als Bankkonto betrachten. Wenn die Differenz $\Phi_i - \Phi_{i-1}$ negativ ist, dann unterschätzt a_i die tatsächlichen Kosten c_i. Die Differenz wird durch das Entnehmen des Potentialverlustes aus dem Konto abgedeckt. Es gilt nun:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} a_i + (\Phi_0 - \Phi_n) \leq \sum_{i=1}^{n} a_i + \Phi_0. \tag{3.1}$$

Wenn wir also $a_i \leq (2-1/m)\text{OPT}(r_i)$ zeigen könnten und Φ_0 unabhängig von der Sequenz σ ist, so haben wir bewiesen, daß MTF die gewünschte Kompetitivität besitzt.

Wir definieren Φ_i, d.h. den Wert der Potentialfunktion vor Anfrage r_i, wie folgt:

$$\begin{aligned}\Phi_i &:= \text{Anzahl der Inversionen}\\ &= |\{ (a,b) : a \text{ steht vor } b \text{ in der Liste von MTF, aber } b \text{ vor } a \text{ in der von OPT}\}|\end{aligned}$$

Wir stellen uns vor, daß bei der Anfrage r_i zuerst MTF die Liste nach dem Element durchsucht und dann umstellt und danach OPT auf seiner Liste arbeitet.

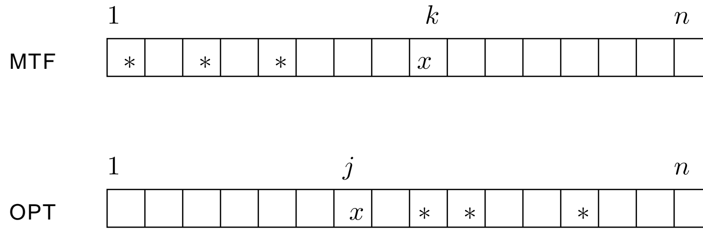


Abbildung 3.1: Die Konfigurationen der Listen von MTF und OPT zum Zeitpunkt der Anfrage $r_i = x$.

Wir betrachten die Listen von MTF und OPT bei der Anfrage $r_i = x$ (siehe Abbildung 3.1). Das Element x stehe in der Liste von MTF an Position k, in der Liste von OPT an Position j. Wir bezeichnen mit p die Anzahl der kostenpflichtigen und mit f die Anzahl der freien Vertauschungen, die OPT nach der Anfrage r_i vornimmt.

Sei v die Anzahl der Elemente, die vor x in der Liste von MTF, aber hinter x in der Liste von OPT stehen (diese Elemente sind in der Abbildung durch Sterne $*$ angedeutet).

Wenn nun MTF das Element x an den Anfang der Liste verschiebt, dann werden dabei v Inversionen aufgehoben und maximal $k-1-v \leq j-1$ neue Inversionen geschaffen (Vor x stehen in der Liste von MTF außer den Elementen, welche die v Inversionen verursachen, noch $k-1-v$ Elemente. Alle diese Elemente stehen in der Liste von OPT vor x, da sie sonst Inversionen induzieren würden).

Die Potentialdifferenz $\Phi_i - \Phi_{i-1}$ ist also vor dem Umordnen von OPT durch $-v + k - 1 - v \leq j - 1 - v$ nach oben beschränkt. Die amortisierten Kosten sind also bis zu diesem Zeitpunkt durch

$$\underbrace{k}_{=c_i} + j - 1 - v = k - 1 - v + j \leq 2j - 1$$

nach oben beschränkt. Durch Umordnen der Liste kann OPT die Potentialfunktion um maximal $p - f$ erhöhen. Somit gilt:

$$a_i \leq 2j - 1 + (p - f) \leq 2(j + p) - 1 = 2\mathsf{OPT}(r_i) - 1. \tag{3.2}$$

Aus (3.2) ergibt sich durch Summation $\sum_{i=1}^{n} a_i \leq 2\mathsf{OPT}(\sigma) - n$. Da $\mathsf{OPT}(\sigma) \leq nm$ ist, folgt

$$\sum_{i=1}^{n} a_i \leq \left(2 - \frac{1}{m}\right) \mathsf{OPT}(\sigma).$$

Anwendung von (3.1) und Ausnutzen von $\Phi_0 = 0$ zeigt nun die Aussage des Satzes. □

Wir zeigen nun eine untere Schranke für die Kompetitivität jedes deterministischen Algorithmus. Dabei benutzen wir eine Technik, die als *Grausamer Adversary* (*Cruel Adversary*) bekannt ist.

Grausamer Adversary

Cruel Adversary

Satz 3.2 *Sei* ALG *ein beliebiger deterministischer Online Algorithmus für das List-Accessing-Problem. Wenn* ALG *c-kompetitiv ist, dann gilt:* $c \geq 2-2/(m+1)$, *wobei m die Länge der Liste ist.*

Beweis: Der grausame Adversary fragt in jedem Schritt das letzte Element der Liste von ALG an: er wählt dabei die Länge n der Anfragefolge beliebig lang. Für den Online-Algorithmus ALG entstehen insgesamt also Kosten

$$\mathsf{ALG}(\sigma) = nm.$$

Wir betrachten nun die $m!$ statischen Offline-Algorithmen, die den Permutationen der m Elemente in der Liste entsprechen. Jeder dieser Algorithmen sortiert anfangs die Liste gemäß der ihm zugeordneten Permutation und läßt die Liste dann für die komplette Anfragesequenz σ fest. Die gesamten Anfangskosten zum Sortieren für alle diese Algorithmen ist durch eine Konstante b beschränkt, die nur von m abhängt.

Wir betrachten die Gesamtkosten der $m!$ Algorithmen bei der Bearbeitung einer Anfrage $r_i = x$. Für jede der m möglichen Positionen von x gibt es genau $(m-1)!$ Permutationen, welche x an dieser Position haben. Also sind die Gesamtkosten

$$\sum_{j=1}^{m} j(m-1)! = (m-1)!\frac{m(m+1)}{2}.$$

Damit sind die Gesamtkosten auf σ für die $m!$ statischen Algorithmen maximal:

$$n(m-1)!\frac{m(m+1)}{2} + m!b.$$

Der beste der statischen Algorithmen hat Kosten, die höchstens so groß wie der Durchschnitt

$$\frac{1}{m!}\left(n(m-1)!\frac{m(m+1)}{2} + m!b\right) = n\frac{m+1}{2} + b.$$

Somit folgt:

$$\frac{\mathsf{ALG}(\sigma)}{\mathsf{OPT}(\sigma)} \geq \frac{nm}{n\frac{m+1}{2}+b} = \frac{2m}{m+1+2b/n} \overset{n\to\infty}{\to} \frac{2m}{m+1} = \left(2 - \frac{2}{m}\right).$$

□

Was ist mit den Algorithmen TRANS und FC? Beide Strategien klingen »vernünftig«. Insbesondere ist TRANS eine »konservativere« Variante von MTF: das gesuchte Element wird nicht komplett an den Anfang der Liste geschoben, sondern nur eine Position weiter nach vorne.

Satz 3.3 *Es gibt für jedes $\varepsilon > 0$ beliebig lange Anfragesequenzen σ mit* $\mathsf{TRANS}(\sigma)/\mathsf{OPT}(\sigma) \geq 2m/3 - \varepsilon$. *(Damit ist* TRANS *nicht kompetitiv, wenn wir m nicht als Konstante ansehen.)*

Beweis: Wir benutzen wieder einen grausamen Adversary, um eine schlechte Folge σ für TRANS zu konstruieren. Dabei ist die Strategie des grausamen Adversaries im Fall von TRANS sehr einfach: es werden abwechselnd die zwei Elemente angefragt, welche an den letzten beiden Positionen der Liste stehen.

Wenn der grausame Adversary das letzte Element der Liste von TRANS anfragt, dann schiebt TRANS das Element an die vorletzte Position. Bei der nächsten Anfrage wandert es wieder an die letzte Position, da es mit dem aktuell letzten Element vertauscht wird. Die Kosten von TRANS auf einer so konstruierten Folge $\sigma = r_1, \ldots, r_n$ sind also wieder

$$\mathsf{TRANS}(\sigma) = nm.$$

Der Adversary kann nach den ersten beiden Anfragen die beiden Elemente durch freie Vertauschungen an die ersten beiden Positionen schieben. Er hat dann für jedes Paar von Anfragen Kosten 3. Seine Gesamtkosten sind also höchstens

$$\mathsf{OPT}(\sigma) \leq 2m + 3(n-2)/2.$$

Somit ergibt sich:

$$\frac{\mathsf{TRANS}(\sigma)}{\mathsf{OPT}(\sigma)} \geq \frac{nm}{3(n/2-1)+2m} \to \frac{2m}{3}.$$

□

In Übung 3.3 wird gezeigt, daß auch FC nicht kompetitiv ist.

3.2 Das Bahncard-Problem

Wir betrachten das Bahncard-Problem aus Abschnitt 1.1 in einer allgemeineren Version. Seien $B > 0$ (der Kaufpreis einer Bahncard), $T > 0$ (die Gültigkeitsdauer einer Bahncard) und $\beta \in [0,1]$ (der Discount-Faktor einer Bahncard) fest. Beim *(B,T,β)-Bahncard-Problem* erhält ein Algorithmus eine Folge $\sigma = r_1, \ldots, r_n$ von Fahrtanfragen $r_i = (t_i, p_i)$, welche chronologisch geordnet sind $t_1 \leq \cdots \leq t_n$. Die Zahl p_i gibt den *regulären Preis* der Fahrt i an.

(B,T,β)-Bahncard-Problem

Die Aufgabe des Algorithmus ALG ist es, auf jede Fahrtanfrage durch Kauf einer Fahrkarte zu reagieren (ALG kann also niemals Fahrten ablehnen). Vor dem Kauf der Fahrkarte, kann ALG noch für Kosten C eine Bahncard kaufen. Eine Bahncard, die zum Zeitpunkt t gekauft wird, ist im Intervall $[t, t+T)$ gültig. Die Kosten von ALG bei Anfrage r_i sind

$$c_A(r_i) = \begin{cases} \beta p_i & \text{wenn } \mathsf{ALG} \text{ eine gültige Bahncard zum Zeitpunkt } t_i \text{ besitzt} \\ p_i & \text{sonst.} \end{cases}$$

Ein Online-Algorithmus für das Bahncard-Problem erhält die Anfrage r_{i+1} erst, wenn er die Anfrage r_i bearbeitet hat. Ein Offline-Algorithmus kennt die komplette Folge σ im Voraus.

Wir nennen

$$\bar{C} := \frac{B}{1-\beta}$$

die *kritischen Kosten*. Der Wert $\bar{C}$ bezeichnet den Kostenschwellwert, ab dem es billiger wird, eine Bahncard zu kaufen, wenn aktuelle Fahrkosten von $\bar{C}$ anliegen.

Warum ist das Bahncard-Problem eine Verallgemeinerung des Skifahrerproblems, das wir in Kapitel 2 genauer untersucht haben? Man erhält das Skifahrerproblem als Spezialfall des Bahncard-Problems, wenn die Gültigkeitsdauer T einer Bahncard unendlich und der Discount-Faktor β gleich Null ist. Das Deutsche Bahncard-Problem erhält man mit $B = 240$, $T = 1$ Jahr und $\beta = 1/2$.

Beobachtung 3.4 *Der Algorithmus, der niemals eine Bahncard kauft, ist $1/\beta$-kompetitiv.*

Für das Deutsche Bahncard-Problem folgt aus der Beobachtung 3.4, daß man bei ständiger Verweigerung des Kaufs einer Bahncard 2-kompetitiv ist. Der folgende »Algorithmus« wird gerne von den DB-Kundenberatern vorgeschlagen:

Algorithmus DBKB (DB-Kundenberater) Wenn keine gültige Bahncard vorliegt und die aktuelle Anfrage mindestens Fahrkosten $\bar{C}$ hat, dann kaufe eine Bahncard. Ansonsten kaufe keine Bahncard.

Wie gut ist nun der Algorithmus DBKB? Das folgende Lemma zeigt, daß der DB-Kundenberater im Worst-Case auch nicht besser als der triviale Algorithmus ist, der niemals eine Bahncard kauft.

Satz 3.5 DBKB *ist nicht besser als $1/\beta$-kompetitiv.*

Beweis: Der Adversary gibt DBKB eine Folge von n Aufträgen der Form $r_i =$

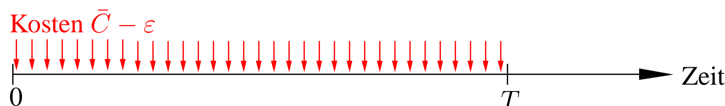


Abbildung 3.2: Beweis von Satz 3.5

$(i \cdot \frac{T-\varepsilon}{n}\varepsilon, \bar{C} - \varepsilon)$, wobei $\varepsilon > 0$ klein ist. Der Algorithmus DBKB kauft bei keiner Anfrage eine Bahncard, hat also Kosten

$$\text{DBKB}(\sigma) = n(\bar{C} - \varepsilon).$$

Wenn n nur groß genug ist, dann ist die optimale Offline-Lösung offenbar, bei der ersten Fahrkartenanfrage eine Bahncard zu kaufen. Daher gilt dann:

$$\text{OPT}(\sigma) = B + \frac{n(\bar{C} - \varepsilon)}{\beta}. \tag{3.3}$$

Somit haben wir:

$$\frac{\text{DBKB}(\sigma)}{\text{OPT}(\sigma)} = \frac{n(\bar{C} - \varepsilon)}{B + \frac{n(\bar{C}-\varepsilon)}{\beta}} \stackrel{n\to\infty}{\longrightarrow} \frac{1}{\beta}.$$

Wie oben angedeutet, strebt der Quotient aus den Kosten von DBKB(σ) und OPT(σ) für $n \to \infty$ gegen $1/\beta$. Dies zeigt das gewünschte Ergebnis. □

Korollar 3.6 *Für das Deutsche Bahncard-Problem ist der Algorithmus* DBKB *auch nur* 2*-kompetitiv.* □

Das Ergebnis aus Satz 3.5 erscheint frustierend. Immerhin erscheint doch die Strategie des DB-Kundenberaters ganz vernünftig. Gibt es am Ende gar keinen besseren Algorithmus?

Um ein Gefühl für die Schwierigkeit des Online-Problems zu bekommen, beschäftigen wir uns zunächst einmal mit der Struktur eines optimalen Offline-Algorithmus.

Wir nennen ein Zeitintervall I *teuer*, wenn die Summe der Preise der Aufträge in I mindestens $\bar{C}$ erreichen. Ansonsten heißt I *billig*.

Definition 3.7 (Reduzierte und reguläre Anfrage) *Sei* $\sigma = r_1, \ldots, r_n$ *eine beliebige Anfragefolge und* ALG *ein beliebiger Algorithmus, der* σ *bearbeitet. Für eine Anfrage* $r_i = (t_i, p_i)$ *nennen wir* βp_i *den* ***reduzierten Preis*** *und* p_i *den* ***reduzierten Preis****. Hat* ALG *zum Zeitpunkt* t_i *eine Bahncard, so nennen wir* r_i *eine* ***reduzierte Anfrage*** *(für* ALG*). Ansonsten heißt* r_i *eine* ***reguläre Anfrage*** *(für* ALG*).*

Lemma 3.8 *Sei* $\sigma = r_1, \ldots, r_n$ *eine beliebige Anfragesequenz und* OPT *ein optimaler Offline-Algorithmus. Dann können wir O. B. d. A. folgendes annehmen:*

(i) OPT *kauft niemals eine Bahncard bei einer reduzierten Anfrage.*

(ii) Sei I *ein Zeitintervall der Länge höchstens* T *und* $p^I(\sigma) := \sum_{i:t_i \in I} p_i \geq \bar{C}$*. Dann hat* OPT *mindestens eine reduzierte Anfrage in* I*.*

Beweis:

(i) Zum Zeitpunkt der reduzierten Anfrage besitzt OPT nach Definition bereits eine Bahncard. Aufschieben des Kaufs bis zur nächsten regulären Anfrage erhöht die Kosten nicht.

(ii) Angenommen, OPT habe nur reguläre Anfragen in I. Dann kauft OPT während des gesamten Zeitintervalls I keine Bahncard. Sei r_j die erste Anfrage im Intervall I. Wir modifizieren die Lösung von OPT so, daß zum Zeitpunkt t_j eine Bahncard gekauft wird. Diese ist dann für das komplette Zeitintervall I gültig, da I Länge höchstens T besitzt. Durch Kauf der Bahncard erhöhen sich die Kosten für OPT zunächst um B, die Kosten für Fahrkarten in I reduzieren sich dann aber um den Faktor β. Somit ist der Kostenanstieg:

$$B - (1-\beta)p_I(\sigma) \leq B - (1-\beta)\bar{C} = 0.$$

Folglich ist die modifizierte Lösung ebenfalls optimal.

□

In Übung 3.1 wird ein optimaler Offline-Algorithmus für das Bahncard-Problem erarbeitet.

Algorithmus BCSUM Wenn bei der Anfrage $r_i = (t_i, p_i)$ keine gültige Bahncard vorhanden ist, dann kaufe eine Bahncard, falls die regulären Kosten $p^{(t_i - T, t_i]}(\sigma)$ für BCSUM im Zeitintervall $(t_i - T, t_i]$ mindestens den Wert $\bar{C}$ erreichen (bei den Kosten im Intervall $(t_i - T, t_i]$ werden die Kosten für r_i mitgerechnet.

Beobachtung 3.9 *Wenn* BCSUM *zum Zeitpunkt t eine Bahncard kauft, dann ist das Zeitintervall $(t - T, t]$ teuer.*

Satz 3.10 BCSUM *ist $(2 - \beta)$-kompetitiv.*

Beweis: Sei $\sigma = r_1, \dots, r_n$ eine beliebige Anfragefolge. Für ein Zeitintervall I bezeichnen wir mit $c^I_{\mathsf{ALG}}(\sigma)$ die Fahrpreiskosten (ohne Kosten für die Bahncards!) von Algorithmus ALG im Zeitintervall I.

Seien $\tau_1 < \tau_2 < \dots < \tau_k$ die Zeitpunkte, zu denen OPT eine Bahncard kauft. Dies unterteilt die Gesamtzeit in *Epochen* $[\tau_j, \tau_{j+1})$, $j = 0, \dots, k$, wobei wir $\tau_0 := 0$ und $\tau_{k+1} := +\infty$ setzen.

Zunächst argumentieren wir, daß BCSUM frühestens in der ersten Epoche $[\tau_1, \tau_2)$ eine Bahncard kauft. Angenommen BCSUM kaufe zum Zeitpunkt $t < \tau_1$ eine Bahncard. Dann ist nach Definition von BCSUM das Intervall $(t - T, t]$ teuer. Nach Lemma 3.8 hätte dann OPT mindestens eine reduzierte Anfrage in diesem Zeitintervall, hätte also eine Bahncard zu einem Zeitpunkt $t' \leq t < \tau_1$ gekauft, was der Definition von τ_1 widerspricht.

Wir betrachten die Epochen $[\tau_j, \tau_{j+1})$ für $j \geq 1$ genauer. Da OPT zum Zeitpunkt τ_j eine Bahncard kauft, muß das Zeitintervall $[\tau_j, \tau_j + T)$ teuer sein (sonst könnten wir die Kosten für OPT verringern, indem wir auf diesen Kauf der Bahncard verzichten). Weiterhin ist das Intervall $[\tau_j + T, \tau_{j+1})$ billig (sonst könnten wir durch zusätzlichen Kauf einer Bahncard zum Zeitpunkt $\tau_j + T$ die Kosten wiederum verringern). Mit Hilfe von Beobachtung 3.9 folgt daher, daß BCSUM in jeder Epoche $[\tau_j, \tau_{j+1})$ $(j \geq 1)$ höchstens eine Bahncard kauft. Wir hatten bereits oben gesehen, daß BCSUM in $[\tau_0, \tau_1)$ keine Bahncard kauft. Die Gesamtkosten für Bahncards von BCSUM für σ ist also durch kB nach oben beschränkt.

Da OPT im billigen Intervall $[\tau_j + T, \tau_{j+1})$ $j = 0, \dots, k$ keine Bahncard besitzt, gilt für die Fahrpreiskosten:

$$c^{[\tau_j + T, \tau_{j+1})}_{\mathsf{BCSUM}}(\sigma) \leq c^{[\tau_j + T, \tau_{j+1})}_{\mathsf{BCSUM}}(\mathsf{OPT}) \quad \text{für } j = 0, \dots, k. \tag{3.4}$$

Wir betrachten nun ein teures Intervall $I_j = [\tau_j, \tau_j + T)$ und teilen dieses in drei disjunkte Teilintervalle I^1_j, I^2_j, I^3_j auf, wobei auch einige dieser Teilintervalle leer sein können. Während I^1_j und I^3_j hat BCSUM eine Bahncard, im Zeitintervall I^3_j hat BCSUM keine Bahncard. Man beachte, daß diese Partitionierung immer durchführbar ist. Die Fahrpreiskosten von BCSUM während I sind dann:

$$c^{I_j}_{\mathsf{BCSUM}}(\sigma) = \beta p^{I^1_j}(\sigma) + p^{I^2_j}(\sigma) + \beta p^{I^3_j}(\sigma). \tag{3.5}$$

Die Fahrpreiskosten von OPT während I sind

$$c^{I_j}_{\mathsf{OPT}}(\sigma) = \beta(p^{I^1_j}(\sigma) + p^{I^2_j}(\sigma) + p^{I^3_j}(\sigma)). \tag{3.6}$$

Dazu kommen für OPT noch Kosten B für eine Bahncard.

Sei q so gewählt, daß der Quotient

$$\frac{B + \beta(p^{I_q^1}(\sigma) + p^{I_q^2}(\sigma)) + p^{I_q^3}(\sigma)}{B + \beta(p^{I_q^1}(\sigma) + p^{I_q^2}(\sigma) + p^{I_q^3}(\sigma))}$$

Es gilt für reelle Zahlen $a \geq b > 0$ und $x \geq 0$ die Ungleichung:

$$\frac{a+x}{b+x} \leq \frac{a}{b}.$$

maximal ist. Aus (3.4), (3.5), (3.6) und der Beobachtung von oben, daß BCSUM insgesamt höchstens Kosten kB für Bahncards hat, ergibt sich:

$$\begin{aligned}
\frac{\mathsf{BCSUM}(\sigma)}{\mathsf{OPT}(\sigma)} &\leq \frac{kB + \sum_{j=0}^{k} c_{\mathsf{OPT}}^{[\tau_j+T,\tau_{j+1})}(\sigma) + \sum_j \beta(p^{I_j^1}(\sigma) + p^{I_j^2}(\sigma)) + p^{I_j^3}(\sigma)}{kB + \sum_{j=0}^{k} c_{\mathsf{OPT}}^{[\tau_j+T,\tau_{j+1})}(\sigma) + \beta\sum_j \beta(p^{I_j^1}(\sigma) + p^{I_j^2}(\sigma) + p^{I_j^3}(\sigma))} \\
&\leq \frac{B + \sum_j \beta(p^{I_j^1}(\sigma) + p^{I_j^2}(\sigma)) + p^{I_j^3}(\sigma)}{B + \beta\sum_j (p^{I_j^1}(\sigma) + p^{I_j^2}(\sigma) + p^{I_j^3}(\sigma))} \\
&\leq \frac{B + \beta(p^{I_q^1}(\sigma) + p^{I_q^2}(\sigma)) + p^{I_q^3}(\sigma)}{B + \beta(p^{I_q^1}(\sigma) + p^{I_q^2}(\sigma) + p^{I_q^3}(\sigma))} \\
&\leq \frac{B + p^{I_q^3}(\sigma)}{\beta p^{I_q^3}(\sigma)} \\
&\leq \frac{B + \bar{C}}{B + \beta\bar{C}} \qquad (\text{da } p^{I_q^3}(\sigma) \leq \bar{C}) \\
&= 2 - \beta.
\end{aligned}$$

□

Korollar 3.11 *Für das Deutsche Bahncard-Problem ist der Algorithmus* BCSUM $3/2$*-kompetitiv. Insbesondere schlägt er den DB-Kundenberateralgorithmus* DBKB. □

Satz 3.12 *Sei* ALG *ein beliebiger deterministischer Online Algorithmus für das Bahncard-Problem. Wenn* ALG c*-kompetitiv ist, dann gilt:* $c \geq 2 - \beta$.

Beweis: Sei $\varepsilon > 0$ eine kleine Konstante. Der Adversary gibt nun solange Fahraufträge mit Kosten ε, bis ALG eine Bahncard kauft. Die Fahraufträge werden dabei beliebig dicht gesetzt, so daß alle Aufträge im Zeitraum $[0, T)$ erfolgen. Man beachte, daß ALG irgendwann eine Bahncard kaufen will, wenn er besser als $1/\beta$-kompetitiv sein möchte. Sobald ALG die Bahncard gekauft hat, beendet der Adversary die Anfragesequenz. Sei s die Summe der Kosten bis zu dem Zeitpunkt, wo ALG die Bahncard kauft, wobei der letzte Auftrag nicht mitgerechnet wird. Dann gilt:

$$\mathsf{ALG}(\sigma) = s + B + \beta\varepsilon$$

und

$$\mathsf{OPT}(\sigma) = \begin{cases} s + \varepsilon & \text{falls } s + \varepsilon \leq \bar{C} \\ B + \beta(s + \varepsilon) & \text{falls } s + \varepsilon > \bar{C}. \end{cases}$$

Somit folgt:

$$\frac{\mathsf{ALG}(\sigma)}{\mathsf{OPT}(\sigma)} = \begin{cases} \frac{B+s+\beta\varepsilon}{s+\varepsilon} & \text{falls } s+\varepsilon \leq \bar{C} \\ \frac{B+s+\beta\varepsilon}{B+\beta(s+\varepsilon)} & \text{falls } s+\varepsilon > \bar{C}. \end{cases}$$
$$\geq 2 - \beta - \frac{\varepsilon(1-\beta)^2}{B}.$$

Dabei haben wir ausgenutzt, daß beide Quotienten ihr Minimum bei $s = \bar{C} - \varepsilon$ erreichen. Da nun $\varepsilon > 0$ beliebig klein gewählt werden kann, folgt die Behauptung. □

Übungsaufgaben

Übung 3.1
Geben Sie einen polynomialen Algorithmus für das Offline Bahncard-Problem an.

Übung 3.2 (Potentialfunktionen und amortisierte Kosten)
Beim Beweis der Kompetitivität von MTF haben wir eine Potentialfunktion und amortisierte Kosten benutzt. Potentialfunktionen sind ein elegantes Hilfsmittel. In dieser Aufgabe soll der Nutzen von Potentialfunktionen an einfachen Beispielen aus dem Bereich der Datenstrukturen illustriert werden.

Wir betrachten eine anfängliche Datenstruktur D_0, auf der n Operationen ausgeführt werden. Für $i = 1, \ldots, n$ seien c_i die Kosten eines Algorithmus ALG bei der $i$ten Operation. Mit D_i bezeichnen wir die Datenstruktur nach der $i$ten Operation. Eine *Potentialfunktion* Φ ordnet jeder Datenstruktur D_i eine reelle Zahl $\Phi(D_i)$ zu. Die *amortisierten Kosten* $\hat{c}_i$ bei der $i$ten Operation sind

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

Die Intuition hinter eine Potentialfunktion und den amortisierten Kosten ist die folgende: $\Phi(D_i)$ mißt, »wie gut« der aktuelle Status der Struktur ist. Man kann $\Phi(D_i)$ als Bankkonto betrachten. Wenn die Differenz $\Phi(D_i) - \Phi(D_{i-1})$ negativ ist, dann unterschätzt $\hat{c}_i$ die tatsächlichen Kosten c_i. Die Differenz wird durch das Entnehmen des Potentialverlustes aus dem Konto abgedeckt.

Für die amortisierten Kosten gilt:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i + \Phi(D_0) - \Phi(D_n). \tag{3.7}$$

Wir analysieren im folgenden Stackoperationen. Ein Stack ist ein Last-in-First-Out Speicher S, auf dem die folgenden Operationen definiert sind:

- PUSH(S, x) legt das Objekt x oben auf den Stack.
- POP(S) liefert das oberste Objekt auf dem Stack und entfernt es vom Stack (wenn der Stack leer ist, dann bricht die Operation mit Fehler ab).

Beide Operationen kosten $\mathcal{O}(1)$ Zeit. Wir erlauben jetzt noch eine weitere Operation $\text{MULTIPOP}(S, k)$, welche die obersten k Objekte des Stacks entfernt. Diese Operation benötigt $\mathcal{O}(k)$ Zeit.

(a) Zeigen Sie, wie man mit herkömmlicher Worst-Case-Analyse eine Zeitschranke von $\mathcal{O}(n^2)$ für eine Folge von n Stackoperationen (beginnend mit dem leeren Stack) erhält. Nehmen Sie an, daß für die MULTIPOP-Operationen jeweils genügend Elemente auf dem Stack liegen.

(b) Benutzen Sie jetzt das Potential $\Phi(S) := |S|$ und amortisierte Kosten, um eine bessere Worst-Case-Zeitschranke von $\mathcal{O}(n)$ zu bekommen.

Übung 3.3 (Untere Schranke für FC)
Beweisen Sie, daß es für jedes $\varepsilon > 0$ beliebig lange Anfragesequenzen σ mit $\text{FC}(\sigma)/\text{OPT}(\sigma) \geq (m+1)/2 - \varepsilon$ gibt.

Übung 3.4 (Page Replication)
Beim *Page Replication* Problem ist ein Rechner-Netzwerk, modelliert durch einen ungerichteten Graphen $G = (V, E)$, gegeben. Eine Datenbank befindet sich im Netzwerkknoten $s \in V$. Es erfolgen nun Datenbankanfragen in den Knoten des Netzwerks. Eine Anfrage im Knoten $x \in V$ wird dadurch bearbeitet, daß eine Verbindung zu einem Knoten $y \in V$ hergestellt wird, an dem die Datenbank vorhanden ist. Die Kosten für diese Verbindung sind $d(x, y)$, die Länge eines kürzesten Weges zwischen x und y. Ein Algorithmus kann nun zusätzlich Kopien der Datenbank in den Netzwerkknoten erzeugen. Beim Kopieren der Datenbank vom Knoten u (der bereits eine Kopie besitzt) zum Knoten v entstehen Kosten $D \cdot d(u, v)$.

(a) Modellieren Sie das Skifahrerproblem aus Abschnitt 1.1 als Spezialfall des Page Replication Problems. (Hinweis: Der benötigte Graph ist sehr klein und einfach!)

(b) Zeigen Sie, daß der folgende Algorithmus RCP $(2 - 1/D)$-kompetitiv für das Page Replication Problem auf Bäumen ist. (Hinweis: Benutzen Sie die Hauptidee aus dem kompetitiven Algorithmus für das Skifahrerproblem):

Algorithmus RCP

Der Algorithmus RCP betrachtet den Baum G als Wurzelbaum mit Wurzel s, wobei $s \in V$ derjenige Knoten ist, der anfangs die Datenbank besitzt. Für jeden Knoten $v \in V$ benutzt der Algorithmus einen Zähler $z[v]$, der anfangs gleich Null ist. Bei einer Anfrage im Knoten v werden alle Zähler auf dem Pfad von v zum nahesten Vorfahren im Baum, der eine Kopie besitzt, um eins erhöht. Wenn der Zähler in v den Wert $D - 1$ erreicht, dann wird die Datenbank zu allen Knoten auf dem Weg vom nähesten Vorgänger im Baum mit einer Kopie zu v kopiert.

4

Paging

Referenzwerke: [10, Kapitel 3 und 4]

Das Paging Problem ist eines der grundlegendsten und ältesten Online Probleme. Dieses Problem hat die Einführung der kompetitiven Analyse in [30] motiviert.

Beim *Paging* ist ein zweistufiges Speichersystem gegeben. Jeder Level kann Speicherseiten mit fester Größe aufnehmen. Die erste Stufe ist der *langsame Speicher*, welcher eine feste Menge $P = \{p_1, \ldots, p_N\}$ von N Seiten speichert. Die zweite Stufe, der *schnelle Speicher* (Cache), kann eine beliebige k-elementige Teilmenge ($k < N$) von P speichern.

langsamer Speicher

schneller Speicher (Cache)

Das Speichersystem erhält eine Folge $\sigma = r_1, \ldots, r_n$ von Seitenanfragen. Wird die Seite p_i angefragt, so muß das System diese Seite im Cache zur Verfügung stellen. Wenn p_i bereits im Cache ist (man spricht dann von einem »*Cache Hit*«), so muß das System nichts tun. Ist p_i jedoch nicht im Cache (man spricht hier von einem »*Cache Miss*«), so muß das System die Seite p_i in den Cache kopieren und erhält einen *Seitenfehler* (»page fault«). Beim Kopieren in den Cache muß das System entscheiden, welche der vorhandenen k Seiten im Cache durch die Seite p_i überschrieben wird.

Cache Hit

Cache Miss

Seitenfehler

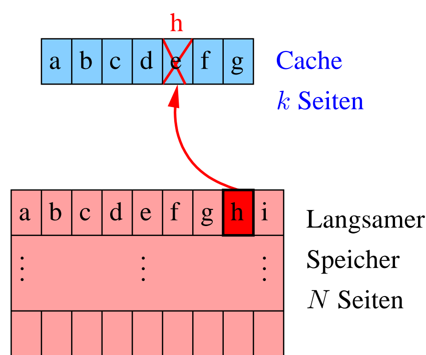


Abbildung 4.1
Die ersten Anfragen aus σ auf die Seiten b, c und d sind *cache hits*. Die nächste Anfrage erfolgt auf die Seite h, die nicht im Cache ist. Der Paging Algorithmus muß nun h in den Cache bringen und dabei eine Seite im Cache überschreiben (im Beispiel die Seite e).

Beim *Seitenfehlermodell (Page Fault Model)* ist die zu minimierende Zielfunktion einfach die Anzahl der Page Faults. Im *Zugriffskostenmodell (Access Cost Model)* kostet ein Cache Hit eine Einheit, während ein Cache Miss Kosten s

Seitenfehlermodell (Page Fault Model)

Zugriffskostenmodell (Access Cost Model)

verursacht. Wir betrachten im folgenden nur das Seitenfehlermodell.

Eine typische Anwendung des Paging Problems ist das Caching von Festplattenzugriffen. Hier ist der langsame Speicher die Festplatte und der schnelle Speicher der Hauptspeicher (RAM). Eine weitere Anwendung ergibt sich beim Caching von Hauptspeicherzugriffen, wo der Schnellspeicher der Prozessorcache und der langsame Speicher der Hauptspeicher ist.

4.1 Ein optimaler Offline Algorithmus

Wir betrachten zunächst das zugehörige Offline-Problem. Hier ist die Anfragefolge $\sigma = r_1, \ldots, r_n$ komplett bekannt. Für das Offline-Problem läßt sich ein einfacher Algorithmus angeben, welcher für jede Folge σ eine minimale Anzahl von Seitenfehlern erzeugt.

Algorithmus LFD (Longest-Forward-Distance) Überschreibe bei einem Cache Miss diejenige Seite im Cache, deren nächster Zugriff am weitesten in der Zukunft liegt.

Beispiel 4.1 Sei die Größe des Cache $k = 4$ und der anfängliche Cache-Inhalt gleich $abcd$. Wir betrachten die Bearbeitung der Anfragefolge $\sigma = aabegbbcdadge$. Die ersten drei Seitenanfragen sind Cache Hits. Bei der Anfrage der Seite e entfernt LFD die Seite a aus dem Cache, da diese von den Seiten a, b, c und d in der Zukunft am spätesten angefragt wird. ◁

Der Algorithmus LFD ist offensichtlicherweise ein *Offline-Algorithmus*. Er benötigt die Kenntnis der Zukunft, um zu entscheiden, welche Seite er aus dem Cache entfernen (überschreiben) soll.

Um zu beweisen, daß LFD eine minimale Anzahl von Seitenfehlern erzeugt, benötigen wir folgendes Lemma.

Lemma 4.2 *Sei* ALG *ein Paging Algorithmus und sei $\sigma = r_1, \ldots, r_n$ eine beliebige Anfragefolge. Dann gibt es für $i = 1, \ldots, n$ einen Algorithmus* ALG_i *mit folgenden Eigenschaften:*

$\underbrace{r_1, \ldots, r_{i-1}}_{\text{wie ALG}} \; \underbrace{r_i}_{\text{LFD-Regel}} \; r_{i+1}, \ldots$

1. ALG_i *bearbeitet die ersten $i - 1$ Anfragen genau wie* ALG.
2. *Wenn die $i$te Anfrage ein Seitenfehler ist, dann verwirft* ALG_i *diejenige Seite, deren nächste Anfrage am weitesten in der Zukunft liegt (d.h. gemäß* LFD*-Regel)*
3. $\text{ALG}_i(\sigma) \leq \text{ALG}(\sigma)$.

Beweis: Wir konstruieren ALG_i aus ALG. Wenn die $i$te Anfrage kein Seitenfehler ist, dann ist nichts zu zeigen. Wir nehmen daher an, daß die Anfrage $r_i = f$ einen Seitenfehler bei Algorithmus ALG produziert.

Sei p diejenige Seite, welche ALG aus dem Cache entfernt und durch $r_i = f$ ersetzt. Sei ferner q diejenige Seite, welche beim Seitenfehler auf r_i gemäß

LFD-Regel: $\sigma = \ldots, r_i = f, \ldots, q, \ldots, p, \ldots$

LFD-Regel ersetzt wurde. Wir können annehmen, daß $p \neq q$ gilt, denn sonst ist nichts mehr zu zeigen.

ALG: $Y \cup \{p, q\} \longrightarrow Y \cup \{f, q\}$
ALG$_i$: $Y \cup \{p, q\} \longrightarrow Y \cup \{p, f\}$

Der Algorithmus ALG$_i$ ersetzt nun q im Cache durch die Seite f. Der Inhalt des Caches von ALG$_i$ nach Bearbeiten der Anfrage r_i ist $X \cup \{p\}$, der von ALG ist $X \cup \{q\}$, wobei X die $k-1$ gemeinsamen Seiten im Cache sind.

Nach der Anfrage r_i verarbeitet ALG$_i$ den Rest der Folge σ bis zu dem ersten Zeitpunkt r_j, wo die Seite p das nächste Mal gefragt wird, wie folgt:

ALG: $X' \cup \{q\} \longrightarrow X' \cup \{z\}$
ALG$_i$: $X' \cup \{p\} \longrightarrow X' \cup \{z\}$

- Wenn ALG die Seite q aus dem Cache entfernt, dann entfernt ALG$_i$ die Seite p.
- Wenn ALG die Seite $x \neq q$ aus dem Cache entfernt, dann entfernt auch ALG$_i$ die Seite x aus dem Cache.

Man zeigt leicht durch Induktion, daß bis zum ersten Zeitpunkt, zu dem ALG die Seite q entfernt oder $r_j = p$ gilt, die Caches von ALG$_i$ und ALG die Form $X' \cup \{p\}$ bzw. $X' \cup \{q\}$ mit einer gemeinsamen Menge X' aus $k-1$ Seiten haben. Tritt also bis zur ersten Anfrage von p irgendwann der erste Fall ein, so sind ab diesem Zeitpunkt beide Caches identisch und beide Algorithmen arbeiten ab dann gleich. In diesem Fall ist nichts mehr zu zeigen, weil dann die Anzahl der Seitenfehler beider Algorithmen für die Folge σ identisch sind.

Bei Anfrage der Seite p (diese Anfrage kommt nach Definition von p *vor* der nächsten Anfrage von q) in r_j hat ALG einen Seitenfehler, während ALG$_i$ die Seite p im Cache hat. ALG ersetzt daraufhin eine Seite r im Cache durch p. Wenn $r = q$ ist, dann sind die beiden Caches identisch, und wir sind fertig. Ansonsten ist der Cache von ALG von der Form $X'' \cup \{q\}$ und der von ALG$_i$ von der Form $X'' \cup \{r\}$.

Wenn das nächste Mal q gefragt wird, dann hat ALG$_i$ einen Seitenfehler, während ALG einen Cache Hit hat. Nun ersetzt ALG$_i$ die Seite r durch q, und beide Caches werden identisch.

Insgesamt hat ALG$_i$ also bis zum Identifizieren der beiden Algorithmen höchstens so viele Seitenfehler wie ALG. □

Korollar 4.3 LFD *ist ein optimaler Offline-Algorithmus für das Paging.*

Hier kennt man einen optimalen Offline-Algorithmus!

Beweis: Sei OPT ein optimaler Offline-Algorithmus. Sei $\sigma = r_1, \ldots, r_n$ beliebig. Aus Lemma 4.2 erhalten wir einen Algorithmus OPT$_1$ mit OPT$_1(\sigma) \leq$ OPT(σ). Wir wenden das Lemma mit $i = 2$ auf OPT$_1$ an und erhalten OPT$_2$ mit OPT$_2(\sigma) \leq$ OPT$_1(\sigma) \leq$ OPT(σ). Fortsetzen liefert dann OPT$_n$, welcher die Folge σ gemäß LFD abarbeitet. □

Lemma 4.4 *Es gelte $N = k+1$. Sei $\sigma = r_1, \ldots, r_n$ eine beliebige Anfragefolge. Dann gilt* LFD$(\sigma) \leq \lceil |\sigma|/k \rceil$, *d.h. höchstens jede k-te Anfrage ergibt einen Seitenfehler.*

Beweis: Angenommen, LFD habe einen Seitenfehler bei Anfrage r_i und verwerfe die Seite p seines Cacheinhalts X. Der nächste Seitenfehler von LFD ist wegen $N = k+1$ auf der Seite p. Bis p gefragt wird, müssen aber wegen der LFD-Regel alle $k-1$ Seiten aus $X \setminus \{p\}$ gefragt worden sein. Somit hat LFD höchstens alle k Anfragen einen Seitenfehler. □

4.2 Deterministische Online-Algorithmen

Beim Online-Paging muß der Online-Algorithmus bei einem Seitenfehler auf der Anfrage r_i seine Entscheidung, welche Seite er im Cache überschreibt, *ohne* Kenntnis der zukünftigen Anfragen $r_{i+1}, r_{i+2}, \ldots$ treffen.

Seitenersetzungsstrategien

Folgende Algorithmen (Seitenersetzungsstrategien) sind gebräuchlich:

FIFO (First-In/First-Out) Entferne die Seite aus dem Cache, die am längsten im Cache ist.

LIFO (Last-In/First-Out) Entferne die Seite aus dem Cache, die zuletzt in den Cache gebracht wurde.

LFU (Least-Frequently-Used) Entferne eine Seite, die am seltensten gefragt wurde.

LRU (Least-Recently-Used) Entferne die Seite, deren letzter Zugriff am weitesten zurückliegt.

Zunächst zeigen wir, daß LIFO und LFU nicht kompetitiv sind.

Wir beginnen mit LIFO. Sei $X = \{p_1, \ldots, p_k\}$ der anfängliche Cacheinhalt von LIFO und $p_{k+1} \notin X$ eine beliebige weitere Seite. Wir betrachten die Folge

$$\begin{aligned}\sigma &= p_1, p_2, \ldots, p_k, (p_{k+1}, p_k)^\ell \\ &= p_1, p_2, \ldots, p_k, \underbrace{(p_{k+1}, p_k), \ldots, (p_{k+1}, p_k)}_{\ell \text{ mal}}\end{aligned}$$

Ab der $(k+1)$ten Anfrage hat LIFO bei jeder Anfrage einen Seitenfehler. Es gilt daher: $\mathsf{LIFO}(\sigma) = 2\ell$. Andererseits gilt: $\mathsf{OPT}(\sigma) = 1$. Da wir ℓ beliebig groß wählen können, gibt es keine Konstanten c und α, so daß $\mathsf{LIFO}(\sigma) \leq c\mathsf{OPT}(\sigma) + \alpha$ gilt.

Für LFU benutzen wir die Folge

$$\sigma = p_1^\ell, \ldots, p_{k-1}^\ell, (p_k, p_{k+1})^\ell.$$

Nach den ersten $(k-1)\ell$ Anfragen hat LFU bei jeder Anfrage einen Seitenfehler. Es folgt: $\mathsf{LFU}(\sigma) \geq 2(\ell - 1)$. Andererseits gilt: $\mathsf{OPT}(\sigma) = 1$. Wiederum kann ℓ beliebig groß gewählt werden.

Phasenpartition:

$$\sigma = \overbrace{r_1, \ldots, r_{j_1}}^{k \text{ versch. Seiten}}, \overbrace{r_{j_1+1}, \ldots, r_{j_2}}^{k \text{ versch. Seiten}}, \ldots, \overbrace{r_{j_r+1}, \ldots, r_n}^{\leq k \text{ versch. Seiten}}$$

4.2.1 Phasenpartitionen und Markierungsalgorithmen

Zur Analyse der Algorithmen benutzen wir ein wichtiges Hilfsmittel, die k-*Phasenpartition* einer Anfragefolge σ. Die Phase 0 besteht aus der leeren Folge. Für $i \geq 0$ besteht die Phase $i + 1$ aus der maximalen Teilfolge von σ, die auf die Phase i folgt und die aus höchstens k verschiedenen Seitenanfragen besteht. Die k-Phasenpartition einer Anfragefolge ist immer eindeutig definiert und unabhängig von einem bestimmten Algorithmus.

Markierung

Sei $\sigma = r_1, \ldots, r_n$ eine beliebige Anfragefolge. Wir weisen jeder Seite (im gesamten langsamen Speicher) eine *Markierung* in folgender Weise zu: Am

Anfang jeder Phase werden die Markierungen aller Seiten gelöscht. Während einer Phase markieren wir eine Seite, wenn sie das erste Mal gefragt wird. Man beachte, daß die Markierung der Seiten immer noch unabhängig von einem bestimmten Algorithmus ist.

