Kompetitive Analyse für Online-Algorithmen — Eine kommentierte Bibliographie

Teil I

Prof. Thomas Ottmann   Dr. Sven Schuierer   Christoph A. Hipke

Institut für Informatik
Albert-Ludwigs-Universität Freiburg

August 1994

Inhaltsverzeichnis

Kapitel 1

Einleitung

I mean how do you know what you're going to do till you do *it? The answer is, you don't.*
— J.D.Salinger *The Catcher in the Rye*

In vielen Anwendungen stellt sich das Problem, daß ein Algorithmus eine Folge von Anfragen bearbeiten muß und bei der Bearbeitung jeder Anfrage keine Information über die noch folgenden Anfragen hat. Er muß seine Entscheidungen also ohne Kenntnis der Zukunft treffen. Solche Probleme heißen *Online-Probleme*.

Es gibt viele Beispiele für Online-Probleme, man denke nur an dynamische Datenstrukturen, die Verwaltung eines Speicher- oder Festplattencaches, die Steuerung von Aufzügen in einem Gebäude oder das Finden von Verbindungen in einem Kommunikationsnetzwerk.

Online-Probleme in diesem Sinne sind nicht neu und wurden schon seit geraumer Zeit untersucht. Neu ist jedoch das Konzept der *kompetitiven Analyse*, das von Sleator und Tarjan [ST85] eingeführt wurde und einen Online-Algorithmus nicht mit einem anderen Online-Algorithmus, sondern mit einem optimalen Offline-Algorithmus vergleicht, der dasselbe Problem löst.

Die Stärke dieses Konzepts zeigt sich darin, daß es universell anwendbar ist, während ein direkter Vergleich zweier Online-Algorithmen nicht immer zu Resultaten führt. Denn ein Algorithmus A könnte auf einer Folge von Anfragen bessere Ergebnisse erzielen und ein anderer Algorithmus B auf einer anderen Folge. Ein Vergleich mit einem Offline-Algorithmus ergibt zum einen eine stärkere Aussage, die im Spezialfall natürlich auch für Online-Algorithmen gilt, zum anderen ist er oft einfacher zu erhalten, da man für viele Probleme optimale Offline-Algorithmen kennt.

Weiterhin ist eine kompetitive Analyse aussagekräftiger als eine worst-case oder average-case Analyse, es werden zum Beispiel keine Annahmen über die Verteilung der Anfragen gemacht wie in einer average-case Analyse. Kompetitivität bedeutet daher auch, daß der Online-Algorithmus Eigenschaften der Verteilung, wie lokale Häufung, bemerken muß, um im Vergleich nicht zu schlecht abzuschneiden.

Die Methode der kompetitiven Analyse läßt sich auch auf den Fall randomisierter Algorithmen übertragen. Dann muß jedoch genauer spezifiziert werden, wie der Offline-Algorithmus, mit dem verglichen wird, beschaffen ist. Hier benutzt man das Modell eines *Gegenspielers*, der eine Anfragefolge für den Online-Algorithmus erzeugt und diese auch selbst bearbeitet. Man unterscheidet dann zwischen unwissenden Gegenspielern, die die Anfragen im voraus erzeugen müssen, und adaptiven Gegenspielern, die ihre Anfragen von den Antworten des Online-Algorithmus abhängig machen können.

Diese Arbeit möchte einen Einblick geben in die Gebiete, die im Bereich der Online-Algorithmen untersucht werden, und die Ergebnisse, die bereits erzielt wurden. Da die kompetitive Analyse ein so allgemeines und viel benutztes Prinzip ist, kann natürlich kein Anspruch auf Vollständigkeit erhoben werden. Wir beschränken uns weitgehend auf neuere Arbeiten aus dem Gebiet der Online-Algorithmen, unsere Schwerpunkte bilden dabei die Darstellung der Grundlagen der kompetitiven Analyse und der theoretischen Modelle, sowie die Untersuchung geometrischer Online-Algorithmen. In Kapitel 2 werden zunächst die grundlegenden Definitionen dargelegt und einige theoretische Fragestellungen untersucht. Es schließt sich in Kapitel 3 und 4 eine Übersicht über die wichtigsten derzeitigen Anwendungsgebiete der kompetitiven Analyse an, in der exemplarisch einige Online-Probleme und kompetitive Lösungsstrategien erläutert werden. Das fünfte Kapitel ist Online-Algorithmen mit Voraussicht gewidmet, hier wird untersucht, wie sich die Ergebnisse aus den vorhergehenden Abschnitten verändern, wenn dem Online-Algorithmus nicht nur die nächste, sondern noch weitere Anfragen im voraus bekannt sind.