Ein *Markierungsalgorithmus* ist ein Algorithmus mit der Eigenschaft, daß er niemals eine markierte Seite aus dem Cache entfernt.

Markierungsalgorithmus

Lemma 4.5 LRU *ist ein Markierungsalgorithmus.*

Beweis: Angenommen LRU würde in einer Phase die markierte Seite p bei Anfrage r_i aus dem Cache entfernen. Wir betrachten die erste Anfrage r_j $(j < i)$ auf p in dieser Phase. Nach der Anfrage r_j ist p markiert und die Seite im Cache von LRU, deren letzter Zugriff am jüngsten ist. Sei X der Cacheinhalt von LRU zu diesem Zeitpunkt.

$\overbrace{\ldots, r_j = p, \ldots, r_i, \ldots}^{\text{Phase}}$

Damit p von LRU entfernt wird, muß p diejenige Seite werden, deren letzter Zugriff am weitesten zurückliegt. Das bedeutet, daß nach der Anfrage r_j jede der $k-1$ Seiten aus $X \setminus \{p\}$ in der aktuellen Phase angefragt worden sein muß. Da $r_j = p$ wurden bis zum Entfernen von p somit k verschiedene Seiten in der Phase gefragt. Nun wird p von LRU bei der Anfrage r_i nach Annahme entfernt. Es folgt $r_i \notin X$, da sonst kein Seitenfehler vorläge. Damit wurden in der Phase $k+1$ verschiedene Seiten angefragt. Dies widerspricht der Definition der Phasenpartition. □

$r_j = p, \overbrace{\ldots, \ldots,}^{\text{mind. } X \setminus \{p\}} r_i$

Satz 4.6 *Jeder Markierungsalgorithmus ist k-kompetitiv für das Paging Problem mit Cachegröße k.*

Beweis: Sei ALG ein Markierungsalgorithmus und $\sigma = r_1, \ldots, r_n$ eine beliebige Anfragefolge. Wir betrachten die k-Phasenpartition von σ.

In jeder Phase hat der Markierungsalgorithmus ALG höchstens k Seitenfehler: Bei einem Seitenfehler wird eine Seite markiert und in den Cache gebracht. Da der Markierungsalgorithmus nie eine markierte Seite aus dem Cache entfernt, kann er in jeder Phase pro markierter Seite höchstens einen Seitenfehler haben. Da jede Phase höchstens k verschiedene Seitenanfragen enthält, folgt, daß ALG höchstens k Seitenfehler pro Phase hat.

Der optimale Offline-Algorithmus OPT hat mindestens so viele Seitenfehler wie es Phasen gibt: Wir unterteilen die Folge σ in *Segmente*. Jedes Segment startet bei der zweiten Anfrage der zugehörigen Phase und endet mit der ersten Anfrage der folgenden Phase. Zu Beginn eines Segments hat OPT die zuletzt gefragte Seite p, d.h. die Seite der ersten Anfrage der Phase, im Speicher. Bis zum Ende der Phase werden noch $k-1$ von p verschiedene Seiten, bis zum Ende des Segments noch k von p verschiedene Seiten gefragt. Da OPT nur k Seiten in Cache halten kann, muß OPT im Segment mindestens einen Seitenfehler haben. □

$\overbrace{r_i, \underbrace{r_{i+1}}_{\text{Start Seg. } i}, \ldots, r_j,}^{\text{Phase } i} \underbrace{r_{j+1}}_{\text{Ende Seg. } i},$

Korollar 4.7 LRU *ist k-kompetitiv für das Paging Problem mit Cachegröße k.* □

Im Gegensatz zu LRU ist FIFO kein Markierungsalgorithmus. Sei $p_1, \ldots, p_k$ der anfängliche Cacheinhalt von FIFO und p_{k+1} eine beliebige weitere Speicherseite. Wir betrachten die folgende Anfragefolge: $\sigma = p_1, p_2, \ldots, p_k, p_{k+1}$. Bei

Anfrage von p_{k+1} entfernt FIFO eine Seite aus $\{p_1, \ldots, p_k\}$ aus dem Cache, um sie durch p_{k+1} zu ersetzen. Sei dies o. B. d. A. p_1. Die nächsten beiden Anfragen sind auf p_2 und dann wieder auf p_1. Bei der Anfrage auf p_1 erfolgt wieder ein Seitenfehler. FIFO entfernt nun eine der ältesten Seiten im Cache, d.h. eine der Seiten $p_2, \ldots, p_k$. Dabei wird eventuell die markierte Seite p_2 aus dem Cache entfernt.

Trotzdem ist FIFO k-kompetitiv, wie der folgende Satz zeigt:

Satz 4.8 FIFO *ist k-kompetitiv für das Paging Problem mit Cachegröße k.*

Beweis: Der Beweis ist eine leichte Modifikation desjenigen von Satz 4.6. Wir benutzen wieder die k-Phasenpartition einer Folge $\sigma = r_1, \ldots, r_n$ und zeigen, daß FIFO in jeder Phase höchstens k Seitenfehler hat.

Es habe FIFO in der Phase i einen Seitenfehler auf Seite p. Damit ein erneuter Seitenfehler auf p entsteht, muß p aus dem Cache entfernt werden. Dazu muß p aber die älteste Seite im Cache werden. Damit dies geschieht, müssen alle $k-1$ anderen Seiten im Cache gefragt werden. Weiterhin muß noch eine weitere Seite gefragt werden, da sonst p nicht ausgelagert würde. Somit werden bis zum nächsten Seitenfehler auf p mindestens $k + 1$ verschiedene Seiten angefragt.

Somit ist gezeigt, daß FIFO in jeder Phase pro gefragter Seite nur maximal einen Seitenfehler hat. Da jede Phase nur k verschiedene Seiten enthält, folgt die Aussage des Satzes. □

4.2.2 Eine untere Schranke für deterministische Algorithmen

Wir beweisen nun, daß die Kompetitivität von LRU und FIFO bestmöglich für deterministische Algorithmen ist.

Satz 4.9 *Sei* ALG *ein beliebiger deterministischer Online Algorithmus für Paging. Wenn* ALG *c-kompetitiv ist, dann gilt: $c \geq k$.*

Beweis: Sei $S = \{p_1, \ldots, p_{k+1}\}$ eine beliebige $(k + 1)$-elementige Teilmenge der Seiten. Wir beschränken uns bei den Seitenanfragen nur auf Seiten aus S und zeigen, daß es eine beliebig lange (und teure) Folge σ mit Anfragen aus S gibt, so daß $|\sigma| = \text{ALG}(\sigma) \geq k \cdot \text{OPT}(\sigma)$ gilt.

Dazu können wir o. B. d. A. annehmen, daß ALG anfangs die Seiten $\{p_1, \ldots, p_k\}$ im Cache hat. Wir definieren eine Sequenz σ induktiv: $r_1 := p_{k+1}$. Für $i \geq 1$ sei p diejenige Seite, welche ALG bei der Anfrage r_1 aus dem Cache entfernt. Dann setzen wir $r_{i+1} := p$.

Offenbar ist $\text{ALG}(\sigma) = |\sigma|$. Nach Lemma 4.4 gilt $\text{OPT}(\sigma) \leq |\sigma|/k$. □

4.3 Ein randomisierter Algorithmus für Paging

In diesem Abschnitt stellen wir einen randomisierten Algorithmus RANDMARK für das Paging Problem vor. Der Algorithmus ist relativ einfach und $2H_k$-

kompetitiv gegen den blinden Gegner OBL. Dabei ist

$$H_k = 1 + \frac{1}{2} + \cdots + \frac{1}{k}$$

die $k$te *harmonische Zahl*.

harmonische Zahl

Es gilt: $\ln k < H_k \leq 1 + \ln k$.

RANDMARK (Randomisiertes Markieren) Anfangs sind alle Seiten unmarkiert.

Bei einer Anfrage auf eine Seite p, die nicht im Cache ist, wird p markiert in den Cache gebracht. Dabei wird eine zufällig ausgewählte unmarkierte Seite im Cache überschrieben (wenn alle Seiten im Cache bereits markiert sind, werden zuerst alle Markierungen gelöscht).

Bei einer Anfrage auf eine Seite p im Cache wird diese Seite p markiert.

Satz 4.10 RANDMARK *ist* $2H_k$*-kompetitiv gegen den blinden Gegner* OBL.

Beweis: Wir nehmen an, daß RANDMARK und der Gegner mit dem gleichen Cache-Inhalt starten (ansonsten können wir die Anfangskosten auf die additive Konstante α aufschlagen).

Sei $\sigma = r_1, \ldots, r_n$ eine beliebige Anfragesequenz. Wir betrachten wieder die k-Phasenpartition von σ. Man beachte, daß eine Seite, die in einer Phase von RANDMARK markiert wird, nicht vor dem Ende der Phase aus dem Cache entfernt wird. RANDMARK ist also tatsächlich ein (randomisierter) Markierungsalgorithmus. Wenn P_i die Menge der Seiten bezeichnet, die in der $i$ten Phase angefragt werden, so ist am Ende der $i$ten Phase der Cacheinhalt von RANDMARK daher exakt P_i.

Für jede Phase i nennen wir diejenigen Seiten, die umittelbar vor dem Start der Phase im Cache von RANDMARK sind *alte Seiten*. Eine Seite, die in der $i$ten Phase angefragt wird, und nicht im Cache vor dem Start der Phase ist, nennen wir *neue Seite*.

alte Seite

neue Seite

Wir betrachten eine beliebige Phase i. Sei m_i die Anzahl der neuen Seiten in Phase i. Da jede neue Seite zu Beginn der Phase nicht im Cache von RANDMARK ist, hat RANDMARK für jede neue Seite einen Seitenfehler. Da markierte Seiten nicht vor dem Ende der Phase entfernt werden, hat RANDMARK in Phase i *genau* m_i Seitenfehler für neue Seiten.

Wir berechnen nun die erwartete Anzahl von Seitenfehlern von RANDMARK in Phase i für alte Seiten. Sei ℓ die Anzahl der neuen Seiten, die angefragt worden sind, bevor die $j$te alte Seite (d.h. das $j$te mal eine alte Seite) verlangt wird. Zum Zeitpunkt, zu dem die $j$te alte Seite angefragt wird, sind dann genau ℓ alte Seiten aus dem Cache entfernt worden. Diese wurden aus den $k-(j-1)$ unmarkierten Seiten im Cache gleichverteilt ausgewählt (sobald eine der $j-1$ alten Seiten angefragt wird, bleibt/kommt sie bis zum Ende der Phase in den Cache von RANDMARK). Somit ist die Wahrscheinlichkeit, daß die $j$te alte Seite bei dieser Anfrage nicht im Cache ist, $\ell/(k-(j-1))$. Da $\ell \leq m_i$, ist die Wahrscheinlichkeit eines Seitenfehlers bei der $j$ten alten Seite höchstens $m_i/(k-(j-1))$.

Insgesamt erhalten wir für den Erwartungswert der Seitenfehler X_i von RANDMARK in Phase i:

$$\mathbb{E}[X_i] \leq m_i + \sum_{j=1}^{k-m_i} \frac{m_i}{k-j+1} = m_i(H_k - H_{m_i} + 1) \leq m_i H_k.$$

Weiterhin gilt natürlich $\mathbb{E}[\text{RANDMARK}(\sigma)] = \sum_i \mathbb{E}[X_i]$. Wir zeigen nun, daß $\text{OPT}(\sigma) \geq \sum_i m_i/2$ gilt. Dies zeigt dann die Behauptung des Satzes.

In der $(i-1)$ten und $i$ten Phase werden nach Definition insgesamt mindestens $k+m_i$ verschiedene Seiten gefragt. Da OPT nur k Seiten im Cache haben kann, hat OPT für die $(i-1)$te und $i$te Phase zusammen mindestens m_i Seitenfehler. Anwendung auf Paare von aufeinanderfolgenden Phasen liefert nun, daß $\text{OPT}(\sigma) \geq \sum_i m_{2i}$ und $\text{OPT}(\sigma) \geq \sum_i m_{2i+1}$. Somit ist

$$\text{OPT}(\sigma) \geq \frac{1}{2}\left(\sum_i m_{2i} + \sum_i m_{2i+1}\right) = \sum_i \frac{m_i}{2}.$$

Dies wollten wir zeigen. □

Übungsaufgaben

Übung 4.1 ((h,k)-Paging)
Bei (h,k)-Paging vergleichen wir die Kosten eines Online-Algorithmus mit Cachegröße k mit denen eines optimalen Offline-Algorithmus mit Cachegröße $h \leq k$. Zeigen Sie, daß LRU und FIFO $k/(k-h+1)$-kompetitiv für das (h,k)-Paging Problem sind. Beweisen Sie ferner, daß kein deterministischer Algorithmus besser als $k/(k-h+1)$-kompetitiv sein kann.

Übung 4.2 (Paging gegen den Adaptiven Offline Adversary)
Warum wird das Ergebnis von Theorem 4.10 falsch, wenn wir den blinden Gegner OBL durch den adaptiven Offline Adversary ADOFF ersetzen?

5

Ein Ausflug in die Spieltheorie

Referenzwerke: [10, Kapitel 6 und 8], [27, Kapitel 2]

5.1 Zwei-Personen Nullsummenspiele

Wir betrachten das klassische Stein-Schere-Papier Spiel: Zacharias und Sybille verstecken ihre Hände hinter dem Rücken und formen eines von drei möglichen Zeichen: Stein (Faust geschlossen), Schere (zwei Finger) oder Papier (flache Hand). Dann zeigen sie gleichzeitig ihr gewähltes Zeichen. Der Gewinner wird gemäß der folgenden Regeln bestimmt: Stein schlägt die Schere (zerstört sie), die Schere schlägt das Papier (zerschneidet es) und das Papier schlägt den Stein (wickelt ihn ein). Gleiche Zeichen werden als Unentschieden gewertet. Der Verlierer zahlt 1 DM an den Gewinner.

Wir können das obige Spiel in Matrixform darstellen: Die Zeilen der Matrix stehen für Zacharias' Wahlmöglichkeiten, die Spalten repräsentieren die Möglichkeiten für Sybille; die Einträge der Matrix zeigen die Geldsumme, die Sybille an Zacharias zahlen muß (siehe Tabelle 5.1).

	Stein	Schere	Papier
Stein	0	1	-1
Schere	-1	0	1
Papier	1	-1	0

Tabelle 5.1: Matrix für das Stein-Schere-Papier Spiel

Unser Stein-Schere-Papier Spiel ist eine sogenanntes Zwei-Personen Nullsummenspiel.

Definition 5.1 (Zwei-Personen Nullsummenspiel)
Ein ***Zwei-Personen Nullsummenspiel*** *in strategischer Form besteht aus einer* $n \times m$*-Auszahlungsmatrix* $M = (m_{ij})$ *mit Einträgen aus* $\mathbb{R}$*. Die Strategien des Zeilenspielers* Z *entsprechen den Zeilen von* M*, die Strategien der Spaltenspielerin* S *entsprechen den Spalten der Matrix* M*.*

Der Eintrag m_{ij} gibt an, wieviel S an Z zahlen muß, wenn Z die Strategie i und S die Strategie j wählt.

Der Ausdruck «Nullsummenspiel»stammt daher, daß der Gewinn des Gewinners dem Verlust des Verlierers entspricht.

Bei einem Zwei-Personen Nullsummenspiel ist es das Ziel des Zeilenspielers seinen Gewinn zu maximieren. Analog möchte die Spaltenspielerin ihren Verlust minimieren.

Wenn Z die Strategie i wählt, dann ist ihm ein Gewinn von $\min_j m_{ij}$ gewiß, unabhängig davon, welche Strategie S auswählt. Eine *optimale Strategie* für R ist es daher, die Strategie i^* zu wählen, welche $\min_j m_{ij}$ maximiert. Wir setzen

$$V_Z := \max_i \min_j m_{ij}.$$

Analog definiert man eine optimale Strategie für die Spaltenspielerin und den Wert

$$V_S := \min_j \max_i m_{ij}.$$

Man sieht leicht, daß immer $V_Z \leq V_S$ gilt. Im allgemeinen ist diese Ungleichung auch strikt. Wenn $V_Z = V_S$, dann nennt man den gemeinsamen Wert auch den *Wert des Spiels*.

Bisher haben wir nur deterministische Strategien für die Spieler betrachtet. Man nennt diese auch *reine Strategien*. Eine *gemischte Strategie* ist eine Wahrscheinlichkeitsverteilung p auf Menge der reinen Strategien.

Wenn Z die gemischte Strategie p und S die gemischte Strategie q benutzt, so ist die Auszahlung an Z eine Zufallsvariable, deren Erwartungswert durch

$$\sum_{i=1}^{n} \sum_{j=1}^{m} p_i m_{ij} q_j = p^T M q$$

gegeben ist.

Wie sehen nun optimale gemischte Strategien aus? Z kann seinen erwarteten Profit maximieren, indem er p so wählt, daß $\min_q p^T M q$ maximiert wird. Wir definieren:

$$v_Z := \max_p \min_q p^T M q$$
$$v_S = \min_q \max p^T M q.$$

Es ist wieder leicht zu sehen, daß $v_Z \leq v_S$ gilt. Das Überraschende ist aber, daß für gemischte Strategien die Spielwerte für Z und S immer übereinstimmen:

Satz 5.2 (Von Neumann Minmax Theorem) *Für jedes Zwei-Personen Nullsummenspiel gilt:*

$$\max_p \min_q p^T M q = \min_q \max p^T M q. \tag{5.1}$$

Beweis: Siehe z.B. [10, Kapitel 8]. □

Man nennt den Wert aus Gleichung (5.1) den *Wert des Spiels*. Ein Paar (p^*, q^*) von gemischten Strategien, welche die linke Seite von (5.1) maximieren, bzw. die rechte Seite minimieren, nennt man einen Sattelpunkt; die Strategien p^* und q^* nennt man *optimale gemischte Strategien*.

Wir ziehen nun eine wichtige Konsequenz aus den Minmax Theorem. Wenn p fest ist, dann ist $p^T Mq$ eine lineare Funktion in q. Das Minimum wird dann für das q angenommen, welches eine 1 an der Stelle j hat, bei der $p^T M$ den kleinsten Eintrag hat.

Das bedeutet das Folgende: Kennt S die Verteilung p von Z, so ist ihre optimale (gemischte) Strategie sogar eine reine Strategie (Analoges gilt umgekehrt). Sei e_k der Einheitsvektor mit einer 1 an der k ten Stelle. Dann ergibt sich aus (5.1):

Satz 5.3 (Loomi's Lemma) *Für jedes Zwei-Personen Nullsummenspiel gilt:*

$$\max_p \min_j p^T M e_j = \min_q \max_i e_i^T M q.$$

5.2 Yao's Prinzip und seine Anwendungen auf Online-Probleme

Aus Loomi's Lemma kann man noch eine einfache, aber sehr nützliche Folgerung ziehen, die als *Yao's Prinzip* bekannt ist. Sei $\bar{p}$ eine feste Verteilung über den reinen Strategien des Zeilenspielers. Dann gilt:

$$\begin{aligned} \min_q \max_i e_i^T M q &= \max_p \min_j p^T M e_j \qquad \text{(nach Loomi's Lemma)} \\ &\geq \bar{p}^T \min_j M e_j. \end{aligned}$$

Die Ungleichung

$$\min_q \max_i e_i^T M q \geq \bar{p}^T \min_j M e_j$$

ist als *Yao's Prinzip* bekannt.

Sei Π ein Online-Problem. Wir stellen uns Π als Spiel zwischen den Online-Spieler und dem Adversary vor. Aus technischen Gründen setzen wir hier voraus, daß es für das Problem nur endlich viele Online-Algorithmen $\{\mathsf{ALG}_j\}$ und nur endlich viele Eingabesequenzen $\{\sigma_i\}$ gibt. Die Ergebnisse lassen sich auf den allgemeinen Fall übertragen, allerdings sind hier die Beweise technischer, da man sich unter anderem mit der Frage beschäftigen muß, ob bestimmte Erwartungswerte überhaupt existieren.

Angenommen, wir wollten beweisen, daß kein randomisierter Algorithmus eine Kompetitivität besser als $c \geq 1$ gegen den blinden Gegner erreichen kann. Dazu müssen wir zeigen, daß für jede Verteilung q über der Menge $\{\mathsf{ALG}_i\}$ der deterministischen Online-Algorithmen $\{\mathsf{ALG}_j\}$ eine Eingabesequenz σ_i existiert mit:

$$\mathbb{E}_q\left[\mathsf{ALG}_j(\sigma_i)\right] \geq c \cdot \mathsf{OPT}(\sigma_i).$$

Dies ist äquivalent dazu, folgende Ungleichung zu beweisen:

$$\min_q \max_i \left\{\mathbb{E}_q\left[\mathsf{ALG}_j(\sigma_i)\right] - c \cdot \mathsf{OPT}(\sigma_i)\right\} \geq 0. \tag{5.2}$$

Wir konstruieren folgendes Zwei-Personen Nullsummenspiel: Der Online-Spieler hat die reinen Strategien $\{\mathsf{ALG}_j\}$, der Adversary die reinen Strategien $\{\sigma_i\}$. Wählt der Online-Spieler den Algorithmus ALG_j und der Gegner die Eingabe σ_i, so ist die «Auszahlung» an den Gegner $m_{ij} := \mathsf{ALG}_j(\sigma_i) - c \cdot \mathsf{OPT}(\sigma_j)$. Der Online-Spieler möchte seinen Verlust minimieren, der Gegner den Gewinn maximieren.

Die Ungleichung (5.2) schreibt sich dann im Rahmen unseres Spiels als $\min_q \max_i e_i^T Mq \geq 0$. Mit Yao's Prinzip folgt, daß es, um (5.2) zu beweisen, genügt, eine Verteilung $\bar{p}$ auf den Eingabesequenzen zu finden, so daß $\min_j \bar{p}^T Me_j \geq 0$ gilt, d.h.

$$\min_j \mathbb{E}_{\bar{p}} \left[\mathsf{ALG}_j(\sigma_i) - c \cdot \mathsf{OPT}(\sigma_i)\right] \geq 0$$
$$\Leftrightarrow \min_j \mathbb{E}_{\bar{p}} \left[\mathsf{ALG}_j(\sigma_i)\right] \geq c \cdot \mathbb{E}_{\bar{p}} \left[\mathsf{OPT}(\sigma_i)\right].$$

Hierbei bezeichnet $\mathbb{E}_{\bar{p}} \left[\mathsf{ALG}(\sigma_i)\right]$ die erwarteten Kosten des *deterministischen* Algorithmus ALG bezüglich der Verteilung $\bar{p}$ auf den Eingabesequenzen. Wir haben somit folgenden Satz (im endlichen Fall) bewiesen:

Satz 5.4 (Yao's Prinzip für Online-Probleme) *Sei $\bar{p}$ eine Verteilung auf den Eingabesequenzen $\{\sigma_i\}$ eines Online-Problems mit der Eigenschaft, daß für jeden deterministischen Online-Algorithmus* ALG *gilt:*

$$\mathbb{E}_{\bar{p}} \left[\mathsf{ALG}(\sigma_i)\right] \geq c \cdot \mathbb{E}_{\bar{p}} \left[\mathsf{OPT}(\sigma_i)\right]. \tag{5.3}$$

Dann ist c eine untere Schranke für die Kompetitivität des besten randomisierten Algorithmus gegen den blinden Gegner. □

Wo liegt der Gewinn durch Yao's Prinzip? Um eine untere Schranke mittels (5.3) zu beweisen, müssen wir «nur» jeden *deterministischen* Online Algorithmus *im Durchschnitt* schlecht aussehen lassen. Dies ist normalerweise viel einfacher, als einen beliebigen *randomisierten* Algorithmus hereinzulegen.

5.3 Beispiele für die Anwendung von Yao's Prinzip

Wir betrachten als erstes das Skifahrerproblem mit Kaufkosten $B := 10$. Wir konstruieren eine Verteilung über Eingabesequenzen wie folgt: Mit Wahrscheinlichkeit $1/2$ wird nur einmal Skifahren angefragt, mit Wahrscheinlichkeit $1/2$ gibt es 20 Tage Skiwetter.

Die erwarteten optimalen Kosten sind dann

$$\mathbb{E}\left[\mathsf{OPT}(\sigma)\right] = \frac{1}{2} \cdot 1 + \frac{1}{2} 10 = \frac{11}{2}.$$

Sei ALG_j der Online-Algorithmus, welcher am j ten Tag Skier kauft. Dann gilt:

$$\mathbb{E}\left[\mathsf{ALG}_j(\sigma)\right] = \begin{cases} 10 & \text{für } j = 1 \\ \frac{1}{2} \cdot 1 + \frac{1}{2}(j - 1 + 10) & \text{für } j = 2, \dots, 20. \\ 20 & \text{für } j > 20. \end{cases}$$
$$\geq \frac{1}{2} + \frac{11}{2} = 6.$$

Folglich gilt:

$$\frac{\mathbb{E}\left[\mathsf{ALG}_j(\sigma)\right]}{\mathbb{E}\left[\mathsf{OPT}(\sigma)\right]} \geq \frac{6}{\frac{11}{2}} = \frac{12}{11}$$

für alle j. Mit Hilfe von Yao's Prinzip erhalten wir somit eine untere Schranke von $12/11$ für die Kompetitivität eines randomisierten Algorithmus für das Skifahrerproblem mit Kaufkosten $B = 10$.

Unser nächstes Beispiel erfordert etwas mehr Einsatz. Wir betrachten das Paging Problem aus Kapitel 4 und beweisen folgenden Satz:

Satz 5.5 (Untere Schranke für Paging) *Jeder randomisierte Algorithmus für Paging mit Cachegröße k hat Kompetitivität größer oder gleich H_k gegen den blinden Gegner.*

Beweis: Unsere Konstruktion arbeitet auf einer Teilmenge P der Größe $k+1$ der gesamten Seiten: dem anfänglichen Cacheinhalt und einer zusätzlichen Seite. Sei $\bar{p}$ eine Wahrscheinlichkeitsverteilung über den Anfragesequenzen mit der Eigenschaft, daß die ℓte Anfrage eine zufällige Seite aus P gleichverteilt und unabhängig von allen vorherigen Anfragen ist.

Offenbar hat dann jeder deterministische Paging Algorithmus mit Cachegröße k bei jeder Anfrage mit Wahrscheinlichkeit $1/(k+1)$ einen Seitenfehler. Dies ergibt

$$\mathbb{E}_{\bar{p}}\left[\mathsf{ALG}(\sigma^n)\right] = \frac{n}{k+1}. \tag{5.4}$$

für Anfragesequenzen der Länge n. Nach Yao's Prinzip ist daher

$$r := \lim_{n\to\infty} \frac{n}{(k+1)\mathbb{E}_{\bar{p}}\left[\mathsf{OPT}(\sigma^n)\right]}$$

eine untere Schranke für die Kompetitivität gegen den blinden Gegner. Unser Ziel ist es $r \geq H_k$ zu beweisen. Dies ist zu folgender Ungleichung äquivalent:

$$\lim_{n\to\infty} \frac{n}{\mathbb{E}_{\bar{p}}\left[\mathsf{OPT}(\sigma^n)\right]} = (k+1)H_k. \tag{5.5}$$

Um (5.5) zu beweisen, müssen wir den optimalen Offline-Algorithmus OPT genauer untersuchen. Sei dazu σ^n eine zufällig gemäß unserer Verteilung $\bar{p}$ gezogene Anfragesequenz. Wir unterteilen die Anfragesequenz σ^n wie folgt in stochastische Phasen: Die Unterteilung entspricht unserer k-Phasenpartitionierung mit dem Unterschied, daß wir jetzt stochastische Phasen haben und die Anfänge/Enden der Phasen durch Zufallsvariablen angegeben werden.

Sei X_i, $i = 1, 2, \ldots$ die Folge der Zufallsvariablen, wobei X_i die Anzahl der Anfragen in der $i$ten Phase angibt. Man beachte, daß die X_i gleichverteilt und unabhängig sind. Die $j$te Phase beginnt dann mit der Anfrage $r_{S_j-1}+1$, wobei $S_j := \sum_{i=1}^{j-1} X_j$. Wir definieren nun noch die Anzahl $N(n)$ der vollständigen Phasen:

$$N(n) := \max\{\, j : S_j \leq n \,\}.$$

Unser Beweis von (5.5) verläuft jetzt in folgenden Schritten:

(i) Wir zeigen, daß

$$\mathsf{OPT}(\sigma^n) \leq N(n) + 1. \tag{5.6}$$

(ii) Wir beweisen

$$\lim_{n\to\infty} \frac{n}{\mathbb{E}_{\bar{p}}[N(n)]} = \mathbb{E}_{\bar{p}}[X_i]\,. \tag{5.7}$$

(iii) Der Erwartungswert von X_i erfüllt

$$\mathbb{E}_{\bar{p}}[X_i] = (k+1)H_k. \tag{5.8}$$

Aus (5.6)– (5.8) folgt dann die Ungleichung (5.5).

Zu (i) Am Ende der Phase j hat OPT alle in der Phase gefragten Seiten im Cache (mit Ausnahme der letzten möglicherweise unvollständigen Phase). Die Phase $j+1$ beginnt mit einem Seitenfehler. Wenn OPT nun die Seite aus dem Cache entfernt, welche als erstes in Phase $j+2$ gefragt wird, so kann OPT jede Phase mit genau einem Seitenfehler bearbeiten. Dies zeigt (5.6).

Zu (ii) Die Familie von Zufallsvariable $\{\, N(n) \,:\, n \in \mathbb{N} \,\}$ bildet einen sogenannten Erneuerungsprozess. Die Gleichung (5.7) ist dann Folge des sogenannten *Erneuerungssatzes* (Beweis siehe z.B. [10, Anhang E]).

Zu (iii) Die Berechnung des Erwartungswertes von X_i führt uns auf das sogenannte «Coupon-Sammler-Problem»: Gegeben seien L Coupons (jeweils in unendlichem Vorrat). In jeder Runde zieht ein Sammler zufällig, gleichverteilt und unabhängig von den vorherigen Runden einen Coupon. Wie viele Runden muß der Sammler im Erwartungswert warten, bis er alle Coupons hat?

Unser Problem schildert sich im Licht des Coupon-Sammler-Problems wie folgt: Eine Phase endet einen Schritt vorher, bevor wir alle $(k+1)$ «Coupons»für Seiten gesammelt haben. Somit ist der Erwartungsert von X_i genau der erwarteten Anzahl der Runden im Coupon-Sammler-Problem minus 1.

Wir lösen das Coupon-Sammler-Problem. Sei Z_i, $i = 1, \ldots, L-1$ die Zufallsvariable die anzeigt, wieviele Runden nötig sind um einen neuen Coupon zu sammeln, wenn man bereits i Coupons besitzt. Wenn man bereits i Coupons besitzt, ist die Wahrscheinlichkeit, in der nächsten Runde einen neuen Coupon zu erwischen, genau $(L-i)/L$. Somit haben wir

$$\mathbb{E}[Z_i] = \frac{L}{L-i}$$

und damit

$$\mathbb{E}\left[\sum_{i=1}^{L-1} Z_i\right] = \sum_{i=1}^{L-1} \frac{L}{L-i} = LH_L.$$

In unserem Fall ist $L = k+1$ und $\mathbb{E}_{X_i}\,[=]\,\mathbb{E}\left[\sum_{i=1}^{k} Z_i\right] - 1 = (k+1)H_k$.

Dies beendet den Beweis. □

6

Telekommunikation und Netzwerk Routing

Referenzwerke: [10, Kapitel 12 und 13]

Online Routing in Netzwerken ist eine Aufgabe, die in zahlreichen Anwendungen in der Telekommunikation auftritt. Beispielsweise sind das Verteilen von Telephongesprächen oder von Video-on-Demand in natürlicher Weise Online-Probleme, da zukünftige Anfragen in der Regel nicht bekannt sind. Es stellt sich dann auch das Problem, ob man eventuell eine bearbeitbare Anfrage (Telephongespräch, Filmabruf) ablehnt, um eventuell Engpässe im Netzwerk zu vermeiden (siehe Abbildung 6.1).

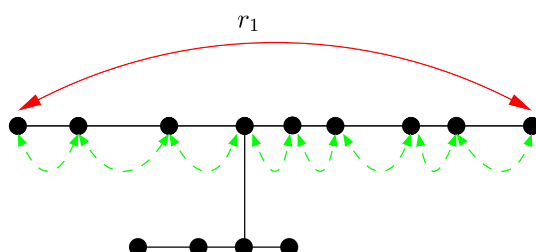


Abbildung 6.1
Im Beispiel ist die Leitungskapazität jeder Netzwerkkante gleich 1, d.h. über jede Kante kann maximal ein Telephongespräch gleichzeitig geroutet werden. Akzeptiert ein Algorithmus den Anruf r_1, so müssen alle weiteren gestrichelt eingezeichneten Anrufe abgelehnt werden.

Aus theoretischer Sicht ist beim *Online Routing* ein Netzwerk $G = (V, E)$ gegeben. Eine Anfrage $r_j = (s_j, t_j, b_j)$ ist die Aufforderung, eine Verbindung zwischen den Knoten s_j und t_j mit Bandbreite b_j herzustellen. Wird diese Verbindung auf dem Pfad P im Netzwerk G geroutet, so erhöht sich die Last auf jeder Kante von P um die Bandbreite b_j. Man unterscheidet zwei Hauptziele beim Routing:

Lastbalancierung In diesem Fall dürfen keine Anfragen abgelehnt werden. Das Optimierungsziel ist es, die maximale Last auf einer Kante, d.h. die Zielfunktion,

$$\max_{e \in E} \sum_{j\colon r_j \text{ wurde über } e \text{ geroutet}} b_j$$

zu minimieren.

Durchsatz Bei der Durchsatzmaximierung sind im Netzwerk zusätzlich Kanten-Kapazitäten $u\colon E \to \mathbb{R}_{\geq 0}$ gegeben. Anfragen dürfen abgelehnt

werden. Eine Anfrage $r_j = (s_j, t_j, b_j)$ kann allerdings nur dann angenommen werden, wenn noch genügend Kapazität im Netz zur Verfügung steht, d.h., wenn ein Pfad P zwischen s_j und t_j existiert, so daß jede Kante auf P noch b_j freie Kapazität besitzt.

Das Optimierungsziel ist es, den »Durchsatz«, d.h. die Summe der Bandbreiten der akzeptierten Anfragen, zu maximieren.

6.1 Lastbalancierung

Wir betrachten das Lastbalancierungsproblem in etwas allgemeinerer Form. In diesem Fall ist bei einer Anfrage $r_j = (s_j, t_j, b_j)$ der dritte Parameter b_j eine Funktion $b_j : E \to \mathbb{R}_{\geq 0}$. Wird die Anfrage r_j über eine Kante $e \in E$ geroutet, so erhöht sich die Last auf der Kante $e \in E$ um $b_j(e)$.

Scheduling

Dieses verallgemeinerte Lastbalancierungsproblem enthält als Spezialfall eine Zahl von *Schedulingproblemen*. Beim *Scheduling* hat man N Maschinen. Eine Anfrage (Job) ist ein N-Tupel $r_j = (r_j(1), \ldots, r_j(N))$, wobei $r_j(i) \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ die Last ist, die der Job j auf der Maschine i verursacht. Ziel ist es, die Jobs so auf die Maschinen zu verteilen, daß die Gesamtbearbeitungszeit (»Makespan«) minimiert wird. Man möchte also die Zeit minimieren, zu der die letzte Maschine fertig wird. Dies ist offenbar äquivalent dazu, die maximale Last auf den Maschinen zu minimieren.

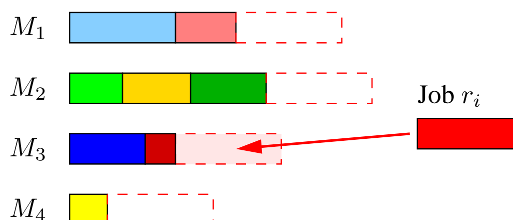
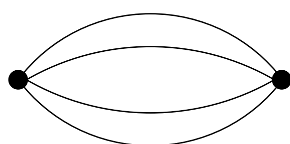


Abbildung 6.2
Scheduling auf 4 Maschinen $M_1, \ldots, M_4$. Der neue Job r_i erzeugt gleiche Last auf jeder der 4 Maschinen (durch die gestrichelten Blöcke angedeutet). Er wird im Beispiel der Maschine M_3 zugewiesen.



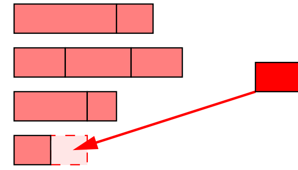
Das oben genannte Schedulingproblem läßt sich als Netzwerk-Lastbalancierungsproblem formulieren: Dabei besteht der Graph nur aus zwei Knoten s und t. Für jede Maschine i existiert eine Kante e_i von s nach t, so daß man insgesamt ein Büschel von N Parallelen hat. Ein Job $(r_j(1), \ldots, r_j(N))$ entspricht dann dem Routingrequest (s, t, b_j) mit $b_j(e_i) = r_j(i)$.

Bei *identischen Maschinen* ist $r_j(1) = r_j(2) = \cdots = r_j(N)$ für alle Jobs. Bei *verwandten Maschinen* hat jede Maschine i einen konstanten Geschwindigkeitsfaktor α_i und $r_j(i) = v_j/\alpha_i$ für einen Wert v_j. Bei *Jobs mit Maschinenrestriktionen* ist jeder Jobeintrag $r_j(i)$ entweder v_j oder ∞ mit der Bedeutung, daß ein Job nur auf Maschinen bearbeitet werden kann, wo er eine endliche Last verursacht. Im allgemeinen Fall, wo keinerlei Einschränkungen an die Jobs gemacht werden, spricht man von *unverwandten Maschinen*.

6.1.1 Spezialfall: Scheduling für identische Maschinen

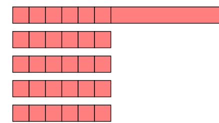
Wir betrachten zunächst den einfachsten Fall der Lastbalancierung im Scheduling bei identischen Maschinen.

Algorithmus $\textsf{GRAHAM}$ Weise den Job $r_j = (v_j, \ldots, v_j)$ einer beliebigen Maschine mit der aktuell geringsten Last zu.

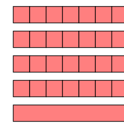


Satz 6.1 *Der Algorithmus* $\textsf{GRAHAM}$ *ist* $(2 - 1/N)$*-kompetitiv. Diese Schranke ist scharf.*

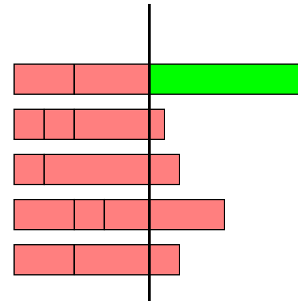
Beweis: Wir zeigen zuerst, daß $\textsf{GRAHAM}$ nicht besser als $(2-1/N)$-kompetitiv ist. Dazu wählen wir eine Folge, die aus $N(N-1)$ Jobs der Größe 1 und zuletzt aus einem Job der Größe N besteht. $\textsf{GRAHAM}$ hat nach der Zuweisung der ersten $N(N-1)$ Jobs auf jeder Maschine Last $N-1$. Der letzte Job erzeugt dann auf einer Maschine Last $2N-1$.



Die optimale Offline-Lösung benutzt zur Zuweisung der ersten $N(N-1)$ Jobs nur $N-1$ Maschinen und legt den letzten Job auf die noch freie Maschine. Dies ergibt eine maximale Last von N.



Sei $\sigma = (r_1, \ldots, r_n)$ eine beliebige Anfragenfolge. O. B. d. A. sei Maschine 1 die Maschine, auf der $\textsf{GRAHAM}$ die maximale Last bei Eingabe von σ erzeugt. Sei w die Last des letzten Jobs, der auf Maschine 1 gelegt wird, und l die vorhergehende Last auf Maschine 1. Dann gilt $\textsf{GRAHAM}(\sigma) = l + w$.



Die Last jeder anderen Maschine ist mindestens l, denn zum Zeitpunkt, zu dem der Job auf 1 gelegt wurde, war Maschine 1 eine Maschine mit geringster Last. Also ist die Summe aller Jobgrößen mindestens $Nl + w$. Somit folgt

$$\textsf{OPT}(\sigma) \geq \frac{Nl + w}{N} = l + \frac{w}{N}.$$

Mit $\textsf{OPT}(\sigma) \geq w$ ergibt sich:

$$\begin{aligned} \textsf{GRAHAM}(\sigma) = w + l &\leq w + \textsf{OPT}(\sigma) - \frac{w}{N} \\ &= \textsf{OPT}(\sigma) + \left(1 - \frac{1}{N}\right) w \\ &\leq \textsf{OPT}(\sigma) + \left(1 - \frac{1}{N}\right) \textsf{OPT}(\sigma) \\ &= \left(2 - \frac{1}{N}\right) \textsf{OPT}(\sigma) \end{aligned}$$

Dies war zu zeigen. □

6.1.2 Virtual Circuit Routing in Netzwerken

Wir gehen nun zum generellen Lastbalancierungsproblem in Netzwerken über, welches in der Literatur auch unter den Namen »*Virtual Circuit Routing*« bekannt ist.

Virtual Circuit Routing

Im folgenden sei $\sigma = (r_1, \ldots, r_n)$ mit $r_j = (s_j, t_j, b_j)$, $b_j \colon E \to \mathbb{R}_{\geq 0}$ eine Anfragefolge. Wir nehmen zunächst an, daß wir eine obere Schranke $\Lambda \geq \textsf{OPT}(\sigma)$ kennen. Wir zeigen später, wie wir dieses scheinbare Problem beheben können.

Für $j = 1, \ldots, n$ sei P_j der Weg, auf denen der folgende Algorithmus die Anfrage r_j routet. Wir bezeichnen mit

$$L_j(e) = \sum_{\substack{i \leq j: \\ e \in P_i}} b_i(e)$$

Last

die *Last* auf der Kante e nach Routing der ersten j Anfragen. Es sei

$$\widetilde{L}_j(e) = \frac{L_j(e)}{\Lambda}$$

skalierte Last

die *skalierte Last* auf e und

$$\widetilde{b}_j(e) = \frac{b_j(e)}{\Lambda}$$

skalierte Bandbreite

die *skalierte Bandbreite* der Anfrage r_j, wenn sie auf Kante e geroutet würde.

Algorithmus ROUTE-EXP$_\Lambda$ Wähle $0 < \gamma < 1$ und setze $a = 1 + \gamma$.

1. Sei $r_j = (s_j, t_j, b_j)$ die aktuelle Anfrage.
2. Berechne einen kürzesten Weg P_j von s_j nach t_j in G bezüglich der Kantengewichte

$$c_j(e) = a^{\widetilde{L}_{j-1}(e)+\widetilde{b}_j(e)} - a^{\widetilde{L}_{j-1}(e)} = a^{\widetilde{L}_{j-1}(e)} \left(a^{\widetilde{b}_j(e)} - 1 \right). \quad (6.1)$$

3. Route die Anfrage r_j auf dem Pfad P_j.

Lemma 6.2 *Für* $\gamma > 0$ *und* $a = 1 + \gamma$ *gilt:*

$$a^x - 1 \leq \gamma x \qquad \text{für alle } x \in [0, 1].$$

Beweis: Ausrechnen. □

Satz 6.3 *Wenn* $\text{OPT}(\sigma) \leq \Lambda$, *dann ist* $\text{ROUTE-EXP}_\Lambda(\sigma) \leq \mathcal{O}(\log m) \cdot \Lambda$.

Beweis: Wir bezeichnen mit P_j und P_j^* die Wege, auf denen ALG_Λ bzw. OPT die Anfrage r_j routen. Da P_j ein kürzester Weg bezüglich der Kantenwichte aus (6.1) ist, gilt:

$$\sum_{e \in P_j} c_j(e) \leq \sum_{e \in P_j^*} c_j(e).$$

Setzt man die Definition von c_j ein, so ergibt sich zusammen mit der Eigenschaft $\widetilde{L}_j(e) \leq \widetilde{L}_n(e)$, daß gilt:

$$\begin{aligned}
\sum_{e \in P_j} a^{\widetilde{L}_{j-1}(e)} \left(a^{\widetilde{b}_j(e)} - 1 \right) &\leq \sum_{e \in P_j^*} a^{\widetilde{L}_{j-1}(e)} \left(a^{\widetilde{b}_j(e)} - 1 \right) \\
&\leq \sum_{e \in P_j^*} a^{\widetilde{L}_n(e)} \left(a^{\widetilde{b}_j(e)} - 1 \right) \\
&\leq \gamma \sum_{e \in P_j^*} a^{\widetilde{L}_n(e)} \widetilde{b}_j(e) \quad \text{nach Lemma 6.2.}
\end{aligned}$$

Summation über $j = 1, \dots, n$ ergibt

$$\begin{aligned}
&\sum_{j=1}^{n} \sum_{e \in P_j} a^{\widetilde{L}_{j-1}(e)} \left(a^{\widetilde{b}_j(e)} - 1\right) \le \sum_{j=1}^{n} \gamma \sum_{e \in P_j^*} a^{\widetilde{L}_n(e)} \widetilde{b}_j(e) \\
\Leftrightarrow \quad &\sum_{e \in E} \sum_{j : e \in P_j} a^{\widetilde{L}_{j-1}(e)} \left(a^{\widetilde{b}_j(e)} - 1\right) \le \gamma \sum_{e \in E} a^{\widetilde{L}_n(e)} \sum_{j : e \in P_j^*} \widetilde{b}_j(e) \\
&\le \gamma \sum_{e \in E} a^{\widetilde{L}_n(e)}. \qquad (6.2)
\end{aligned}$$

Dabei haben wir benutzt, daß

$$\sum_{j : e \in P_j^*} \widetilde{b}_j(e) = \frac{1}{\Lambda} \sum_{j : e \in P_j^*} b_j(e) \le \frac{\mathsf{OPT}(\sigma)}{\Lambda} \le 1.$$

Für festes $e \in E$ gilt:

$$\begin{aligned}
\sum_{j : e \in P_j} a^{\widetilde{L}_{j-1}(e)} \left(a^{\widetilde{b}_j(e)} - 1\right) &= \sum_{j : e \in P_j} a^{\overbrace{\widetilde{L}_{j-1}(e) + \widetilde{b}_j(e)}^{=\widetilde{L}_j(e)}} - a^{\widetilde{L}_{j-1}(e)} \\
&= a^{\widetilde{L}_n(e)} - 1.
\end{aligned}$$

Aus (6.2) folgt damit

$$\begin{aligned}
&\sum_{e \in E} a^{\widetilde{L}_n(e)} - m \le \gamma \sum_{e \in E} a^{\widetilde{L}_n(e)} \\
\Leftrightarrow \quad &\sum_{e \in E} a^{\widetilde{L}_n(e)} \le \frac{m}{1-\gamma} \\
\Rightarrow \quad &\max_{e \in E} a^{\widetilde{L}_n(e)} \le \frac{m}{1-\gamma} \\
\Rightarrow \quad &\max_{e \in E} \widetilde{L}_n(e) \le \log_a \left(\frac{m}{1-\gamma}\right) \\
\Rightarrow \quad &\max_{e \in E} L_n(e) \le \Lambda \cdot \log_a \left(\frac{m}{1-\gamma}\right)
\end{aligned}$$

Wenn $0 < \gamma < 1$ fest ist, dann gilt $\log_a \left(\frac{m}{1-\gamma}\right) \in \mathcal{O}(\log m)$ und die Aussage des Satzes folgt. □

Wir zeigen nun, daß die fehlende Kenntnis einer geeigneten Schranke $\Lambda \ge \mathsf{OPT}(\sigma)$ nicht tragisch ist.

Satz 6.4 *Sei ALG_Λ ein parametrisierter Online Algorithmus für das Lastbalancierungsproblem mit folgender Eigenschaft: Wenn $\mathsf{OPT}(\sigma) \le \Lambda$, dann gilt $\mathsf{ALG}_\Lambda(\sigma) \le c \cdot \Lambda$. Dann gibt es einen strikt $4c$-kompetitiven Algorithmus ALG für das Lastbalancierungsproblem.*

Beweis: Der Algorithmus ALG arbeitet in Phasen. Jede Phase entspricht einer neuen Schätzung für den Parameter Λ. Anfangs ist $\Lambda := \Lambda_0 = \mathsf{OPT}(r_1)$. In der Phase k wird Λ auf den Wert $2^k\Lambda_0$ gesetzt.

Am Anfang jeder Phase »säubert der Algorithmus ALG sein Gedächtnis«: Er ignoriert alle in vorherigen Phasen vorgenommenen Zuweisungen. Alle Zuweisungen in der Phase k werden von ALG durch Konsultieren von ALG_{Λ_k} vorgenommen. Dies funktioniert wie folgt.

$\sigma' r_j = \underbrace{r_{j-s}, \ldots, r_{j-1}}_{\text{(in Phase } k \text{ zugew., evtl. leer)}}, r_j$

Sei r_j die aktuelle Anfrage und sei σ' die Teilfolge aller Anfragen von σ, die bereits in Phase k zugewiesen worden sind. Der Algorithmus ALG testet, ob $\mathsf{ALG}_{\Lambda_k}(\sigma' r_j) \leq c \cdot \Lambda_k$ ist. Falls ja, dann wird r_j gemäß ALG_{Λ_k} verarbeitet. Wenn $\mathsf{ALG}_{\Lambda_k}(\sigma' r_j) > c \cdot \Lambda_k$, dann wird die Phase k beendet und die Phase $k+1$ gestartet.

Wir müssen zeigen, daß $\mathsf{ALG}(\sigma) \leq 4c \cdot \mathsf{OPT}(\sigma)$ gilt. Dazu nehmen wir an, daß der Algorithmus in Phase h endet. Falls $h = 0$, dann gilt

$$\mathsf{ALG}(\sigma) = \mathsf{ALG}_{\Lambda_0}(\sigma) \leq c \cdot \Lambda_0 = c \cdot \mathsf{OPT}(r_1) \leq c \cdot \mathsf{OPT}(\sigma).$$

In diesem Fall ist nichts mehr zu zeigen. Sei daher $h > 0$. Wir betrachten die (möglicherweise leere) Teilsequenz $\sigma^{(h-1)}$, die in Phase $h-1$ verarbeitet wurde, und die Anfrage r_j, die zum Beenden der Phase $h-1$ führte. Es gilt:

$$\mathsf{ALG}_{\Lambda_{h-1}}(\sigma' r_j) > c \cdot \Lambda_{h-1}. \tag{6.3}$$

Es muß $\mathsf{OPT}(\sigma' r_j) > \Lambda_{h-1}$ gelten, denn sonst wäre nach Voraussetzung an ALG_Λ die Bedingung (6.3) nicht verletzt. Somit ist

$$\mathsf{OPT}(\sigma) \geq \mathsf{OPT}(\sigma' r_j) > \Lambda_{h-1} = 2^{h-1}\Lambda_0. \tag{6.4}$$

Daher folgt:

$$\mathsf{ALG}(\sigma) = \sum_{k=0}^{h} \mathsf{ALG}_{\Lambda_k}(\sigma^{(k)}) \leq \sum_{k=0}^{h} c 2^k \Lambda_0 = (2^{h+1}-1)c\Lambda_0 \overset{(6.4)}{\leq} 4c \cdot \mathsf{OPT}(\sigma).$$

Dies beendet den Beweis. □

Aus den Sätzen 6.3 und 6.4 ergibt sich nun das folgende Korollar.

Korollar 6.5 *Es gibt einen $\mathcal{O}(\log m)$-kompetitiven Algorithmus für das Lastbalancierungsproblem.* □

6.2 Durchsatzmaximierung

Admission Control

Das Problem der Durchsatzmaximierung ist in der Literatur auch unter dem Begriff *Admission Control* zu finden. Dieser Name wird aus der Aufgabenstellung motiviert, zu entscheiden, welche Anfragen man akzeptieren und welche man ablehnen soll.

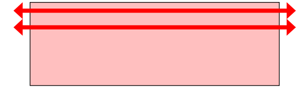
Im folgenden sei $\sigma = (r_1, \ldots, r_n)$ mit $r_j = (s_j, t_j, b_j)$ eine Anfragefolge. Sei ALG ein Online Algorithmus. Analog zu Abschnitt 6.1.2 definieren wir als $L_j(e)$

die Last auf der Kante e nach Bearbeitung (Routen, Ablehnung) der ersten j Anfragen.

Seien A_j und R_j die Menge der Indizes der akzeptierten bzw. abgelehnten Anfragen aus $r_1, \ldots, r_j$. Für $j \in A_j$ sei wieder P_j der Weg, auf dem r_j geroutet wurde. Dann ist

$$L_j(e) = \sum_{\substack{k \in A_j \\ e \in P_k}} b_k.$$

Sei D der Durchmesser des Graphen $G = (V, E)$, d.h. die Länge (in Kanten) des längsten einfachen Weges in G. Wir definieren $\mu := 2(D+1)$. Im folgenden machen wir folgende einschränkende Annahme:



Bandbreiteneinschränkung: Die einzelnen Anfragen benötigen nur relativ wenig Brandbreite im Vergleich zur Kantenkapazität.

Annahme 6.6 (Bandbreiteneinschränkung) *Es gilt für jede Anfrage $r_j = (s_j, t_j, b_j)$ die Ungleichung:*

$$b_j \leq \frac{U}{\log \mu},$$

wobei $U := \min\{\, u(e) : e \in E \,\}$ die geringste Kapazität einer Kante im Netzwerk ist und $\log$ den Logarithmus zur Basis 2 bezeichnet.

Anschaulich bedeutet die Annahme 6.6, daß die Bandbreiten der einzelnen Anfragen „relativ klein" zur Bandbreite sind, die auf den einzelnen Kanten zur Verfügung steht.

Algorithmus LOWBANDWITH Akzeptiere Anfragen nach folgender Strategie:

1. Sei $r_j = (s_j, t_j, b_j)$ die aktuelle Anfrage.
2. Berechne einen kürzesten Weg W von s_j nach t_j in G bezüglich der Kantengewichte

$$c_j(e) = \frac{1}{D}\left(\mu^{L_{j-1}(e)/u(e)} - 1\right) \tag{6.5}$$

3. Falls $c_j(W) \leq 1$, dann setze $P_j := W$ und route die Anfrage r_j auf dem Pfad P_j.
4. Ansonsten lehne r_j ab.

Lemma 6.7 *Unter der Annahme 6.6 liefert der Algorithmus LOWBANDWITH ein gültiges Routing der akzeptierten Anfragen, d.h. die Kapazitäten auf den Kanten werden nicht überschritten.*

Beweis: Wenn $L_{j-1}(e) + b_j > u(e)$, dann gilt

$$L_{j-1}(e) > u(e) - b_j \geq u(e) - \frac{U}{\log \mu} \geq u(e)\left(1 - \frac{1}{\log \mu}\right).$$

In diesem Fall ist

$$c_j(e) > \frac{1}{D}\left(\mu^{1-1/\log \mu} - 1\right) \geq \frac{1}{D}\left(\frac{\mu}{2} - 1\right) = 1.$$

Somit wird die Kante e vom Algorithmus in keinem Pfad mehr verwendet. □

Lemma 6.8 (Untere Schranke für den Profit) *Es gilt für $j = 1, \dots, n$:*

$$2 \log \mu \sum_{k \in A_{j-1}} b_k \geq \sum_{e \in E} u(e) c_j(e). \tag{6.6}$$

Beweis: Wir beweisen die Aussage durch Induktion nach j. Für $j = 1$ ist $L_{j-1}(e) = 0$ für alle $e \in E$, also $c_j(e) = 0$ für alle $e \in E$.

Sei die Aussage bereits für $j - 1$ bewiesen. Wir zeigen, daß sie auch nach der Bearbeitung der $j$ten Anfrage weiterhin gilt. Wenn die $j$te Anfrage abgelehnt wird, dann verändert sich keine der beiden Seiten der Ungleichung (6.6). In diesem Fall ist nichts zu zeigen. Wir können also annehmen, daß die Anfrage $r_j = (s_j, t_j, b_j)$ vom Algorithmus geroutet wird.

Sei P_j der für das Routen benutzte Pfad. Für $e \in P_j$ gilt:

$$\begin{aligned}
D(c_{j+1}(e) - c_j(e)) &= \mu^{L_j(e)/u(e)} - \mu^{L_{j-1}(e)/u(e)} \\
&= \mu^{(L_{j-1}(e)+b_j)/u(e)} - \mu^{L_{j-1}(e)/u(e)} \\
&= \mu^{L_{j-1}(e)/u(e)} \left(\mu^{b_j/u(e)} - 1\right) \\
&= \mu^{L_{j-1}(e)/u(e)} \left(2^{\overbrace{\log \mu \cdot b_j/u(e)}^{\leq 1}} - 1\right) \\
&\leq \mu^{L_{j-1}(e)/u(e)} \log \mu \cdot \frac{b_j}{u(e)} \quad \text{(nach Lemma 6.2)} \\
&= D\left(c_j(e) + \frac{1}{D}\right) \log \mu \cdot \frac{b_j}{u(e)} \quad \text{(nach Def. von } c_j(e)\text{).}
\end{aligned}$$

Wenn man über alle Kanten in P_j aufsummiert, erhält man, daß sich die rechte Seite der Ungleichung (6.6) um höchstens

$$\begin{aligned}
\log \mu \cdot b_j \sum_{e \in P_j} \left(c_j(e) + \frac{1}{D}\right) &\leq \log \mu \cdot b_j \Big(\underbrace{\sum_{e \in P_j} c_j}_{\substack{\leq 1 \\ \text{da } r_j \text{ akzeptiert}}} + \underbrace{\sum_{e \in P_j} 1/D}_{\substack{\leq 1 \\ \text{da } D \text{ Diameter}}} \Big) \\
&\leq 2 \log \mu \cdot b_j
\end{aligned}$$

erhöht. Daher bleibt die Ungleichung (6.6) erhalten. □

Lemma 6.9 (Schranke für die abgelehnten Anfragen) *Sei Q die Menge derjenigen Indizes $j \in \{1, \dots, n\}$ mit der Eigenschaft, daß in OPT$[\sigma]$ die Anfrage r_j geroutet wird, der Algorithmus* LOWBANDWITH *bei Eingabe von σ diese Anfrage aber ablehnt. Dann gilt für $j = 1, \dots, n$:*

$$\sum_{k \in Q} b_k \leq \sum_{e \in E} u(e) c_n(e).$$

Beweis: Da r_j von LOWBANDWITH abgelehnt wurde, gibt es bei Bekanntwerden von r_j für den Algorithmus keinen Weg mit Kosten höchstens 1. Insbesondere gilt für den von OPT verwendeten Weg P_j^*, daß $\sum_{e \in P_j^*} c_j(e) > 1$. Da die

Kosten c_j einer Kante nie fallen, gilt auch $\sum_{e \in P_j^*} c_n(e) > 1$. Somit haben wir

$$\begin{aligned}\sum_{j \in Q} b_j &= \sum_{j \in Q} b_j \cdot 1 \\ &< \sum_{j \in Q} \sum_{e \in P_j^*} b_j c_n(e) \\ &= \sum_{e \in E} c_n(e) \sum_{\substack{j \in Q: \\ e \in P_j^*}} b_j \\ &\le \sum_{e \in E} c_n(e) u(e)\end{aligned}$$

Dies war zu zeigen. □

Satz 6.10 *Unter der Bandbreiteneinschränkung (Annahme 6.6) ist der Algorithmus* LOWBANDWITH $\mathcal{O}(\log D)$*-kompetitiv.*

Beweis: Nach Lemma 6.9 gilt:

$$\begin{aligned}\mathsf{OPT}(\sigma) - \mathsf{LOWBANDWITH}(\sigma) &\le \sum_{e \in E} c_n(e) u(e) \\ &\le 2 \log \mu \cdot \mathsf{LOWBANDWITH}(\sigma) \quad \text{(nach Lemma 6.8).}\end{aligned}$$

Daher folgt:

$$\mathsf{LOWBANDWITH}(\sigma) \ge \frac{1}{2 \log \mu - 1} \cdot \mathsf{OPT}(\sigma) \ge \Omega\left(\frac{1}{\log D}\right) \cdot \mathsf{OPT}(\sigma).$$

Dies beendet den Beweis. □

6.3 Routing in optischen Netzwerken

Optische Netzwerke sind die Telekommunikationsnetzwerke der Zukunft. Daten werden als optische Signale über Glasfaserkabel gesendet und an den Netzwerkknoten optisch verschaltet. Letztere Eigenschaft erfordert eine andere mathematische Modellierung als die in Abschnitt 6.2 für herkömmliche Netzwerke beschriebene. Die hohe Übertragungskapazität in optischen Netzwerken wird durch das sogenannte *Wavelength Division Multiplexing (WDM)* ermöglicht, bei dem eine Faser virtuell vervielfacht wird, indem die verfügbare Bandbreite in mehrere Wellenlängen(bereiche) partitioniert wird. Mehrere Signale können nun gleichzeitig dieselbe Faser benutzen, indem man sie in verschiedenen Wellenlängen sendet. Das zugrundeliegende Modell für die weiteren Betrachtungen in diesem Abschnitt sieht wie folgt aus.

optisches Netzwerk

Ein *optisches Netzwerk* (G, W) ist ein Graph $G = (V, E)$ zusammen mit einer Menge von Wellenlängen $W := \{\lambda_1, \ldots, \lambda_\chi\}$. Jede Kante in G entspricht einer optischen Faser, und auf jeder Faser steht jede Wellenlänge $\lambda_i \in W$ genau einmal zur Verfügung. Eine Anfrage (Call) $r_i = (s_i, t_i)$ in einem optischen Netzwerk spezifiert zwei Knoten $s_i \neq t_i \in V$, zwischen denen eine (optische)

Verbindung hergestellt werden soll. Eine optische Verbindung wird durch einen *Lichtweg* realisiert, wobei ein Lichtweg aus einem Pfad P in G und einer Wellenlänge $\lambda \in W$ besteht. Eine Menge von Lichtwegen in einem optischen Netzwerk muß die folgende Bedingung erfüllen: Zwei Lichtwege, die die gleiche Kante benutzen, müssen verschiedene Wellenlängen besitzen. Oder anders ausgedrückt: auf jeder Kante darf jede Wellenlänge nur einmal vergeben werden. Diese essentielle Forderung heißt Wellenlängenkonfliktbedingung.

Lichtweg

Wellenlängenkonflikt-bedingung

Call Admission

Das Ziel bei der *Call Admission* in optischen Netzwerken ist die Durchsatzmaximierung. D.h., man muß für jede Anfrage entscheiden, ob sie angenommen oder abgelehnt wird und möchte die Zahl der angenommenen Anfragen maximieren. Wird eine Anfrage angenommen, so muß ein passender Lichtweg eingerichtet werden(«Routing Problem»). Dabei darf die Wellenlängenkonfliktbedingung nicht verletzt werden.

Besteht die Menge W nur aus einer einzigen Wellenlänge, so reduziert sich das Routing Problem darauf, kantendisjunkte Wege in dem gegebenen Graphen zu finden, und man spricht vom Problem der *Disjoint Path Allcoation* (DPA).

Ein optisches Netzwerk $(G < W)$ mit $G = (V, E)$ und $W = \{\lambda_1, \ldots, \lambda_\chi\}$ kann man auch als χ Kopien des Graphen G betrachten, jede mit einer anderen Wellenlänge. Das Routing Problem setzt sich dann zusammen aus dem Problem, eine Wellenlänge auszuwählen und dem Problem, in der entsprechenden Kopie von G einen Weg zu finden, der kantendisjunkt zu allen Wegen ist, die bereits in dieser Kopie geroutet sind.

Der folgende Algorithmus *First-Fit-Coloring* reduziert das Problem der Online Call Admission auf den Spezialfall der Disjoint Path Allocation.

Algorithmus FFC Setze $i = 1$.

1. Übergebe die aktuelle Anfrage an einen DPA-Algorithmus ALG, der in der Kopie von G mit der Wellenlänge λ_i arbeitet. Lehnt dieser Algorithmus die Anfrage ab, erhöhe i um 1 und verfahre analog. Nimmt er sie an, route sie auf dem vorgeschlagenen Weg in Wellenlänge λ_i.
2. Lehne die Anfrage nur ab, wenn sie von keinem der χ DPA-Algorithmen angenommen wurde.

FFC ist eine Art Greedy-Algorithmus, da er die DPA-Algorithmen, die auf den χ Kopien von G arbeiten, der Reihe nach durchgeht und sofort damit aufhört, sobald die Anfrage von einem angenommen wird. Die zusätzliche Dimension, die das Problem dadurch erhält, daß mehrere Wellenlängen zur Verfügung stehen, wird damit umgangen. Bemerkenswert ist hierbei vor allem, daß dies nur zu einer geringen Verschlechterung der Kompetitivität führt, wie der nächste Satz zeigt.

Satz 6.11 *Ist der von* FFC *verwendete* DPA*-Algorithmus* ALG *c-kompetitiv, so ist* FFC *$(c+1)$-kompetitiv.*

Beweis: Sei ALG_i der DPA-Algorithmus ALG, der auf der i-ten Kopie von G arbeitet (d.h., der die Wellenlänge λ_i in G verwaltet), $i = 1, \dots, \chi$. Sei σ eine beliebige Anfragesequenz. Offensichtlich entspricht der von FFC erzielte Profit auf σ der Summe der Profite, die die einzelnen DPA-Algorithmen ALG_i erzielen. Allerdings erhalten die verschiedenen ALG_i auch verschiedene Anfragesequenzen: ALG_1 sieht ganz σ, ALG_2 nur noch den Teil, den ALG_1 abgelehnt hat, ALG_3 bearbeitet nur noch den Teil von σ, den sowohl ALG_1 als auch ALG_2 abgelehnt haben usw. Bezeichnen wir mit F_i die Menge der Anfragen in σ, die FFC annimmt und in Wellenlänge λ_i routet, dann gilt $|F_i| = \mathsf{ALG}_i\left(\sigma \setminus \bigcup_{l<i} F_l\right)$ für $i = 1, \dots, \chi$. Somit ist

$$\mathsf{FFC}(\sigma) = \sum_{i=1}^{\chi} |F_i| = \sum_{i=1}^{\chi} \mathsf{ALG}_i \left(\sigma \setminus \bigcup_{l<i} F_l \right). \tag{6.7}$$

Umgekehrt wollen wir nun den Profit abschätzen, den der optimale Offline-Algorithmus erzielen kann. Dazu sei mit $\mathsf{OPT}[i, \pi]$ die Menge der Anfragen in einer Sequenz π bezeichnet, die der optimale Offline-Algorithmus annimmt und in Wellenlänge λ_i routet. Da der optimale Offliner in einer einzelnen Wellenlänge nicht mehr Anfragen routen kann als der optimale Offline-Algorithmus OPT_1, der nur eine Wellenlänge zu seiner Verfügung hat, gilt $|\mathsf{OPT}[i, \pi]| \leq \mathsf{OPT}_1(\pi)$ für $i = 1, \dots, \chi$. Damit folgt

$$\mathsf{OPT}(\pi) = \sum_{i=1}^{\chi} |\mathsf{OPT}[i, \pi]| \leq \sum_{i=1}^{\chi} \mathsf{OPT}_1(\pi). \tag{6.8}$$

Der Algorithmus OPT_1 ist nach Definition gerade der optimale Offline-Algorithmus für das DPA-Problem und erzielt nach Voraussetzung auf jeder beliebigen Anfragesequenz nicht mehr als c-mal soviel Profit wie ALG. Insgesamt erhalten wir folgende Abschätzung:

$$\begin{aligned}
\mathsf{OPT}(\sigma) &\leq \left| \bigcup_{l=1}^{\chi-1} F_l \right| + \mathsf{OPT}\left(\sigma \setminus \bigcup_{l=1}^{\chi-1} F_l \right) \\
&\overset{(6.8)}{\leq} \sum_{l=1}^{\chi} |F_l| + \sum_{i=1}^{\chi} \mathsf{OPT}_1 \left(\sigma \setminus \bigcup_{l=1}^{\chi-1} F_l \right) \\
&\leq \mathsf{FFC}(\sigma) + \sum_{i=1}^{\chi} \mathsf{OPT}_1 \left(\sigma \setminus \bigcup_{l<i} F_l \right) \\
&\overset{\text{Vor.}}{\leq} \mathsf{FFC}(\sigma) + \sum_{i=1}^{\chi} c \cdot \mathsf{ALG}_i \left(\sigma \setminus \bigcup_{l<i} F_l \right) \\
&\overset{(6.7)}{=} (c+1) \cdot \mathsf{FFC}(\sigma).
\end{aligned}$$

Bei der ersten Ungleichung vergrößern wir den optimalen Profit nur, indem wir OPT erlauben, die Anfragen in $\bigcup_{l=1}^{\chi-1} F_l$ anzunehmen, ohne dafür seine Ressourcen zu verbrauchen. Die dritte Ungleichung resultiert aus der Tatsache, daß der optimale Profit nicht kleiner werden kann, wenn man einer Sequenz Anfragen hinzufügt.
Damit ist Satz 6.11 bewiesen. □

Wir haben gesehen, daß man das Problem der Online Call Admission auf das der Disjoint Path Allocation reduzieren kann, ohne große Einbußen beim Kompetitivitätsratio in Kauf nehmen zu müssen. Allerdings kann man sich dies natürlich nur zunutze machen, wenn man «gute» DPA-Algorithmen kennt. Leider sind kompetitive Algorithmen für das Problem der Disjoint Path Allocation nur für spezielle Graphen bekannt. Der randomisierte Algorithmus *Classify-and-Randomly-Select* (CRS) ist auf dem Pfad mit N Knoten $(\log_2 N)$-kompetitiv, wie in Übungsaufgabe 6.2 gezeigt wird. Weiterhin gibt es kompetitive Algorithmen für Bäume und Gitter, ansonsten sind jedoch keine kompetitiven Algorithmen für Disjoint Path Allocation bekannt.

In Übungsaufgabe 6.2 wird ebenfalls gezeigt, daß kein deterministischer Algorithmus auf dem Pfad mit N Knoten (und damit in Netzwerken im allgemeinen) besser als $(N-1)$-kompetitiv sein kann. Wir wollen nun untersuchen, welche untere Schranke hier für randomisierte Algorithmen gilt. Dazu wenden wir Yao's Prinzip für Maximierungsprobleme an.

Satz 6.12 *Auf einem Pfad mit N Knoten ist kein randomisierter Algorithmus für das Problem der Disjoint Path Allocation besser als $\frac{\lceil \log_2 N \rceil}{2}$-kompetitiv gegen den blinden Gegner.*

Beweis: Sei $2^k < N \leq 2^{k+1}$. Wir werden eine Wahrscheinlichkeitsverteilung auf der Menge der Anfragesequenzen konstruieren, bezüglich derer für jeden beliebigen deterministischen Algorithmus ALG folgendes gilt:

$$\mathbb{E}\left[\mathsf{OPT}(\sigma)\right] \geq \frac{\lceil \log_2 N \rceil}{2} \quad \text{und} \quad \mathbb{E}\left[\mathsf{ALG}(\sigma)\right] \leq 1.$$

Mit Yao's Prinzip für Maximierungsprobleme folgt aus der Ungleichung $\frac{\mathbb{E}[\mathsf{OPT}(\sigma)]}{\mathbb{E}[\mathsf{ALG}(\sigma)]} \geq \frac{\lceil \log_2 N \rceil}{2}$, daß $\frac{\lceil \log_2 N \rceil}{2}$ eine untere Schranke für den Kompetitivitätsratio jedes randomisierten Algorithmus gegen den blinden Gegner ist.

Sei $D := 2^k$, und seien die Knoten von G von 0 bis $N-1$ numeriert. Wir betrachten zunächst die folgenden Mengen von Anfragen:

$$\begin{aligned} C_1 &:= \{(0, D)\}, \\ C_2 &:= \left\{\left(0, \frac{D}{2}\right), \left(\frac{D}{2}, D\right)\right\}, \\ &\vdots \\ C_{1+\log_2 D} &:= \{(1,2), (2,3), \ldots, (D-1, D)\}. \end{aligned}$$

D.h. die Menge $C_i = \left\{\left(0, \frac{D}{2^{i-1}}\right), \left(\frac{D}{2^{i-1}}, 2\frac{D}{2^{i-1}}\right), \ldots, \left((2^{i-1}-1)\frac{D}{2^{i-1}}, D\right)\right\}$ enthält 2^{i-1} Anfragen. Nach Konstruktion steht jede Anfrage in C_i im Konflikt mit genau zwei Anfragen in C_{i+1}, mit vieren in C_{i+2} usw..

Unsere randomisierte Eingabesequenz konstruieren wir nun folgendermaßen. Wir wählen $l \in \{1, 2, \ldots, 1 + \log_2 D\}$ mit Wahrscheinlichkeit

$$p_l := \frac{2^{-l}}{1 - \frac{1}{2D}}.$$

Dann geben wir nacheinander die Anfragen der Mengen $C_1, C_2, \ldots, C_l$ und beenden anschließend die Sequenz.

Was ist der erwartete optimale Profit auf dieser randomisierten Eingabesequenz? Der meiste Profit läßt sich erzielen, wenn alle Anfragen der Mengen C_i mit $i < l$ abgelehnt und alle 2^{l-1} Anfragen in C_l angenommen werden. Der erwartet optimale Profit beträgt also

$$\begin{aligned} \mathbb{E}\left[\mathsf{OPT}(\sigma)\right] &= \sum_{l=1}^{1+\log_2 D} \frac{2^{-l}}{1-\frac{1}{2D}} \cdot 2^{l-1} \\ &= \sum_{l=1}^{1+\log_2 D} \frac{1}{2} \cdot \underbrace{\frac{2D}{(2D-1)}}_{<1} > \frac{1+\log_2 D}{2} = \frac{\lceil \log_2 N \rceil}{2}. \end{aligned}$$

Wir müssen nun noch den erwarteten Profit eines beliebigen deterministischen Online-Algorithmus ALG abschätzen. Dazu benötigen wir zwei Lemmata.

Lemma 6.13 *Pr* $[\mathsf{ALG}$ *sieht* $C_i \mid \mathsf{ALG}$ *sieht* $C_{i-1}] \leq \frac{1}{2}$.

Das heißt also, daß die Sequenz nach der letzten Anfrage in C_i mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ aufhört.

Beweis: Es ist zu zeigen, daß Pr $[l \geq i \mid l \geq i-1] \leq \frac{1}{2}$ ist. Die bedingte Wahrscheinlichkeit errechnet sich als

$$\Pr\left[l \geq i \mid l \geq i-1\right] = \frac{\Pr\left[l \geq i\right]}{\Pr\left[l \geq i-1\right]} = \frac{\Pr\left[l \geq i\right]}{\Pr\left[l \geq i\right] + \Pr\left[l = i-1\right]},$$

so daß die Behauptung äquivalent ist zu

$$\Pr\left[l \geq i\right] \;\leq\; \Pr\left[l = i-1\right].$$

Berücksichtigt man, daß Pr $[l \geq i] = \sum_{k=i}^{1+\log_2 D} \Pr\left[l = k\right]$ ist und setzt die Wahrscheinlichkeiten ein, mit dem Hauptnenner multipliziert, so bleibt zu zeigen, daß

$$\sum_{k=i}^{1+\log_2 D} 2^{-k} \leq 2^{-(i-1)}$$

ist. Dies gilt, da $\sum_{k=i}^{1+\log_2 D} 2^{-k} \leq 2^{-i} \sum_{k=0}^{\infty} 2^{-k} = 2 \cdot 2^{-i}$ ist.

Damit ist Lemma 6.13 bewiesen. □

Als zweites betrachten wir, was der Online-Algorithmus (im Erwartungswert) gewinnt, wenn er eine Anfrage ablehnt.

Lemma 6.14 *Sei für eine Anfrage* $r \in C_i$, *die nicht mit zuvor angenommenen Anfragen konfliktiert,* $\mathbb{E}\left[r\right]$ *der maximal zu erwartende Profit, den* ALG *durch Annahme oder Ablehnen von* r *erzielen kann. Dann gilt* $\mathbb{E}\left[r\right] \leq 1$ *für alle* $r \in C_i$ *und* $i = 1, \ldots, 1 + \log_2 D$.

Beweis: Wir beweisen Lemma 6.14 durch Rückwärtsinduktion nach i. Für den Induktionsanfang $i = 1 + \log_2 D$ gilt die Behauptung offensichtlich, weil die Annahme von r genau Profit 1 bringt, während das Ablehnen von r keinen weiteren Profit ermöglicht, da keine mit r in Konflikt stehenden Anfragen mehr kommen können.

Es gelte die Behauptung für $i = 1 + \log_2 D, \ldots, k$. Wir haben zu zeigen, daß sie auch für $i = k - 1$ gilt.

Sei $r \in C_i$. Falls ALG r annimmt, so resultiert daraus unmittelbar Profit 1. Lehnt ALG die Anfrage ab, so resultiert daraus nach Lemma 6.13 mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ überhaupt kein Profit für ALG, da die Sequenz nach C_i aufhört. Mit Wahrscheinlichkeit höchstens $\frac{1}{2}$ hört sie jedoch noch nicht auf, und ALG kann spätere Anfragen annehmen, die sich mit r überschneiden. Es gilt somit

$$\mathbb{E}\left[\text{ALG's durch Ablehnen von } r \in C_i \text{ resultierender Profit}\right] \leq \frac{1}{2}(\mathbb{E}\left[r_1\right]+\mathbb{E}\left[r_2\right]),$$

wobei r_1 und r_2 genau die beiden mit r konfliktierenden Anfragen aus C_{i+1} seien. Nach Induktionsvoraussetzung gilt für diese $\mathbb{E}\left[r_j\right] \leq 1, j = 1, 2$, und wir erhalten $\mathbb{E}\left[r\right] \leq \frac{1}{2}(1 + 1) = 1$, wie behauptet. □

Wendet man Lemma 6.14 auf den ersten Call $\tilde{r} = (0, D) \in C_1$ an, so folgt die Behauptung von Satz 6.12, da der erwartete Profit von ALG auf der randomisierten Sequenz gleich $\mathbb{E}\left[\tilde{r}\right]$ ist. □

Übungsaufgaben

Übung 6.1 (Scheduling auf zwei verwandten Maschinen)
Beim Scheduling auf zwei verwandten Maschinen hat man eine schnelle erste und eine langsame zweite Maschine gegeben. Die Ausführung eines Jobs der Größe r benötigt r Zeiteinheiten auf der schnellen und αr Zeiteinheiten auf der langsamen Maschine ($\alpha \geq 1$).

Wir betrachten den folgenden Online-Algorithmus LIST, welcher Graham's Algorithmus (siehe Seite 44) verallgemeinert: Sei $r_j = (v_j, \alpha v_j)$ der aktuelle Job und F (L) die aktuelle Last der schnellen (langsamen) Maschine. Dann weist LIST den Job r_j der schnellen Maschine zu, wenn $F + v_j \leq L + \alpha v_j$, d.h. er weist r_j derjenigen Maschine zu, auf der er zuerst beendet wird.

(a) Warum verallgemeinert LIST den Algorithmus GRAHAM von Seite 44?

(b) Zeigen Sie, daß für $\alpha \leq (1 + \sqrt{5})/2$ der Algorithmus LIST $\min\left\{\frac{2\alpha+1}{\alpha+1}, 1 + \frac{1}{\alpha}\right\}$-kompetitiv ist (Hinweis: Benutzen Sie Ideen aus dem Beweis von Satz 6.1).

(c) Beweisen Sie das Kompetitivitätsresultat aus (b) auch für den Fall, daß $\alpha > (1 + \sqrt{5})/2$ gilt.

(d) Beweisen Sie, daß für $\alpha \geq (1 + \sqrt{5})/2$ kein deterministischer Algorithmus besser als $1 + 1/\alpha$-kompetitiv sein kann.

Übung 6.2 (Disjoint Path Allocation (DPA))
Ein Spezialfall des Durchsatzmaximierungsproblems ergibt sich, wenn jede Kante nur Kapazität eins besitzt. Jede Anfrage r_j hat dann in diesem Fall auch nur eine Bandbreitenanforderung von eins. Das Optimierungsziel ist es also, eine maximale Anzahl kantendisjunkter Wege mit den entsprechenden Endpunkten aus den Anfragen im Graphen $G = (V, E)$ zu konstruieren.

Bei der kompetitiven Analyse läßt man in diesem Fall keine additiven Konstanten zu, da sonst mit $\alpha := m$ jeder Algorithmus 1-kompetitiv wäre. Wir betrachten das Problem für den »einfachen Fall«, daß der Graph $G = (V, E)$ ein Pfad aus N Ecken ist. G hat also $m := N - 1$ Kanten und Durchmesser $D = N - 1$.

(a) Zeigen Sie, daß jeder deterministische Algorithmus auf G höchstens D-kompetitiv ist. Geben Sie einen D-kompetitiven Algorithmus an.

(b) Warum widerspricht das Ergebnis aus (a) nicht Satz 6.10?

(c) Sei $N = 2^p$ eine Zweierpotenz. Wir partitionieren die Kanten von G wie folgt in »Separationsmengen«: Sei e_1 diejenige Kante, durch deren Entfernen der Graph G in zwei Pfade G' und G'' mit je 2^{p-1} Ecken zerfällt. Wir setzen $E_1 = \{e_1\}$. Wir teilen jeden der beiden Pfade G' und G'' wieder jeweils in der Mitte. Seien $E_2 = \{e_{21}, e_{22}\}$ die entsprechenden Kanten. Rekursive Fortsetzung liefert eine Partition von E in Mengen $E_1, E_2, \dots, E_p$.

Jede Anfrage r_j kann nun mit Hilfe der Partition klassifiziert werden. Wir nennen eine Anfrage $r_j = (s_j, t_j)$ eine *Level-i-Anfrage*, wenn $i = \min\{s : E_s \cap E' \neq \emptyset\}$, wobei E' die Kanten auf dem Weg zwischen s_j und t_j sind.

Algorithmus CRS (Classify-and-Randomly-Select) Wähle $\ell \in \{1, 2, p = \log_2 N\}$ zufällig und gleichverteilt. Akzeptiere nur Level-ℓ-Anfragen (und von diesen nur diejenigen, die nicht mit bereits akzeptierten Anfragen kollidieren).

Beweisen Sie, daß CRS $\log_2 N$-kompetitiv ist. (Hinweis: Seien o_i und c_i die Anzahl der von OPT bzw. CRS akzeptierten Level i-Anfragen. Zeigen sie zunächst, daß $c_\ell \geq o_\ell$ gilt.)

7

Metrische Tasksysteme

In diesem Kapitel untersuchen wir ein sehr allgemeines Online-Problem, in dessen Form sich viele andere Online-Probleme formulieren lassen. Dabei lernen wir einen wichtigen Algorithmus, den *Work Function Algorithmus* kennen, der uns noch im Kapitel 8 zum *k-Server Problem* beschäftigen wird.

Work Function Algorithmus

k-Server Problem

Bei einem metrischen Tasksystem ist eine Maschine gegeben, die online Aufträge (*tasks*) verarbeiten muß. Die Maschine kann sich in einem von endlich vielen Zuständen befinden. Die Kosten für die Bearbeitung eines Tasks hängen vom Zustand ab.

task

Formal ist ein *metrisches Tasksystem* ein Paar (M, T), wobei M ein endlicher *metrischer Raum* und T eine endliche Menge von Funktionen $\tau\colon M \to \mathbb{R} \cup \{+\infty\}$, den Tasks, ist. Bearbeitung eines Tasks τ im Zustand s verursacht Kosten $\tau(s)$. Durch Wechsel des Zustands von s nach s' entstehen Kosten $d(s, s')$.

metrischer Raum

Sei X eine Menge und $d\colon X \times X \to \mathbb{R}_{\geq 0}$ eine Abbildung. Die Abbildung d ist eine *Metrik*, falls für alle x, y, z aus X folgendes gilt:

$$d(x,y) = 0 \iff x = y;$$
$$d(x,y) = d(y,x);$$
$$d(x,z) \leq d(x,y) + d(y,z).$$

Das Paar (X, d) heißt dann *metrischer Raum.*

Ein Algorithmus startet in einem festgelegten Startzustand s_0 und erhält nun eine endliche Folge $\sigma = r_1, \ldots, r_n$ von Tasks $r_i \in T$, die er sequentiell bearbeiten muß. Bei Ankunft des Tasks r_i hat der Algorithmus die Möglichkeit, vor Bearbeiten von r_i den Zustand noch zu wechseln. Ein Online-Algorithmus erhält den Task r_{i+1} erst, nachdem er r_i bearbeitet hat. Ziel ist es, die Gesamtkosten, bestehend aus Bearbeitungs- und Zustandsübergangskosten, zu minimieren.

Beispiel 7.1 Wir können das Eismaschinenoptimierungsproblem aus Abschnitt 1.1 als Metrisches Tasksystem formulieren: Luigis Eismaschine besitzt zwei Zustände V und S, d.h. $M = \{V, S\}$, wobei $d(V, S) = 1$. Die Menge der zulässigen Tasks ist $T = \{v, s\}$ mit

$$\begin{aligned} v &= (1, 4) \\ &\doteq (\text{Kosten für Vanilleproduktion im Zustand } V, \text{Kosten im Zustand } S) \\ s &= (2, 2). \end{aligned}$$

$\triangleleft$

Sei $s_0 \in M$ ein Anfangszustand und $\sigma = r_1, \ldots, r_n$ eine Folge von Tasks aus T. Eine Folge $\bar{s} = s_0, s_1, \ldots, s_n$ mit $s_i \in M$ nennen wir einen *Service*

Schedule für s_0 und σ. Wir definieren

$$C(s_0, \sigma, \bar{s}) := \sum_{i=1}^{n} (d(s_{i-1}, s_i) + r_i(s_i))$$
$$\mathsf{OPT}(s_0, \sigma) := \min_{\bar{s}} C(s_0, \sigma, \bar{s}).$$

7.1 Arbeitsfunktionen

Arbeitsfunktion

Sei (M, T) ein Metrisches Tasksystem, s_0 ein Anfangszustand und $\sigma = r_1, \ldots, r_n$ eine Folge von Tasks. Wir definieren die *Arbeitsfunktion (Work Function)* $w_\sigma \colon M \to \mathbb{R}$, die mit s_0 und σ assoziiert ist, wie folgt: $w(s)$ sind die minimalen Kosten, um die Folge σ ausgehend vom Zustand s_0 zu bearbeiten und dabei im Zustand s zu enden.

Definition 7.2 (Arbeitsfunktion (Work Function))

Work Function

*Die **Arbeitsfunktion** (Work Function) w_σ zum Startzustand s_0 und zur Folge $\sigma = r_1, \ldots, r_n$ ist defi niert durch:*

$$w_\sigma(s) := \min_{\bar{s}} \left(C(s_0, \sigma, \bar{s}) + d(s_n, s) \right).$$

Beobachtung 7.3 *Die Arbeitsfunktionen besitzen folgende Eigenschaften:*

(i) $w_\sigma(s) \leq w_\sigma(s') + d(s, s')$ *für alle* $s, s' \in M$.

(ii) $w_{\sigma r}(s) \geq w_\sigma(s)$ *für alle* $s \in M$ *und alle* $r \in T$.

(iii) $w_{\sigma r}(s) = \min_{x \in M} (w_\sigma(x) + r(x) + d(x, s))$.

(iv) $\mathsf{OPT}(s_0, \sigma) = \min_{s \in M} w_\sigma(s)$.

Lemma 7.4 *Wenn* $w_{\sigma r}(x) = w_\sigma(y) + r(y) + d(y, x)$ *gilt, dann folgt* $w_{\sigma r}(y) = w_\sigma(y) + r(y)$.

Beweis: Wir müssen zeigen, daß

$$w_\sigma(y) + r(y) \leq w_\sigma(z) + r(z) + d(z, y)$$

für alle $z \in M$ gilt (vgl. Beobachtung 7.3 (iv)). Wir haben

$$w_\sigma(y) + r(y) + d(y, x) \leq w_\sigma(z) + r(z) + d(z, x).$$

Also ist

$$w_\sigma(y) + r(y) \leq w_\sigma(z) + r(z) + d(z, x) - d(y, x) \leq w_\sigma(z) + r(z) + d(z, y).$$

Dies war zu zeigen. □

7.2 Eine untere Schranke

7.3 Der Algorithmus WFA

Sei $\sigma = r_1, \ldots, r_n$ eine beliebige Anfragesequenz. Wir bezeichnen mit $\sigma_i = r_1, \ldots, r_i$ den Präfix bestehend aus den ersten i Tasks.

Algorithmus WFA Sei $\sigma_i = r_1, \ldots, r_i$ die bisherige Anfragefolge und $r = r_{i+1}$ die aktuelle Anfrage. Sei ferner $s = s_i$ der aktuelle Zustand von WFA. Dann bearbeitet WFA die Anfrage r in einem Zustand $s' = s_{i+1}$ mit

$$d(s, s') + w_{\sigma_i r}(s') = \min_{x \in M} (d(s, x) + w_{\sigma_i r}(x)) \qquad (7.1)$$

und

$$w_{\sigma_i r}(s') = w_{\sigma_i}(s') + r(s'). \qquad (7.2)$$

Lemma 7.5 *In jedem Schritt kann WFA einen Zustand $s' = s_{i+1}$ auswählen, der (7.1) und (7.2) erfüllt.*

Beweis: Sei $s' \in M$ ein Zustand, der (7.1) erfüllt. Nach Beobachtung 7.3 (iii) gibt es ein $z \in M$, für das

$$w_{\sigma_i r}(s') = w_{\sigma_i}(z) + r(z) + d(z, s') \qquad (7.3)$$

gilt. Nach Lemma 7.4 gilt $w_{\sigma_i r}(z) = w_{\sigma_i}(z) + r(z)$, somit erfüllt z Bedingung (7.2). Wir zeigen, daß z auch Bedingung (7.1) genügt. Aus (7.3) folgt

$$w_{\sigma_i r}(s') + d(z, s) = w_{\sigma_i}(z) + r(z) + d(z, s') + d(z, s),$$

was äquivalent ist zu

$$w_{\sigma_i}(z) + r(z) + d(z, s) = w_{\sigma_i r}(s') + d(z, s) - d(z, s').$$

Mit Hilfe der Dreiecksungleichung und nach Wahl von s' als Zustand, der (7.1) erfüllt, sowie mit Lemma 7.4 folgt

$$\begin{aligned} w_{\sigma_i}(z) + r(z) + d(z, s) &\leq w_{\sigma_i r}(s') + d(s, s') \\ &= \min_{x \in M} (d(s, x) + w_{\sigma_i r}(x)) \\ &\leq d(s, z) + w_{\sigma_i r}(z) \\ &\overset{(7.4)}{=} d(s, z) + w_{\sigma_i}(z) + r(z). \end{aligned}$$

Somit gilt überall Gleichheit, insbesondere also

$$\min_{x \in M} (d(s, x) + w_{\sigma_i r}(x)) = d(s, z) + w_{\sigma_i r}(z),$$

d.h. z erfüllt auch Bedingung (7.1), was zu zeigen war.