Insgesamt liegt uns daran, die untersuchten Probleme darzulegen, und die Ergebnisse, die erzielt wurden, im Überblick zu präsentieren. Wo es sich anbietet, wird auch die Analyse der Algorithmen genauer erläutert, jedoch konnten wir längere und kompliziertere Analysen meist nicht aufnehmen, hier wird der Leser auf die Literatur verwiesen. Diese ist am Ende eines jeden Abschnitts aufzufinden.

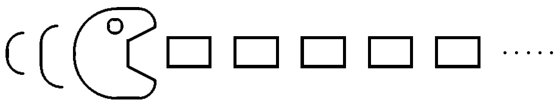

Abbildung 1.1: Ein Online-Algorithmus bei der Arbeit.

Kapitel 2

Grundlagen

2.1 Kompetitivität

Im Verhältnis zu ihrer Größe ist die
gewöhnliche Hausspinne achtmal schneller
als der schnellste Olympia-Sprinter.
— H. HAEFS *Handbuch des nutzlosen Wissens*

Wie bereits in der Einleitung angedeutet, vergleicht man bei der kompetitiven Analyse einen Online-Algorithmus A mit jedem Offline-Algorithmus. Da es in den meisten Fällen einen optimalen Offline-Algorithmus OPT gibt, genügt es dann, A mit diesem Algorithmus zu vergleichen.

Man kann dies auch so deuten, daß das Verhältnis des Ergebnisses von A zum optimalen Ergebnis betrachtet wird. Wir formulieren das Ergebnis hier zunächst als Kostenfunktion, die minimiert werden soll. Die Definitionen übertragen sich analog auf eine zu maximierende Gewinnfunktion.

Allgemein nennt man nun den Online-Algorithmus A *kompetitiv*, wenn es eine Konstante c gibt, so daß die Kosten von A für jede beliebige Folge von Anfragen nicht mehr als das c-fache der Kosten betragen, die ein optimaler Offline-Algorithmus benötigt.

2.1.1 Deterministische Algorithmen

Für deterministische Algorithmen bereitet die exakte Definition der Kompetitivität von Online-Algorithmen keine Probleme. Bezeichnet σ eine Folge von Anfragen, und C_X die Kostenfunktion für einen Algorithmus X, so heißt ein Algorithmus A *c-kompetitiv*, wenn

$$C_A(\sigma) \leq c \cdot C_S(\sigma) + a$$

für alle Anfragefolgen σ und alle Offline-Algorithmen S ist. Dabei ist zunächst noch nicht festgelegt, ob die Kostenfunktion die Rechenzeit des Algorithmus oder eine andere Größe widerspiegelt. Wichtig ist, daß die Ungleichung für *alle* Anfragefolgen gilt und nicht nur im Erwartungswert für eine Wahrscheinlichkeitsverteilung der Anfragefolgen. Die additive Konstante a darf nur von der Anfangskonfiguration des Problems abhängen.

Vielfach verzichtet man in der Literatur auf diese additive Konstante, dann ist der *kompetitive Faktor* c eines Online-Algorithmus A gegeben durch

$$c = \max_S \max_\sigma C_A(\sigma)/C_S(\sigma) = \max_\sigma C_A(\sigma)/C_{opt}(\sigma).$$

Andererseits hilft die additive Konstante in einigen Fällen, Spezialfälle für sehr kurze Anfragefolgen, bzw. sehr geringe Kosten des Gegenspielers zu vermeiden. Ein Beispiel dafür zeigt der Abschnitt 4.3.1.

Die gleiche Definition läßt sich anschaulich ausdrücken mit dem Modell eines *Gegenspielers* (*adversary*), das heißt einem Algorithmus ADV, der zum einen eine Folge von Anfragen σ erzeugt und sie zum anderen auch selbst bearbeitet. Ein Algorithmus A heißt dann c-kompetitiv, wenn er zur Bearbeitung der von einem beliebigen Gegenspieler ADV erzeugten Folge höchstens das c-fache der Kosten verursacht, die ADV selbst erzeugt.