□

Lemma 7.6 *Sei s_i der aktuelle Zustand von* WFA *zum Zeitpunkt, zu dem $r = r_{i+1}$ bekannt wird und s_{i+1} der Folgezustand, in dem r_{i+1} von* WFA *bearbeitet wird. Dann gilt:*

$$\begin{aligned} w_{\sigma_{i+1}}(s_i) &= w_{\sigma_{i+1}}(s_{i+1}) + d(s_{i+1}, s_i) \\ w_{\sigma_{i+1}}(s_{i+1}) &= w_{\sigma_i}(s_{i+1}) + r_{i+1}(s_{i+1}). \end{aligned}$$

Beweis: Die zweite Gleichung ist nach Konstruktion von WFA erfüllt (siehe Bedingung (7.2)). Zum Beweis der ersten Gleichung müssen wir zeigen, daß für jedes $x \in M$ gilt:

$$w_{\sigma_{i+1}}(s_{i+1}) + d(s_{i+1}, s_i) \leq w_{\sigma_{i+1}}(x) + d(x, s_i).$$

Diese Bedingung ist aber ebenfalls nach Konstruktion von WFA erfüllt (siehe Bedingung (7.1)). □

Satz 7.7 *Der Algorithmus* WFA *ist $(2N - 1)$-kompetitiv für Tasksysteme mit N Zuständen.*

Beweis: Sei $\sigma = r_1, \ldots, r_n$ eine beliebige Folge von Tasks und $s_0, s_1, \ldots, s_n$ der Service Schedule von WFA für σ: der Task r_i wird im Zustand s_i bearbeitet. Sei $D := \max_{s,s' \in M} d(s, s')$. Es gilt:

$$\begin{aligned} \mathsf{WFA}(\sigma) &= \sum_{i=1}^{n} (d(s_{i-1}, s_i) + r_i(s_i)) \\ &= \sum_{i=0}^{n-1} (d(s_i, s_{i+1}) + r_{i+1}(s_{i+1})) \\ &= \sum_{i=0}^{n-1} (w_{\sigma_{i+1}}(s_i) - w_{\sigma_i}(s_{i+1})) \qquad \text{(Nach Lemma 7.6)} \\ &= \sum_{i=0}^{n-1} (w_{\sigma_{i+1}}(s_i) - w_{\sigma_i}(s_i)) \\ &\quad + \sum_{i=0}^{n-1} (w_{\sigma_{i+1}}(s_{i+1}) - w_{\sigma_i}(s_{i+1})) \\ &\quad + w_{\sigma_0}(s_0) - w_{\sigma_n}(s_n) \\ &\leq 2 \sum_{i=0}^{n-1} \max_{x \in M} \big(w_{\sigma_{i+1}}(x) - w_{\sigma_i}(x)\big) - w_{\sigma_n}(s_n). \end{aligned} \tag{7.4}$$

Dabei haben wir ausgenutzt, daß $w_{\sigma_0}(s_0) = w_{\emptyset}(s_0) = d(s_0, s_0) = 0$ gilt. Wir zeigen durch ein Potentialfunktionsargument, daß

$$\sum_{i=0}^{n-1} \max_{x \in M} \big(w_{\sigma_{i+1}}(x) - w_{\sigma_i}(x)\big) \leq N \cdot \mathsf{OPT}(\sigma) + N \cdot D \tag{7.5}$$

gilt. Dieses Ergebnis, zusammen mit

$$\mathsf{OPT}(\sigma) = w_\sigma(s^*) := \min_{x \in M} w_\sigma(x) \leq w_\sigma(s_n)$$

in (7.4) eingesetzt, zeigt dann

$$\mathsf{WFA}(\sigma) \leq (2N-1)\mathsf{OPT}(\sigma) + N \cdot D,$$

also die Behauptung des Satzes.

Sei die Potentialfunktion $\Phi_i = \Phi(w_{\sigma_i}) = \sum_{x \in M} w_{\sigma_i}(x)$. Dann ist

$$\max_{x \in M} \big(w_{\sigma_{i+1}}(x) - w_{\sigma_i}(x)\big) \leq \sum_{x \in M} \big(w_{\sigma_{i+1}}(x) - w_{\sigma_i}(x)\big) = \Phi_{i+1} - \Phi_i. \quad (7.6)$$

Summation von (7.6) über $i = 0, \ldots, n-1$ liefert:

$$\begin{aligned}
\sum_{i=0}^{n-1} \max_{x \in M} \big(w_{\sigma_{i+1}}(x) - w_{\sigma_i}(x)\big) &\leq \Phi(w_{\sigma_n}) - \Phi(w_\emptyset) \\
&= \sum_{x \in M} w_{\sigma_n}(x) - \sum_{x \in M} \underbrace{w_{\sigma_0}(x)}_{d(s_0,x)} \\
&\leq \sum_{x \in M} w_{\sigma_n}(s^*) + \sum_{x \in M} (d(x,s^*) - d(s_0,x)) \\
&\leq N \cdot \mathsf{OPT}(\sigma) + \sum_{x \in M} d(s_0, s^*) \\
&\leq N \cdot \mathsf{OPT}(\sigma) + N \cdot D.
\end{aligned}$$

Dies impliziert (7.5) und beendet den Beweis. □

8

Das k-Server Problem

Referenzwerke: [10, Kapitel 10], [23, Kapitel 13]

Das k-Server Problem ist eine natürliche Verallgemeinerung des Paging Problems. Sei (X, d) ein *metrischer Raum.* Ein Algorithmus bewegt k mobile *Server*, deren Standorte immer Punkte des metrischen Raums X sind. Der Algorithmus erhält eine Anfragefolge $\sigma = r_1, \dots, r_n$, wobei r_i ein Punkt aus X ist. Wir sagen, daß eine Anfrage r_i beantwortet wird, wenn ein Server auf r_i plaziert ist. Der Algorithmus muß nun alle Anfragen beantworten, indem er die Server im Raum X bewegt. Seine Kosten $\mathsf{ALG}(\sigma)$ sind als die Gesamtstrecke definiert, welche seine Server bei der Anfragefolge $\sigma = r_1, \dots, r_n$ zurücklegen. Das k-Server Problem hat Anwendungen bei der Einsatzplanung von Versorgungsfahrzeugen (etwa Pannenhilfe).

Sei X eine Menge und $d\colon X \times X \to \mathbb{R}_{\geq 0}$ eine Abbildung. Die Abbildung d ist eine *Metrik*, falls für alle x, y, z aus X folgendes gilt:

$$d(x, y) = 0 \iff x = y;$$
$$d(x, y) = d(y, x);$$
$$d(x, z) \leq d(x, y) + d(y, z).$$

Das Paar (X, d) heißt dann *metrischer Raum.*

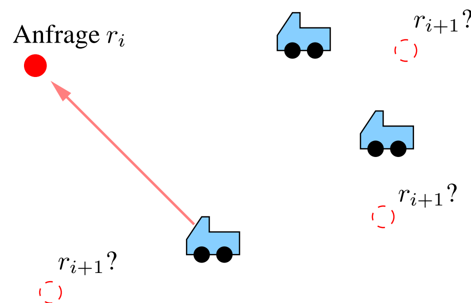
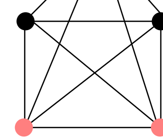


Abbildung 8.1
Bei Bekanntwerden der Anfrage r_i muß einer der Server zum Punkt r_i bewegt werden. Für den Online-Algorithmus wird erst dann die nächste Anfrage r_{i+1} bekannt.

Das Paging Problem ergibt sich als Spezialfall, wenn $d(x, y) = 1$ für alle $x, y \in X$ gilt. Die k Server repräsentieren die k Plätze im Cache, $N = |X|$ ist die Gesamtzahl der Speicherseiten. Daraus folgt bereits, daß für spezielle metrische Räume kein deterministischer Algorithmus mit Kompetitivität $k < c$ existieren kann. Wie wir in Abschnitt 8.2 zeigen werden, ist k sogar für *jeden* metrischen Raum mit wenigstens $k + 1$ Punkten eine untere Schranke.



Modellierung des Paging Problems mit $N = 5$ und $k = 2$ als k-Server Problem.

Die wohl berühmteste Vermutung in der Online-Optimierung ist die sogenannte *k-Server Vermutung* [26]:

Vermutung 8.1 (k-Server Vermutung) *Für jeden metrischen Raum gibt es einen deterministischen k-kompetitiven Algorithmus für das k-Server Problem.*

Die k-Server Vermutung ist bis heute noch nicht bewiesen. In der Tat ist es zunächst überhaupt nicht klar, daß es für jeden metrischen Raum einen Algorithmus gibt, dessen Kompetitivität nur von k und nicht etwa von der Anzahl

der Punkte im metrischen Raum abhängt. Das erste Resultat dieser Art erzielten Fiat et al. [13], die einen $\mathcal{O}((k!)^3)$-kompetitiven Algorithmus finden konnten. In Abschnitt 8.4 zeigen wir das Resultat von Koutsoupias und Papadimitriou [25] nach dem der »Work Function Algorithmus« (WFA) $(2k-1)$-kompetitiv ist.

8.1 Faule Algorithmen

Wir werden zunächst einige grundlegende Eigenschaften von Algorithmen für das k-Server Problem zeigen. Diese ermöglichen es uns, bei den folgenden Beweisen die die Klasse der zu betrachtenden Algorithmen einzuschränken.

Wir nennen einen Algorithmus *faul*, wenn er bei jeder Anfrage höchstens einen Server bewegt (und dies auch nur dann, wenn er keinen Server auf dem aktuellen Anfragepunkt hat). Wir zeigen nun, daß faule Algorithmen alle anderen Algorithmen dominieren:

Lemma 8.2 *Sei* ALG *ein Algorithmus für das k-Server Problem. Dann gibt es einen faulen Algorithmus* ALG' *mit* $\text{ALG}'(\sigma) \leq \text{ALG}(\sigma)$ *für jede Anfragefolge σ. Wenn* ALG *ein Online-Algorithmus ist, dann ist auch* ALG′ *ein Online-Algorithmus.*

Beweis: Der Beweis folgt durch einfache Induktion unter Zuhilfenahme der Dreiecksungleichung.

Der modifizierte Algorithmus ALG′ arbeitet wie folgt: Angenommen ALG arbeite faul bis zur $i$ten Anfrage und bewege dann unnötigerweise einen Server s von x nach y. Die dabei entstehenden Kosten sind $d(x, y)$. ALG′ arbeitet bis zur $i$ten Anfrage genau wie ALG, unterläßt dann jedoch die Bewegung des Servers s. Sei die nächste Anfrage, die der Server s von ALG bearbeitet bei z. Dann bewegt ALG′ den Server s direkt von x nach z. Da $d(x, z) \leq d(x, y) + d(y, z)$, sind die Kosten von ALG′ nicht größer als die von ALG. □

8.2 Eine untere Schranke für deterministische Algorithmen

Satz 8.3 *Sei $M = (X, d)$ ein beliebiger metrischer Raum mit $|X| \geq k + 1$ Punkten. Jeder determinitische c-kompetitive Algorithmus* ALG *für das k-Server Problem in M erfüllt dann $c \geq k$.*

Beweis: Sei ALG ein c-kompetitiver deterministischer Algorithmus für das k-Server Problem in $M = (X, d)$. Ohne Einschränkung nehmen wir an, daß ALG »faul« ist. Wir zeigen nun, daß es beliebig lange Anfragefolgen σ gibt, so daß $\text{ALG}(\sigma) \geq k\,\text{OPT}(\sigma)$ gilt.

Dazu konstruieren wir eine Anfragefolge σ und k Algorithmen $B_1, \ldots, B_k$, so daß $\text{ALG}(\sigma) = \sum_{j=1}^{k} B_k(\sigma)$ gilt. Folglich gibt es einen Algorithmus B_{j_0} mit $B_{j_0}(\sigma) \leq k\text{ALG}(\sigma)$.

Sei S die Menge der Serverpositionen, die ALG anfangs besetzt hat zusammen mit einen neuen Punkt. Wir können o. B. d. A. annehmen, daß $|S| = k$ ist. Wir werden nur Anfragen an Punkte aus S in der Folge verwenden. Die Anfrage r_1 erfolgt an dem Punkt aus S, an dem ALG keinen Server besitzt.

Sei $\sigma = r_1, \ldots, r_n$ die so konstruierte Folge und r_{n+1} der Punkt aus S, der nach σ von keinem Server von ALG besetzt ist. Dann gilt:

$$\text{ALG}(\sigma) = \sum_{i=1}^{n} d(r_{i+1}, r_i) = \sum_{i=1}^{n} d(r_i, r_{i+1}).$$

Wir konstruieren nun die Algorithmen $B_1, \ldots, B_k$. Seien die anfangs von ALG durch Server besetzten Punkte aus S die Punkte $x_1, \ldots, x_k$. Für $j = 1, \ldots, k$ ist dann Algorithmus B_i wie folgt definiert: Anfangs hat B_j Server auf allen Punkten aus S außer auf x_j. Wird im Verlaufe der Folge σ ein Punkt r_i gefragt, der nicht von B_j besetzt wird, dann bewegt B_j einen Server von r_{i-1} nach r_i.

Sei S_j die Menge der Punkte, auf denen B_j Server hat. Anfangs sind die Mengen S_j unterschiedlich. Wir zeigen durch Induktion, daß diese Mengen auch während der Bearbeitung von σ verschieden bleiben. Seien die Mengen bis zur Anfrage r_{i-1} verschieden. Wir betrachten die Anfrage r_i. Eine Menge S_j verändert sich nur, wenn r_i nicht in ihr enthalten ist (nur dann bewegt Algorithmus B_j einen Server). Wenn r_i also in zwei Mengen S_j und $S_{j'}$ liegt, dann bleiben diese auch weiterhin verschieden.

Da die Mengen S_j bis vor der Anfrage r_i unterschiedlich sind, hat genau ein Algorithmus B_j von $B_1, \ldots, B_k$ keinen Server auf r_i. Für diesen Algorithmus B_j ändert sich die Menge S_j, indem r_{i-1} durch r_i ersetzt wird. Da aber alle Algorithmen $B_1, \ldots, B_k$ die letzte Anfrage r_{i-1} bearbeitet haben und folglich für $j' \neq j$ nach der $i$ten Anfrage $r_{i-1} \in S_{j'}$ gilt, folgt, daß die Mengen weiterhin verschieden bleiben.

Wir haben bereits gesehen, daß bei der Anfrage r_i genau einem der Algorithmen $B_1, \ldots, B_k$ Kosten $d(r_{i-1}, r_i)$ enstehen. Also ist:

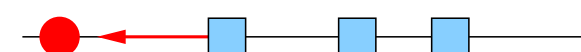
$$\sum_{j=1}^{k} B_j(\sigma) = \sum_{i=2}^{n} d(r_{i-1}, r_i) = \sum_{i=1}^{n-1} d(r_i, r_{i+1}).$$

Die Summe auf der rechten Seite der letzten Gleichung entspricht den Kosten von ALG auf σ bis auf den letzten Term, den man für lange Folgen vernachlässigen kann. □

8.3 Das k-Server Problem auf der Linie

Wir betrachten im folgenden den Spezialfall, daß der metrische Raum X die reelle Gerade $\mathbb{R}$ mit der Metrik $d(x, y) = |x - y|$ ist.

Algorithmus DC (Double Coverage) Wenn die aktuelle Anfrage r_i außerhalb der konvexen Hülle der Server liegt, dann bearbeite die Anfrage mit den dichtesten Server.



Ansonsten liegt die Anfrage zwischen zwei Servern. Bewege beide Server mit gleicher Geschwindigkeit auf r_i zu, bis der erste Server r_i erreicht. Wenn zwei Server auf dem gleichen Punkt sitzen, dann wird ein beliebiger von ihnen ausgewählt.

Der Algorithmus DC ist kein fauler Algorithmus. Durch die Techniken aus Lemma 8.2 kann man DC jedoch zu einem faulen Algorithmus modifizieren. Für die Analyse ist jedoch die oben angegebene Version von DC geeigneter. Zum Beweis der Kompetitivität von DC benutzen wir eine häufig angewandte Technik bei der kompetitiven Analyse: eine *Potentialfunktion*.

Potentialfunktion

Satz 8.4 DC *ist k-kompetitiv für das k-Server Problem in $\mathbb{R}$.*

Matching

Beweis: Wir bezeichnen mit M_i das minimale *Matching* zwischen den Servern von DC und OPT nach der $i$ten Anfrage. Sei ferner S_i die Summe aller Distanzen zwischen den Servern s_j von DC nach der $i$ten Anfrage: $S_i = \sum_{l<j} d(s_l, s_j)$. Dann benutzen wir folgendes Potential:

$$\Phi_i := k \cdot M_i + S_i.$$

Behauptung 8.5 *Wir stellen uns vor, daß bei einer Anfrage r_i zuerst* OPT *einen Server bewegt und danach* DC *einen (oder mehrere) Server bewegt. Für das Potential Φ gelten dann folgende Aussagen:*

(i) Es gilt: $\Phi_i \geq 0$ für alle i.

*(ii) Wenn der Gegner (*OPT*) einen Server zum Request r_i um δ bewegt, dann erhöht sich das Potential höchstens um $k\delta$.*

(iii) Wenn danach DC *seine Server um insgesamt δ bewegt, dann verringert sich das Potential um mindestens δ.*

Bevor wir die Behauptung 8.5 beweisen, zeigen wir zunächst, daß sie die im Satz behauptete Kompetitivität von DC impliziert.

amortisierte Kosten

Wir definieren die *amortisierten Kosten* a_i von DC für die Anfrage r_i wie folgt: Sei $\text{DC}(r_i)$ die Gesamtstrecke, die die Server von DC bei der Anfrage r_i zurücklegen. Dann ist

$$a_i := \text{DC}(r_i) + \Phi_i - \Phi_{i-1}. \tag{8.1}$$

Wir benutzen zunächst Behauptung 8.5 (ii) und (iii), um a_i abzuschätzen. Wir stellen uns wie in der Behauptung vor, daß bei der Anfrage r_i zuerst OPT und dann DC seine Server bewegt. Es folgt dann aus den Eigenschaften des Potentials Φ, daß

$$\Phi_i - \Phi_{i-1} \leq k \cdot \text{OPT}(r_i) - \text{DC}(r_i).$$

Also ist

$$a_i \leq k \cdot \text{OPT}(r_i). \tag{8.2}$$

Aus der Definition der amortisierten Kosten in (8.1) folgt

$$\text{DC}(\sigma) = \sum_{i=1}^{n} \text{DC}(r_i) = \sum_{i=1}^{n} (a_i + \Phi_{i-1} - \Phi_i) = \sum_{i=1}^{n} a_i + \Phi_0 - \Phi_n.$$

Da nach Behauptung 8.5 (i) $\Phi_n \geq 0$ ist, folgt, daß

$$\mathsf{DC}(\sigma) \leq \Phi_0 + \sum_{i=1}^{n} a_i \overset{(8.2)}{\leq} \Phi_0 + \sum_{i=1}^{n} k \cdot \mathsf{OPT}(r_i) = k \cdot \mathsf{OPT}(\sigma) + \Phi_0.$$

Mit $\alpha := \Phi_0$ ist dann die k-Kompetitivität von DC gezeigt.

Das folgende Lemma ist durch Austauschargumente einfach zu beweisen.

Lemma 8.6 *Sei $k \geq 2$ und $X = \{x_1, \dots, x_k\} \subset \mathbb{R}$, $Y = \{y_1, \dots, y_k\} \subset \mathbb{R}$. Sei $x \in X$.*

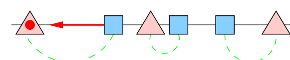
(i) Sei $y \in Y$ mit $d(x, y) = \min_{x' \in X} d(x', y)$. Gilt $x \leq y_i$ für $i = 1, \dots, k$ (oder $x \geq y_i$ für $i = 1, \dots, k$), dann gibt es ein minimales Matching zwischen den Punkten in X und Y, in dem x der Punkt y zugeordnet ist.

(ii) Gilt $x \in [y, y']$, wobei $y, y' \in Y$ mit der Eigenschaft, daß das offene Intervall $]y, y'[$ keine Punkte aus X enthält, dann dann gibt es ein minimales Matching zwischen den Punkten in X und Y, in dem x einer der Punkte y, y' zugeordnet ist.

Beweis: Übung 8.2. □

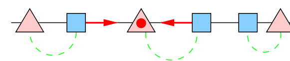
Beweis von Behauptung 8.5: Die Aussage (i) ist trivial. Die Aussage (ii) ist ebenfalls sehr einfach: Wenn OPT einen Server bewegt, dann ändert sich die S_i-Komponente von Φ nicht. Der Wert des minimalen Matchings zwischen den Servern von DC und OPT kann sich offenbar nicht mehr vergrößern als der Abstand, den OPT seinen Server bewegt.

Um (iii) zu beweisen, unterscheiden wir, ob DC einen oder zwei Server bewegt. Wenn DC nur einen Server s bewegt, muß dieser nach Definition von DC ein Eckpunkt der konvexen Hülle der Server sein. Das bedeutet aber, daß Bewegung um δ den Abstand zu jedem der anderen $k-1$ Server um δ erhöht. Somit erhöht sich die S_i-Komponente um $(k-1)\delta$. Da s der dichteste Server zur Anfrage r_i ist, gibt es nach Lemma 8.6 ein minimales Matching, in dem s demjenigen Server von OPT zugeordnet ist, der bereits auf r_i sitzt. Somit erniedrigt sich die Matching-Komponente in Φ um $k\delta$. Insgesamt verringert sich Φ also um mindestens δ.



Server von DC (Quadrate) und von OPT (Dreiecke). Die gestrichelten Linien veranschaulichen das Matching.

Es verbleibt der Fall, daß DC zwei Server s_1 und s_2 um je $\delta/2$ bewegt. Einer der beiden Server ist der dichteste zur Anfrage r_i und nach Lemma 8.6 in einem minimalen Matching zum Server von OPT zugeordnet, der auf r_i sitzt. Seine Bewegung erniedrigt M_i also um $\delta/2$. Da der andere Server sich höchstens $\delta/2$ von seinem Matchingpartner wegbewegt, folgt, daß sich die Matchingkomponente in Φ nicht erhöhen kann. Wir betrachten nun S_i. Für jeden Server $s \neq s_1, s_2$ ändert sich die Summe $d(s_1, s) + d(s_2, s)$ nicht: Ein Summand erhöht sich um $\delta/2$, der andere verringert sich um $\delta/2$. Allerdings verringert sich der Abstand zwischen s_1 und s_2 um insgesamt $2 \cdot \delta/2 = \delta$. Also sinkt Φ insgesamt um mindestens δ. □



8.4 Der Work Function Algorithmus

In diesem Abschnitt werden wir folgendes Resultat beweisen:

Satz 8.7 ([25]) *Es gibt einen deterministischen Algorithmus* WFA, *welcher in jedem metrischen Raum* $(2k-1)$*-kompetitiv ist.*

Sei $M = (X, d)$ ein metrischer Raum. Eine *Konfiguration* eines Algorithmus ALG nennen wir eine Multimenge von k Punkten aus X, welche die aktuelle Positionen der Server von ALG repräsentieren. Wir verwenden im Weiteren folgende Notationen:

- Für Konfigurationen C_1 und C_2 bezeichnen wir mit $C_1 + C_2$ und $C_1 - C_2$ die Multimengenvereinigung bzw. -differenz.
- Wir schreiben kurz $C_1 + p$ für $C_1 + \{p\}$ und $C_1 - p$ für $C_1 - \{p\}$, wobei C_1 eine Konfiguration und $p \in X$ ein Punkt des metrischen Raums ist.
- Für zwei Konfigurationen C_1 und C_2 bezeichnen wir mit $D(C_1, C_2)$ den *Konfigurationsabstand*, d.h. das Gewicht eines (gewichts-) minimalen Matchings zwischen C_1 und C_2.

8.4.1 Arbeitsfunktionen

Definition 8.8 (Arbeitsfunktion (Work Function)) *Sei C eine Konfiguration und σ eine Anfragefolge. Dann definieren wir die* ***Arbeitsfunktion (Work Function)*** $w_\sigma(C) = w_\sigma(C_0, C)$ *als die optimalen Offline-Kosten, um ausgehend von der Startkonfiguration C_0 alle Anfragen aus σ zu bearbeiten und mit der Konfiguration C zu enden.*

Aus den Arbeitsfunktionen kann man die optimalen Offline-Kosten für eine Anfragefolge σ leicht bestimmen. Es gilt:

$$\text{OPT}(\sigma) = \min_C w_\sigma(C). \tag{8.3}$$

Sei nun $\sigma = r_1, \dots, r_n$ fixiert. Wir bezeichnen mit $\sigma_i := r_1, \dots, r_i$ den Präfix der ersten i Anfragen von σ. Wir leiten nun her, wie man die Arbeitsfunktionen durch dynamische Programmierung berechnen kann.

Offenbar gilt für jede Konfiguration C, daß $w_\emptyset(C) = D(C_0, C)$. Nehmen wir an, daß $w_{\sigma_i}(C)$ für alle Konfigurationen C bereits bekannt ist. Sei nun $r := r_{i+1}$ die nächste Anfrage und C eine Konfiguration. Wie sieht $w_{\sigma_i r}(C)$ aus?

Wenn $r \in C$ gilt, dann ist offenbar $w_{\sigma_i r}(C) = w_{\sigma_i}(C)$ (längere Anfragefolgen können die Kosten nicht erhöhen). Wenn $r \notin C$, dann muß der optimale Algorithmus beim Bearbeiten von $\sigma_i r$ zuerst die Anfrage r bearbeiten, wobei er Konfiguration $\tilde{C}$ mit $r \in \tilde{C}$ annimmt, und danach von $\tilde{C}$ zur Konfiguration C wechseln. Es gilt also:

$$\begin{aligned} w_{\sigma_i r}(C) &= \min_{\tilde{C}: r \in \tilde{C}} \left(w_{\sigma_i r}(\tilde{C}) + D(\tilde{C}, C) \right) \\ &= \min_{\tilde{C}: r \in \tilde{C}} \left(w_{\sigma_i}(\tilde{C}) + D(\tilde{C}, C) \right) \end{aligned} \tag{8.4}$$

Für die letzte Gleichung haben wir unsere Beobachtung von oben ausgenutzt, daß im Fall $r \in \tilde{C}$ gilt: $w_{\sigma_i r}(\tilde{C}) = w_{\sigma_i}(\tilde{C})$. Man sieht nun leicht, daß die Gleichung (8.4) auch für den Fall $r \in C$ gültig bleibt (in diesem Fall ist C bei der Minimierung auf der rechten Seite als Wahl von $\tilde{C}$ zugelassen). Somit haben wir eine Rekursionsgleichung für die Arbeitsfunktionen gefunden. Mit Hilfe dynamischer Programmierung läßt sich somit $w_\sigma(C)$ für jede Anfragefolge und jede Konfiguration berechnen. Insbesondere kann man $\texttt{OPT}(\sigma)$ mit Hilfe von (8.3) bestimmen.

Wir vereinfachen die rechte Seite von (8.4) nun noch ein wenig, indem wir zeigen, daß es ausreicht, über solche Konfigurationen $\tilde{C}$ zu minimieren, die sich von C durch genau einen Punkt unterscheiden, d.h., die von der Form $\tilde{C} = C - x + r$ mit $x \in X$ sind. Bei den folgenden Rechnungen verwenden wir folgende einfache Eigenschaft der Arbeitsfunktionen:

$$w_{\sigma_i}(A) \leq w_{\sigma_i}(B) + D(A, B) \quad \text{für alle Konfigurationen } A, B. \tag{8.5}$$

Sei $\tilde{C}$ eine Konfiguration, für die das Minimum auf der rechten Seite von (8.4) angenommen wird. Sei $\tilde{C} - C = Y + r$ für eine nichtleere Multimenge Y. Wir zeigen nun, daß wir im Fall $Y \neq \emptyset$ eine Menge $\hat{C}$ finden können, die ebenfalls die rechte Seite von (8.4) minimiert und die gleichzeitig $|\hat{C} - C| < |\tilde{C} - C|$ erfüllt.

Mit $M: \tilde{C} \to C$ bezeichnen wir das minimale Matching zwischen $\tilde{C}$ und C mit Gewicht $D(\tilde{C}, C)$. Wähle $y \in Y$ mit $z := M(y) \in C - \tilde{C}$. Wir betrachten nun die Menge $\hat{C} := \tilde{C} - y + z$. Diese erfüllt $|\hat{C} - C| < |Y| = |\tilde{C} - C|$. Ferner gilt:

$$\begin{aligned}
w_{\sigma_i}(\hat{C}) + D(\hat{C}, C) &\leq w_{\sigma_i}(\hat{C}) + \sum_{x \in \tilde{C} - y} d(x, M(x)) + d(z, z) \\
&= w_{\sigma_i}(\tilde{C} - y + z) + \sum_{x \in \tilde{C} - y} d(x, M(x)) + d(z, z) \\
&\leq w_{\sigma_i}(\tilde{C}) + d(y, z) + \sum_{x \in \tilde{C} - y} d(x, M(x)) \\
&= w_{\sigma_i}(\tilde{C}) + \sum_{x \in \tilde{C}} d(x, M(x)) \\
&= w_{\sigma_i}(\tilde{C}) + D(\tilde{C}, C).
\end{aligned}$$

Folgende Notation verwenden wir zur Verkürzung im Rest des Abschnitts:

$$\begin{aligned}
w(C) &:= w_{\sigma_i}(C), \\
w'(C) &:= w_{\sigma_i r}(C).
\end{aligned}$$

Wir fassen die Ergebnisse von oben in einem Lemma zusammen.

Lemma 8.9 *Sei $\sigma = r_1, \ldots, r_n$ eine Anfragefolge, C eine beliebige Konfiguration und $r := r_{i+1}$. Dann gilt:*

(i) $w_\emptyset(C) = D(C_0, C)$

(ii) $w'(C) = \min_{x \in C} (w(C - x + r) + d(x, r))$. □

8.4.2 Der Algorithmus WFA

Algorithmus WFA Sei σ_i die bisherige Anfragefolge und C die aktuelle Konfiguration von WFA(welche nach Bearbeiten von σ_i angenommen wird). Sei $r = r_{i+1}$ die nächste Anfrage. WFA bearbeitet r mit einem Server $s \in C$ mit

$$s = \operatorname*{argmin}_{x \in C} (w(C - x + r) + d(x, r)) .$$

Es folgt aus der Konstruktion von WFA, daß die Kosten von WFA für die Anfrage $r = r_{i+1}$ genau $d(s, r)$ sind. Ist $C' = C - s + r$ die Folgekonfiguration von WFA, so folgt dann mit Hilfe von Lemma 8.9:

$$w'(C) = w(C - s + r) + d(s, r) = w(C') + d(s, r) = w'(C') + d(s, r). \quad (8.6)$$

Für die letzte Gleichheit haben wir $r \in C'$ benutzt.

8.4.3 Analyse des Algorithmus WFA

Wir werden nun Satz 8.7 über eine Reihe von Lemmata beweisen. Sei $\sigma = r_1, \ldots, r_n$ fixiert. Wir nehmen o. B. d. A. an, daß WFA und OPT in der gleichen Konfiguration starten und enden. Die letztere Annahme ist wegen Ungleichung (8.5) statthaft. Ist nämlich A die Endkonfiguration von WFA nach Bearbeiten von σ und $\mathsf{WFA}(\sigma) \leq c\, w_\sigma(B) + b$, so folgt mit (8.5):

$$\mathsf{WFA}(\sigma) \leq c \min_C w_\sigma(C) + (b + c \max_{C,C'} D(C, C')) = c\,\mathsf{OPT}(\sigma) + \tilde{b}.$$

Die Konstante $\tilde{b}$ ist dabei unabhängig von der Anfragefolge.

Sei $\sigma_i = r_1, \ldots, r_i$ die bisherige Anfragefolge und C die aktuelle Konfiguration von WFA. Sei $r := r_{i+1}$ die nächste Anfrage, die WFA durch Bewegen eines Servers von s nach r bearbeitet. Sei $C' = C - s + r$ die Folgekonfiguration von WFA. Wir definieren die *Offline Pseudokosten* für Anfrage r_i durch

$$\mathrm{PC}_i := w'(C') - w(C). \quad (8.7)$$

Es gilt dann:

$$\sum_{i=1}^{n} \mathrm{PC}_i = \mathsf{OPT}(\sigma) - w_\emptyset(C_0) = \mathsf{OPT}(\sigma). \quad (8.8)$$

Weiterhin seien die *erweiterten Kosten* für Anfrage r_i durch

$$\mathrm{EC}_i := \max_X (w'(X) - w(X)) \quad (8.9)$$

definiert. Die erweiterten Kosten entsprechen dem maximalen Anstieg einer Arbeitsfunktion bei der aktuellen Anfrage.

Lemma 8.10 *Sei b eine Konstante, die unabhängig von σ ist. Wenn*

$$\sum_{i=1}^{n} \mathrm{EC}_i \leq (c + 1)\,\mathsf{OPT}(\sigma) + b,$$

dann ist WFA c-kompetitiv.

Beweis: Wir betrachten die Summe der Offline Pseudokosten $\text{PC}_i = w'(C') - w(C)$ und der Kosten von WFA beim Bearbeiten der $i$ten Anfrage:

$$\text{PC}_i + \mathsf{WFA}(\sigma_i) = w'(C') - w(C) + d(s, r).$$

Nach Gleichung (8.6) gilt $w'(C') = w'(C) - d(s, r)$. Somit ist

$$\text{PC}_i + \mathsf{WFA}(\sigma_i) = w'(C) - w(C) \leq \max_X (w'(X) - w(X)) = \text{EC}_i.$$

Summation über alle Anfragen ergibt mit (8.7):

$$\mathsf{OPT}(\sigma) + \mathsf{WFA}(\sigma) \leq \sum_{i=1}^{n} \text{EC}_i \leq (c+1)\,\mathsf{OPT}(\sigma) + b.$$

Dies zeigt, daß WFA c-kompetitiv ist. □

Lemma 8.10 hat den Vorteil, daß wir uns nicht länger mit den Konfigurationen von WFA beschäftigen müssen. Um Kompetitivität zu zeigen, genügt es die erweiterten Kosten abzuschätzen. Auf der anderen Seite überschätzen die erweiterten Kosten möglicherweise die tatsächlichen Kosten von WFA, so daß wir eventuell keine optimalen Kompetitivitätsresultate zeigen.

Im folgenden schieben wir den Beweis einiger Lemmata kurzzeitig auf, um die Struktur des gesamten Beweises deutlicher zu machen. Im Hinblick auf Lemma 8.10 beschäftigen wir uns zuächst mit den erweiterten Kosten. Wir untersuchen, wie eine Konfiguration aussieht, bei der die Arbeitsfunktion maximal ansteigt.

Definition 8.11 (Maximum bezüglich w) *Sei w die aktuelle Arbeitsfunktion und $r = r_{i+1}$ die nächste Anfrage. Eine Konfi guration A heißt **Maximum bezüglich** w, wenn*

$$A = \operatorname*{argmax}_X (w'(X) - w(X))\,.$$

Für ein Maximum A bezüglich w gilt dann $\text{EC}_i = w'(A) - w(A)$.

Definition 8.12 (Minimum für r bezüglich w) *Sei w die aktuelle Arbeitsfunktion und r ein beliebiger Punkt des metrischen Raums M. Eine Konfi guration A heißt dann ein **Minimum für** r **bezüglich** w, wenn*

$$A = \operatorname*{argmin}_X \left(w(X) - \sum_{x \in X} d(x, r) \right).$$

Lemma 8.13 (Dualitätslemma) *Sei w die aktuelle Arbeitsfunktion und $r = r_{i+1}$ die nächste Anfrage. Jedes Minimum für r beüglich w ist auch ein Maximum bezüglich w.*

Sei nun A ein Minimum für r bezüglich der aktuellen Arbeitsfunktion w. Wir definieren

$$\text{MIN}_w(r) := w(A) - \sum_{a \in A} d(a, r).$$

Zur Analyse der erweiterten Kosten führen wir eine (Art) Potentialfunktion (vgl. Beweis von Satz 8.4) ein. Sei $U = \{u_1, \ldots, u_k\}$ die aktuelle Konfiguration von OPT. Dann sei

$$\phi(U, w) := \sum_{u \in U} \mathrm{MIN}_w(u).$$

Sei nun wieder $r = r_{i+1}$ die nächste Anfrage, die von OPT durch einen Server bearbeitet wird, der vorher auf u_j steht. Die Folgekonfiguration von OPT ist also $U' = U - u_j + r$.

Lemma 8.14 *Es gilt*

$$\phi(U', w) - \phi(U, w) \geq -kd(u_j, r).$$

Beweis: Wir setzen die Definition von ϕ ein:

$$\begin{aligned} \phi(U', w) - \phi(U, w) &= \sum_{u' \in U'} \mathrm{MIN}_w(u') - \sum_{u \in U} \mathrm{MIN}_w(u) \\ &= \mathrm{MIN}_w(r) - \mathrm{MIN}_w(u_j). \end{aligned} \tag{8.10}$$

Es gilt nun:

$$\begin{aligned} \mathrm{MIN}_w(r) &= \min_A \left(w(A) - \sum_{a \in A} d(a, r) \right) \\ &\geq \min_A \left(w(A) - \sum_{a \in A} (d(a, u_j) + d(u_j, r)) \right) \quad \mathrm{MIN}_w(u_j) - kd(u_j, r). \end{aligned}$$

Setzt man dies in (8.10) ein, so folgt die Behauptung. □

Lemma 8.15 *Es gilt*

$$\phi(U', w') - \phi(U', w) \geq EC_i = \max_X \left(w'(X) - w(X) \right).$$

Beweis: Siehe weiter hinten. □

Beweis von Satz 8.7 Um den Satz unter Zuhilfenahme von Lemma 8.10 zu beweisen, müssen wir zeigen, daß

$$\sum_{i=1}^{n} \mathrm{EC}_i \leq 2k\, \mathsf{OPT}(\sigma) + b$$

für eine Konstante b gilt. Aus den Lemmas 8.14 und 8.15 wissen wir folgendes:

$$\phi(U', w) - \phi(U, w) \geq -kd(u_j, r) \tag{8.11}$$

$$\phi(U', w') - \phi(U', w) \geq \mathrm{EC}_i. \tag{8.12}$$

Summieren wir die Ungleichungen (8.11) und (8.12), so ergibt sich:

$$\phi(U', w') - \phi(U, w) \geq \mathrm{EC}_i - kd(u_j, r) = \mathrm{EC}_i - k\, \mathsf{OPT}(\sigma_i). \tag{8.13}$$

Durch Summation über alle Anfragen $\sigma_1, \ldots, \sigma_n$ erhalten wir aus (8.13):

$$\sum_{i=1}^{n} \mathrm{EC}_i \leq k\, \mathsf{OPT}(\sigma) + \phi(U_\sigma, w_\sigma) - \phi(U_\emptyset, w_\emptyset)$$

Hierbei bezeichnen $U_\emptyset = C_0$ und $U_\sigma = C_n$ die Start- bzw. Endkonfiguration von OPT (und WFA). Es genügt nun zu zeigen, daß folgende zwei Ungleichungen erfüllt sind:

(i) $\phi(U_\sigma, w_\sigma) \leq k w_\sigma(U_\sigma) = k\, \mathsf{OPT}(\sigma)$.

(ii) $\phi(U_\emptyset, w_\emptyset) \geq -2 \sum_{a,b \in C_0} d(a,b) = \text{const.}$

Zu (i): Es gilt:

$$\phi(U_\sigma, w_\sigma) = \sum_{u \in U_\sigma} \mathrm{MIN}_{w_\sigma}(u) \tag{8.14}$$

Ferner gilt nach Definition von $\mathrm{MIN}_{w_\sigma}(u)$, daß

$$\mathrm{MIN}_{w_\sigma}(u) \leq \min_{y \notin U_\sigma} \left(w_\sigma(U_\sigma - u + y) - \sum_{x \in U_\sigma - u + y} d(x,u) \right) \tag{8.15}$$

Sei nun y^* so gewählt, daß die rechte Seite von (8.15) minimiert wird. Dann ergibt sich aus (8.15)

$$\begin{aligned} \mathrm{MIN}_{w_\sigma}(u) &\leq w_\sigma(U_\sigma - u + y^*) - \sum_{x \in U_\sigma - u + y^*} d(x,u) \\ &\leq w_\sigma(U_\sigma - u + y^*) - d(y^*, u) \\ &\leq w_\sigma(U_\sigma). \end{aligned}$$

Dieses Ergebnis in (8.14) eingesetzt zeigt (i).

Zu (ii): Nach Definition von ϕ haben wir:

$$\phi(U_\emptyset, w_\emptyset) = \sum_{u \in U_\emptyset} \mathrm{MIN}_{w_\emptyset}(u) = \sum_{u \in C_0} \mathrm{MIN}_{w_\emptyset}(u). \tag{8.16}$$

Wir betrachten wieder die einzelnen Terme $\mathrm{MIN}_{w_\emptyset}(u)$ in der Summe. Sei dazu $u \in C_0$ fest und $A = \{a_1, \ldots, a_k\}$ eine Konfiguration, mit

$$\mathrm{MIN}_{w_\emptyset}(u) = w_\emptyset(A) - \sum_{a \in A} d(a,u) \overset{\text{Lemma 8.9 (i)}}{=} D(C_0, A) - \sum_{a \in A} d(a,u). \tag{8.17}$$

O. B. d. A. seien die Elemente von $A = \{a_1, \ldots, a_k\}$ und $C_0 = \{u_1, \ldots, u_k\}$ so numeriert, daß a_i im mimimalen Matching u_i zugeordnet ist. Dann dann können wir aus (8.16) ableiten, daß

$$\begin{aligned} \mathrm{MIN}_{w_\emptyset}(u) &= \sum_{i=1}^{k} d(u_i, a_i) - \sum_{i=1}^{k} d(a_i, u) \\ &= \sum_{i=1}^{k} \left(d(u_i, a_i) - d(a_i, u) \right) \\ &\geq -\sum_{i=1}^{k} d(u_i, u) \qquad \text{(nach der Dreiecksungleichung).} \end{aligned}$$

Einsetzen in (8.16) liefert:

$$\phi(U_\emptyset, w_\emptyset) \geq -\sum_{u \in C_0} \sum_{i=1}^{k} d(u_i, u) = -2 \sum_{a,b \in C_0} d(a, b).$$

Dies beendet den Beweis von Satz 8.7.

8.4.4 Beweis des Dualitätslemmas

Wir holen zunächst den Beweis des Dualitätslemmas (Lemma 8.13) nach. Dazu benötigen wir weitere Hilfsaussagen. Ein wichtiges Hilfsmittel ist dabei die Quasi-Konvexität der Arbeitsfuntionen.

Definition 8.16 (Quasi-Konvexität) *Eine Arbeistfunktion w heißt **quasikonvex**, wenn für alle Konfigurationen X und Y und jeden Punkt $x \in X$ gilt:*

$$\min_{y \in Y} (w(X - x + y) + w(Y - y + x)) \leq w(X) + w(Y). \tag{8.18}$$

Lemma 8.17 (Quasi-Konvexitätslemma) *Alle Arbeitsfunktionen sind quasikonvex.*

Beweis: Wir zeigen hier eine etwas allgemeinere Eigenschaft:

Eigenschaft (GQ) Sei w eine Arbeitfunktion, X und Y zwei Konfigurationen. Dann existiert eine Bijektion $g\colon X \to Y$, so daß folgende Ungleichung gilt:

$$w(X_1 + g(X_2)) + w(g(X_1) + X_2) \leq w(X) + w(Y)$$
$$\text{für alle Partitionen } X = X_1 \cup X_2 \text{ von } X \tag{8.19}$$

Die Eigenschaft (GQ) impliziert die Quasi-Konvexität: Man setzt einfach $X_1 := X - x$ und $X_2 = x$.

Behauptung 8.18 *Wenn eine Bijektion g die Ungleichung (8.19) erfüllt, dann gibt es auch eine Bijektion $\bar{g}$, mit $\bar{g}(x) = x$ für alle $x \in X \cap Y$, die ebenfalls (8.19) erfüllt.*

Beweis: Wir wählen unter allen Bijektionen, welche (8.19) erfüllen, eine solche Bijektion g aus, welche $|\{\, x \in X \cap Y : g(x) = x \,\}|$ maximiert. Wir zeigen nun, daß g alle Elemente im Schnitt $X \cap Y$ invariant läßt.

Dazu nehmen wir an, es gäbe $a \in X \cap Y$ mit $g(a) \neq a$. Wir definieren nun $g'\colon X \to Y$ mit

$$g'(x) := \begin{cases} g(x) & x \in X \setminus \{a, g^{-1}(x)\} \\ a & x = a \\ g(a) & x = g^{-1}(x). \end{cases}$$

Die Abbildung g' vertauscht also die Bilder von a und $g^{-1}(a)$. Wir zeigen nun, daß g' ebenfalls (8.19) erfüllt, was der Wahl von g widerspricht (g' läßt mehr Elemente im Schnitt invariant).

Sei $X = X_1 \cup X_2$ eine beliebige Partion von X und o. B. d. A. $a \in X_2$. Falls $g^{-1}(a) \in X_2$, dann gilt $g'(X_1) = g(X_1)$ und $g'(X_2) = g(X_2)$. In diesem Fall ist Ungleichung (8.19) erfüllt, da g ihr genügt. Wir können also annehmen, daß $g^{-1}(a) \in X_1$ gilt. Es gilt nun:

$$
\begin{aligned}
&w(X_1 + g'(X_2)) + w(g'(X_1) + X_2) \\
&= w(X_1 + g'(X_2 - a) + \underbrace{g'(a)}_{=a}) + w(g'(X_1) + X_2 - a + a) \\
&= w(X_1 + a + g'(X_2 - a)) + w(g'(X_1) + a + X_2 - a) \\
&= w(X_1 + a + g(X_2 - a)) + w(g(X_1) + a + X_2 - a) \quad \text{da } g'|_{X_2 - a} = g|_{X_2 - a} \\
&\leq w(X) + w(Y).
\end{aligned}
$$

Für die letzte Ungleichung haben wir ausgenutzt, daß g die Bedingung (8.19) erfüllt. Dies beendet den Beweis der Behauptung 8.18. □

Wir beweisen nun die Eigenschaft (GQ) für die Arbeitsfunktionen unter Zuhilfenahme von Behauptung 8.18. Wir benutzen Induktion nach der Länge der Anfragefolge σ.

Induktionsanfang: $\sigma = \emptyset$

Es gilt $w_\sigma(C) = w_\emptyset(C) = D(C, C_0)$ für alle Konfigurationen C, wobei C_0 die Anfangskonfiguration ist. Somit ist für Konfigurationen X, Y:

$$w(X) + w(Y) = D(X, C_0) + D(Y, C_0).$$

Seien $M_X\colon X \to C_0$ und $M_Y\colon Y \to C_0$ die zugehörigen minimalen Matchings. Wir definieren die Bijektion $g\colon X \to Y$ durch $g(x) := M_Y^{-1}(M_X(x))$. Diese Funktion ordnet also einem Punkt $x \in X$ den Matchingpartner von $M_X(x)$ in Y zu. Man beachte, daß gilt:

$$M_X(x) = M_Y(g(x)) \quad \text{für alle } x \in X.$$

Ist nun $X_1 \cup X_2 = X$ eine beliebige Partition von X, so gilt:

$$
\begin{aligned}
w(X_1 + g(X_2)) + w(g(X_1) + X_2) &= D(X_1 + g(X_2), C_0) + D(g(X_1) + X_2, C_0) \\
&\leq \sum_{x \in X_1} d(x, M_X(x)) + \sum_{x \in X_2} d(g(x), M_X(x)) \\
&\quad + \sum_{x \in X_1} d(g(x), M_X(x)) + \sum_{x \in X_2} d(x, M_X(x)) \\
&= \sum_{x \in X_1} d(x, M_X(x)) + \sum_{x \in X_2} d(g(x), M_Y(g(x))) \\
&\quad + \sum_{x \in X_1} d(g(x), M_Y(g(x))) + \sum_{x \in X_2} d(x, M_X(x)) \\
&= D(X, C_0) + D(Y, C_0).
\end{aligned}
$$

Dies zeigt den Induktionsanfang.

Induktionsschritt: $i \to i+1$.

Wir nehmen an, daß w die Eigenschaft (GQ) besitzt. Sei r die neue Anfrage. Wir müssen zeigen, daß w' ebenfalls die Eigenschaft (GQ) hat. Seien dazu die Konfigurationen X und Y. Wir wissen, daß

$$\begin{aligned} w'(X) &= w(X - x + r) + d(x,r) && \text{für ein } x \in X\text{; und} \\ w'(Y) &= w(Y - y + r) + d(y,r) && \text{für ein } y \in Y \end{aligned}$$

gilt. Nach Induktionsvoraussetzung existiert eine Bijektion $g\colon X - x + r \to Y - y + r$, welche den Bedingungen der Eigenschaft (GQ) genügt. Nach Behauptung 8.18 können wir o. B. d. A. annehmen, daß $g(r) = r$ gilt.

Wir definieren nun eine Bijektion $g'\colon X \to Y$ wie folgt:

$$g'(z) = \begin{cases} g(z) & \text{falls } z \neq x \\ y & \text{falls } z = x. \end{cases}$$

Unser Ziel ist es nun zu beweisen, daß g' die Eigenschaften von (GQ) erfüllt. Sei dazu $X = X_1 \cup X_2$ wieder eine beliebige Partition von X. Wir nehmen o. B. d. A. an, daß $x \in X_1$ gilt. Es gilt dann:

$$\begin{aligned} w'(X) + w'(Y) &= w(X - x + r) + d(x,r) + w(Y - y + r) + d(y,r) \\ &= w((X_1 - x + r) + X_2) + w(Y - y + r) + d(x,r) + d(y,r) \\ &\geq w(X_1 - x + r + g(X_2)) + w(g(X_1 - x + r) + X_2) + d(x,r) + d(y,r) \qquad \text{da} \\ &= w(X_1 - x + r + g'(X_2)) + w(g'(X_1 - x + r) + X_2) + d(x,r) + d(y,r) \\ &= w(X_1 - x + r + g'(X_2)) + w(g'(X_1) - y + r + X_2) + d(x,r) + d(y,r) \\ &\geq w'(X_1 + g'(X_2)) + w'(g'(X_1) + X_2). \end{aligned}$$

Dies beendet den Beweis. □

Lemma 8.19 *Sei w die aktuelle Arbeitsfunktion und r die nächste Anfrage. Wenn A ein Minimum für r bezüglich w ist, dann ist A auch ein Minimum von r bezüglich w'.*

Beweis: Wir müssen zeigen, daß

$$w'(A) + \sum_{a \in A} d(a,r) \leq w'(B) + \sum_{b \in B} d(b,r)$$

für jede Konfiguration B gilt. Wir wissen, daß

$$w'(B) = w(B - b' + r) + d(b',r) \qquad \text{für ein } b' \in B.$$

Sei $\bar{a} \in A$ beliebig. Da A ein Minimum von r bezüglich w ist, gilt:

$$\begin{aligned} w(A) - \sum_{a \in A} d(a,r) &\leq w(B - b' + \bar{a}) - \sum_{b \in B - b' + \bar{a}} d(b,r) \\ &= w(B - b' + \bar{a}) - \sum_{b \in B} d(b,r) + d(b',r) - d(\bar{a},r). \end{aligned} \tag{8.20}$$

Wir verwenden die Quasikonvexität von w für die Konfigurationen $X = B - b' + r$, $Y = A$ und den Punkt $x = r$. Dies ergibt:

$$\min_{\tilde{a}\in A} (w(B - b' + \tilde{a}) + w(A - \tilde{a} + r)) \leq w(B - b' + r) + w(A). \tag{8.21}$$

Addition der Ungleichungen (8.20) und (8.21) ergibt:

$$\begin{aligned} &- \sum_{a\in A} d(a,r) + \min_{\tilde{a}\in A} (w(B - b' + \tilde{a}) + w(A - \tilde{a} + r)) \\ \leq &w(B - b' + \bar{a}) - d(\bar{a}, r) + \underbrace{w(B - b' + r) + d(b', r)}_{=w'(B)} - \sum_{b\in B} d(b,r). \end{aligned}$$

Wir wählen $\bar{a}$ als ein solches $\tilde{a}$, für welches das Minimum auf der linken Seite angenommen wird. Bringen wir noch $w(B - b' + \bar{a}) - d(\bar{a}, r)$ auf die andere Seite, so erhalten wir:

$$\begin{aligned} &w'(B) - \sum_{b\in B} d(b,r) \\ \geq &- \sum_{a\in A} d(a,r) + w(B - b' + \bar{a}) + w(A - \bar{a} + r) - w(B - b' + \bar{a}) + d(\bar{a}, r) \\ &= w(A - \bar{a} + r) + d(\bar{a}, r) - \sum_{a\in A} d(a,r) \\ &\geq w'(A) - \sum_{a\in A} d(a,r). \end{aligned}$$

Dies war zu zeigen. □

Wir können nun das Dualitätslemma 8.13 beweisen. Sei A ein Minimum von r bezüglich w. Wir müssen zeigen, daß für jede Konfiguration B die Ungleichung

$$w'(A) - w(A) \geq w'(B) - w(B)$$

gilt. Sei $a' \in A$ mit $w'(A) = w(A - a' + r) + d(a', r)$. Sei ferner $\bar{b} \in B$ beliebig. Da A ein Minimum von r bezüglich w ist, gilt:

$$\begin{aligned} w(A) - \sum_{a\in A} d(a,r) &\leq w(A - a' + \bar{b}) - \sum_{a\in A - a' + \bar{b}} d(a,r) \\ &= w(A - a' + \bar{b}) - \sum_{a\in A} d(a,r) + d(a', r) - d(\bar{b}, r). \end{aligned}$$

Daraus ergibt sich unmittelbar:

$$w(A) + d(\bar{b}, r) - d(a', r) \leq w(A - a' + \bar{b}). \tag{8.22}$$

Wiederum nutzen wir die Quasikonvexität von w aus. Diesmal benutzen wir die Konfigurationen $X = A - a' + r$, $Y = B$ und den Punkt $x = r$. Dann erhalten wir:

$$\min_{\tilde{b}\in B} \Big(w(A - a' + \tilde{b}) + w(B - \tilde{b} + r)\Big) \leq w(A - a' + r) + w(B). \tag{8.23}$$

Addition von (8.22) und (8.23) zeigt dann:

$$\begin{aligned}&w(A) + d(\bar{b}, r) - d(a', r) + \min_{\tilde{b} \in B} \left(w(A - a' + \tilde{b}) + w(B - \tilde{b} + r) \right) \\ \leq & w(A - a' + \bar{b}) + w(A - a' + r) + w(B).\end{aligned}$$

Durch Umstellen ergibt sich unter Zuhilfenahme von $w(A-a'+r)+d(a',r) = w'(A)$:

$$\begin{aligned}&\min_{\tilde{b} \in B} \left(w(A - a' + \tilde{b}) + w(B - \tilde{b} + r) \right) - w(A - a' + \bar{b}) + d(\bar{b}, r) - w(B) \\ \leq & w'(A) - w(A),\end{aligned}$$

wobei diese Ungleichung für alle $\bar{b} \in B$ gilt. Daher gilt auch

$$\begin{aligned}w'(A) - w(A) &\geq \min_{\tilde{b} \in B} \left(w(B - \tilde{b} + r) + d(\tilde{b}, r) \right) - w(B) \\ &= w'(B) - w(B).\end{aligned}$$

Dies beendet den Beweis des Dualitätslemmas.

8.4.5 Beweis von Lemma 8.15

Wir müssen zeigen, daß für $U' = U - u_j + r$ gilt:

$$\phi(U', w') - \phi(U', w) \geq \text{EC}_i = \max_X \left(w'(X) - w(X) \right).$$

Sei A ein Minimum von r bezüglich w. Dann gilt nach dem Dualitätslemma:

$$w'(A) - w(A) = \max_X \left(w'(X) - w(X) \right) \tag{8.24}$$

Ferner gilt nach Definition des Minimums:

$$w(A) - \sum_{a \in A} d(a, r) = \text{MIN}_w(r). \tag{8.25}$$

Nach Lemma 8.19 ist A auch ein Minimum von r bezüglich w', d.h.:

$$w'(A) - \sum_{a \in A} d(a, r) = \text{MIN}_{w'}(r). \tag{8.26}$$

Aus (8.25) und (8.26) folgt:

$$\begin{aligned}\text{MIN}_{w'}(r) - \text{MIN}_w(r) &= w'(A) - w(A) \\ &= \max_X \left(w'(X) - w(X) \right) \quad \text{(nach (8.24))}.\end{aligned} \tag{8.27}$$

Sei nun x ein beliebiger Punkt des metrischen Raums. Dann gilt

$$\begin{aligned}\text{MIN}_w(x) &= \min_A \left(w(A) - \sum_{a \in A} d(a, x) \right) \\ &\leq \min_A \left(w(A) - \sum_{a \in A} d(a, x) \right) \quad \text{(da } w'(A) \geq w(A) \text{ für alle } A\text{)} \\ &= \text{MIN}_{w'}(x).\end{aligned}$$

Also ist

$$\mathrm{MIN}_{w'}(x) - \mathrm{MIN}_w(x) \geq 0. \tag{8.28}$$

Wir setzen jetzt die Definition der Potentialfunktion ein:

$$\begin{aligned}
\phi(U', w') - \phi(U', w) &= \sum_{u' \in U'} (\mathrm{MIN}_{w'}(u') - \mathrm{MIN}_w(u')) \\
&\geq \mathrm{MIN}_{w'}(u') - \mathrm{MIN}_w(u') && \text{(wegen (8.28))} \\
&= \max_X (w'(X) - w(X)) && \text{(wegen (8.27)).}
\end{aligned}$$

Dies wollten wir zeigen.

Übungsaufgaben

Übung 8.1 (Naheliegender Algorithmus für das k-Server Problem)
Ein naheliegender Algorithmus für das k-Server Problem ist der folgende: Sei r_i die aktuelle Anfrage. Sofern kein Server auf r_i steht, benutze einen Server, der am dichtesten an der Anfrage r_i liegt, um r_i zu bearbeiten.

Beweisen Sie, daß der obige Algorithmus nicht kompetitiv für das k-Server Problem in $\mathbb{R}$ ist.

Übung 8.2 (Eigenschaften von Matchings auf der reellen Achse)
Beweisen Sie Lemma 8.6.

Übung 8.3 (k-Server Problem und Approximation metrischer Räume)
Wir betrachten das k-Server Problem auf einem endlichen metrischen Raum (X, d). Sei (X', d') ein weiterer metrischer Raum mit folgenden Eigenschaften:

(i) $X \subseteq X'$, d.h. alle Punkte aus X sind auch in X'.

(ii) Für alle $x, y \in X$ gilt:

$$d(x, y) \leq d'(x, y). \tag{8.29}$$

(iii) Es gibt eine Konstante $\beta \geq 1$, so daß für alle $x, y \in X$ gilt:

$$d'(x, y) \leq \beta d(x, y). \tag{8.30}$$

Im folgenden soll bewiesen werden, wie man durch Approximation metrischer Räume bekannte Algorithmen auf »einfachen« Räumen benutzen kann, um kompetitive Algorithmen für »kompliziertere« metrische Räume zu konstruieren.

(a) Zeigen Sie, daß ein c-kompetitiver Algorithmus für das k-Server Problem in (X', d') einen βc-kompetitiven Algorithmus für das k-Server Problem in (X, d) liefert.

(b) Es sei $G = (V, E)$ ein ungerichteter Graph mit Kantengewichten $d\colon E \to \mathbb{R}_{\geq 0}$. Wir betrachten das k-Server Problem auf dem metrischen Raum (X, d), wobei $X = V$ und $d(x, y)$ als die Länge des kürzesten Weges zwischen $x \in V$ und $y \in V$ definiert sei. Es sei $D(G) := \max_{x,y \in V} d(x, y)$ der Durchmesser von G.

Zeigen Sie, daß jeder k-kompetitive Paging Algorithmus benutzt werden kann, um einen kD-kompetitiven Algorithmus für das k-Server Problem auf G zu erhalten.

(c) Sei $G = (V, E)$ ein einfacher Kreis mit n Knoten und Einheitsgewichten auf den Kanten, d.h. $V = \{v_1, \ldots, v_n\}$, $E = \big\{(v_i, v_{(i+1) \bmod n}) : i = 1, \ldots, n\big\}$ und $d(v_i, v_{i+1}) = 1$ für alle i.

Zeigen Sie, daß es einen randomisierten $2k$-kompetitiven Algorithmus für das k-Server Problem auf G gegen OBL gibt.

9

Transport- und Logistik-Probleme

Offline-Transportprobleme, in denen Objekte zwischen vorgegebenen Quellen und Zielen transportiert werden sollen, sind klassische Probleme der Kombinatorischen Optimierung. In diesem Abschnitt betrachten wir Online-Transportprobleme, bei denen zukünftige Aufträge dem Online-Algorithmus unbekannt sind.

9.1 Problemstellung und Zeitstempel-Modell

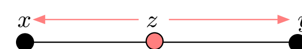
In allen bisherigen Online-Problemen haben wir von den Online-Algorithmen nach jeder Anfrage immer eine unmittelbare und unwiderrufliche Entscheidung gefordert. Häufig ist für Planungsprobleme jedoch das folgende *Zeitstempel-Modell* (auch *Real-Time Modell* genannt) aussagekräftiger: Jede Anfrage ist mit einem Zeitstempel versehen, der den Zeitpunkt markiert, zu dem die Anfrage dem Online-Algorithmus bekannt wird. Der Online-Algorithmus kann Anfragen »sammeln« und ihre Abarbeitung planen.

Zeitstempel-Modell

Wir definieren nun das in diesem Kapitel untersuchte Online-Transportproblem, welches wir OLDARP (Online-»Dial-a-Ride-Problem«) nennen.

Dial-a-Ride-Problem

Gegeben ist ein metrischer Raum (X, d) mit einem ausgezeichneten Ursprung o. Wir fordern, daß für je zwei Punkte $x, y \in X$ der kürzeste Weg zwischen x und y kontinuierlich in X enthalten ist und Länge $d(x, y)$ hat.[1]



Ein Beispiel für einen metrischen Raum, welcher die obige Bedingung erfüllt, ist der Euklidische Raum $\mathbb{R}^m$. Ein weiteres Beispiel ist der metrische Raum (X, d), der durch einen Graphen $G = (V, E)$ mit Kantengewichten induziert wird. Man definiert hier $d(x, y)$ als die Länge des kürzesten Weges zwischen $x \in V$ und $y \in V$. Man setzt dann die Funktion d auf die unendlich vielen Punkte auf den Kanten des Graphen fort. Da die Abstände sowieso gemäß kürzester Wege definiert sind, kann man auch o. B. d. A. davon ausgehen, daß die Kantengewichte die Dreiecksungleichung erfüllen.

Eine Anfrage (Transportauftrag) ist ein Tripel $r_i = (t_i, a_i, b_i)$ mit folgender

Transportauftrag

[1] Formaler fordern wir, daß es eine Abbildung $p\colon [0, 1] \to M$ gibt, so daß $p(0) = x$ und $p(1) = y$ sowie $d(x, p(\tau)) = \tau d(x, y)$ und $d(y, p(\tau)) = (1 - \tau)d(x, y)$ für alle $0 \leq \tau \leq 1$ gilt. Die Menge $p([0, 1]) \subseteq M$ heißt dann kürzester Weg zwischen x und y.

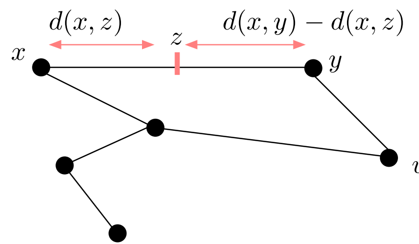


Abbildung 9.1
Ein gewichteter Graph induziert einen metrischen Raum.

Release-Zeit

$\sigma_{\leq t}$

$\sigma_{\geq t}$

Server

Bedeutung: $t_i \in \mathbb{R}_{\geq 0}$ ist der Zeitpunkt, zu dem die Anfrage r_i dem Online-Algorithmus bekannt wird, seine sogenannte *Release-Zeit*. Die beiden Punkte $a_i \in X$ und $b_i \in X$ bezeichnen die Quelle und das Ziel, zwischen denen das neue Objekt transportiert werden soll. Die Anfragefolge $\sigma = r_1, \ldots, r_n$ ist gemäß der Release-Zeiten der Aufträge sortiert, d.h. es gilt $t_1 \leq t_2 \leq \cdots \leq t_n$. Für $t \in \mathbb{R}_{\geq 0}$ bezeichnen wir mit $\sigma_{\leq t}$ und $\sigma_{\geq t}$ diejenigen Aufträge in σ mit Release-Zeit höchstens bzw. mindestens t.

Zum Bearbeiten der Transportaufträge steht einem Algorithmus ein *Server* zur Verfügung. Dieser hat Kapazität C, d.h. er kann maximal C Objekte gleichzeitig tragen. Der Server befindet sich zum Zeitpunkt 0 im Ursprung o, und kann sich mit konstanter Einheitsgeschwindigkeit im metrischen Raum bewegen.

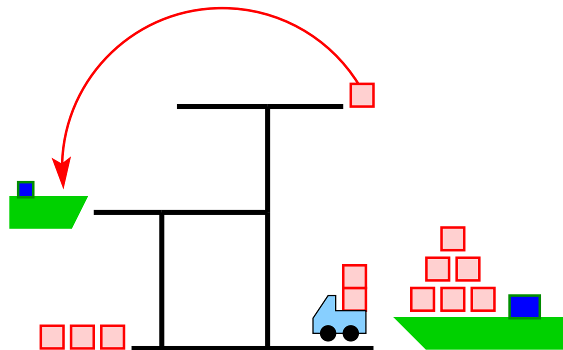


Abbildung 9.2
Situation bei OLDARP: Ein Server mit Kapazität $C = 2$ erledigt Transportaufträge. Während seiner Arbeit werden neue Aufträge bekannt.

Ein Online-Algorithmus kennt weder die Anzahl der Transportaufträge in σ noch den Zeitpunkt t_n, zu dem der letzte Auftrag bekanntgegeben wird. Er muß zu jedem Zeitpunkt t die Arbeitsweise des Servers nur durch Kenntnis der Aufträge aus $\sigma_{\leq t}$ bestimmen. Der Offline-Gegner hingegen kennt zum Zeitpunkt 0 bereits die vollständige Auftragsfolge.

Transportplan

Eine zulässiger *Transportplan* ist ein Routing für den Server, das im Ursprung o startet und endet, wobei jeder Transportauftrag abgearbeitet wird (aber nicht vor seiner Release-Zeit). Um die Qualität einer Lösung zu messen, bieten sich für OLDARP folgende Zielfunktionen an:

Makespan

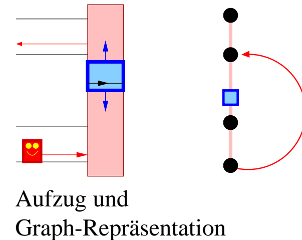
Fertigstellungszeit $C^{\max}$ Mit $C^{\max}$ bezeichnen wir denjenigen Zeitpunkt, zu dem der Server wieder in den Ausgangspunkt o zurückgekehrt ist, nachdem alle Aufträge erledigt worden sind. Diese Zielfunktion wird im Scheduling auch *Makespan* genannt.

Maximale Flußzeit $F^{\max}$ Die Flußzeit eines Auftrages $r_i = (t_i, a_i, b_i)$ ist die Differenz $T_i - t_i$ aus der Zeit T_i, zu der r_i bearbeitet worden ist, und der

Release-Zeit t_i des Auftrags.

Durchschnittliche Flußzeit F^{avg} Der Durchschnitt aller Flußzeiten.

Maximale und durchschnittliche Wartezeit $W^{\max}$ und W^{avg} Die Wartezeit eines Auftrags r_i entspricht seiner Flußzeit minus seiner Bearbeitungsdauer.

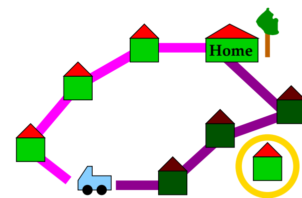


Aufzug und Graph-Repräsentation

Anwendungen von OLDARP finden sich beim Gütertransport, bei der Einsatzplanung von Taxis (daher leitet sich der Name »Dial-a-Ride« ab) und bei der Steuerung von Aufzügen. Im Falle des Aufzuges ist der zugrundeliegende metrische Raum durch einen Graphen $G = (V, E)$ induziert, der ein Pfad ist.

Traveling Salesperson Problem

Ein wichtiger Spezialfall von OLDARP ergibt sich, wenn für jede Anfrage $r_i = (t_i, a_i, b_i)$ gilt $a_i = b_i$, d.h. wenn für jeden Transportauftrag Quelle und Ziel identisch sind. In diesem Fall spielt die Kapazität des Servers keine Rolle. Der Server muß nur jeden Auftragspunkt mindestens einmal besuchen. Man erhält also eine Online-Variante des *Traveling Salesperson Problems* [8, 7].



Online-TSP (OLTSP)

Präemption

Wir betrachten im folgenden für OLDARP nur den Fall, daß die Kapazität C des Servers gleich eins ist, d.h. der Server kann jeweils nur ein Objekt gleichzeitig tragen. Weiterhin erlauben wir dem Server keine *Präemption*, d.h. ein einmal aufgenommenes Objekt darf von Server nur am vorgesehenen Zielort abgeladen werden.

9.2 Optimierung der Fertigstellungszeit

Referenzwerke: [4, 5]

$L^*(t, x, \sigma)$

Für eine Auftragsfolge σ, eine Zeit $t \in \mathbb{R}_{\geq 0}$ und einen Punkt $x \in X$ des metrischen Raums bezeichnen wir mit $L^*(t, x, \sigma)$ die Bearbeitungszeit eines kürzesten Transportplans, der zum Zeitpunkt t in x startet, alle Aufträge in σ bedient und im Ursprung o endet.

Es gilt dann:

1. $\mathsf{OPT}(\sigma) = L^*(0, o, \sigma)$.
2. Für $t' \geq t$ gilt $L^*(t', x, \sigma) \leq L^*(t, x, \sigma)$. Dabei kann die Ungleichung strikt sein: Es sei $X = \mathbb{R}$ und σ bestehe nur aus der Anfrage $r_1 = (1, 1, o)$. Dann gilt $L^*(0, 1, \sigma) = 2$, da die Anfrage r_1 erst zum Zeitpunkt 1 bekannt wird und der Server auf seinem Startpunkt 1 eine Zeiteinheit warten muß. Andererseits ist $L^*(1, 1, \sigma) = 1$.
3. Für $x, y \in X$ gilt: $L^*(t, y, \sigma) \leq d(x, y) + L^*(t, x, \sigma)$.

Wir betrachten nun zwei Online-Algorithmen für OLDARP:

Algorithmus REPLAN Wenn ein neuer Auftrag bekannt wird, dann führt der Server einen eventuell gerade bearbeiteten Auftrag noch fertig aus und errechnet dann einen neuen optimalen Transportplan, der in seiner aktuellen Position startet, alle noch nicht bearbeiteten Aufträge erledigt und im Ursprung o endet.

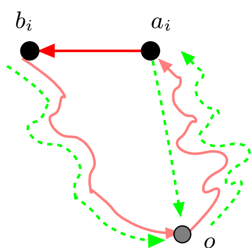
Algorithmus IGNORE Im Gegensatz zu REPLAN plant IGNORE erst neu, wenn er seinen aktuellen Transportplan vollständig abgearbeitet hat (und wieder in den Ursprung zurückgekehrt ist).

Im folgenden beweisen wir, daß beide Algorithmen $5/2$-kompetitiv für die Optimierung der Fertigstellungszeit $C^{\max}$ sind. Dafür benötigen wir noch ein Hilfsresultat.

Lemma 9.1 *Sei* $\sigma = r_1, \ldots, r_n$ *eine Auftragsfolge. Dann gilt für beliebiges* $t \geq t_n$ *und* $r_i = (t_i, a_i, b_i) \in \sigma$*:*

$$L^*(t, b_i, \sigma \setminus r_i) \leq L^*(t, o, \sigma) - d(a_i, b_i) + d(a_i, o).$$

Dabei bezeichne $\sigma \setminus r_i$ *die Anfragefolge* $r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n$.



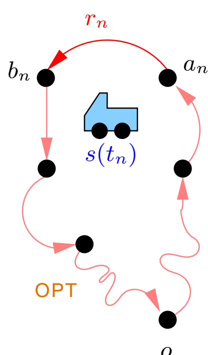
Aus S^* wird eine Tour konstruiert, die in b_i startet (gestrichelt).

Beweis: Sei S^* ein optimaler Transportplan der Länge $L^*(t, o, \sigma)$. Dieser bearbeite die Aufträge in σ in der Reihenfolge $r_{j_1}, \ldots, r_{j_n}$ mit $r_i = r_{j_k}$. Wenn wir nun in b_i zum Zeitpunkt t starten und die Aufträge in der Reihenfolge

$$r_{j_{k+1}}, \ldots, r_{j_n}, r_{j_1}, \ldots, r_{j_{k-1}}$$

bearbeiten und den Server danach zum Ursprung fahren, ergibt sich ein gültiger Transportplan der Länge höchstens $L^*(t, o, \sigma) - d(a_i, b_i) + d(a_i, o)$. □

Satz 9.2 *Algorithmus* REPLAN *ist* $5/2$*-kompetitiv für die Zielfunktion* $C^{\max}$.



1. Fall: Der Restaufwand ist höchstens $d(s(t_n), o) + \text{OPT}(\sigma)$

Beweis: Sei $\sigma = r_1, \ldots, r_n$ eine beliebige Auftragsfolge. Wir betrachten den Zeitpunkt t_n, zu dem der letzte Auftrag bekanntgegeben wird. Sei $s(t_n)$ die Position des REPLAN-Servers zum Zeitpunkt t_n.

Wenn der Server zum Zeitpunkt t_n gerade kein Objekt befördert, so plant er sofort neu. Seine Fertigstellungszeit erfüllt also in diesem Fall

$$\begin{aligned} \text{REPLAN}(\sigma) &\leq t_n + L^*(t_n, s(t_n), \sigma) \\ &\leq t_n + d(s(t_n), o) + L^*(t_n, o, \sigma) \\ &\leq t_n + d(s(t_n), o) + L^*(0, o, \sigma) \\ &= t_n + d(s(t_n), o) + \text{OPT}(\sigma). \end{aligned} \tag{9.1}$$

Es gilt $\text{OPT}(\sigma) \geq t_n$, da der optimale Offline-Server definitiv nicht *vor* Bekanntwerden des letzten Auftrags fertig sein kann. Da der REPLAN-Server zum Punkt $s(t_n)$ gefahren ist, folgt, daß es einen Auftrag $r_j \in \sigma$ mit $\max\{d(a_j, o), d(b_j, o)\} \geq d(s(t_n), o)$ gibt. Mit Hilfe der Dreiecksungleichung ergibt sich

$$\text{OPT}(\sigma) \geq 2 \max\{d(a_j, o), d(b_j, o)\} \geq 2d(s(t_n), o).$$

Benutzt man diese Ungleichung zusammen mit $\text{OPT}(\sigma) \geq t_n$ in (9.1), so folgt, daß im ersten Fall $\text{REPLAN}(\sigma) \leq 5/2 \cdot \text{OPT}(\sigma)$ wie behauptet gilt.

Im zweiten Fall trägt der REPLAN-Server zum Zeitpunkt t_n ein Objekt aus einem Auftrag $r_i = (t_i, a_i, b_i)$. Er benötigt noch $d(s(t_n), b_i)$ Zeit, um diesen

Auftrag fertigzustellen. Danach plant er im Punkt b_i neu. Es gilt also

$$\begin{aligned}
\mathsf{REPLAN}(\sigma) &\leq t_n + d(s(t_n), b_i) + L^*(t_n, b_i, \sigma \setminus \{r_i\}) \\
&\leq t_n + d(s(t_n), b_i) \\
&\quad + d(a_i, o) - d(a_i, b_i) + L^*(t_n, o, \sigma) \quad \text{(Lemma 9.1)} \\
&= t_n + d(s(t_n), b_i) \\
&\quad + d(a_i, o) - d(a_i, s(t_n)) - d(b_i, s(t_n)) + L^*(t_n, o, \sigma) \\
&\leq t_n + d(a_i, o) + L^*(t_n, o, \sigma) \\
&\leq 5/2 \cdot \mathsf{OPT}(\sigma).
\end{aligned}$$

2. Fall: Der Restaufwand ist höchstens $d(a_i, o) + \mathsf{OPT}(\sigma)$

Dabei haben wir neben $\mathsf{OPT}(\sigma) \geq t_n$ die Tatsache benutzt, daß $\mathsf{OPT}(\sigma) \geq d(o, a_i) + d(a_i, b_i) + d(b_i, o) \geq 2d(o, a_i)$ gilt. □

Satz 9.3 *Algorithmus* IGNORE *ist* $5/2$*-kompetitiv für die Zielfunktion* $C^{\max}$.

Beweis: Wir betrachten wieder den Zeipunkt t_n, zu dem der letzte Auftrag der Folge $\sigma = r_1, \ldots, r_n$ bekannt wird. Wenn der IGNORE-Server zu diesem Zeitpunkt im Ursprung ist und alle Aufträge mit Zeitstempel kleiner als t_n bearbeitet hat, dann ist seine Fertigstellungszeit höchstens $t_n + \mathsf{OPT}(\sigma) \leq 2 \cdot \mathsf{OPT}(\sigma)$.

Es verbleibt also der Fall, daß der IGNORE-Server zum Zeitpunkt t_n einen Transportplan S für eine Teilfolge σ' der Aufträge in σ bearbeitet. Sei t_S die Zeit, zu der IGNORE mit der Bearbeitung des Plans S beginnt. Dieser Plan hat Länge $L^*(t_S, o, \sigma')$. Sei $\sigma_{\geq t_S}$ die Menge der Aufträge, die IGNORE dann im letzten Transportplan bearbeitet. Es gilt

$$\begin{aligned}
\mathsf{IGNORE}(\sigma) &= t_S + L^*(t_S, o, \sigma') + L^*(t_n, o, \sigma_{\geq t_S}) \\
&\leq t_S + \mathsf{OPT}(\sigma) + L^*(t_n, o, \sigma_{\geq t_S}). \qquad (9.2)
\end{aligned}$$

Sei r_j der erste Auftrag in $\sigma_{\geq t_S}$, den der optimale Offline-Algorithmus OPT bearbeitet. Man beachte, daß $t_j \geq t_S$ gilt, da r_j erst *nach* Start des Transportplans S zur Zeit t_S bekannt wurde. Dann folgt

$$\mathsf{OPT}(\sigma) \geq t_j + L^*(t_j, a_j, \sigma_{\geq t_S}) \geq t_j + L^*(t_n, a_j, \sigma_{\geq t_S}), \qquad (9.3)$$

da OPT zum Zeitpunkt t_j noch alle Aufträge aus $\sigma_{\geq t_S}$ zu bearbeiten hat. Aus (9.2) ergibt sich nun

$$\begin{aligned}
\mathsf{IGNORE}(\sigma) &\leq t_S + \mathsf{OPT}(\sigma) + d(o, a_j) + L^*(t_n, a_j, \sigma_{\geq t_S}) \\
&\leq t_j + L^*(t_n, a_j, \sigma_{\geq t_S}) + \mathsf{OPT}(\sigma) + d(o, a_j) \quad (\text{da } t_j \geq t_S) \\
&\leq \mathsf{OPT}(\sigma) + \mathsf{OPT}(\sigma) + d(o, a_j) \quad (\text{nach (9.3)}) \\
&\leq 5/2 \cdot \mathsf{OPT}(\sigma).
\end{aligned}$$

Dies war zu zeigen. □

Im folgenden geben wir nun einen Algorithmus an, der eine bessere Kompetitivität als $5/2$ erreicht. Die Idee des Algorithms SMARTSTART ist es, im wesentlichen IGNORE zu emulieren, dabei aber sicherzustellen, daß jeder Teil-Transportplan „rechtzeitig“ beendet wird (was dies bedeutet, wird unten noch

genau erklärt): Wenn ein Teilplan zu lange dauert, dann legt sich der Algorithmus eine Weile schlafen und betrachtet die Situation beim Aufwachen erst wieder.

Der Algorithmus benutzt einen festen »Wartezeit-Skalierungsparameter« $\theta > 1$. Von Zeit zu Zeit befragt der Algorithmus sein »Arbeiten-oder-Schlafen« Orakel: Diese Routine berechnet einen kürzesten Transportplan S für alle noch unbearbeiteten Aufträge, der im Ursprung o startet und endet. Wenn dieser Transportplan nicht später als zur Zeit θt beendet werden kann (wobei t die aktuelle Zeit ist), dann liefert das Orakel $(S, \text{Arbeiten})$, ansonsten liefert es $(S, \text{Schlafen})$.

Der SMARTSTART Server kann sich in drei Zuständen befinden:

Stilliegend In diesem Fall hat der Server alle bekannten Aufträge bearbeitet und wartet untätig im Ursprung auf neue Aufträge.

Schlafend Der Algorithmus kennt noch zu bearbeitende Aufträge, weiß aber auch, daß sie »lange Bearbeitungszeit« benötigen. Er schläft im Ursprung bis zu seiner Weckzeit.

Arbeitend In diesem Zustand bearbeitet der Algorithmus gerade einen Transportplan.

Wir formalisieren die Arbeitsweise des Algorithmus, indem wir spezifizieren, wie er sich in den einzelnen Zuständen verhält.

Algorithmus SMARTSTART

- Wenn der Algorithmus zum Zeitpunkt T stilliegt und ein neuer Auftrag bekannt wird, so konsultiert er sein „Arbeiten-oder-Schlafen“ Orakel. Wenn das Ergebnis $(S, \text{Arbeiten})$ ist, dann wechselt er in den Arbeitszustand und bearbeitet S. Ansonsten wechselt er in den Schlafzustand und stellt seinen Wecker auf Zeit t', wobei $t' \geq T$ die früheste Zeit ist, so daß $t' + l(S) \leq \theta t'$. Hier ist $l(S)$ die Bearbeitungsdauer des vom Orakel gelieferten Transportplans. Es ist also $t' = \min\{\, t \geq T : t + l(S) \leq \theta t \,\}$.
- Im Schlafzustand tut der Algorithmus einfach nichts bis zum Wecken. Wenn er zur Zeit T geweckt wird, dann konsultiert er wieder das „Arbeiten-oder-Schlafen“ Orakel. Wenn das Ergebnis $(S, \text{Arbeiten})$ ist, wechselt er in den Arbeitszustand, ansonsten schläft der Algorithmus weiter, wobei die neue Weckzeit dann $\min\{\, t \geq T : t + l(S) \leq \theta t \,\}$ ist.
- Während des Arbeitens, d.h. während der Server einen Transportplan bearbeitet, werden alle neuen Anfragen ignoriert. Sobald der Transportplan beendet (und der Server somit wieder im Ursprung) ist, befragt er wieder das „Arbeiten-oder-Schlafen“ Orakel. Abhängig von dessen Antwort wird dann der nächste Zustand gewählt.

Satz 9.4 *Der Algorithmus* SMARTSTART *ist* $\max\left\{\theta, 1 + \frac{1}{\theta-1}, \frac{\theta}{2} + 1\right\}$*-kompetitiv für die Zielfunktion* $C^{\max}$*. Insbesondere ist der Algorithmus bei Wahl von* $\theta = 2$ *dann* 2*-kompetitiv.*

Beweis: Sei $\sigma = r_1, \ldots, r_n$ eine beliebige Folge von Aufträgen und $\sigma_{=t_n}$ die Menge der Aufträge mit Releasezeit t_n. Wir unterscheiden drei Fälle, je nach dem Zustand des Algorithmus zum Zeitpunkt t_n.

1. Fall: Der Server liegt still.

In diesem Fall konsultiert der Algorithmus zum Zeitpunkt t_n sofort sein „Arbeiten-oder-Schlafen“ Orakel. Sei S der vom Orakel berechnete Transportplan. Der SMARTSTART-Server startet mit dem Plan S zum Zeitpunkt $t' = \min\{\, t \geq t_n : t + l(S) \leq \theta t \,\}$.

Wenn $t' = t_n$, dann ist der Algorithmus nach Konstruktion spätestens zum Zeitpunkt $\theta t_n \leq \theta\, \mathsf{OPT}(\sigma)$ fertig. Ist $t' > t_n$, so folgt $t' + l(S) = \theta t'$. Es gilt dann $\mathsf{OPT}(\sigma) \geq l(S) = (\theta - 1)t'$. Folglich ist

$$\mathsf{SMARTSTART}(\sigma) = t' + l(S) = \theta t' \leq \theta \cdot \frac{\mathsf{OPT}(\sigma)}{\theta - 1} = \left(1 + \frac{1}{\theta - 1}\right) \mathsf{OPT}(\sigma).$$

2. Fall: Der Server schläft.

Die Argumente sind hier im wesentlichen die gleichen wie im ersten Fall. Sei σ' die Teilfolge der von SMARTSTART zum Zeitpunkt t_n noch nicht bearbeiteten Aufträge (inklusive der Aufträge in $\sigma_{=t_n}$). Sei S ein kürzester Transportplan für die Aufträge in σ'. Der SMARTSTART-Server startet mit der Bearbeitung von S zum Zeitpunkt $t' = \min\{\, t \geq t_n : t + l(S) \leq \theta t \,\}$. Wenn $t' = t_n$, so gilt $\mathsf{SMARTSTART}(\sigma) \leq \theta\, \mathsf{OPT}(\sigma)$. Ansonsten gilt $\mathsf{OPT}(\sigma) \geq \mathsf{OPT}(\sigma') \geq l(S) = (\theta - 1)t'$. Wie oben sind dann die Kosten von SMARTSTART durch $\left(1 + \frac{1}{\theta-1}\right) \mathsf{OPT}(\sigma)$ beschränkt.

3. Fall: Der Server arbeitet.

Wenn der Server nach Beendigung des aktuellen Transportplans in den Schlafzustand übergeht, dann zeigen die Argumente von oben, daß $\mathsf{SMARTSTART}(\sigma) \leq \left(1 + \frac{1}{\theta-1}\right) \mathsf{OPT}(\sigma)$ gilt.

Wir nehmen daher an, daß der SMARTSTART-Server seinen letzten Transportplan S' unmittelbar nach Beendigung des aktuellen Transportplans S startet. Sei t_S die Zeit, zu dem SMARTSTART mit dem Transportplan S begonnen hat. Dann gilt

$$\mathsf{SMARTSTART}(\sigma) = t_S + l(S) + l(S'). \tag{9.4}$$

Nach Konstruktion des Algorithmus gilt auch

$$t_S + l(S) \leq \theta t_S. \tag{9.5}$$

Seien $\sigma_{\geq t_S}$ die Anfragen, die bekanntgegeben wurden, nachdem SMARTSTART mit dem Transportplan S zum Zeitpunkt t_S begonnen hatte. Man beachte, daß $\sigma_{\geq t_S}$ genau die Aufträge sind, die SMARTSTART im letzten Plan S' bearbeitet.

Sei $r_f \in \sigma_{\geq t_S}$ der erste Auftrag aus $\sigma_{\geq t_S}$, den OPT bearbeitet. Wie im Beweis von Satz 9.3 folgt nun

$$\mathsf{OPT}(\sigma) \geq t_S + L^*(t_n, a_f, \sigma_{\geq t_S}). \tag{9.6}$$

Da außerdem der Transportplan der Länge $L^*(t_n, a_f, \sigma_{\geq t_S})$ in a_f startet und in o endet, folgt aus der Dreiecksungleichung, daß $L^*(t_n, a_f, \sigma_{\geq t_S}) \geq d(o, a_f)$. Somit ergibt sich aus (9.6)

$$\mathsf{OPT}(\sigma) \geq t_S + d(o, a_f). \tag{9.7}$$

Andererseits gilt

$$l(S') \leq d(o, a_f) + L^*(t_n, a_f, \sigma_{\geq t_S}) \leq \mathsf{OPT}(\sigma) - t_S + d(o, a_f). \tag{9.8}$$

Wenn man (9.8) in (9.4) verwendet, so erhält man

$$\begin{aligned}
\mathsf{SMARTSTART}(\sigma) &\leq \theta t_S + l(S') && \text{(nach (9.5))}\\
&\leq (\theta - 1)t_S + d(o, a_f) + \mathsf{OPT}(\sigma) && \text{(nach (9.8))}\\
&\leq \theta\,\mathsf{OPT}(\sigma) + (2-\theta)d(o, a_f) && \text{(nach (9.7))}\\
&\leq \begin{cases} \theta\,\mathsf{OPT}(\sigma) + (2-\theta)\frac{\mathsf{OPT}(\sigma)}{2} & \text{, falls } \theta \leq 2\\ \theta\,\mathsf{OPT}(\sigma) & \text{, falls } \theta > 2 \end{cases}\\
&\leq \max\left\{\frac{\theta}{2} + 1, \theta\right\} \mathsf{OPT}(\sigma).
\end{aligned}$$

Dies beendet den Beweis. □

Wie in [5] gezeigt wird, kann man das Ergebnis des letzten Satzes noch etwas verallgemeinern. Man betrachtet dabei den Fall, daß SMARTSTART für sein »Arbeiten-oder-Schlafen« Orakel einen ρ-Approximationsalgorithmus einsetzt: Ein ρ-Approximationsalgorithmus liefert für eine Menge von Aufträgen einen Transportplan, der höchstens ρ-mal so lange wie der kürzeste ist (In der bisherigen Version von SMARTSTART und im Beweis von Satz 9.4 haben wir $\rho = 1$ angenommen).

Satz 9.5 *Für alle $\theta \geq \rho$, $\theta > 1$, ist Algorithmus* SMARTSTART *c_ρ-kompetitiv mit*

$$c_\rho = \max\left\{\theta, \rho\left(1 + \frac{1}{\theta - 1}\right), \frac{\theta}{2} + \rho\right\}.$$

Die beste Wahl von θ ist $\frac{1}{2}\left(1 + \sqrt{1 + 8\rho}\right)$ und ergibt eine Kompetitivität von $\frac{1}{4}\left(4\rho + 1 + \sqrt{1 + 8\rho}\right)$.

Beweis: Siehe [5]. □

Die Bedeutung von Approximationsalgorithmen ist die folgende: Das Offline-Problem, zu einer gegebenen Auftragsmenge einen kürzesten Transportplan zu bestimmen, ist im allgemeinen Fall NP-hart [17]. Da in der Praxis allerdings auch schnelle Antworten vom Algorithmus erwartet werden (es macht keinen Sinn, einen Aufzug stundenlang rechnen zu lassen und dann den gerade errechneten optimalen Plan abzufahren), möchte man gerne effiziente Algorithmen mit polynomialer Laufzeit haben.

Die Offline-Variante von OLDARP ist auch unter dem Namen *Stacker Crane Problem* bekannt. In [17] wird ein polynomialer $9/5$-Approximationsalgorithmus für den allgemeinen Fall vorgestellt. Für Bäume

gibt es in [16] einen $5/4$-Approximationsalgorithmus. Wenn der metrische Raum ein Pfad ist, kann das Offline-Problem in Polynomialzeit gelöst werden [6, 22].