Allgemein lassen sich verschiedene Arten von Gegenspielern betrachten, nämlich solche, die die Folge σ *online*, d.h. jede Anfrage direkt nach deren Erzeugung, oder *offline*, d.h. alle Anfragen, nachdem bereits die gesamte Folge σ erzeugt und somit bekannt ist, bearbeiten.

Diese Unterscheidung spielt jedoch bei deterministischen Online-Algorithmen zunächst noch keine Rolle, da die Entscheidungen des Online-Algorithmus hier ohnehin vorhersehbar sind.

2.1.2 Randomisierte Algorithmen

Man nennt Algorithmen *randomisiert*, wenn ihre Entscheidungen außer von ihrer Eingabe noch von der Ausgabe eines Zufallsgenerators abhängen. Solche Algorithmen haben also die Möglichkeit, eine imaginäre Münze zu werfen und je nach Ergebnis zu verzweigen.

Will man den Begriff der Kompetitivität auf randomisierte Algorithmen übertragen, so verwendet man statt der Kostenfunktion den Erwartungswert der Kosten. Ein randomisierter Algorithmus heißt also kompetitiv, wenn die erwarteten Kosten innerhalb eines konstanten Faktors der erwarteten Kosten des Gegenspielers liegen.

Dabei muß man allerdings unterscheiden zwischen einem Gegenspieler, dem alle zufälligen Entscheidungen, die der Algorithmus A bei der Bearbeitung der Anfragen ausführt, mitgeteilt werden, und einem Gegenspieler, der dieses Wissen nicht hat. Ersterer wird *adaptiv* genannt, letzterer *unwissend* (*oblivious*). Daß diese Unterscheidung notwendig ist, sieht man daran, daß A im Falle eines unwissenden Gegenspielers seine zufälligen Entscheidungen dazu nutzen kann, seinen Zustand vor dem Gegenspieler zu verheimlichen, so daß dieser die Folge σ nicht maximal ungünstig wählen kann.

Kompetitivität gegen einen unwissenden Gegenspieler entspricht der Forderung, daß

$$E[C_A(\sigma)] \leq c \cdot C_S(\sigma) + a$$

für alle Anfragen σ und für alle Offline-Algorithmen S ist. Für die Kosten von S wird hier kein Erwartungswert benötigt, da ohnehin alle Algorithmen S betrachtet werden.

Der Erwartungswert für die Kosten von A wird über alle Zufallsentscheidungen von A gebildet. Im Fall eines adaptiven Gegenspielers hängt die Folge σ von diesen Zufallsentscheidungen ab, daher kann man Kompetitivität dort nicht auf diese Art definieren. Stattdessen definiert man die Kosten $C_A(G)$ von A in Abhängigkeit von dem Gegenspieler G. Die Folge σ wird von diesem Gegenspieler erzeugt und ist somit implizit gegeben. Umgekehrt hängen im allgemeinen die Kosten $C_G(A)$ des Gegenspielers vom Algorithmus A ab. Insgesamt nennen wir einen randomisierten Algorithmus A c-kompetitiv gegen einen adaptiven Gegenspieler, falls

$$E[C_A(G)] \leq c \cdot C_G(A) + a$$

für alle Gegenspieler G gilt.

Leicht einzusehen ist, daß ein adaptiver Gegenspieler im Falle eines deterministischen Algorithmus keine Vorteile gegenüber einem unwissenden Gegenspieler hat, da die Entscheidungen von A nur von der Folge σ abhängen.

Im Falle eines unwissenden Gegenspielers kann man sich ebenso gut vorstellen, daß dieser die Folge der Anfragen bereits ganz erzeugt hat, bevor der Online-Algorithmus die Anfragen einzeln erhält. Hier braucht man also nicht zwischen Online- und Offline-Gegenspielern zu unterscheiden.

Insgesamt ergeben sich vier verschiedene Arten von Kompetitivität:

- Kompetitivität deterministischer Online-Algorithmen
- Kompetitivität randomisierter Online-Algorithmen gegen unwissende Gegenspieler
- Kompetitivität randomisierter Online-Algorithmen gegen adaptive Online-Gegenspieler
- Kompetivitität randomisierter Online-Algorithmen gegen adaptive Offline-Gegenspieler

Bei dieser Aufzählung wächst nach unten hin die Stärke des Gegenspielers. Einige Aussagen über das Verhältnis der Gegenspieler untereinander werden wir in Abschnitt 2.2.4 gewinnen.