Für den Spezialfall des *Online Traveling Salesperson Problem* (OLTSP) ist der bekannte Algorithmus von Christofides [11] ein $3/2$-Approximationsalgorithmus. Für $\rho = 3/2$ wird die beste Kompetitivität von SMARTSTART für $\theta = \frac{1+\sqrt{13}}{2}$ erreicht. Sie beträgt dann $\frac{7+\sqrt{13}}{4} \approx 2.6514$.

9.2.1 Verallgemeinerung auf mehrere Server

Das Problem OLDARP kann in naheliegender Weise dahingehend verallgemeinert werden, daß mehr als ein Server die Aufträge bearbeiten: Beim Problem k-OLDARP hat man k-Server mit Kapazitäten $C_1, \ldots, C_k \in \mathbb{N}$. Die Kapazität C_j bedeutet, daß der Server j gleichzeitig bis zu C_j Gegenstände tragen kann. Die Zielfunktion $C^{\max}$ („Fertigstellungszeit") mißt den Zeitpunkt, zu dem der letzte der k-Server wieder in den Ursprung zurückgekehrt ist, nachdem alle Aufträge bearbeitet worden sind.

Die Algorithmen IGNORE und SMARTSTART werden für k-OLDARP dergestalt modifiziert, daß sie immer einen Transportplan berechnen, in dem die Länge des längsten Plans für die k-Server minimiert wird. Man kann nun folgende Resultate zeigen:

Satz 9.6 *Für die modifizierten Algorithmen* IGNORE *und* SMARTSTART *gilt:*

1. IGNORE *ist $5/2$-kompetitiv für k-OLDARP bezüglich der Zielfunktion $C^{\max}$.*

2. *Mit $\theta = 2$ ist* SMARTSTART *2-kompetitiv bezüglich der Zielfunktion $C^{\max}$.*

Beweis: Siehe [5]. □

9.3 Optimierung der Fluß- und Wartezeiten

Referenzwerke: [21, 20]

In diesem Abschnitt zeigen wir, daß es für die Optimierung der Fluß- und Wartezeiten keine kompetitiven Algorithmen geben kann. Sei $\sigma = r_1, \ldots, r_n$ eine beliebige Auftragsfolge und $N := \sum_{i=1}^{n} d(a_i, b_i)$. Man beachte, daß die Minimierung der durchschnittlichen Fluß- bzw. Wartezeit äquivalent zur Minimierung der Summe der Fluß- bzw. Wartezeiten ist. Mit $F_{\text{ALG}}(\sigma)$ und $W_{\text{ALG}}(\sigma)$ bezeichnen wir die Summe der Fluß- bzw. Wartezeiten, wenn die Folge σ durch den Algorithmus ALG bearbeitet wird.

Es gilt dann

$$\frac{W_{\text{ALG}}(\sigma)}{W_{\text{OPT}}(\sigma)} = \frac{F_{\text{ALG}}(\sigma) - N}{F_{\text{OPT}}(\sigma) - N} \geq \frac{F_{\text{ALG}}(\sigma)}{F_{\text{OPT}}(\sigma)}.$$

Somit impliziert ein c-kompetitiver Algorithmus für die Zielfunktion W^{avg} auch einen c-kompetitiven Algorithmus für die Zielfunktion F^{avg}. Im Umkehrschluß folgt, daß sich ein negatives Resultat für die Zielfunktion F^{avg} unmittelbar auf W^{avg} überträgt. Eine analoge Rechnung zeigt, daß sich $F^{\max}$ und $W^{\max}$ ebenso verhalten.

Satz 9.7 (Untere Schranken für die Minimierung der durchschn. Flußzeit) *Für* OLDARP *in* $(\mathbb{R}_{\geq 0}, |.|)$ *mit der Zielfunktion* F^{avg} *gilt:*

1. *Jeder c-kompetitive deterministische Algorithmus hat Kompetitivität $c \geq n-1$, wobei $n := |\sigma|$.*
2. *Jeder c-kompetitive randomisierte Algorithmus hat Kompetitivität $c \in \Omega(\sqrt{n})$ gegen den blinden Gegner, wobei $n := |\sigma|$.*

Beweis: Wir zeigen nur die erste Aussage des Satzes. Einen Beweis des zweiten Teils kann man in [31] finden.

Der Adversary gibt zum Zeitpunkt 0 einen Auftrag $r_1 = (0, 1/2, o)$. Sei t der Zeitpunkt, zu dem der Online-Algorithmus ALG mit der Bearbeitung von r_1 beginnt. Zum Zeitpunkt $t+\varepsilon$ gibt der Adversary $n-1$ neue Aufträge, jeder mit Startpunkt $\varepsilon/2$ und Zielpunkt o.

Nach Bearbeitung von r_1 muß der Online Algorithmus noch die ganzen »kleinen Aufträge« bearbeiten. Wenn er dies unmittelbar und schnellstmöglichst erledigt, so hat der $i$te Auftrag der $n-1$ »kleinen Aufräge« eine Flußzeit von $t+1+i\varepsilon-(t+\varepsilon) = 1+(i-1)\varepsilon$. Die Summe der Flußzeiten der Aufträge für ALG ist also mindestens:

$$\underbrace{t+1}_{\text{Auftrag } r_1} + \sum_{i=1}^{n-1}(1+(i-1)\varepsilon) = t+1+(n-1)+\varepsilon\sum_{i=1}^{n-1}(i-1).$$

Der Offline-Gegner bearbeitet zuerst die $n-1$ »kleinen Aufträge« und dann r_1:

$$\begin{aligned}\text{OPT}(\sigma) &= \underbrace{\varepsilon\sum_{i=1}^{n-1}(i-1)+(n-1)\varepsilon/2}_{(n-1)\text{ kleine Aufträge}} + \underbrace{t+\varepsilon+\varepsilon(n-1)-\varepsilon/2+1}_{\text{Auftrag } r_1} \\ &= \varepsilon\sum_{i=1}^{n-1}(i-1)+(n-1)\frac{3}{2}\varepsilon+t+\varepsilon/2+1.\end{aligned}$$

Für großes n und geeignet kleines $\varepsilon > 0$ folgt also

$$\frac{\text{ALG}(\sigma)}{\text{OPT}(\sigma)} \sim n-1.$$

Dies beendet den Beweis. □

Satz 9.8 (Untere Schranke für die Minimierung der max. Flußzeit) *Für das Problem* OLDARP *in* $(\mathbb{R}_{\geq 0}, |.|)$ *mit der Zielfunktion* $F^{\max}$ *gilt: Für kein $c \geq 0$ gibt es einen c-kompetitiven deterministischen Algorithmus.*

Beweis: Angenommen, ALG sei c-kompetitiv. Der Adversary wartet bis zum Zeitpunkt $t = c$. Zu diesem Zeitpunkt kann der Server von ALG nicht gleichzeitig im Ursprung o und im Punkt c sein. Es gilt für seine Position $s(t)$:

$$\max\{d(s(t), o), d(s(t), c)\} \geq \frac{c}{2}.$$

Sei $x \in \{o, c\}$ der Punkt, welcher das Maximum auf der rechten Seite annimmt. Dann gibt der Adversary zum Zeitpunkt $t = c$ die Anfrage $(c, x, x + \varepsilon)$. Die Kosten von ALG sind dann mindestens $c/2$. Der Adversary hat seinen Server zum Zeitpunkt c bereits in x sitzen. Somit hat der Adversary nur Kosten ε und es gilt für die Sequenz $\sigma = (c, x, x + \varepsilon)$

$$\frac{\mathsf{ALG}(\sigma)}{\mathsf{OPT}(\sigma)} \geq \frac{c}{2\varepsilon}.$$

Da ε beliebig klein gewählt werden kann, folgt, daß ALG nicht c-kompetitiv sein kann. □

Aus den Sätzen 9.7 und 9.8 folgt nun:

Satz 9.9 (Untere Schranken für die Minimierung der Wartezeiten)

1. *Für* OLDARP *in* $(\mathbb{R}_{\geq 0}, |.|)$ *mit der Zielfunktion* W^{avg} *gilt:*
   (a) *Jeder* c*-kompetitive deterministische Algorithmus hat Kompetitivität* $c \geq n - 1$*, wobei* $n := |\sigma|$.
   (b) *Jeder* c*-kompetitive randomisierte Algorithmus hat Kompetitivität* $c \in \Omega(\sqrt{n})$ *gegen den blinden Gegner, wobei* $n := |\sigma|$.
2. *Für* OLDARP *in* $(\mathbb{R}_{\geq 0}, |.|)$ *mit der Zielfunktion* $W^{\max}$ *gilt: Für kein* $c \geq 0$ *gibt es einen* c*-kompetitiven deterministischen Algorithmus.*

9.3.1 Vertretbare Belastungen

Im folgenden fassen wir eine Anfragefolge $\sigma = r_1, \ldots, r_n$ auch als *Menge* von Anfragen $\sigma = \{r_1, \ldots, r_n\}$ auf. Dies ermöglicht es uns, die üblichen Notationen für Funktionen etc. zu verwenden.

Für eine Zielfunktion C für OLDARP bezeichnen wir mit $C_{\mathsf{ALG}}(\sigma)$ die Kosten des Algorithmus ALG bei der Anfragemenge σ unter C. Diese etwas von der bisherigen Notation abweichende Schreibweise ist daher nötig, da wir in den Rechnungen in diesem Abschnitt sowohl die Fertigstellungszeit $C^{\max}$ als auch die Flußzeiten $F^{\max}$ und F^{avg} benötigen.

Wir nehmen in diesem Abschnitt an, daß IGNORE einen ρ-Approximationsalgorithmus benutzt, um die auftretenden Offline-Instanzen zu lösen: Sobald der aktuelle Transportplan beendet und der Server wieder im Ursprung ist, erstellt IGNORE mit Hilfe des Approximationsalgorithmus einen bezüglich der Fertigstellungszeit $C^{\max}$ »fast optimalen« Transportplan für die aufgelaufenen Aufträge.

Man beachte, daß IGNORE iterativ für Teilmengen der Anfragefolge die Fertigstellungszeit optimiert, auch wenn das »globale Gesamtziel« die Minimierung der Flußzeiten ist.

Definition 9.10 *Sei σ eine Anfragemenge für* OLDARP *und $r = (t, a, b) \in \sigma$ eine beliebige Anfrage. Wir setzen:*

$$t(r) := t, \qquad a(r) := a, \qquad b(r) := b.$$

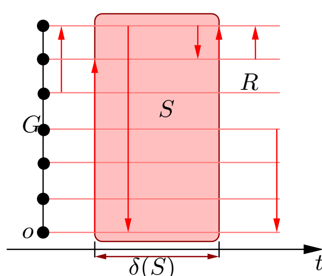
Die Offline-Version *von r ist der Auftrag*

$$r^{offline} := (0, a, b),$$

die Offline-Version von σ ist dann die Menge

$$\sigma^{offline} := \left\{ r^{offline} \; : \; r \in \sigma \right\}.$$

Ein wichtiger Parameter zur Beurteilung der »Belastung« eines Systems ist die Zeitspanne, in der neue Aufträge bekannt werden.



Release-Spanne einer Anfragemenge. Der metrische Raum wird durch einen Pfad-Graphen (vertikal dargestellt) induziert. Die Zeitachse verläuft horizontal.

Definition 9.11 *Sei σ eine endliche Anfragemenge für* OLDARP*. Die* Release-Spanne *$\delta(\sigma)$ von σ ist definiert als*

$$\delta(\sigma) := \max_{r \in \sigma} t(r) - \min_{r \in \sigma} t(r).$$

Definition 9.12 (Lastschranke) *Sei σ eine Anfragemenge für* OLDARP*. Eine schwach monoton wachsende Funktion $f\colon \mathbb{R} \to \mathbb{R}$ heißt eine* Lastschranke *für σ, wenn für alle $\delta \in \mathbb{R}$ und jede endliche Teilmenge $S \subseteq \sigma$ mit $\delta(S) \leq \delta$ gilt:*

$$C_{\text{OPT}}^{\max}(S^{offline}) \leq f(\delta).$$

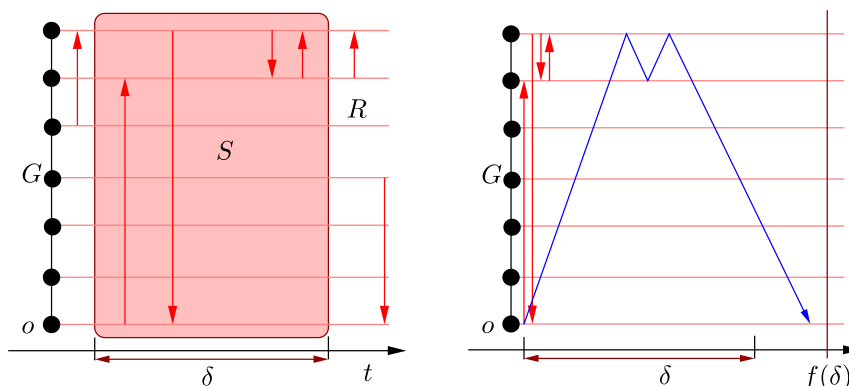
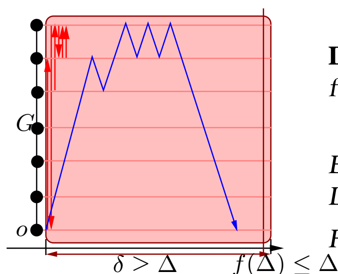


Abbildung 9.3 Lastschranke f für eine Anfragemenge. Wenn $\delta(S) \leq \delta$, dann gilt: $C_{\text{OPT}}^{\max}(S^{offline}) \leq f(\delta)$. Die Menge $S^{offline}$ entsteht aus S durch Verändern der Release-Zeiten aller Anfragen aus S auf 0. Die blaue Kurve zeigt die »Spur« des optimalen Offline Servers auf $S^{offline}$.

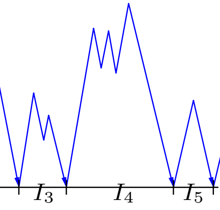


Definition 9.13 *Eine Lastschranke f ist* (Δ,ρ)-vertretbar *für $\Delta, \rho \in \mathbb{R}$, wenn für alle $\delta \geq \Delta$ gilt:*

$$\rho f(\delta) \leq \delta.$$

Eine Anfragemenge σ ist (Δ,ρ)-vertretbar*, wenn es für σ eine (Δ,ρ)-vertretbare Lastschranke gibt.*

Für $\rho = 1$ nennen wir eine Anfragemenge einfach Δ-vertretbar.



des IGNORE-Servers beim Bearbeiten von Aufträgen auf dem Pfadgraphen.

Wir betrachten nun die Arbeitsweise von IGNORE genauer. Der IGNORE Algorithmus organisiert seine Planung in Phasen. Diese Phasen induzieren eine Partitionierung der Zeitachse in Intervalle: Wir nehmen o. B. d. A. an, daß die ersten Anfragen zur Zeit 0 auftreten.[2] Sei somit $\delta_0 := 0$ der Zeitpunkt, zu dem die ersten Anfragen bekannt werden. Diese Anfragen werden vom IGNORE-Server in seinem ersten Transportplan bearbeitet. Für $i > 0$ sei δ_i die Länge des Transportplans von IGNORE für diejenigen Anfragen, die in den letzten δ_{i-1} Zeiteinheiten ignoriert wurden (weil der Server gerade Aufträge bearbeitete). Dann teilt sich die Zeitachse in die Intervalle

$$[\delta_0 = 0, \delta_0], (\delta_0, \delta_1], (\delta_1, \delta_1 + \delta_2], (\delta_1 + \delta_2, \delta_1 + \delta_2 + \delta_3], \ldots$$

Wir bezeichnen die obigen Intervalle mit $I_0, I_1, I_2, \ldots$. Sei ferner R_i die Menge der Aufträge, die in I_i bekanntgegeben werden. Die Vereinigung aller R_i ergibt dann die komplette Anfragefolge.

Am Ende von Intervall I_i löst IGNORE ein Offline-Problem: alle zu bearbeitenden Anfragen sind verfügbar in dem Sinne, daß sie vor der aktuellen Zeit bekanntgegeben wurden. Der IGNORE-Server startet seine Arbeit sofort nach Ende von I_i. Der Transportplan ist dann δ_{i+1} Zeiteinheiten (am Ende von I_{i+1}) später bearbeitet, und der Server steht dann wieder im Ursprung. Es gilt dann:

$$\delta_{i+1} \leq \rho \cdot C^{\max}_{\mathsf{OPT}}(R_i{}^{\mathit{offline}}), \tag{9.9}$$

da der Plan ein ρ-approximativer Offline-Transportplan für die Aufträge aus R_i ist.

Wir zeigen nun das erste Hauptergebnis dieses Abschnittes:

Satz 9.14 *Seien $\Delta > 0$ und $\rho \geq 1$. IGNORE setze einen ρ-Approximationsalgorithmus zum Lösen der auftretenden Offline-Instanzen ein. Für alle Instanzen von OLDARP mit (Δ, ρ)-vertretbaren Anfragefolgen ist die maximale Flußzeit nach oben durch 2Δ beschränkt.*

Beweis: Wir nehmen o. B. d. A. an, daß die ersten Anfragen zur Zeit 0 auftreten. Sei $r = (t, a, b)$ ein beliebiger Auftrag aus R_i, d.h. $t \in I_i$. Nach Konstruktion von IGNORE ist r dann spätestens nach Beendigung des Schedules zum Ende des Intervals I_{i+1}, d.h. $\delta_i + \delta_{i+1}$ Zeiteinheiten später, bearbeitet. Folglich ist die Flußzeit von r bei Bearbeitung durch IGNORE höchstens $\delta_i + \delta_{i+1}$.

Es genügt nun zu zeigen, daß für alle $i > 0$ die Abschätzung $\delta_i \leq \Delta$ erfüllt ist.

Sei dazu $f\colon \mathbb{R} \to \mathbb{R}$ eine (Δ, ρ)-vertretbare Lastschranke. Dann gilt:

$$C^{\max}_{\mathsf{OPT}}(R_i{}^{\mathit{offline}}) \leq f(\delta_i),$$

da $\delta(R_i) \leq \delta_i$. Mit Ungleichung (9.9) folgt nun:

$$\delta_{i+1} \leq \rho \cdot C^{\max}_{\mathsf{OPT}}(R_i{}^{\mathit{offline}}) \leq \rho \cdot f(\delta_i) \leq \max\{\delta_i, \Delta\}. \tag{9.10}$$

Da die Releasespanne der Anfragen, die in I_1 von IGNORE bearbeitet werden, gleich 0 ist (dies sind nach Definition die Anfragen mit Release-Zeit genau 0), gilt: $\delta_1 \leq \max\{0, \Delta\} = \Delta$. Aus (9.10) folgt nun mit Induktion nach i, daß $\delta_i \leq \Delta$ für alle $i > 0$ gilt. □

[2] Diese Annahme ist deshalb o. B. d. A. möglich, weil die Flußzeit eine translationsinvariante Funktion ist.

Korollar 9.15 *Seien $\Delta > 0$ und $\rho \geq 1$.* IGNORE *setze einen ρ-Approximationsalgorithmus zum Lösen der auftretenden Offline-Instanzen ein. Für alle Instanzen von* OLDARP *mit (Δ, ρ)-vertretbaren Anfragefolgen ist die durchschnittliche Flußzeit nach oben durch 2Δ beschränkt.* □

Während wir für IGNORE bei vertretbarer Belastung obere Schranken für die Flußzeiten nachweisen können, verhält sich der REPLAN-Algorithmus deutlich schlechter.

Satz 9.16 *Es gibt eine Instanz von* OLDARP *mit vertretbarer Belastung, so daß die maximale und durchschnittliche Flußzeit von* REPLAN *unbeschränkt sind.*

Beweis: Der Graph G für die Instanz ist ein Pfad aus vier Knoten a, b, c und d. Die Abstände sind $d(a, b) = d(c, d) = \varepsilon$ und $d(b, c) = \ell - 2\varepsilon$. Abbildung 9.4 zeigt den Graphen und die im folgenden spezifizierten Aufträge.

Zum Zeitpunkt 0 wird ein Auftrag von a nach d bekanntgegeben. Ab Zeit $3/2\ell - 2\varepsilon$ erscheinen im Abstand von $\ell - 2\varepsilon$ Zeiteinheiten Paare von neuen Aufträgen von b nach a bzw. von c nach d.

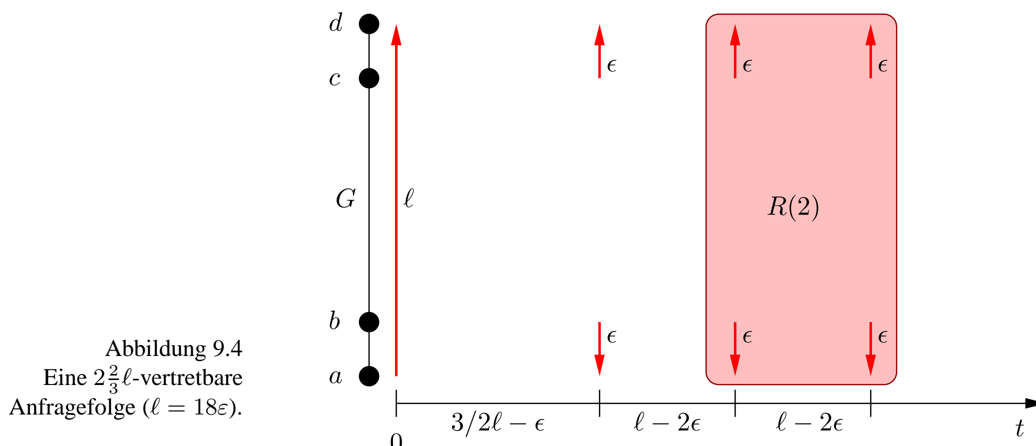


Abbildung 9.4
Eine $2\frac{2}{3}\ell$-vertretbare Anfragefolge ($\ell = 18\varepsilon$).

Wir zeigen nun, daß für $\ell = 18\varepsilon$ die Auftragsmenge $\Delta_0 := 2\frac{2}{3}\ell$-vertretbar ist.

Wir betrachten zunächst Mengen R_k von k aufeinander folgenden Paaren von Aufträgen von b nach a bzw. von c nach d. Dann gilt für die Release-Spanne

$$\delta(R_k) = (k - 1)(\ell - 2\varepsilon).$$

Um die Menge ${R_k}^{\mathit{offline}}$ von Aufträgen zu bearbeiten, benötigt man Offline eine Zeit von

$$C^{\max}_{\mathsf{OPT}}({R_k}^{\mathit{offline}}) = 2\ell + (k - 1) \cdot 4\varepsilon.$$

Um das kleinste Δ zu finden, so daß R_k Δ-vertretbar ist, lösen wir die Ungleichung

$$\delta(R_k) \geq C^{\max}_{\mathsf{OPT}}({R_k}^{\mathit{offline}}) \tag{9.11}$$

nach k auf und erhalten:

$$k \geq 1 + \left\lceil \frac{2\ell}{\ell - 6\varepsilon} \right\rceil = 4.$$

Wenn wir Δ somit durch

$$\Delta := C_{\mathsf{OPT}}^{\max}(R_4{}^{\mathit{offline}}) = 2\frac{2}{3}\ell$$

definieren, so ist R_k dann Δ-vertretbar: Die Funktion $f\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ mit

$$f(\delta) := \begin{cases} \Delta & \text{, falls } \delta < \Delta, \\ \delta & \text{, falls } \delta \geq \Delta, \end{cases}$$

ist per Definition Δ-vertretbar und nach Wahl von Δ auch eine Lastschranke für R_k (Wenn $\delta(R_k) \geq \Delta$, so ist $k \geq 4$, und Ungleichung (9.11) ist erfüllt. Für $k \leq 4$ gilt andererseits: $C_{\mathsf{OPT}}^{\max}(R_k{}^{\mathit{offline}}) \leq C_{\mathsf{OPT}}^{\max}(R_4{}^{\mathit{offline}}) \leq \Delta$).

Es verbleibt zu zeigen, daß der erste Auftrag von a nach d die Δ-Vertretbarkeit nicht zerstört. Dies folgt aber daraus, daß Hinzunahme dieses Auftrags zu einem beliebigen R_k die Offline-Kosten um höchstens $\ell + \varepsilon = 19\varepsilon$ vergrößert, die Release-Spanne andererseits aber mindestens um $3/2\ell - 2\varepsilon = 25\varepsilon$ erhöht.

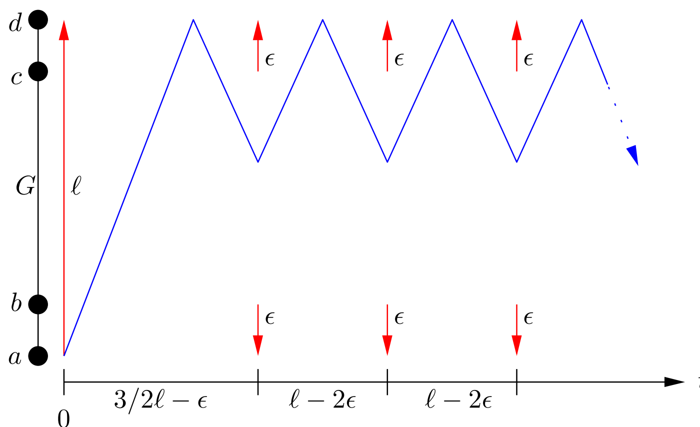


Abbildung 9.5
Die Spur des REPLAN-Servers auf der $2\frac{2}{3}\ell$-vertretbaren Anfragefolge.

Wie verhält sich der REPLAN-Algorithmus auf der vorgegebenen Instanz? Da ein neues Paar von Anfragen genau dann bekanntgegeben wird, wenn der Server noch näher an den oberen Anfragen ist, wird die Bearbeitung aller Aufträge am unteren Ende des Graphen weiter aufgeschoben. Abbildung 9.5 zeigt die »Spur« des REPLAN-Servers. Die maximale Flußzeit wird vom Auftrag $(3/2\ell - \varepsilon, b, a)$ erreicht. Diese wächst mit der Länge der Folge und ist somit unbeschränkt. Somit ist $F^{\max}$ für REPLAN unbeschränkt.

Da außerdem alle Aufträge von b nach a bis zum Schluß aufgeschoben werden, bedeutet dies, daß die Hälfte aller Aufträge unbeschränkte Flußzeiten aufweisen. Folglich ist auch F^{avg} für REPLAN in der vorgestellten Instanz unbeschränkt. □

Abbildung 9.6 zeigt die Spur des IGNORE-Servers für die $2\frac{2}{3}\ell$-vertretbaren Anfragefolge, die für REPLAN katastrophal bezüglich der Flußzeiten ist. Nach einer »ineffizienten« Startphase pendelt sich der IGNORE-Server auf einer stabilen Arbeitsweise ein. Das Beispiel zeigt übrigens auch, daß das Ergebnis von Satz 9.14 scharf für IGNORE ist.

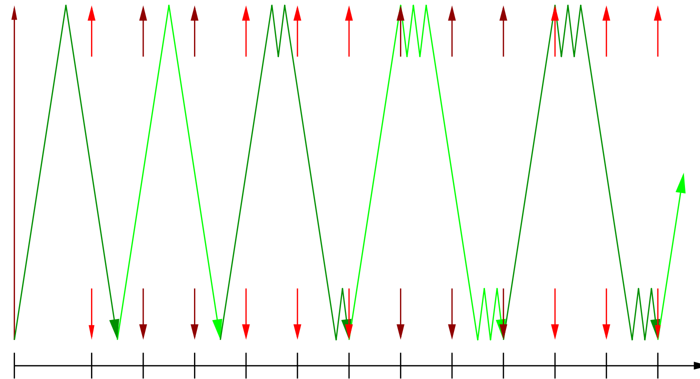


Abbildung 9.6
Die Spur des IGNORE-Servers auf der $2\frac{2}{3}\ell$-vertretbaren Anfragefolge.

Übungsaufgaben

Übung 9.1 (Online-TSP)
Das Online-TSP (OLTSP) ist ein Spezialfall von OLDARP: Hier ist bei jeder Anfrage der Start- und Zielpunkt identisch. Somit können wir eine Anfrage auch einfacher als $r_i = (t_i, x_i)$ schreiben, wobei t_i die Release-Zeit der Anfrage und x_i die neue zu besuchende Stadt ist. Wir betrachten im Folgenden die Zielfunktion $C^{\max}$.

(a) Zeigen Sie, daß REPLAN für das Online-TSP auf dem metrischen Raum $(\mathbb{R}, d)$ mit $d(x,y) = |x - y|$, d.h. auf der reellen Achse mit dem üblichen Euklidischen Abstand, 2-kompetitiv ist.

(b) Analysieren Sie für den Fall des metrischen Raums $(\mathbb{R}_{\geq 0}, d)$ den folgenden Algorithmus:

Algorithmus MRIN (Move-Right-if-Necessary) Wenn eine neue Anfrage $r_i = (t_i, x_i)$ bekannt wird, und diese Anfrage befindet sich rechts vom aktuellen Serverort $s(t_i)$, d.h. $x_i \geq s(t_i)$, dann bewege den Server nach rechts, solange sich noch unbearbeitete Anfragen rechts vom Server befinden. Wenn sich keine unbearbeiteten Anfragen mehr rechts vom Server befinden, dann bewege den Server nach links zum Nullpunkt.

Teil II

Online-Algorithmen in der Praxis

10

Online-Optimierung in flexiblen Fertigungsanlagen

Referenzwerke: [2]

Flexible Fertigungsanlagen (FMS, englisch: *flexible manufacturing system*)

Als *Flexible Fertigungsanlagen (FMS, englisch:* flexible manufacturing system*)* bezeichnet man i. d. R. industrielle Produktionsstätten, in denen verschiedene Maschinen und/oder Roboter über automatische Materialhandling-Systeme wie Förderbänder, führerlose Transportfahrzeuge, Aufzugsysteme u. ä. miteinander verbunden sind und von einer EDV-Anlage gesteuert werden.

Durch die Computersteuerung kann man z. B. die Fertigungsstraße für Meßbecher in einer Plastikspritzerei auf die Produktion von Mülleimern umkonfigurieren: Die Maschinen werden umgerüstet, eine neue Zeittaktung wird eingestellt und aus der Verpackungsstraße wird nun für jeden neuen Mülleimer ein Mülleimerkarton angeliefert, wo vorher ein Meßbecherkarton für zwölf Meßbecher benötigt wurde.

Flexible Fertigungsanlagen sind somit die Voraussetzung dafür, daß der Betrieb auf Änderungen in der Nachfrage schnell reagieren kann. Durch den technischen Fortschritt der letzten zwanzig Jahre haben sich diese Anlagen schneller entwickelt als die Methoden zu ihrer Steuerung. Hier lauert eine Fülle von Online-Optimierungsprobleme, die so komplex sind, daß sie sich nicht immer mit den Methoden aus Teil I analysieren lassen. Umgekehrt kann man viele dieser Probleme wieder auf neue Elementarprobleme schrumpfen, die eine mathematische Untersuchung erlauben.

Wir beschränken uns in diesem Kapitel darauf, ein spezielles FMS und die in ihm auftretenden Probleme genauer zu beschreiben. In den Kapiteln 12, und 13 gehen wir dann auf ausgesuchte Online-Optimierungsprobleme in diesem FMS noch einmal genauer ein.

10.1 Herlitz PBS AG: das Versandlager in Falkensee (Stand 1999)

Ein Palettenfördersystem, wie es im Versandlager der Herlitz PBS AG in Falkensee bei Berlin installiert ist, besteht aus Be- und Entladestationen für LKW,

Palettensortern, Förderbändern (horizontale Fördertechnik), Aufzügen und Fördertürmen (vertikale Fördertechnik), Meß- und Bearbeitungsstationen, sowie einem Hochregallagersystem mit automatischen Bediengeräten. Der Fluß der Paletten vom Wareneingang zur Verarbeitung oder Lagerung sowie zur Auslieferung wird von einem Lagerverwaltungsrechner gesteuert. Während des zeitlichen Ablaufs muß z. B. entschieden werden, auf welchem Weg die Palette sich zu ihrem Zielpunkt bewegen soll, wann eine Palette in welche Lagergasse eingelagert werden soll oder ob eine Palette in eine Staustrecke verzweigen soll. Für die Abwicklung des Palettentransports gibt es Nebenbedingungen, wie z. B. das sogenannte *FIFO-Prinzip*: Paletten müssen in der Reihenfolge ihres Produktionsdatums zur Weiterverarbeitung bzw. Auslieferung.

Die Aufgabe besteht darin, für einen schnellstmöglichen störungsfreien Materialfluß zu sorgen, der alle Nebenbedingungen einhält. So kann z. B. eine Kapazitätsüberschreitung einer Staustrecke zur stundenlangen Blockade eines wichtigen Abschnitts des Fördersystems führen. Auf der anderen Seite kann die Abnahme von Paletten am Wareneingang nicht beliebig lange hinausgezögert werden, nur um Staus zu vermeiden. Außerdem existieren für den Versand am Warenausgang feste Termine, die unbedingt eingehalten werden müssen.

Da sowohl Aufträge als auch Materiallieferungen am Anfang eines Produktionstages noch nicht feststehen, müssen laufend Ad-hoc-Entscheidungen auf der Basis unvollständigen Wissens getroffen werden, die später nicht wieder rückgängig gemacht werden können. Eine Steuerung, die zu einem bestimmten Zeitpunkt nach Datenlage optimal ist, kann nach Eingang eines neuen Auftrags oder einer neuen Materiallieferung schon nachteilige Auswirkungen haben, die auch durch Reoptimierung nicht immer wettzumachen sind (Online-Anforderung). Alle Entscheidungen müssen zusätzlich zu bestimmten Terminen gefällt werden (Echtzeit-Anforderung). Systeme dieser Größe werden nach Aussage unseres Partners Herlitz PBS AG derzeit mittels einfachster Heuristiken gesteuert.

Abbildung 10.1 zeigt ein Schema der Palettenförderlogistik auf der sogenannten *Z-Ebene* des Lagers. Dies ist eine Zwischenebene über dem Erdgeschoß, die verhindert, daß die Anzahl der Kreuzungen im Erdgeschoß zu groß wird. Über diese Ebene werden Paletten vom Wareneingang in das Hochregallager oder in eine der Produktionsebenen gebracht. Umgekehrt werden auszulagernde Paletten über die Z-Ebene zum Warenausgang befördert.

Andere Ebenen stellen die Verbindung zu Produktionshallen her. In einer Etage werden z. B. Glückwunschkarten kommissioniert: Die geschieht ebenfalls mithilfe automatischer Fördertechnik: Kommissioniermobile fahren an den Glückwunschkartenregalen entlang und halten automatisch dort, wo Karten für einen dem Mobil zugewiesenen Auftrag »gepickt« werden müssen.

Da man nach dem heutigen Stand der Technik für ein derartig großes System kein integriertes Optimierungsmodell lösen kann, versucht man bei der Analyse, kritische Module zu isolieren und ihr Verhalten zu studieren.

Probleme sind z. B.:

Lagerplatzzuweisung Eine Palette mit einem bestimmten Artikel (z. B. Aktenordner) oder einem bestimmten Material (z. B. Karton) soll in das Hochregallager eingelagert werden. Weise ihr einen Lagerplatz zu, so daß

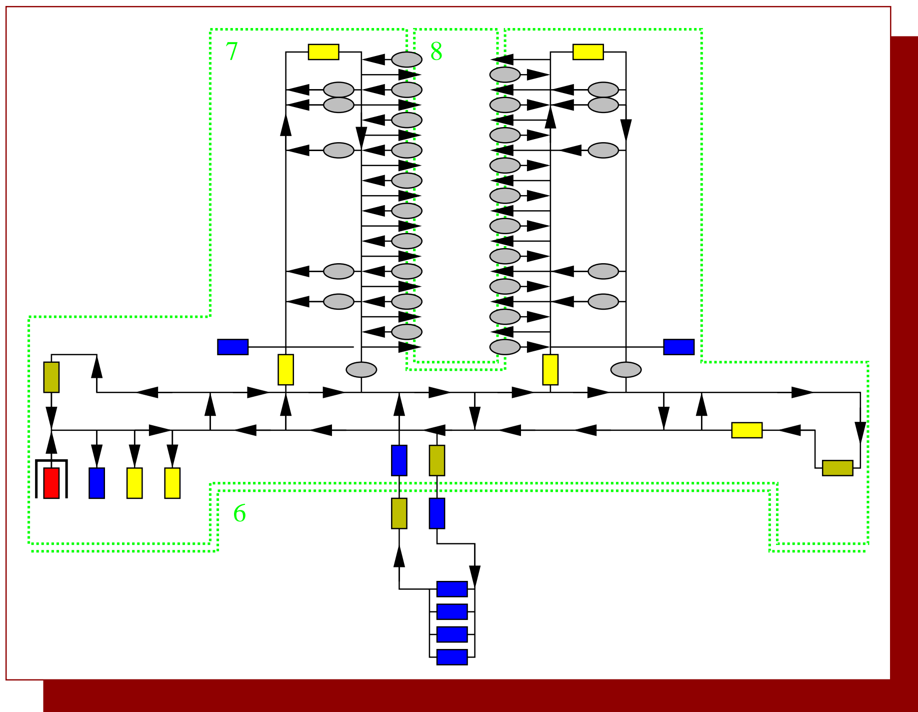


Abbildung 10.1: Skizze der Palettenförderlogistik: Pfeile geben die Förderrichtung auf Rollen- oder Kettenförderbändern an. Gelbe Rechtecke symbolisieren sogenannte *innere I-Punkte*, an denen Paletten über einen Barcodeleser oder ein Benutzerterminal relokalisiert werden. Ovale kennzeichnen Schnittstellen zu Spezialfördergeräten (Aufzug, Regalbediengerät). Grüne Linien deuten logische Systemgrenzen an.

- auch bei Ausfall eines Regalbediengerätes immer eine Palette dieses Typs ausgelagert werden kann (Streuung);
- das Einlagern möglichst schnell geht (kürzeste Wege, Stauvermeidung).

Wegezuweisung Eine Palette soll von A nach B transportiert werden. Finde den kürzesten Weg, der garantiert staufrei sein wird, wenn die Palette ihn benutzt.

Steuerung der Regalbediengeräte Ordne Ein- und Auslageraufträge so an, daß die Leerbewegungen des Regalbediengerätes minimiert werden.

Steuerung der Aufzüge Ordne die Aufträge eines Aufzugs so an, daß

- die mittlere Bedienzeit möglichst kurz ist;
- die maximale Bedienzeit eines Auftrags nicht zu groß wird.

Kommissionieren Ordne Aufträge so den Kommissioniermobilen zu, daß die Gesamtfertigstellungszeit der Aufträge eines Tages minimal wird.

Für alle diese Probleme kann man noch weitere Nebenbedingungen formulieren. In den Kapiteln 12 und 13 werden wir drei Probleme näher beleuchten.

11

Diskrete ereignisbasierte Simulation

oder: Probieren geht über Studieren

Referenzwerke: [2, 29]

Die Online-Optimierungsprobleme einer flexiblen Fertigungsanlage (FMS) sind bei weitem komplizierter als die in der Theorie behandelbaren Elementarprobleme. Daher ist man dazu gezwungen, zur Entscheidungsfindung auch Methoden zu benutzen, die nicht im strengen Sinne "wissenschaftlich" sind. *Simulation*, also das geplante "Ausprobieren" von Algorithmen in einem Modell des FMS auf dem Computer zählt dabei zu den häufig eingesetzen Methoden.

11.1 Zielsetzung

> „Ziel der Simulation ist es, quantitative Aussagen über das Verhalten eines Ausschnitts aus der realen Welt zu erhalten."

schreibt Siegert in seinem Lehrbuch [29]. Da häufig Experimente im realen System zu kostspielig, aufwendig oder gar unmöglich sind, möchte man ein Modell der Realität in angemessener Genauigkeit im Computer ablaufen lassen.

Eine Möglichkeit, dies zu realisieren, beschreiben wir im folgenden.

11.2 Begriffe

Wenn von der Simulation eines Systems gesprochen wird, so weiß in der Regel jeder ungefähr, was damit gemeint ist. Um eine Simulation zu konstruieren, muß man sich über einige Begriffe genauere Gedanken zu machen. Was also ist

- ein System,
- ein Modell,
- ein Ereignis,

- eine Simulation?

Wir besprechen im folgenden diese Begriffe nur im Zusammenhang mit diskreter ereignisorientierter Simulation. Andere Simulationsansätze existieren, siehe [2, 29].

11.2.1 Das System

System
Komponenten

Soll ein Aspekt der realen Welt auf dem Computer ausprobiert werden, so muß man sich erst einmal im Klaren darüber werden, welchen Ausschnitt der realen Welt man betrachten möchte. Diesen Ausschnitt nennen wir das (zu untersuchende) *System*. In der Regel besteht das System aus *Komponenten*. Um z. B. die Wartezeit an einer Supermarktkasse zu untersuchen, betrachtet man den Ausschnitt »Kassenbereich« des Supermarkts.

Umwelt

Den Teil der realen Welt, der nicht im System enthalten ist, nennen wir *Umwelt*. Das System interagiert mit seiner Umwelt, indem es für bestimmte Eingaben bestimmte Ausgaben erzeugt.

Rückkopplung

Manchmal hängen die Eingaben der Umwelt von den Ausgaben des Systems ab (*Rückkopplung*); will man dies berücksichtigen, so muß man den Teil der Umwelt, der dafür verantwortlich ist, in das System integrieren. Sind zum Beispiel zu einer bestimmten Zeit die Schlangen an den Supermarktkassen und damit die Bedienzeiten besonders lang (Ausgabe des Systems), so werden um diese Zeit evtl. einige Leute die Zeit mit weiteren Einkäufen überbrücken und somit später und mit mehr Artikeln im Kassenbereich erscheinen (Eingabe). Um diesen Effekt mit zu untersuchen, muß das System auch noch das Einkaufsverhalten der Kunden enthalten. Da häufig Rückkopplungen auftreten, muß man eine Entscheidung treffen, welche davon man vernachlässigen will, um ein überschaubares System zu erhalten.

dynamischen
zeitdiskrete

Ändert sich ein System mit der Zeit, so spricht man von einem *dynamischen* System. Systeme, die sich nur zu bestimmten Zeitpunkten ändern, nennt man *zeitdiskrete* dynamische Systeme. Im Supermarktbeispiel handelt es sich um ein zeitdiskretes dynamischens System, da es sich nur zu den Zeitpunkten ändert, zu denen ein Kunde den Kassenbereich betritt, um zu bezahlen, und zu denen ein Kunde den Kassenbereich verläßt, nachdem er bezahlt hat.

11.2.2 Das Modell

Dies ist der Punkt, an dem man das System in die Sprache der Mathematik zu übersetzen versucht. Entscheidungen, die dabei getroffen werden, sind daher nicht Bestandteil der Mathematik und somit auch i. d. R. nicht beweisbar richtig oder falsch.

Komponenten
Komponentendaten
Attribute
Strategien

Zur Modellierung eines Systems muß man festlegen, welche *Komponenten* und welche *Komponentendaten* oder *Attribute* man berücksichtigen will. Wichtige Komponentendaten sind *Strategien*, die bestimmen, wie sich eine Komponente verhält, wenn es mehrere Möglichkeiten für sein Verhalten gibt. Manche Komponentendaten sind zeitabhängig, manche nicht.

Im Supermarktbeispiel könnte man die Komponenten »Kasse« und »Kunde« betrachten. Komponentendaten sind

- geöffnet/geschlossen, Kassierer bzw. Kassiererin schnell/langsam, Länge der Schlange (für die Kasse);
- Eintrittszeitpunkt, Waren im Warenkorb, Kassenauswahlstrategie, Austrittszeitpunkt (für Kunden).

Die Eingaben der Umwelt für die Supermarktkasse sind die Kunden, die bezahlen wollen, Ausgaben sind die Kunden, die bezahlt haben. Aus den Ein- und Ausgabedaten kann man nun interessante Daten ermitteln wie die durchschnittliche/maximale Wartezeit der Kunden oder die durchschnittliche/maximale Länge der Warteschlangen.

Der i. allg. zeitabhängige Vektor aller Komponentendaten wird als *Systemzustand* oder *Zustandsvektor* bezeichnet. Bei einem zeitdiskreten dynamischen System gibt es zwischen zwei Zeitpunkten immer nur endlich viele verschiedene Systemzustände, verbunden durch *Systemübergänge*. Daher lassen sich die Modelle zeitdiskreter Systeme leicht auf dem Computer abbilden.

Systemzustand

Zustandsvektor

Systemübergänge

11.2.3 Ereignisse

Eine Möglichkeit, Systemübergänge in zeitdiskreten dynamischen Systemen zu beschreiben, sind *Ereignisse*. Ereignisse sind Datensätze, die Informationen über

Ereignisse

- eine Systemübergangsfunktion, die jedem Systemzustand einen neuen Systemzustand zuordnet,
- eine Folgeereignisfunktion, die jedem Systemzustand eine Menge von Folgeereignissen zusammen mit dem Zeitpunkt ihres Eintreffens zuordnet.

enthalten.

Für das Supermarktbeispiel können wir das Ereignis »ein Kunde stellt sich an Kasse i an« betrachten, das wie folgt definiert ist:

- Im neuen Systemzustand ist die Warteschlange der Kasse i um Eins verlängert.
- Es gibt kein Folgeereignis.

Das Ereignis »ein Kunde wird an Kasse i bedient« kann wie folgt definiert werden:

- Im neuen Systemzustand ist die Warteschlange der Kasse i um Eins verkürzt.
- Das Folgeereignis ist definiert als »ein Kunde verläßt den Kassenbereich« (konstante Funktion) zur Zeit »aktuelle Zeit plus Anzahl der Artikel im Warenkorb des Kunden mal fünf Sekunden« (zeitunabhängig aber abhängig vom Kunden).

11.2.4 Die Simulation

Simulation

Unter einer *Simulation* eines zeitdiskreten dynamischen Systems verstehen wir das Berechnen der Ausgabedaten des Systems aus den Eingabedaten des Systems durch fortlaufendes Verfolgen der Systemübergänge. *Diskrete ereignisbasierte Simulation* ist die Simulation eines zeitdiskreten dynamischen Systems durch *Abarbeiten von Ereignissen* in der Reihenfolge ihres Eintreffens. Abarbeiten eines Ereignisses bedeutet

Diskrete ereignisbasierte Simulation

Abarbeiten von Ereignissen

- Ändern des Systemzustandes durch die Systemübergangsfunktion des Ereignisses,
- Erzeugen der Nachfolgeereignisse und ihrer Eintreffzeiten.

Iteration

Die Abarbeitung des aktuellen Ereignisses bezeichnen wir als *Iteration* der Simulation.

Die ersten Ereignisse, die abgearbeitet werden, sind solche, die die Eingaben der Umwelt modellieren. Im Supermarktbeispiel könnte man das »Umwelt-Eingabe-Ereignis« mit Namen »ein Kunde kommt in den Kassenbereich« wie folgt definieren:

- Der Systemzustand bleibt gleich.
- Als Folgeereignis wird aus der Kassenauswahlstrategie des Kunden die Kasse i ausgewählt. Dann wird das Folgeereignis »ein Kunde stellt sich an Kasse i an« zur Zeit »eine Sekunde später« (konstante Funktion) erzeugt.

11.3 Das Simulationssystem AMSEL

Mit den Überlegungen des vorigen Abschnitts kann man im Prinzip eine Simulation implementieren. Wir beschreiben im folgenden exemplarisch die Struktur der Simulationsbibliothek AMSEL [1] genauer. Die Einführung eines konkreten Simulationssystems hat den Vorteil, daß wir für einige Beispiele konkrete Modelle vorstellen können. Ansonsten ist die Struktur eines diskreten ereignisbasierten Simulationssystems ähnlich dem folgenden, wobei kommerzielle Systeme den Vorgang des Modellierens mit benutzerfreundlichen Oberflächen besser unterstützen, dafür aber keinen Einblick – und damit auch keinen Eingriff – in ihre interne Funktionsweise erlauben.

11.3.1 Die Ereignisliste

Ereignisliste

Die *Ereignisliste* regelt intern die Abarbeitung der Ereignisse. Die Ereignisliste enthält nach jeder Iteration alle Ereignisse, die schon bekannt aber noch nicht eingetroffen sind, und zwar sortiert nach Eintreffzeiten.

In einer Iteration wird folgendes getan:

- Das nächste Ereignis wird zurückgeben (aktuelles Ereignis);

- Das aktuelle Ereignis wird abgearbeitet.
- Das aktuelle Ereignis wird gelöscht.

11.3.2 Objekte und Module

In AMSEL sind Komponenten noch einmal in zwei Klassen eingeteilt:

- *Objekte* sind bewegliche Komponenten des Systems, z. B. die Kunden im Supermarkt. *Objekte*
- *Module* sind feststehende Komponenten des Systems, z. B. die Kasse im Supermarkt. *Module*

11.3.3 Ereignistypen

Ereignisse lassen sich einteilen in *Streckenereignisse* und *neutrale Ereignisse*. *Streckenereignisse* *neutrale Ereignisse*

Streckenereignisse modellieren den Objektfluß im System. Solche Ereignisse können beschrieben werden durch »Objekt o erreicht Ereignispunkt e«. Zusammen mit dem Streckenereignis wird immer das Objekt in die Ereignisliste eingetragen, das das Ereignis ausgelöst hat. Zu den Nachfolgeereignissen von Streckenereignissen gehört immer ein weiteres Streckenereignis.

Objekte werden durch sogenannte G-Ereignisse generiert (Generierung); dies modelliert das Hereinfließen eines Objektes in das System. Objekte können dabei beliebige Daten erhalten, die z. B. ihren Zielpunkt im System beschreiben.

Objekte können durch Module fließen. Das Herein-/Herausfließen eines Objektes in ein Modul wird durch E-/A-Ereignisse beschrieben (Eintritt/Austritt). Module können eine Kapazität haben, so daß sich nur eine bestimmte Anzahl von Objekten gleichzeitig in ihnen befinden darf. Wird ein E-Ereignis abgearbeitet, so wird geprüft, ob die freie Kapazität des Moduls noch ausreicht, um das neue Objekt aufzunehmen. Wenn nicht, so wird das Objekt in die Warteschlange des E-Ereignisses eingereiht.

Objekte verlassen das System durch ein Q-Ereignis (Quit).

Fließt ein Objekt auf ein Warteereignis, so wird es in eine Warteschlange gestellt und wartet auf ein Ereignis, das die Warteschlange aktiviert. Dies kann zur Synchronisation mehrerer Objektströme verwendet werden.

Neutrale Ereignisse beschreiben alle anderen Vorkommnisse von Relevanz.

11.3.4 Warteschlangen

AMSEL verwaltet Warteschlangen in den Ereignissen. Im Prinzip könnte man die Warteschlangensteuerung durch spezielle Ereignistypen definieren. Einfacher ist es jedoch, die Ereignisdaten etwas zu erweitern:

- Jedes Ereignis verwaltet eine eigene Warteschlange, in der sich Objekte anstellen, wenn für sie das Folgeereignis noch nicht eintreten kann/soll, z. B. wenn bei einem Eintrittsereignis das Modul voll ist.

- Neben der Systemupdatefunktion und der Folgeereignisfunktion enthält jedes Ereignis eine Funktion, die für jeden Systemzustand eine weitere Menge von Ereignissen zurückgibt. Die Warteschlangen dieser Ereignisse werden von AMSEL beim Abarbeiten des aktuellen Ereignisses »erweckt«, d. h. es wird das Objekt an der Spitze der Warteschlange auf das Folgeereignis des Warteschlangenereignisses gesetzt, genauer, das Warteschlangenereignis wird mit dem ersten Objekt in der Warteschlange abgearbeitet.

11.3.5 Systemupdates

AMSEL erlaubt Systemupdates an zwei Stellen: einmal vor der Ermittlung der Nachfolgeereignisse und einmal danach. Jedes Ereignis spezifiziert also zwei Systemupdate-Funktionen.

11.3.6 Strategien

Strategien werden durch die Nachfolgeereignis-Funktion verwirklicht. Im Supermarktbeispiel kann die Kassenauswahlstrategie abgebildet werden, indem man aus den Ereignissen »Kunde stellt sich an Kasse i an«, $i = 1, \ldots,$ Anzahl Kassen, eins auswählt.

11.4 Praktische Probleme bei der Simulation

Man stößt bei Entwurf und Durchführung einer Simulation auf Schwierigkeiten, die wir in diesem Abschnitt etwas genauer betrachten wollen.

11.4.1 Erzeugung der Eingabedaten

Das Ergebnis eines Simulationslaufs hängt häufig so empfindlich von den Eingaben ab, daß die Verwendung von unrealistischen Daten zu völlig verzerrten Ergebnissen führen kann.

Bei der Ermittlung der Eingabedaten sind im wesentlichen zwei Möglichkeiten gegeben:

zufällige Daten

- Erzeugung von Daten nach einer Wahrscheinlichkeitsverteilung (*zufällige Daten*).

  Vorteile:

  – Man kann leicht eine beliebig große Menge Stichproben erzeugen und damit eine aussagefähige Statistik erhalten.
  – Man kann eine Vorauswahl vielversprechender Strategien treffen, die man später mit größerem Aufwand weiter untersuchen möchte.

  Nachteile:

  - Um aussagefähige Resultate zu erhalten, muß man eine Ahnung haben, gemäß welcher Wahrscheinlichkeitsverteilung die Daten in der Realität auftreten. Hier sind aber häufig sogar Experten mit großer Erfahrung ratlos.
  - Selbst wenn man durch aufwendige statistische Untersuchungen eine Verteilung ansetzen kann, ist häufig die Streuung so absurd, daß statistische Aussagen nicht sehr wertvoll sind.

- Datenerfassung durch Messen in einem realen System (*reale Daten*).

  reale Daten

  Vorteile:

  - Simulation kann »geeicht« werden, da man die Ausgaben der Simulation mit den in der Realität beobachteten Daten vergleichen kann.
  - Ein Simulationslauf mit realen Daten führt zu aussagekräftigen Ergebnissen, wenn die Daten in irgendeiner Weise »typisch« für die Realität sind.

  Nachteile:

  - Reale Daten sind nur unter großem Zeitaufwand zu beschaffen.
  - Man hat meistens nur kleine Stichproben zur Verfügung, aus denen man nicht entnehmen kann, ob die Daten typisch für die Realität sind.

12

Hochregallagerbediengeräte

oder: Wo ist das Offline-Problem?

Referenzwerke: [2], [3]

Um Online-Algorithmen mithilfe kompetitiver Analyse beurteilen zu können, muß zuerst ein assoziiertes Offline-Opimierunsproblem definiert werden. Für die Offline-Version eines Online-Optimierungsproblems müßte eigentlich gelten, daß jede zulässige Lösung des Offline-Problems online »abgespielt« werden kann, ohne daß Widersprüche auftreten. Eine solche Formulierung ist aber nicht immer einfach – wenn überhaupt – zu finden.

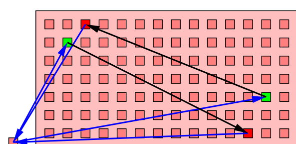
In diesem Kapitel betrachten wir das Problem, die Bewegungen eines Hochregallagerbediengerätes (RBG) zu optimieren. Minimiert werden soll die Gesamt-Leerfahrtzeit des RBGs. Dieses Online-Optimierungsproblem war Kernstück eines Forschungsprojektes des Konrad-Zuse-Zentrums für Informationstechnik Berlin in Kooperation mit einer PC-Fertigungsstätte der Siemens-Nixdorf-Informationssysteme GmbH in Augsburg (SNI).

Sie werden in diesem und den folgenden Kapiteln vielleicht die übliche Struktur eines mathematischen Textes vermissen, der in »Definition«, »Satz«, »Beweis« gegliedert ist. Die hier dargestellten Praxisprobleme sind zu kompliziert, um ihnen mit einer geschliffenen mathmatischen Theorie zu Leibe zu rücken. Es soll hier darum gehen, Probleme und Auswege aufzuzeigen, wenn die Methoden aus Teil I allein nicht mehr greifen.

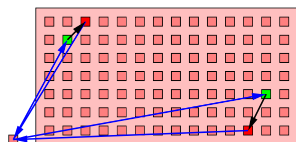
12.1 Problembeschreibung

Ein automatisches Hochregallager besteht aus in Zeilen angeordneten Regalsystemen, die ihrerseits in Zellen eingeteilt sind. In den Gassen zwischen den Regalen befinden sich die Hochregallagerbediengeräte (RBGs), die Paletten ein- und auslagern. In der Regel kann ein RBG genau eine Palette auf einmal transportieren. Die Paletten für die Einlageraufträge sind in einem Materialeingangspuffer gespeichert. Die auszulagernden Aufträge können ungehindert abfließen. Als Einschränkung sei gefordert, daß das RBG immer dann arbeiten muß, wenn Aufträge verfügbar sind.

Die Bewegungen des RBG sollen nun möglichst »effizient« geplant werden. In unserem Fall soll das heißen, daß die Gesamtstrecke an Leerfahrten des RBG



Je zwei Ein- (rot) und Auslageraufträge (grün) mit langen ...



... und mit kurzen Leerfahrten.

möglichst klein sein soll. Intuitiv ist z. B. die Verknüpfung eines Ein- und eines Auslagerauftrages in derselben Zelle des Regals (sofern technisch möglich) wünschenswert.

Bei reibungslosem Betrieb oder geringer Belastung laufen in diesen Systemen typischerweise nicht soviele Aufträge auf, daß sich ein genügend großes Optimierungspotential ergibt. Allerdings zeigen die Erfahrungen bei SNI, daß nach Störungen sich durchaus schon einmal fünfzig Aufträge stauen können.

12.2 Mathematisches Modell

Der Regalbereich mit dem Eingangspuffer kann als endlicher metrischer Raum aufgefaßt werden. Die Punkte in diesem metrischen Raum sind die Regalzellen und der Punkt, an dem das RBG die Ladung aus dem Eingangspuffer holt. Die Distanzen sind die tatsächlichen Distanzen im physikalischen Raum. Ein Auftrag ist gegeben durch eine Freigabezeit, den Startpunkt und den Zielpunkt für die zu transportierende Palette.

Eine Leerfahrt entsteht immer dann, wenn der Zielpunkt eines Auftrags nicht der Startpunkt des als nächstes ausgeführten Auftrags ist. Die Leerfahrten sind also Fahrten »zwischen den Aufträgen«, wobei der Abstand zwischen Aufträgen nicht symmetrisch ist.

Diese Beobachtung legt das folgende Modell nahe: Wir betrachten einen vollständigen gerichteten Graphen $D = (V, A)$ mit Knotenmenge V gleich der Auftragsmenge. Das Bogengewicht des Bogens (i, j) von Auftrag i nach Auftrag j ist gleich der Fahrtstrecke vom Zielpunkt des Auftrags i zum Startpunkt des Auftrags j. Gesucht ist ein gerichteter Weg H (ein sogenannter *Hamilton-Weg*), der alle Knoten aus V genau einmal besucht und dessen Gesamtgewicht minimal unter allen solchen Wegen ist. Wir nennen einen Digraphen, der in dieser Weise aus Aufträgen entsteht einen *Auftragsdigraphen*.

Hamilton-Weg

Auftragsdigraphen

Fügen wir noch einen künstlichen Start- und Stopauftrag hinzu, der mit allen Knoten mit Bögen vom Gewicht Null verbunden ist, so ist ein kürzester *Hamilton-Kreis* in D zu finden. Mit anderen Worten, unser Problem ist äquivalent zu einem *asymmetrischen Travelling-Salesman-Problem (*ATSP*)*.

Hamilton-Kreis

asymmetrischen Travelling-Salesman-Problem (ATSP)

Da wir es mit einem Online-Problem zu tun haben – nennen wir es OLATSP– , sind nicht alle Knoten von Beginn an bekannt, für den Online-Algorithmus erscheinen sie erst mit der Freigabe der zugehörigen Aufträge; erst dann kann er sie in seine Planung einbeziehen.

12.3 Online-Heuristiken

Warum nennen wir die in diesem Abschnitt vorgestellten Verfahren »Heuristiken« und nicht »Algorithmen«? Das liegt daran, daß wir bislang keine mathematische Analyse der Qualität dieser Methoden haben, die für *alle* Instanzen von OLATSP bewiesenermaßen gültig wären. Insbesondere gibt es für keinen der Algorithmen obere Schranken für die Kompetitivität (Übung). In Abschnitt 12.4 werden wir ein Verfahren vorstellen, daß bei der Beurteilung eines Algorithmus im Nachhinein angewendet werden kann.

Hier sind zunächst einige typische Online-Heuristiken (mehr finden sich wieder in [2]):

FIFO Wer zuerst kommt, mahlt zuerst.

PRIORITY Aufträgen wird eine natürliche Zahl, die *Priorität*, zugeordnet. Es wird nach dem FIFO-Prinzip immer ein Auftrag aus der niedrigsten Prioritätsklasse abgearbeitet.

Ein solches Verfahren war bei SNI im Einsatz.

RANDOM Der als nächstes zu bearbeitende Auftrag wird zufällig aus der Menge aller verfügbaren Aufträge gewählt.

GREEDY In jedem Schritt wird der nächste Auftrag in unserem Auftrags-Digraphen D bearbeitet.

BESTINS Jeder neue Auftrag wird so kostengünstig wie möglich in den Fahrplan eingefügt.

LOCAL-OPTIMAL Bei jedem neu erzeugten Auftrag wird eine optimale ATSP-Tour im Auftragsdigraphen D berechnet und solange abgefahren, bis abermals ein neuer Auftrag generiert wird.

12.4 A-posteriori-Analyse

In einer der Übungsaufgaben zeigen Sie, daß eine kompetitive Analyse des OLATSP für alle deterministischen Online-Algorithmen ein Desaster ist. Wir möchten aber dennoch für die Algorithmen, die wir einsetzen, eine Gütegarantie. Die liefert die sogenannte *A-posteriori-Analyse*.

A-posteriori-Analyse

Ein wesentliches Leistungsmerkmal *polyedrischer Optimierungsmodelle* ist, daß sie häufig nicht-triviale Schranken für den Abstand der Kosten einer zulässigen Lösung zu den Kosten einer optimalen Lösung für eine *spezielle Instanz* des Problems liefern. Das Gleiche tun wir nun hier für die Analyse unserer Online-Heuristiken.

Referenzen

Wir sammeln die Aufträge, die unsere Online-Heuristiken bearbeitet haben in der Auftragsmenge σ, und merken uns die Kosten, die die Heuristiken dabei verursacht haben. Danach lösen wir das assoziierte Offline-Problem und messen den *Abstand* $\textsc{Gap}(\sigma)$ nach der Formel

$$\textsc{Gap}(\sigma) = \frac{\mathsf{H}(\sigma) - \mathsf{OPT}(\sigma)}{\mathsf{H}(\sigma)},$$

wobei $\mathsf{H}(\sigma)$ die Kosten der Heuristik H auf σ sind.

Wie sieht nun das Offline-Optimum für die Aufträge eines Tages aus? Wie lautet genau *das* assoziierte Offline-Problem? Wenn wir einfach die Zeiten, zu denen die Aufträge generiert werden, ignorieren, haben wir es mit einem ATSP zu tun. Es kann allerdings passieren, daß i. allg. eine Optimallösung des ATSP im Nachhinein online nicht hätte benutzt werden können, da ein Auftrag in der Offline-Lösung zu einer Zeit geplant werden kann, zu der er online noch nicht verfügbar ist.

Daher ist die Anwendung des Zeitstempelmodells aus Abschnitt 9.1 zwingend notwendig. Hier darf ein Auftrag erst dann abgearbeitet werden, wenn er freigegeben ist. Um das Offline-Optimum zu erhalten, müssen wir also ein ATSP mit Freigabezeiten lösen, nennen wir es ATSPRT. Dies ist nicht ganz einfach und soll hier nicht weiter behandelt werden; Näheres findet sich in [2].

Trotzdem kann es in der Praxis passieren, daß auch eine solche Optimallösung nicht online hätte Verwendung finden können (Übung). In diesem Falle erhalten wir aber wenigstens eine *untere Schranke* für die Kosten $\mathsf{OPT}(\sigma)$ der optimalen Offline-Lösung $\mathsf{OPT}[\sigma]$ eines *nicht explizit bekannten* Offline-Problems, welche auch online hätte benutzt werden können.

Es gilt somit für jede zeitgestempelte Auftragsmenge σ und jede Online-Heuristik H die *A-posteriori-Ungleichung*

A-posteriori-Ungleichung

$$\mathsf{OPT}_{\mathrm{ATSP}}(\sigma) \leq \mathsf{OPT}_{\mathrm{ATSPRT}}(\sigma) \leq \mathsf{OPT}(\sigma) \leq \mathsf{H}(\sigma). \quad (12.1)$$

Diese Art der Beurteilung von Online-Heuristiken ist natürlich besonders interessant, wenn man mithilfe eines Simulationsprogramms diese Berechnungen für viele Datensätze am Computer durchführen kann. In dem SNI-Projekt wurde dies durchgeführt. Es hat sich gezeigt, daß im Falle auftretender Störungen sich die Heuristik LOCAL-OPTIMAL im Durchschnitt am besten verhalten hat. Die vorher bei SNI eingesetzte Heuristik PRIORITY schnitt dagegen nur etwa so gut ab wie RANDOM.

12.5 Fazit

Damit das RBG nicht auf das Optimierungsprogramm warten muß, wenn die Suche nach dem (lokalen) Optimum mal etwas länger dauert, wurde bei SNI ein dreistufiges Verfahren implementiert:

1. Schnelle Heuristik BESTINS liefert im Nu eine Lösung;
2. Eine kompliziertere Heuristik für das ATSP liefert ggf. eine bessere Lösung;
3. Ein *Branch&Bound*-Code löst das ATSP optimal.

Mithilfe eines Simulationsmodells konnte »nachgewiesen« werden, daß bei auftretenden Störungen bis zu 30% der Leerfahrten vermieden werden konnten. Diese Ergebnisse haben SNI dazu bewogen, das Verfahren einzusetzen. Mittlerweile haben sich diese Ergebnisse auch im praktischen Betrieb bestätigt.

Übungsaufgaben

Übung 12.1 (Leerfahrten vs. Fertigstellungszeit)
Beweisen oder widerlegen Sie: Minimierung der Summe der Leerfahrten ist äquivalent zur Minimierung der Fertigstellungszeit im Falle

- alle Freigabezeiten sind null,
- beliebige Freigabezeiten und der Server darf warten, auch wenn Aufträge zur Abarbeitung anstehen,
- der Server darf nicht warten.

Übung 12.2 (Kompetitivität des OLATSP)
Zeigen Sie: Es gibt kein $c < \infty$, für das ein c-kompetitiver Online-Algorithmus für das OLATSP existiert.

Übung 12.3
Finde plausible Szenarien, die zeigen, daß die Ungleichheitszeichen in (12.1) i. allg. nicht durch Gleichheitszeichen ersetzt werden können.

Übung 12.4
Übung: Haben äquivalente Online-Optimierungsprobleme gleiche Kompetitivität? (Bem.: Dieselbe Antwort gilt für Approximationsalgorithmen.)

13

Automatische Glückwunschkartenkommissionierung

oder: Sind alle Algorithmen gleich gut?

Referenzwerke: [24], [3]

13.1 Problembeschreibung

In einer Abteilung des Versandlagers Falkensee der Herlitz PBS AG werden Glückwunschkarten kommisioniert. Die Karten – etwa 4000 Typen – befinden sich in einem Regalsystem bestehend aus vier parallelen Regalen. In den Gängen zwischen den Regalen bewegen sich halbautomatische Kommissionierfahrzeuge, die einem Rundkurs folgen. In der Ladezone wird jedem Fahrzeug durch ein automatisches Scheduling-System eine Menge von maximal neunzehn Aufträgen zugewiesen. Um einen Auftrag zusammenzustellen, holt der auf dem Fahrzeug befindliche »Order-Picker« an den jeweils automatisch angesteuerten »Pick-Positionen« Karten aus dem Regal und legt sie in den für den Auftrag vorgesehenen Karton.

In den Gängen können sich die Fahrzeuge nicht überholen; außerdem dürfen aus Sicherheitsgründen nur maximal zwei Fahrzeuge sich gleichzeitig in einem Gang befinden. Das führte bei dem in der Praxis eingesetzten Scheduling zu signifikanten Verzögerungen in der Bearbeitung, und man war mit der Systemleistung »irgendwie« nicht zufrieden. Aber: Eine Zielfunktion für die Zuordnung der Aufträge auf die Kommissioniermobile, deren Optimierung für einen zufriedenstellenden Betrieb sorgen würde, liegt hier nicht auf der Hand. Eine Möglichkeit, die Effizienz des Systems zu steigern, könnte die Minimierung der Anzahl der Stoppositionen sein, an denen die Kommissionierfahrzeuge halten müssen.

Die Aufträge werden erst im Laufe des Tages bekannt; die Zuweisung kann jedoch i. d. R. aus einem Pool von Aufträgen erfolgen, d. h. das Bekanntwerden des jeweils nächsten Auftrags ist unabhängig von der Antwortzeit einer Zuweisung.

Wir wollen nun hieraus ein mathematisches Online-Optimierungsproblem formulieren. Wir probieren es zunächst mit dem klassischen Sequenzmodell, d. h.

der jeweils nächste Auftrag wird erst bekannt, wenn der vorherige zugewiesen worden ist.

Übung

13.2 Mathematisches Modell

Commissioning Vehicle Problem

Das *Commissioning Vehicle Problem* CVP ist wie folgt gegeben: Eine Instanz des CVP besteht aus einer Menge $L = \{1, \ldots, N\}$ von *Stoppositionen*, einer nicht-leeren endlichen Menge *Mobile* $\{v_1, \ldots, v_q\}$ mit Kapazität jeweils C und einer Sequenz von σ von Aufträgen, wobei ein *Auftrag* r eine Teilmenge $\{l_{m_1}, \ldots, l_{m_r}$ von Stoppositionen aus L ist. Gesucht ist eine Zuweisung von Aufträgen auf Mobile *in der Reihenfolge der Auftragssequenz* σ, für die die folgende Regel gilt: Wird einem Mobil der C-te Auftrag zugewiesen (das Mobil wird *geschlossen*), so wird es gegen ein Neues, leeres Mobil ausgetauscht.

geschlossen

Beim *Online Commissioning Vehicle Problem* OLCVP wird dem Online-Algorithmus Auftrag r_i aus σ erst dann bekannt, wenn er r_{i_1} zugewiesen hat.

Beachte: Aufträge können nicht im Vorhinein auf ein zu erwartendes leeres Mobil gepackt werden, solange dieses nicht tatsächlich ein volles Mobil ersetzt hat (Übung).

Online Commissioning Vehicle Problem

13.3 Kompetitive Analyse

Wir starten mit einer trivialen oberen Schranke. Die bestmögliche Organisation der Stoppositionen ist gegeben, wenn jeweils C Aufträge, die die gleichen Stoppositionen induzieren, einem Mobil zugewiesen werden; die schlechtestmögliche ist gegeben, wenn auf jedes Mobil Aufträge gelegt werden, deren Stoppositionen disjunkt sind. Dann ist die Anzahl der Stops genau so groß, wie wenn man C-mal so viele Mobile mit je C Kopien von einem Auftrag belegt hätte. Beide Extremsituationen unterscheiden sich also nur um den Faktor C. Mit anderen Worten:

Beobachtung 13.1 *Jeder korrekte Algorithmus für das* OLCVP *ist* C-*kompetitiv.*

Erstaunlich ist, daß die folgende schwach wirkende untere Schranke schon recht kompliziert zu beweisen ist.

Satz 13.2 *Falls* $C, q > 1$ *ist, gibt es keinen* c*-kompetitiven Online-Algorithmus für das* OLCVP *mit* $c < 2$.

Beweis: Wir konstruieren eine Folge von Aufträgen aus einem Pool von disjunkten Auftragstypen, so daß jeder Online-Algorithmus auf fast alle Fahrzeuge Aufträge von mindestens zwei verschiedenen Typen laden muß, während der Offline-Gegenspieler jedem Fahrzeug nur Aufträge eines Typs zuweisen kann.

Sei A ein beliebiger Online-Algorithmus für OLCVP.

Schritt 1: Der Auftragstyp T_0 erfordert nur den Stop an Position N. Die Typen $T_1, \ldots, T_{q+1}$ seien definiert durch Teilen der Menge der möglichen Stoppositionen ohne N in $q + 1$ gleichgroße Mengen (wir ignorieren einen eventuellen

Rest). In Formeln:

$$T_0 := \{N\};$$
$$T_i := \left\{(i-1) \cdot \lfloor \tfrac{N-1}{q+1} \rfloor + 1, \ldots, i \cdot \lfloor \tfrac{N-1}{q+1} \rfloor \right\}, \quad i = 1, 2, \ldots, q+1.$$

Sei z die Anzahl der Stops der Typen $T_1, \ldots, T_{q+1}$, also $z := \lfloor \frac{N-1}{q+1} \rfloor$.

Schritt 2: Wir konstruieren aus den obigen Auftragstypen eine unendliche Familie $\Sigma := \{\sigma_1, \sigma_2, \ldots\}$ von Auftragsfolgen. Hier besteht die Auftragsfolge σ_ν aus $q(C-1) + \nu C$ Aufträgen. Alle Auftragssequenzen werden nach der gleichen Vorschrift erzeugt, unterscheiden sich also nur im Abbruchkriterium.

Schritt 3: Der Offline-Gegner eröffnet mit je $C - 1$ Aufträgen des Typs $T_1, T_2, \ldots, T_q$.

Schritt 4: Wir unterscheiden von nun an prinzipiell verschiedene Konfigurationen auf den aktuell verfügbaren, d. h. noch nicht vollen, Mobilen, die sich durch das Verhalten des Online-Algorithmus ergeben. Der Offline-Gegner stellt einen Auftrag aus, je nach dem in welcher Konfiguration sich der Online-Spieler befindet. Wir benutzen für jeden Auftragstyp T_i die Größe $n(i)$, die angibt, wieviel Aufträge des Typs T_i bereits erzeugt worden sind.

Wir nennen ein nicht-leeres Mobil T_j-homogen, $j = 1, \ldots, q+1$, wenn sich nur Aufträge vom Typ T_j auf ihm befinden. Ein Auftragstyp T_j heißt *ärgerlich*, $j = 1, \ldots, q+1$, wenn zum Zeitpunkt seiner Erzeugung kein T_j-homogenes Mobil existiert. Ein Auftragstyp heißt *neu*, wenn keines der verfügbaren Mobile einen Auftrag dieses Typs enthält.

Beobachtung 13.3 *Da nur q Mobile gleichzeitig verfügbar sind, existiert immer mindestens ein ärgerlicher Auftragstyp.*

Der Offline-Gegner prüft der Reihe nach folgende Bedingungen und erzeugt dementsprechend Aufträge:

1. *Situation:* Es sind bereits $q(C-1) + \nu C$ Aufträge erzeugt.

   Aufträge: Der Offline-Gegner stellt keine Auftäge mehr aus. Es wird abgerechnet.

2. *Situation:* Alle Mobile sind mit jeweils $C - 1$ Aufträgen eines Typs T_i beladen, und $n(i) \equiv C - 1 \mod C$ für alle diese i.

   Aufträge: Der Offline-Gegner erzeugt einen Auftrag des Typs T_0.

3. *Situation:* Es existiert ein ärgerlicher Auftragstyp T_j mit $n(j) \equiv C - 1 \mod C$.

   Aufträge: Der Offline-Gegner erzeugt einen Auftrag vom Typ T_j.

4. *Situation:* Es existiert ein ärgerlicher Auftragstyp T_j mit $n(j) \equiv 0 \mod C$.

   Aufträge: Der Offline-Gegner erzeugt $C - 1$ Aufträge vom Typ T_j.

Schritt 5: Wir behaupten nun folgendes:

Behauptung 13.4 *In jeder Situation gilt für jeden Auftragstyp* $T_1, \dots, T_{q+1}$ *entweder* $n(j) \equiv 0 \mod C$ *oder* $n(j) \equiv C - 1 \mod C$.

Behauptung 13.5 *Nach jeder Auftragserzeugung des Offline-Gegners befinden wir uns wieder in einer der oben beschriebenen Situationen.*

Behauptung 13.6 *Wir befinden uns höchstens einmal in Situation 2.*

Behauptung 13.7 *Der Online-Spieler kann höchstens ein Fahrzeug mit* $z + 1$ *Stops schließen. Alle anderen geschlossenen Fahrzeuge müssen mindestens* $2z$ *mal halten.*

Behauptung 13.8 *Der Offline-Gegner muß ein Fahrzeug mit* $z+1$ *Stops schließen, für alle anderen kann er mit* z *Stops auskommen.*

Schritt 6: Wir glauben für einen Moment die Behauptungen und bilanzieren nun die Kosten für σ_ν:

$$\begin{aligned} \mathsf{OPT}(\sigma_\nu) &\leq z(\nu + q) + 1; \\ \mathsf{A}(\sigma_\nu) &\leq 2z(\nu - 1) + z + 1. \end{aligned}$$

Daraus folgt, daß der Quotient beliebig nahe an 2 liegt, sofern nur ν groß genug gewählt wird. Es bleiben also nur noch die Behauptungen 13.4 bis 13.8 zu zeigen.

Schritt 7: Behauptung 13.4 folgt direkt aus der Konstruktion. Wir beweisen nun Behauptung 13.5:

Beweis der Behauptung 13.5 Da nach Beobachtung 13.3 immer mindestens ein ärgerlicher Auftragstyp existiert und nach Behauptung 13.4 für diesen Auftragtypen entweder $n(j) \equiv C - 1 \mod C$ oder $n(j) \equiv 0 \mod C$ gilt, ist stets die Bedingung für Situation 3 oder die Bedingung für Situation 4 erfüllt. □

Beweis der Behauptung 13.6 Nehmen wir an, wir befinden uns zum zweiten Mal in Situation 2. Dann sind alle Mobile T_j-homogen mit $C - 1$ Aufträgen, o. B. d. A. $j = 1, \dots, q$ und $n(j) \equiv C - 1 \mod C$. Wir betrachten die Situation unmitelbar vor Erreichen der Situation 2. Dies kann weder Situation 1 (Abbruch) noch Situation 2 (führt zu einem leeren Mobil) sein. Situation 3 scheidet ebenfalls aus, da Austellen eines einzelnen ärgerlichen Auftrags entweder zu einem inhomogenen oder zu einem nur mit einem Auftrag beladenen Mobil führt. Wir sind also aus Situation 4 in Situation 2 gekommen. Insbesondere gilt zu diesem Zeitpunkt $n(q+1) \equiv 0 \mod C$, weil auch T_{q+1} ärgerlich ist, wodurch im Falle von $n(q + 1) \equiv C - 1 \mod C$ Situation 3 vorgelegen hätte.

Also gilt beim Erreichen der Situation 2: $n(i) \equiv C-1 \mod C$ für $i = 1, \dots, q$ und $n(q+1) \equiv 0 \mod C$. Daraus folgt für die Gesamtanzahl Z aller Aufträge

auf geschlossenen Mobilen:

$$\begin{aligned} Z &= \sum_{i=0}^{q+1} n(i) - q(C-1) \\ &= 1 + \sum_{i=1}^{q} n(i) + n(q+1) - q(C-1) \\ &\equiv 1 + q(C-1) + 0 - q(C-1) \mod C \\ &\equiv 1 \mod C. \end{aligned}$$

Das ist ein Widerspruch zu der Tatsache, daß Z natürlich durch C teilbar sein muß. □

Beweis der Behauptung 13.7 Der Offline-Gegner erzeugt niemals einen Auftrag eines Typs, von dem sich bereits $C-1$ Aufträge auf einem der verfügbaren Mobile befinden. Da nur ein Auftrag des Typs T_0 erzeugt wird, kann nur ein Mobil mit $z+1$ Stops auskommen; alle anderen geschlossenen Mobile müssen mindestens $2z$-mal halten. □

Beweis der Behauptung 13.8 Wird ein Auftrag vom Typ T_j ausgestellt, so nennen wir ein offenes T_j-homogenes verfügbares Mobil *passend*. Wir behaupten, daß der Offline-Gegner außer in Situation 2 immer ein passendes Mobil findet.

Nach Konstruktion der Auftragssequenz wird der Auftrag vom Typ T_0 erzeugt, wenn in der Zuweisung des Online-Algorithmus nur homogene Mobile mit $C-1$ Aufträgen existieren. Den Auftragstypen auf diesem Mobil nennen wir im folgenden den *online-speziellen* Auftragstypen; den Auftragstypen auf dem Mobil, dem der Offline-Gegner den Auftrag vom Typ T_0 zuordnet, bezeichnen wir als den *offline-speziellen* Auftragstypen.

Wir konstruieren induktiv eine Offline-Zuordnung von Aufträgen auf Mobile für die Situationen 3 und 4 wie folgt:

- Findet der Offline-Gegner in Situation 3
  - nach Ausstellung eines *nicht offline-speziellen* Auftrags ein passendes Mobil mit $C-1$ nicht offline-speziellen Aufträgen, so wählt er die Zuweisung auf dieses Mobil. Das ergibt ein leeres Mobil;
  - nach Ausstellung eines *offline-speziellen* Auftrags ein leeres Mobil, so wählt er die Zuweisung auf dieses Mobil. Das ergibt ein Mobil mit einem Auftrag.
- Findet der Offline-Gegner in Situation 4
  - nach Ausstellung von $C-1$ *nicht offline-speziellen* Aufträgen ein leeres Mobil, so wählt er die Zuweisung auf dieses Mobil. Das ergibt ein homogenes Mobil mit $C-1$ Aufträgen;
  - nach Ausstellung von $C-1$ *offline-speziellen* Aufträgen ein passendes Mobil mit einem Auftrag, so wählt er die Zuweisung auf dieses Mobil. Das ergibt ein leeres Mobil.

Jeden der obigen Fälle bezeichnen wir im folgenden als *Glücksfall*.

Wir beweisen nun die Behauptung durch Induktion. Wir starten, indem wir die ersten $q(C-1)$ Aufträge homogen auf Mobile verteilen. Nach dieser Zuweisung befinden wir uns entweder in Situation 3 oder in Situation 2.

Im ersten Fall gilt $n(j) \equiv C - 1 \mod C$ nur für die Aufträge vom Typ T_i, $i = 1, \ldots, q$ auf den Mobilen; für den verbleibenden Auftragstypen T_{q+1} gilt natürlich $n(q + 1) = 0$. Da bislang noch kein offline spezieller-Auftragstyp existiert, landen wir in einem Glücksfall.

Der zweite Fall ist ein Spezialfall der folgenden allgemeineren Untersuchung.

Nach Behauptung 13.6 kommt Situation 2 höchstens einmal vor. Wir müssen also nur noch die Zuweisung in Situation 2 so vornehmen, daß ein Glücksfall eintritt. Danach treten nur noch die Situationen 3 und 4 auf, und damit nur noch Glücksfälle.

In Situation 2 existieren nach Konstruktion der Auftragssequenz in der augenblicklichen Online-Zuweisung q homogene Mobile. Die Offline-Zuweisung hat nach Induktionsannahme ebenfalls alle Mobile homogen mit $C - 1$ Aufträgen bestückt. Für den Auftragstypen T_k, der in der Offline-Zuweisung auf keinem der augenblicklich verfügbaren Mobile vorhanden ist, muß $n(k) \equiv 0 \mod C$ gelten, da nach Induktionsannahme alle geschlossenen Mobile homogen waren. Das bedeutet aber, daß dieser Typ auch auf keinem Mobil der Online-Zuweisung sein kann. Also sehen die Offline-Mobile aus wie die Online-Mobile. Der Offline-Gegner weist nun den Auftrag vom Typ T_0 auf ein beliebiges anderes Mobil zu als der Online-Spieler. Damit gilt, daß der offline-spezielle Auftragstyp nicht als nächstes ausgestellt wird, da er für den Online-Spieler nicht ärgerlich ist.

Der Offline-Gegner hat nun also ein leeres Mobil, und für alle Auftragstypen T_i auf Mobilen des Offline-Gegners (sowie den speziellen Auftragstypen) gilt $n(i) \equiv C - 1 \mod C$; für den nicht offline-speziellen Auftragstypen T_j, von dem sich kein Auftrag auf einem der Mobile befindet, gilt weiterhin $n(j) \equiv 0 \mod C$.

Sind wir nun in Situation 3, so finden wir ein passendes Mobil mit $C-1$ Aufträgen; in Situation 4 finden wir ein leeres Mobil; in beiden Fällen tritt ein Glücksfall ein, was zu zeigen war. □

Damit ist Theorem 13.2 bewiesen. □

13.4 Online-Heuristiken

Eine naheliegende Strategie ist die sogenannte BESTFIT-Heuristik. Der jeweils nächste Auftrag wird auf dasjenige Mobil gepackt, für das er am wenigsten zusätzliche Stops erzeugt. Obwohl dies sehr natürlich aussieht, zeigen Sie in einer Übungsaufgabe folgendes Resultat:

Satz 13.9 BESTFIT *ist nicht besser als C-kompetitiv für das* OLCVP.

In der Praxis zeigt sich, daß ein Auftrag häufig auf allen Mobilen gleichviele neue Stops erzeugt (alle Stoppositionen sind neu). Dann benötigt man einen *tie-breaker*, eine Sekundärregel, die die Zuordnung entscheidet, wenn die Primärregel mehrere Gewinner hat. Hier sind einige mögliche Varianten:

tie-breaker

- Wähle das erste Mobil in der Reihe;
- wähle das letzte Mobil in der Reihe;
- wähle das leerste Mobil;
- wähle das vollste Mobil.

Erlaubt man das Verändern von Zuweisungen auf noch nicht geschlossenen Mobilen, so kann man die Güte der Heuristiken durch *Austauschverfahren* verbessern. Für ein festes $k > 0$ werden im k*-Austauschverfahren* alle k-elementigen Teilmengen der zugewiesenen Aufträge auf nicht gesschlossenen Mobilen betrachtet: die Zuweisung dieser k-elementigen Menge wird ggf. auf die lokal billigste Zuweisung geändert. Dies geschieht, solange man Verbesserungen auf k-elementigen Teilmengen findet.

Austauschverfahren

k*-Austauschverfahren*

Man beachte, daß das Ändern der Zuweisung in der Realität nicht zu physikalischem Umpacken führen muß; die einem Mobil zugeordneten Aufträge müssen ja erst dann auf das Mobil gepackt werden, wenn das Mobil abfährt, d. h. nachdem es geschlossen worden ist.

Da in der Praxis die Minimierung der Gesamtanzahl der Stops kein Selbstzweck ist, betrachtet man in heuristischen Überlegungen auch andere Aspekte des Systemverhaltens wie Gesamtfertigstellungszeit, durchschnittliche/maximale Rundenzeit, Stauzeit etc.. Da eine Unterbrechung der Arbeit des Komissionierers auf dem Mobil unerwünscht ist, spielt die Minimierung der Stauzeit eine große Rolle. Hat man eine Zuweisung gemacht, so kann man entweder mit Simulationsmethoden oder durch einfach Berechnungen die Stauzeit eines Mobils ungefähr prognostizieren. Man kann dann bei der Vorhersage eines langen Staus entweder die Zuweisungen in dieser Hinsicht verbessern oder einfach das nächste Mobil zurückhalten.

Ein Stau im Zuweisungsbereich ist nicht so schlimm wie einer auf der Strecke.

13.5 Fazit

Die Simulations-Erfahrung zeigt, daß durch einen 2-Austauschverfahren noch erhebliche Verbesserungen erzielt werden können. Schon ein 3-Austauschverfahren lohnt den zusätzlichen Rechenaufwand nicht mehr.

In Simulationen konnte gezeigt werden, daß mit den besten heuristischen Verfahren ohne Einbußen in der Systemleistung

- die Gesamtfertigstellungszeit um bis zu vier Stunden verkürzt,
- die Anzahl der Stops um bis zu 6% reduziert,
- die durchschnittliche Stauzeit eines Mobils von etwa fünf Minuten auf vierzig Sekunden gedrückt,

- die Anzahl der benutzten Fahrzeuge von acht auf sechs verkleinert werden konnte.

Wir haben gesehen, daß in diesem speziellen Fall kompetitive Analyse leider nicht zu dem Erfolg beitragen konnte. Dies spiegelt die augenblickliche Situation in der praktischen Online-Optimierung wider: man erreicht Verbesserungen durch mathematisch motivierte Heuristiken, man kann jedoch keine allgemeingültigen Analysen und Gütegarantien abgeben. Simulationen müssen daher als Nachweis der Leistungsfähigkeit von Online-Verfahren herhalten.

Übung 13.1

Beweise: Wenn der Offline-Gegenspieler alle benötigten Fahrzeuge, die für die komplette Zuweisung einer Auftragsequenz benötigt werden, benutzen darf, so existiert kein Online-Algorithmus mit Kompetitivität besser als C.

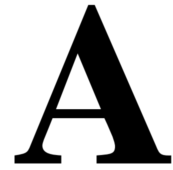
Übung 13.2

Beweisen Sie, daß BESTFIT für das OLCVP nicht besser als C-kompetitiv ist, wenn $q \geq 2$ Mobile zur Verfügung stehen.

Übung 13.3

Diese Aufgabe hat keine »richtige Lösung«; experimentieren sie mit Modellen.

Formulieren Sie ein plausibles Optimierungsproblem im Zusammenhang mit der Zuordnung von Aufträgen auf Kommissioniermobile, welches das Zeitstempelmodell verwendet, und führen Sie eine kompetitive Analyse durch.



Abkürzungen und Symbole

Abkürzungen

O. B. d. A.	Ohne Beschränkung der Allgemeinheit

Symbole

$\forall$	der Allquantor
$\exists$	der Existenzquantor
$[a,b]$	das geschlossene Intervall $\{\, x \in \mathbb{R} \mid a \leq x \leq b \,\}$
$[a,b)$	das halboffene Intervall $\{\, x \in \mathbb{R} \mid a \leq x < b \,\}$, analog auch $(a,b]$
(a,b)	das offene Intervall $\{\, x \in \mathbb{R} \mid a < x < b \,\}$
$\lvert A \rvert$	die Kardinalität der Menge A
$\varnothing$	die leere Menge
$\mathbb{N}$	die Menge der natürlichen Zahlen, $\mathbb{N} := \{0, 1, 2, \ldots\}$
$\Omega(g)$	$\Omega(g) := \{\, f \in M \mid \exists c, n_0 \in \mathbb{N} \colon \forall n \geq n_0 \colon f(n) \geq c \cdot g(n) \,\}$
$\mathcal{O}(g)$	$\mathcal{O}(g) := \{\, f \in M \mid \exists c, n_0 \in \mathbb{N} \colon \forall n \geq n_0 \colon f(n) \leq c \cdot g(n) \,\}$
$\mathbb{R}$	die Menge der reellen Zahlen
$\Theta(g)$	$\Theta(g) := \mathcal{O}(g) \cap \Omega(g)$

B

Komplexität von Algorithmen

B.1 Größenordnung von Funktionen

Sei M die Menge aller reellwertigen Funktionen $f\colon \mathbb{N} \to \mathbb{R}$ auf den natürlichen Zahlen. Jede Funktion $g \in M$ legt dann drei Klassen von Funktionen wie folgt fest:

- $\mathcal{O}(g) := \{\, f \in M \mid \exists c, n_0 \in \mathbb{N}\colon \forall n \geq n_0\colon f(n) \leq c \cdot g(n) \,\}$
- $\Omega(g) := \{\, f \in M \mid \exists c, n_0 \in \mathbb{N}\colon \forall n \geq n_0\colon f(n) \geq c \cdot g(n) \,\}$
- $\Theta(g) := \mathcal{O}(g) \cap \Omega(g)$

Man nennt eine Funktion f *von polynomieller Größenordnung* oder einfach *polynomiell*, wenn es ein Polynom g gibt, so daß $f \in \mathcal{O}(g)$ gilt.

B.2 Berechnungsmodell

Das bei der Laufzeit-Analyse verwendete Maschinenmodell ist das der *Unit-Cost RAM* (Random Access Machine). Diese Maschine besitzt abzählbar viele Register, die jeweils eine ganze Zahl beliebiger Größe aufnehmen können. Folgende Operationen sind jeweils in einem Takt der Maschine durchführbar: Ein- oder Ausgabe eines Registers, Übertragen eines Wertes zwischen Register und Hauptspeicher (evtl. mit indirekter Adressierung), Vergleich zweier Register und bedingte Verzweigung, sowie die arithmetischen Operationen Addition, Subtraktion, Multiplikation und Division [28].

Dieses Modell erscheint für die Analyse der Laufzeit von Algorithmen besser geeignet als das Modell der Turing-Maschine, denn es kommt der Arbeitsweise realer Rechner näher. Allerdings ist zu beachten, daß die Unit-Cost RAM in einem Takt Zahlen beliebiger Größe verarbeiten kann. Durch geeignete Codierungen können damit ausgedehnte Berechnungen in einem einzigen Takt versteckt werden, ferner sind beliebig lange Daten in einem Takt zu bewegen. Damit ist

das Modell echt mächtiger als das der Turing-Maschine. Es gibt keine Simulation einer Unit-Cost RAM auf einer (deterministischen) Turing-Maschine, die mit einem polynomiell beschränkten Mehraufwand auskommt.

Um diesem Problem der zu großen Zahlen vorzubeugen, kann man auf das Modell der *Log-Cost RAM* [28] zurückgreifen. Bei einer solchen Maschine wird für jede Operation ein Zeitbedarf angesetzt, der proportional zum Logarithmus der Operanden, also proportional zur Codierungslänge ist. Eine andere Möglichkeit, das Problem auszuschließen, besteht darin, sicherzustellen, daß die auftretenden Zahlen nicht zu groß werden, also daß ihre Codierungslänge polynomiell beschränkt bleibt. Diese Voraussetzung ist bei den hier vorgestellten Algorithmen stets erfüllt. Der einfacheren Analyse wegen wird daher das Modell der Unit-Cost RAM zugrundegelegt.

B.3 Komplexitätsklassen

Die Komplexität eines Algorithmus ist ein Maß dafür, welchen Aufwand an Ressourcen ein Algorithmus bei seiner Ausführung braucht. Man unterscheidet die Zeitkomplexität, die die benötigte Laufzeit beschreibt, und die Raumkomplexität, die Aussagen über die Größe des benutzten Speichers macht. Raumkomplexitäten werden in diesem Skript nicht untersucht.

Die Komplexität wird in der Regel als Funktion über der Länge der Eingabe angegeben. Man nennt einen Algorithmus *von der (worst-case-) Komplexität* T, wenn die Laufzeit für alle Eingaben der Länge n durch die Funktion $T(n)$ nach oben beschränkt ist.

Die Komplexität von Algorithmen wird in dieser Arbeit als Funktion der Eckenzahl n und Kantenzahl m des eingegebenen Graphen angegeben. Diese Angabe ist detaillierter als die Abhängigkeit der Komplexität von der Eingabelänge: bei ecken- und kantenbewerteten Graphen ist deren Codierungslänge mindestens von der Größenordnung $\Omega(n + m)$.

Besonders wichtig sind in diesem Zusammenhang die Klassen P und NP. Die Klasse P ist die Menge aller Entscheidungsprobleme, die auf einer deterministischen Turing-Maschine in polynomieller Zeit gelöst werden können. Entsprechend ist die Klasse NP definiert als die Menge aller Probleme, deren Lösung auf einer nichtdeterministischen Turing-Maschine in Polynomialzeit möglich ist. Man vergleiche dazu etwa [18].

Eine Transformation zwischen NP-Problemen heißt *polynomielle Reduktion*, wenn sie in polynomieller Zeit Instanzen zweier Probleme aus NP so ineinander überführt, daß die Antwort des Ausgangsproblems auf die Ausgangsinstanz dieselbe ist wie die Antwort des zweiten Problems auf die transformierte Instanz. Ein Problem heißt NP-*vollständig*, wenn jedes andere Problem aus NP polynomiell darauf reduziert werden kann. Der Reduktionsbegriff wird durch Einführen der *Turing-Reduktion* so erweitert, daß Reduktionen zwischen Optimierungsproblemen und Entscheidungsproblemen in NP erfaßt werden. Ein NP-Optimierungsproblem heißt dann NP-*hart*, wenn es von einem NP-vollständigen Entscheidungsproblem turing-reduziert werden kann.

Ein wesentliches Resultat der Komplexitätstheorie besagt, daß NP-harte Optimierungsprobleme nicht in polynomieller Zeit gelöst werden können, es sei

denn, es gilt $\mathsf{P} = \mathsf{NP}$. Dies ist der Grund, warum bei der Untersuchung von NP-harten Optimierungsproblemen auf exakte Lösungen verzichtet wird und stattdessen Näherungen in Betracht gezogen werden.

C

Lösungen zu den Übungsaufgaben

Kapitel 2

Aufgabe 2.1

1. Ein (deterministischer) Online-Algorithmus ALG ist **c-kompetitiv**, wenn es eine Konstante α gibt, so daß für alle (endlichen) Eingabefolgen σ gilt:

$$\mathsf{ALG}(\sigma) \geq \frac{\mathsf{OPT}(\sigma)}{c} + \alpha.$$

   Alternativ:

$$c \cdot \mathsf{ALG}(\sigma) \geq \mathsf{OPT}(\sigma) + \alpha.$$

   Kann $\alpha = 0$ gewählt werden, so nennt man ALG **strikt c-kompetitiv**.

2. Ein randomisierter Online-Algorithmus ALG, verteilt über einer Menge $\{\mathsf{ALG}_x\}$ von deterministischen Algorithmen ist **c-kompetitiv** gegen einen Gegner vom Typ ADV wenn es eine Konstante α gibt, so daß für alle (endlichen) Eingabefolgen σ gilt:

$$E_x[c\mathsf{ALG}(\sigma) - \mathsf{ADV}(\sigma)] \geq \alpha.$$

Aufgabe 2.3

(a) Sei S die aktuelle Größe des belegten Blocks, anfangs $S = 0$. Bei einer Anfrage $r_i > S$ alloziiert der Algorithmus ALG einen neuen Speicherblock der Größe 2^ℓ, wobei $2^{\ell-1} < r_i \leq 2^\ell$. Wir zeigen im folgenden, daß ALG 4-kompetitiv ist.

Sei dazu $\sigma = r_1, \ldots, r_n$ eine beliebige Anfragefolge mit $r_1 \leq \cdots \leq r_n$. Bei der ersten Anfrage r_1 hat ALG Kosten $\mathsf{ALG}(r_1) = 2^\ell$ mit $2^{\ell-1} < r_i \leq 2^\ell$. Danach hat ALG erst bei der ersten Anfrage r_j Kosten, bei der $r_j > 2^\ell$ gilt. Somit lassen sich die Kosten von ALG wie folgt beschränken:

$$\mathsf{ALG}(\sigma) \leq \sum_{k=1}^{p} 2^k, \qquad \text{wobei } 2^{p-1} < r_n \leq 2^p.$$

Wegen $\sum_{k=1}^{p} 2^k = 2^{p+1} - 1$ und $\mathsf{OPT}(\sigma) = r_n > 2^{p-1}$ folgt $\mathsf{ALG}(\sigma) \leq 4 \cdot \mathsf{OPT}(\sigma)$.

(b) Sei ALG c-kompetitiv. Der Adversary konstruiert die Anfragefolge σ induktiv. Er startet mit $r_1 = 1$. Sei L_i die Größe des von ALG neu belegten Speicherblocks bei der Anfrage r_i. Dann setzt der Adversary $r_{i+1} := L_i + 1$.

Für die Kosten von ALG gilt dann:

$$\mathsf{ALG}(\sigma) = \sum_{i=1}^{n} L_i \geq L_n + L_{n-1} = L_n + r_n - 1 \geq 2r_n - 1 = 2\,\mathsf{OPT}(\sigma) - 1.$$

Da wir r_n beliebig groß werden lassen können, folgt, daß ALG höchstens 2-kompetitiv ist.

Wir zeigen jetzt durch genauere Analyse, daß jeder deterministische c-kompetitive Algorithmus sogar $c \geq 3$ erfüllt.

Der Adversary benutzt die gleiche Anfragesequenz wie oben. Es folgt, daß für jedes $n \in \mathbb{N}$ gilt:

$$\begin{aligned} c &\geq \frac{\sum_{i=1}^{n+1} L_i}{L_n + 1} &\text{(C.1)}\\ &= \frac{L_{n+1} + L_n + 1 + \sum_{i=1}^{n-1} L_i - 1}{L_n + 1}\\ &> 1 + \frac{L_{n+1}}{L_n + 1} &\text{(C.2)}\end{aligned}$$

Sei $n \in \mathbb{N}$ so groß, daß $\sum_{i=1}^{n-1} L_i > 1 + c$, also $\sum_{i=1}^{n} L_i > L_n + 1 + c$. Dann gilt für $k \in \mathbb{N}$:

$$\begin{aligned} c(L_{n+k} + 1) &\geq \sum_{i=1}^{n+k+1} L_i \qquad \text{(nach (C.1))}\\ &= \sum_{i=1}^{n} L_i + \sum_{j=1}^{k+1} L_{n+j}\\ &> L_n + 1 + c + \sum_{j=1}^{k+1} L_{n+j}\end{aligned}$$

Dies is äquivalent zu

$$L_{n+k} > \frac{L_n + 1 + \sum_{j=1}^{k+1} L_{n+j} - L_{n+k}}{c - 1} \quad \text{für alle } k \in \mathbb{N}. \tag{C.3}$$

Sei $g(k) := L_{n+k}/L_n$. Man beachte, daß $g(k) \geq 1$ für alle $k \in \mathbb{N}$ gilt. Ungleichung (C.2) ist dann äquivalent zu

$$c > 1 + g(1). \tag{C.4}$$

Aus (C.3) ergibt sich für die Werte $g(k)$.

$$g(k) \geq 1 + \frac{g(k+1)}{c-1} = \frac{c - 1 + g(k+1)}{c - 1}. \tag{C.5}$$

Zusammen mit $g(k) \geq 1$ für alle $k \in \mathbb{N}$ folgt aus (C.4) und (C.5), daß für den Kompetitivitätsfaktor c gilt:

$$c > 1 + \sum_{k=0}^{\infty} \left(\frac{1}{c-1}\right)^k = \frac{c-1}{c-2}$$

Aus der letzten Ungleichung folgt, daß $c^2 - 4c + 3 > 0$. Die kleinste Lösung dieser quadratischen Ungleichung in $[1, +\infty[$ ist $c = 3$.

Aufgabe 2.4

(a) Sei ALG c-kompetitiv mit $c < 4/3$. Der Offline-Gegner gibt nun eine Folge σ aus n Gegenständen der Größe $1/2 - \varepsilon$ gefolgt von n Gegenständen der Größe $1/2 + \varepsilon$. Es gilt $\mathsf{OPT}(\sigma) = n$.

Sei σ' die Teilfolge der ersten n Gegenstände. Nach Bearbeitung von σ' habe ALG $s = \mathsf{ALG}(\sigma')$ Kisten geöffnet. Danach öffne ALG noch weitere b Kisten für den Rest der Folge. Somit gilt $\mathsf{ALG}(\sigma) = s + b$.

In jeder der b Kisten, die ALG nach der $s$ten Kiste öffnet, befindet sich in der Packung von ALG nur höchstens ein Gegenstand, da von den letzten n Gegenständen nur jeweils einer in einer Kiste paßt. In jedem der ersten s Behälter befinden sich maximal zwei Gegenstände. Somit folgt $2s+b \geq 2n$, oder $b \geq 2(n - s)$. Das bedeutet $\mathsf{ALG}(\sigma) \geq 2n - s$.

Da ALG nach Annahme c-kompetitiv ist, gilt

$$s \leq c \cdot \mathsf{OPT}(\sigma') \leq \frac{4}{3} \cdot \frac{n}{2} = \frac{2}{3} \cdot n.$$

Damit folgt aber $\mathsf{ALG}(\sigma) \geq 2n - 2/3 \cdot n = 4/3 \cdot n$. Dies widerspricht der Annahme, daß $c < 4/3$ gilt.

(b) Am Ende des Verpackens einer Folge $\sigma = r_1, \ldots, r_n$ ist höchstens eine Kiste weniger als zur Hälfte gefüllt (wenn zwei der Kisten weniger als bis zur Hälfte gefüllt wären, dann hätte FIRSTFIT den Inhalt der zweiten Kiste noch in die erste gepackt). Somit gilt $\sum_{i=1}^{n} r_i \leq \frac{\mathsf{FIRSTFIT}(\sigma)-1}{2}$. Da jeder Algorithmus mindestens $\sum_{i=1}^{n} r_i$ Behälter benötigt, folgt

$$\mathsf{FIRSTFIT}(\sigma) \leq 2 \cdot \mathsf{OPT}(\sigma) + 1.$$

Aufgabe 2.5

(a) Der Algorithmus liefert ein maximales Matching M: Wäre M nicht maximal, so gäbe es eine Kante $e = (h, d) \in E$, so daß $M \cup \{e\}$ ebenfalls ein Matching ist. Zu dem Zeitpunkt, wo d dem Algorithmus bekannt wird, könnte d somit einem Herren zugewiesen werden können.

Jedes maximale Matching M hat mindestens $n/2$ Kanten: Wenn $|M| < n/2$, dann sind in M weniger als $n/2$ Herren H' zugeordnet. Sei M^* das nach Voraussetzung existierende perfekte Matching in G. In M^* hat jeder der mehr als $n/2$ Herren aus $H \setminus H'$ eine Partnerin. Davon sind jedoch in M weniger als $n/2$ vergeben. Somit kann noch mindestens ein Herr aus $H \setminus H'$ zugeordnet werden, was der Maximalität von H' widerspricht.

(b) Der Adversary schickt zuerst $n/2$ Damen, die gewillt sind, jeden Herren zu heiraten. Der Algorithmus verheirate davon b Stück und lehne $n/2 - b$ ab. Die nächsten b Damen sind jeweils gewillt, nur einen Herren zu heiraten. Die entsprechenden Herren sind die vom Online-Algorithmus verheirateten b Herren. Offenbar muß der Algorithmus alle diese Damen ablehnen. Zum Schluß folgen noch $n/2 - b$ Damen, die nicht wählerisch sind und jeden der n Herren heiraten wollen.

Insgesamt kann der Online-Algorithmus also maximal $b + n/2 - b = n/2$ Paare bilden. Offenbar besitzt der »Sympathiegraph« ein perfektes Matching der Größe n.

Aufgabe 2.6

Sei $a > 0$ (der Fall $a < 0$ verläuft analog) und $\alpha^{2k+1} < a \leq \alpha^{2(k+1)+1}$. Wenn $a \leq \alpha$, dann findet der Algorithmus das Auto im ersten Zug und hat Kosten a. Wir nehmen daher an, daß $a > \alpha$ und somit $k \geq 1$ gilt. Die Wegstrecke des Algorithmus ist dann

$$2\sum_{i=1}^{2k+2} \alpha^i + a = 2\frac{\alpha^{2k+3} - 1}{\alpha - 1} - 2 + a \leq 2\frac{\alpha^2 a - 1}{\alpha - 1} - 2 + a < \left(\frac{2\alpha^2 + \alpha - 1}{\alpha - 1}\right) \cdot a.$$

Der Term $\left(\frac{2\alpha^2+\alpha-1}{\alpha-1}\right)$ ist minimal für $\alpha = 2$ und erreicht dann 9. Der entsprechende Algorithmus ist somit 9-kompetitiv.

Kapitel 4

Aufgabe 4.1

Wir führen den Beweis nur für LRU, indem wir den Beweis von Satz 4.6 adaptieren. Der Beweis für FIFO erfolgt durch analoge Modifikation des Beweises von Satz 4.8.

Um zu beweisen, daß LRU $k/(k - h + 1)$-kompetitiv ist, adaptieren wir den Beweis von Satz 4.6 und zeigen, daß jeder Markierungsalgorithmus $k/(k - h + 1)$-kompetitiv für das (h, k)-Paging Problem ist.

Sei ALG ein Markierungsalgorithmus und $\sigma = r_1, \ldots, r_n$ eine beliebige Anfragefolge. Wir benutzen wieder die k-Phasenpartition von σ. Wie im Beweis von Satz 4.6 gezeigt, hat der Markierungsalgorithmus ALG höchstens k Seitenfehler pro Phase. Wir zeigen nun, daß der optimale Offline-Algorithmus mit Cachegröße h pro Phase mindestens $k - h + 1$ Seitenfehler hat.

Dazu betrachten wir wieder die Segmente aus dem Beweis von Satz 4.6. Jedes Segment startet bei der zweiten Anfrage der zugehörigen Phase und endet mit der ersten Anfrage der folgenden Phase. Zu Beginn eines Segments hat OPT die zuletzt gefragte Seite p, d.h. die Seite der ersten Anfrage der Phase, im Speicher. Bis zum Ende der Phase werden noch $k-1$ von p verschiedene Seiten, bis zum Ende des Segments noch k von p von verschiedene Seiten gefragt. Da OPT nur h Seiten in Cache halten kann, muß OPT im Segment mindestens $k - h + 1$ Seitenfehler haben.

Aufgabe 4.2

Der adaptive Offline Adversary kann sehen, welche Seiten RANDMARK entfernt. Er kann daher erwirken, daß RANDMARK bei jeder Anfrage einen Seitenfehler hat. Er selbst bearbeitet die Folge mit durchschnittlich einem Seitenfehler pro k-Anfragen.

Die Zufallsvariable X_i für die Seitenfehler in Phase i von RANDMARK ist beim adaptiven Gegner keine zufällige Variable mehr.

Kapitel 8

Aufgabe 8.1

Sei $s_1 \leq s_2 \leq \cdots \leq s_k$ die Startpositionen der k Server. Der Adversary gibt nun abwechselnd Anfragen auf den Punkten $x = s_k + 2$ und $y = s_k + 3$. Der Algorithmus bewegt bei der ersten Anfrage seinen Server von s_k auf den Punkt x. Die nächste Anfrage auf y wird ebenfalls von diesem Server beantwortet. Der Server mit der ursprünglichen Position s_k fährt in der weiteren Folge immer zwischen x und y hin und her.

Der optimale Offline-Algorithmus bewegt anfangs einen Server zu x und einen zu y. Seine Kosten sind somit durch eine Konstante beschränkt, während die Kosten des Online-Algorithmus beliebig groß werden.

Aufgabe 8.2

(i) Wir betrachten den Fall $x \leq y_i$, $i = 1, \ldots, k$. Der andere Fall ist analog. Sei M ein minimales Matching und $\tilde{y} := M(x) \in Y$ der Matchingpartner von x. Ist $\tilde{y} = y$, so ist nichts zu zeigen. Sei daher $\tilde{y} \neq y$ und $\tilde{x} := M^{-1}(y)$ der Matchingpartner von y.

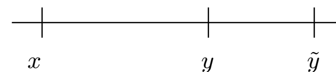
Wir definieren ein neues Matching M', das die Bilder von x und $\tilde{x}$ vertauscht. Formal ist

$$M'(x') := \begin{cases} M(x') & \text{, falls } x' \neq x, \tilde{x} \\ y & \text{, falls } x' = x \\ \tilde{y} & \text{, falls } x' = \tilde{x} \end{cases}$$

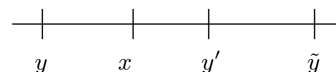
Für die Gewichte $w(M)$ und $w(M')$ gilt dann:

$$\begin{aligned} w(M') - w(M) &= d(x,y) + d(\tilde{x},\tilde{y}) - d(x,\tilde{y}) - d(\tilde{x},y) \\ &= (y - x) - (\tilde{y} - x) + d(\tilde{x},\tilde{y}) - d(\tilde{x},y) \\ &= y - \tilde{y} + d(\tilde{x},\tilde{y}) - d(\tilde{x},y) \\ &= d(\tilde{x},\tilde{y}) - d(y,\tilde{y}) - d(\tilde{x},y) \leq 0. \end{aligned}$$

Somit ist M' ebenfalls ein minimales Matching. Ferner hat M' die gewünschten Eigenschaften.



(ii) Der Beweis verläuft ähnlich zu Teil (i). Sei $\tilde{y} := M(x) \notin \{y, y'\}$. Sei $\tilde{y} \geq y'$ (der Fall $\tilde{y} \leq y$ verläuft analog) und $\tilde{x} := M^{-1}(y')$. Wir definieren wieder M', welches die Bilder von x und $\tilde{x}$ vertauscht. Es folgt dann analog zu oben $w(M') - w(M) = d(\tilde{x},\tilde{y}) - d(y',\tilde{y}) - d(\tilde{x},y') \leq 0$.



Aufgabe 8.3

(a) Sei ALG' ein c-kompetitiver Algorithmus für das k-Serverproblem in (X', d'). O. B. d. A. bewege ALG bei jeder Anfrage höchstens einen Server und dies auch nur, wenn der Server zum Bearbeiten der aktuellen Anfrage benutzt wird (vgl. Abschnitt 66). Wegen $X \subseteq X'$ ist jede Anfragefolge aus Punkten aus X auch eine gültige Anfragefolge von Punkten aus X'.

Unser Algorithmus ALG in (X, d) sieht wie folgt aus: Bei Anfrage r_i »simuliert« ALG den Algorithmus ALG': Wenn ALG' einen Server von x nach r_i zieht, dann zieht auch ALG einen Server von x nach r_i.

Sei OPT' ein optimaler Offline-Algorithmus in (X', d'), OPT einer in (X, d). Aus (8.30) folgt $\mathsf{OPT}'(\sigma) \leq \beta\mathsf{OPT}(\sigma)$, da die Abstände in (X', d') höchstens β mal so groß wie in (X, d) sind.

Wegen (8.29) gilt $\mathsf{ALG}(\sigma) \leq \mathsf{ALG}'(\sigma)$. Damit folgt:

$$\mathsf{ALG}(\sigma) \leq \mathsf{ALG}'(\sigma) \leq c \cdot \mathsf{OPT}'(\sigma) \leq \beta c \cdot \mathsf{OPT}(\sigma).$$

Somit ist ALG βc-kompetitiv.

(b) Wir setzen $X' := X$ und $d'(x, y) = 1$ für alle $x, y \in X$. Die Bedingungen (i), (ii), (iii) sind mit $\beta = D$ erfüllt. Das Ergebnis folgt nun aus (a).

(c) Sei G_i der Graph, der aus G durch Entfernen der Kante (v_i, v_{i+1}) entsteht. Wir können G_i als Pfad auffassen und auf ihm den Algorithmus DC für das k-Server Problem benutzen. Der Beweis von Satz 8.4 zeigt, daß DC k-kompetitiv für das k-Server Problem auf G_i ist.

Unser randomisierter Algorithmus arbeitet wie folgt: Er wählt eine zufällige Zahl $i \in \{1, \ldots, n-1\}$ gleichverteilt aus und benutzt dann den Algorithmus DC auf G_i zum Bearbeiten der Anfragefolge $\sigma = r_1, \ldots, r_n$.

Wir zeigen, daß folgende Eigenschaften gelten:

(i) G_i enthält alle Knoten von G. (Dies ist trivial!)

(ii) Für alle $x, y \in G$ und alle i gilt:

$$d_G(x, y) \leq d_{G_i}(x, y). \tag{C.6}$$

(Diese Ungleichung ist auch trivial!)

(iii) Es gilt für alle $x, y \in G$:

$$\mathbb{E}_i[d_{G_i}(x, y)] \leq 2d_G(x, y) \tag{C.7}$$

Seien $\mathsf{OPT}_i(\sigma)$ die optimalen Offline-Kosten zur Bearbeitung von σ auf G_i. Wie üblich bezeichnen wir mit $\mathsf{OPT}(\sigma)$ die optimalen Kosten (auf G). Sei letztendlich $\mathsf{DC}_i(\sigma)$ der Algorithmus DC bei der Bearbeitung der Folge σ auf dem Graphen G_i.

Analog zu Aufgabenteil (a) folgt dann:

$$\mathbb{E}[\mathsf{ALG}(\sigma)] = \mathbb{E}_i[\mathsf{DC}_i(\sigma)] \leq \mathbb{E}_i[k\mathsf{OPT}_i(\sigma)] = k\mathbb{E}[\mathsf{OPT}_i(\sigma)] \leq 2k\mathsf{OPT}(\sigma).$$

Die k-Kompetitivität von DC geht dabei in die erste Ungleichung, die Eigenschaften von d_{G_i} in die zweite Ungleichung ein.

Zu beweisen ist noch (C.7). Seien $v_a, v_b \in V$ mit $a < b$. Dann gilt

$$\mathbb{E}_i[d_{G_i}(v_a, v_b)] \leq \mathbb{E}_i\left[\sum_{j=a}^{b-1} d_{G_i}(v_j, v_{j+1})\right] = \sum_{k=a}^{b-1} \mathbb{E}_i[d_{G_i}(v_j, v_{j+1})].$$

Daher genügt es zu zeigen, daß für benachbarte Knoten v_j, v_{j+1} gilt: $\mathbb{E}_i[d(v_j, v_{j+1})] \leq 2d_G(v_j, v_{j+1}) = 2$.

Mit Wahrscheinlichkeit $1/n$ wird die Kante (v_j, v_{j+1}) aus G entfernt. In diesem Fall ist $d_{G_i}(v_j, v_{j+1}) = n$. Mit Wahrscheinlichkeit $1 - 1/n$ bleibt die Kante und somit auch der Abstand erhalten. Daher gilt

$$\mathbb{E}_i[d(v_j, v_{j+1})] = \frac{1}{n} \cdot n + \left(1 - \frac{1}{n} \cdot 1\right) = 1 + 1 - \frac{1}{n} \leq 2.$$

Dies wollten wir zeigen.

Kapitel 6

Aufgabe 6.1

(a) Für den Fall $\alpha = 1$ ist LIST identisch mit GRAHAM.

(b) Wir benutzen eine Modifikation des Beweises von Satz 6.1. Sei dazu $\sigma = (r_1, \ldots, r_n)$ eine beliebige Anfragenfolge.

1. Fall: LIST erzeugt auf der schnellen Maschine die maximale Last.

Sei w die Last des letzten Jobs, der auf Maschine 1 gelegt wird und l die vorhergehende Last auf Maschine 1. Dann gilt $\mathsf{LIST}(\sigma) = l + w$. Die Last der langsamen Maschine ist mindestens $l + w - \alpha w = l + (1-\alpha)w$. Also ist die Summe aller Jobgrößen mindestens $(1 + 1/\alpha)l + w/\alpha$. Da man Jobs mit Gesamtlast X auf den zwei Maschinen nur so aufteilen kann, daß mindestens eine Maschine Last $X/(1 + 1/\alpha)$ besitzt, folgt

$$\mathsf{OPT}(\sigma) \geq l + \frac{1}{(1 + 1/\alpha)\alpha} w = l + \frac{1}{1+\alpha} w.$$

Daher gilt

$$\begin{aligned}
\mathsf{LIST}(\sigma) &= l + w \\
&= \mathsf{OPT}(\sigma) + \left(1 - \frac{1}{\alpha + 1}\right) w \\
&= \mathsf{OPT}(\sigma) + \frac{\alpha}{\alpha + 1}\, w \\
&\leq \frac{2\alpha + 1}{\alpha + 1}\, \mathsf{OPT}(\sigma).
\end{aligned}$$

2. Fall: LIST erzeugt die maximale Last auf der langsamen Maschine.

Sei w die Last des letzten Jobs, der auf die langsame Maschine gelegt wird und L die vorhergehende Last. Dann ist $\text{LIST} = L + \alpha w$. Die Last der schnellen Maschine ist mindestens $L + \alpha w - w = L + (\alpha - 1)w$. Die Summe aller Jobgrößen ist daher mindestens $(L + \alpha w)/\alpha + L + (\alpha - 1)w = (1 + 1/\alpha)L + \alpha w$. Es folgt

$$\text{OPT}(\sigma) \geq L + \frac{1}{\alpha + 1} w.$$

Folglich ist

$$\text{LIST}(\sigma) = L + \alpha w = \text{OPT}(\sigma) + \frac{\alpha^2 - \alpha - 1}{\alpha + 1} w.$$

Für $\alpha \leq (1 + \sqrt{5})/2$ gilt $\alpha^2 - \alpha - 1 \leq 0$. Somit ist LIST im zweiten Fall oben sogar 1-kompetitiv. Da für diese kleinen Werte von α auch $(2\alpha + 1)/(\alpha + 1) \leq 1 + 1/\alpha$, also $\min\left\{\frac{2\alpha+1}{\alpha+1}, 1 + \frac{1}{\alpha}\right\} = \frac{2\alpha+1}{\alpha+1}$ gilt, folgt somit die Kompetitivität mit dem gewünschten Faktor.

(c) Sei $\alpha \geq (1 + \sqrt{5})/2$. In diesem Fall ist $\min\left\{\frac{2\alpha+1}{\alpha+1}, 1 + \frac{1}{\alpha}\right\} = 1 + \frac{1}{\alpha}$. Betrachte den Algorithmus, der alle Jobs auf die schnelle Maschine legt. Dieser Algorithmus erzeugt auf der Eingabefolge σ kein besseres Ergebnis als LIST: Wenn LIST einen Job mit auf die langsame Maschine legt, dann ist seine Fertigstellungszeit (auf der langsamen Maschine) höchstens gleich der aktuellen Last auf der schnellen Maschine plus der Jobgröße. Die aktuelle Last auf der schnellen Maschine ist dabei höchstens gleich der Summe aller Jobgrößen ohne diesen Job.

Sei S die Summe aller Jobgrößen. Dann ist $\text{OPT}(\sigma) \geq \frac{S}{1+1/\alpha}$. Der Algorithmus, der alle Jobs auf die schnelle Maschine legt, erzeugt eine Last von S auf dieser Maschine, ist also $1 + 1/\alpha$-kompetitiv.

(d) Da wir die Jobgrößen beliebig skalieren können, können wir davon ausgehen, daß die additive Konstante eines c-kompetitiven Algorithmus gleich Null ist. Sei ALG ein beliebiger Online-Algorithmus für das Schedulingproblem. Der Adversary gibt ihm nacheinander die Jobs $r_1 = (1, \alpha)$ und $r_2 = (\alpha, \alpha^2)$. ALG muß r_1 auf die schnelle Maschine legen, da er sonst einen Makespan von α erzeugt, während $\text{OPT}(r_1) = 1$ gilt (wegen $\alpha \geq (1 + \sqrt{5})/2$ gilt $\alpha \geq 1 + 1/\alpha$). Der zweite Job muß von ALG ebenfalls auf die schnelle Maschine gelegt werden, da sonst gilt:

$$\frac{\text{ALG}(r_1, r_2)}{\text{OPT}(r_1, r_2)} = \frac{\alpha^2}{\alpha} = \alpha.$$

Folglich erzeugt ALG die Last $1 + \alpha$ auf der schnellen Maschine. OPT verteilt hingegen den kurzen Job auf die langsame und den langen Job auf die schnelle Maschine, was einen Makespan von α ergibt.

Aufgabe 6.2

(a) Sei $V = \{v_1, \ldots, v_N\}$ und $E = \{(v_i, v_{i+1}) : i = 1, \ldots, N - 1\}$. Der Adversary gibt zunächst eine Anfrage von v_1 nach v_N. Diese muß ein Online-Algorithmus ALG akzeptieren, da dies die einzige Anfrage sein könnte. Danach gibt der Adversary $N - 1$ Anfragen

$(v_1, v_2), (v_2, v_3), \ldots, (v_{N-1}, v_N)$. Diese muß ALG dann ablehnen. Sein Profit ist also 1 (für die erste Anfrage). Der Adversary lehnt jedoch die erste Anfrage ab und akzeptiert die $N-1$ folgenden, was ihm Profit $N-1$ einträgt.

(b) Die Bandbreiteneinschränkung (Annahme 6.6) ist nicht erfüllt.

(c) Sei $E_\ell = \{e_1, \ldots, e_{2^\ell - 1}\}$. Zwei Level ℓ-Anfragen sind entweder kantendisjunkt oder sie haben eine eindeutige gemeinsame Kante aus E_ℓ. Für jede Level ℓ-Anfrage, die CRS akzeptiert, kann OPT also auch höchstens eine Level ℓ-Anfrage akzeptieren. Somit gilt $c_\ell \geq o_\ell$.

Für den Erwartungswert der von CRS akzeptierten Anfragen folgt:

$$\mathbb{E}[\mathsf{CRS}(\sigma)] = \sum_{\ell=1}^{p} \frac{1}{p} c_\ell \geq \frac{1}{p} \sum_{\ell=1}^{p} o_\ell = \frac{1}{p} \mathsf{OPT}(\sigma).$$

Daher ist CRS $p = \log N$-kompetitiv.

Kapitel 9

Aufgabe 9.1

(a) Sei $\sigma = r_1, \ldots, r_n$ eine beliebige Auftragsfolge und $r_n = (t_n, x_n)$ die letzte Anfrage. Sei s die Position des REPLAN-Servers zum Zeitpunkt t_n. O. B. d. A. sei $s \geq 0$ (der andere Fall verläuft analog). Sei x^+ die maximale Koordinate einer Anfrage in σ. Sei ferner x^- das Minimum von 0 und der minimalen Koordinate einer Anfrage in σ. Dann gilt $\mathsf{OPT}(\sigma) \geq 2(x^+ + |x^-|)$ und $\mathsf{OPT}(\sigma) \geq t_n$.

Es gilt $s \in [x^-, x^+]$. Ab dem Zeitpunkt t_n benötigt der REPLAN-Server noch höchstens $|s - x^+| + x^+ + 2|x^-| = x^+ - s + x^+ + 2|x^-| \leq 2(x^+ + |x^-|) \leq \mathsf{OPT}(\sigma)$ Zeit (Bewegung des Servers von s zu x^+, dann zu 0 und letztendlich zu x^- und wieder zu 0). Somit ist die Gesamtzeit des REPLAN-Servers durch $t_n + \mathsf{OPT}(\sigma) \leq 2\,\mathsf{OPT}(\sigma)$ beschränkt.

(b) Wir zeigen, daß MRIN $3/2$-kompetitiv ist. Dazu benutzen wir Induktion nach der Anzahl n der Anfragen in der Folge σ. Die Behauptung ist für $n \leq 1$ offenbar richtig.

Sei $\sigma = r_1, \ldots, r_n = \sigma' r_n$ eine Eingabefolge mit $n \geq 2$ Anfragen, von denen $r_n = (t, x)$ die letzte ist. Wenn $t = 0$ gilt, dann ist MRIN offensichtlicherweise $3/2$-kompetitiv. Daher nehmen wir an, daß $t > 0$ gilt. Sei $s(t)$ die Position des MRIN-Servers zum Zeitpunkt t. Außerdem sei f die Position der am weitesten von 0 entfernten Anfrage, die MRIN zum Zeitpunkt t noch nicht bearbeitet hat.

Wenn $x \leq f$, dann sind die Kosten von MRIN auf σ gleich denen auf σ', der Folge der ersten $n-1$ Anfragen. Neue Anfragen können die Offline-Kosten nie verringern. Daher folgt die Behauptung in diesem Fall aus der Induktionsannahme.

Sei nun $f < x$. In diesem Fall ist zum Zeitpunkt t die Anfrage in x die am weitesten von 0 entfernte. MRIN beendet seine Arbeit nicht später als zum Zeitpunkt $t + 2x$, d.h. $\text{MRIN}(\sigma) \leq t + 2x$. Andererseits gilt $\text{OPT}(\sigma) \geq \max\{t + x, 2x\}$. Somit folgt:

$$\frac{\text{MRIN}(\sigma)}{\text{OPT}(\sigma)} \leq \frac{t+x}{\text{OPT}(\sigma)} + \frac{x}{\text{OPT}(\sigma)} \leq \frac{t+x}{t+x} + \frac{x}{2x} = \frac{3}{2}.$$

Daher ist MRIN $3/2$-kompetitiv.

Die Kompetitivität von MRIN ist bestmöglich für deterministische Algorithmen. Sei ALG ein beliebiger c-kompetitiver Algorithmus für das OLTSP in $\mathbb{R}_{\geq 0}$. Der Adversary gibt zum Zeitpunkt 0 eine Anfrage im Punkt 1. Sei T der Zeitpunkt, zu dem der Server von ALG diese Anfrage bearbeitet hat und wieder in den Nullpunkt zurückgekehrt ist. Wenn $T \geq 3$, dann gibt der Adversary keine weiteren Anfragen. ALG ist in diesem Fall höchstens $3/2$-kompetitiv, da die einzelne Anfrage in zwei Zeiteinheiten bearbeitet werden kann.

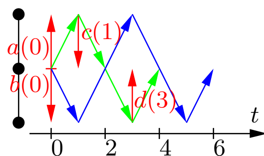
Sei also $T \leq 3$. In diesem Fall gibt der Adversary zum Zeitpunkt T eine Anfrage im Punkt T. Die Kosten von ALG sind also mindestens $T + 2T = 3T$. Andererseits kann der Adversary die Folge in $2T$ Zeiteinheiten abarbeiten.

Kapitel 12

Aufgabe 12.2

Wir betrachten einen vollständigen Digraphen $D = (V, A)$ auf 5 Knoten $o(0), a(0), b(0), c(1), d(3)$ (Generierungszeit in Klammern, Server startet in o) mit folgenden Bogengewichten (alle nicht aufgeführten Gewichte seien null), wobei wir o. B. d. A. annehmen, daß der Online-Server zuerst Knoten a ansteuert (sonst erzeugen wir c mit entsprechend vertauschten Kosten):

$$\begin{aligned} w(0,c) &= 1 & w(0,d) &= 1, \\ w(a,b) &= 1 & w(b,a) &= 1, \\ w(a,c) &= 2 & w(c,b) &= 1, \\ w(b,d) &= 2 & w(d,a) &= 1. \end{aligned}$$



Eine Transportsequenz mit Offline-Optimum (grün) und Online-Ergebnis (blau).

Offline kann man eine kostenlose Tour finden. In dem Moment, in dem der Online-Server a erreicht, wird c erzeugt. Egal, welchen Auftrag wir als nächstes nehmen, wir müssen eine Leerfahrt in Kauf nehmen. Siehe Abbildung C.1 für eine Skizze.

Dieses Beispiel ist das OLATSP-Modell einer Anforderungssequenz für eine eindimensionale Transportbewegung, daher die speziellen Gewichte.

Aufgabe 12.4

Nein. . . .

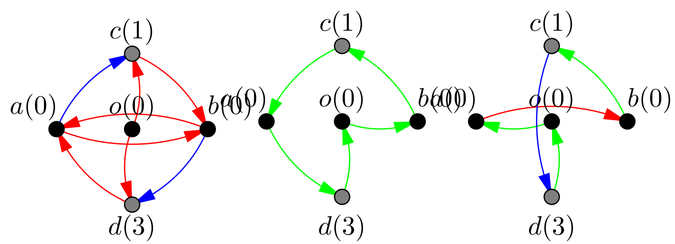


Abbildung C.1: Eine Instanz von OLATSP: Grüne Bögen kosten nichts, rote eine und blaue zwei Mark. Die Tour der optimalen Offline-Lösung in der Mitte kostet nichts.

Literaturverzeichnis

[1] N. Ascheuer, *Amsel—a modelling and simulation environment library*, Online-Documentation at `http://www.zib.de/ascheuer/AMSEL.html`.

[2] ______, *Hamiltonian path problems in the on-line optimization of flexible manufacturing systems*, Ph.D. thesis, Technische Universität Berlin, 1995.

[3] N. Ascheuer, M. Grötschel, S. O. Krumke, and J. Rambau, *Combinatorial online optimization*, Proceedings of the International Conference of Operations Research (OR'98), Springer, 1998, pp. 21–37.

[4] N. Ascheuer, S. O. Krumke, and J. Rambau, *Competitive scheduling of elevators*, Preprint SC 98-34, Konrad-Zuse-Zentrum für Informationstechnik Berlin, November 1998, An improved version of the paper is to appear as [5] in the Proceedings of the 17th International Symposium on Theoretical Aspects of Computer Science.

[5] ______, *Online dial-a-ride problems: Minimizing the completion time*, Proceedings of the 17th International Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, vol. 1770, Springer, 2000, pp. 639–650.

[6] M. J. Atallah and S. R. Kosaraju, *Efficient solutions to some transportation problems with applications to minimizing robot arm travel*, SIAM Journal on Computing **17** (1988), no. 5, 849–869.

[7] G. Ausiello, E. Feuerstein, S. Leonardi, L. Stougie, and M. Talamo, *Serving request with on-line routing*, Proceedings of the 4th Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science, vol. 824, July 1994, pp. 37–48.

[8] ______, *Competitive algorithms for the traveling salesman*, Proceedings of the 4th Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science, vol. 955, August 1995, pp. 206–217.

[9] R. A. Baeza-Yates, J. C. Culberson, and G. J. E. Rawlings, *Searching in the plane*, Information and Computation **106** (1993), no. 2, 234–252.

[10] A. Borodin and R. El-Yaniv, *Online computation and competitive analysis*, Cambridge University Press, 1998.

[11] N. Christofides, *Worst-case analysis of a new heuristic for the traveling salesman problem*, Tech. report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA, 1976.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*, MIT Press, 1990.

[13] A. Fiat, Y. Rabani, and Y. Ravid, *Competitive k-server algorithms*, Proceedings of the 31st Annual IEEE Symposium on the Foundations of Computer Science, 1990, pp. 454–463.

[14] A. Fiat and G. J. Woeginger (eds.), *Online algorithms: The state of the art*, Lecture Notes in Computer Science, vol. 1442, Springer, 1998.

[15] R. Fleischer, *On the Bahncard problem*, Proceedings of the 4th International Conference on Computing and Combinatorics, Lecture Notes in Computer Science, vol. 1449, Springer, 1998, pp. 65–74.

[16] G. N. Frederickson and D. J. Guan, *Nonpreemptive ensemble motion planning on a tree*, Journal of Algorithms **15** (1993), no. 1, 29–60.

[17] G. N. Frederickson, M. S. Hecht, and C. E. Kim, *Approximation algorithms for some routing problems*, SIAM Journal on Computing **7** (1978), no. 2, 178–193.

[18] M. R. Garey and D. S. Johnson, *Computers and intractability (a guide to the theory of NP-completeness)*, W.H. Freeman and Company, New York, 1979.

[19] M. Grötschel, L. Lovász, and A. Schrijver, *Geometric algorithms and combinatorial optimization*, Springer-Verlag, Berlin Heidelberg, 1988.

[20] D. Hauptmeier, S. O. Krumke, and J. Rambau, *The online dial-a-ride problem under reasonable load*, Preprint SC 99-08, Konrad-Zuse-Zentrum für Informationstechnik Berlin, March 1999, To appear as [21] in the Proceedings of the 4th Italian Conference on Algorithms and Complexity.

[21] ——, *The online dial-a-ride problem under reasonable load*, Proceedings of the 4th Italian Conference on Algorithms and Complexity, Lecture Notes in Computer Science, vol. 1767, Springer, 2000, pp. 125–136.

[22] D. Hauptmeier, S. O. Krumke, J. Rambau, and H.-C. Wirth, *Euler is standing in line*, Proceedings of the 25th International Workshop on Graph-Theoretic Concepts in Computer Science, Ascona, Switzerland, Lecture Notes in Computer Science, vol. 1665, Springer, June 1999, pp. 42–54.

[23] D. S. Hochbaum (ed.), *Approximation algorithms for NP-hard problems*, PWS Publishing Company, 20 Park Plaza, Boston, MA 02116–4324, 1997.

[24] N. Kamin, *On-line optimization of order picking in an automated warehouse*, Ph.D. thesis, Technische Universität Berlin, 1998.

[25] E. Koutsoupias and C. Papadimitriou, *On the k-server conjecture*, Journal of the ACM **42** (1995), no. 5, 971–983.

[26] M. Manasse, L. A. McGeoch, and D. Sleator, *Competitive algorithms for online problems*, Proceedings of the 20th Annual ACM Symposium on the Theory of Computing, 1988, pp. 322–333.

[27] R. Motwani and P. Raghavan, *Randomized algorithms*, Cambridge University Press, 1995.

[28] C. M. Papadimitriou, *Computational complexity*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994.

[29] H.-J. Siegert, *Simulation zeitdiskreter systeme*, Oldenbourg, München, Wien, 1991.

[30] D. D. Sleator and R. E. Tarjan, *Amortized efficiency of list update and paging rules*, Communications of the ACM **28** (1985), no. 2, 202–208.

[31] A. P. A. Vestjens, *On-line machine scheduling*, Ph.D. thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1994.