2.1.3 Ein Beispiel

Ein typisches Online-Problem ist das *Bin Packing Problem.* Eine Menge von Paketen mit Größen $1 > r_i \in \mathbb{R}$ soll in Behälter der Größe 1 verpackt werden. Dabei sollen möglichst wenige Behälter verwendet werden. Im Online-Problem erscheinen die Pakete sequentiell und müssen in einen Behälter verpackt werden, bevor das nächste Paket erscheint. Wir möchten hier nur einen kurzen Eindruck eines Online-Problems vermitteln und werden daher nicht genauer auf das Bin Packing Problem eingehen.

Die Anfragen entsprechen in diesem Problem den Paketen, die Kosten der Anzahl der Behälter, die ein Algorithmus zum Verpacken benötigt. Ein optimaler Offline-Algorithmus kann leicht angegeben werden. Im einfachsten Fall probiert er alle Möglichkeiten durch.

Sehen wir uns den einfachen Online-Algorithmus FIRST-FIT an, der alle Behälter nacheinander betrachtet und das Paket in den ersten Behälter packt, in dem noch genug Platz ist.

Satz 2.1.1 FIRST-FIT *ist 2-kompetitiv, d.h. der Algorithmus benutzt höchstens zweimal so viele Behälter wie die optimale Verpackung.*

Beweis: Zu jedem Zeitpunkt kann nur ein Behälter halb leer sein, da sonst der Inhalt des einen halb leeren Behälters in einen anderen halb leeren Behälter passen würde und der Algorithmus FIRST-FIT ihn auch dort hin gepackt hätte. Aus dieser Tatsache ist sofort ersichtlich, daß FIRST-FIT 2-kompetitiv ist. □

Durch einen komplizierteren Beweis läßt sich zeigen, daß FIRST-FIT sogar $\frac{17}{10}$-kompetitiv ist. Der folgende Satz zeigt, daß es Online-Algorithmen nicht möglich ist, wirklich die optimale Anzahl von Behältern zu erreichen.

Satz 2.1.2 *Es gibt Anfragefolgen σ mit minimaler Anzahl von Behältern m, für die jeder Online-Algorithmus mindestens $\frac{4}{3}m$ Behälter benötigt.*

Beweis: Nehmen wir an, es gäbe einen c-kompetitiven Algorithmus A mit $c < \frac{4}{3}$. Gegeben sei nun die Anfragefolge, die aus n Paketen der Größe $\frac{1}{2} - \epsilon$ und n Paketen der Größe $\frac{1}{2} + \epsilon$ besteht. Nach n Schritten ist die optimale Anzahl von Behältern $n/2$. Der Online-Algorithmus A habe b Behälter benutzt. Da A auch für diese Teilfolge kompetitiv sein muß, gilt $b/n < \frac{2}{3}$.

Nach Bearbeitung der gesamten Anfragefolge ist die optimale Anzahl von Behältern n. In allen Behältern, die A nach dem b-ten geöffnet hat, kann sich jedoch nur ein Paket befinden, da zwei große Pakete nicht in einen Behälter passen. In den ersten b Behältern können sich je zwei Pakete befinden. Somit ist die Anzahl der von A benutzten Behälter mindestens $2n-b$. Da A c-kompetitiv ist gilt weiterhin $(2n-b)/n < \frac{4}{3}$, umgeformt $b/n > \frac{2}{3}$. Dies ist ein Widerspruch zur obigen Abschätzung für b/n. □

2.1.4 Literatur — Bin Packing

[CGJ83] E. G. Coffman, M. R. Garey, and D. S. Johnson. Dynamic bin packing. *SIAM J. of Computing*, 12:227–258, 1983.

[Cha92] B. Chandra. Does randomization help in on-line bin packing? *Information Processing Letters*, 43(1):15–19, 1992.

[LC92] K. Li and K. H. Cheng. Heuristic algorithms for on-line packing in three dimensions. *J. of Algorithms*, 13:589–605, 1992.

2.2 Theoretische Aussagen

Ich sehe, daß Herrn Professor die Logik
des Denkens im Stich läßt.
Entweder es gibt Backen, oder es gibt keine.
— Witold Gombrowicz *Ferdydurke*

Will man allgemeine theoretische Aussagen über Online-Probleme erzielen, spielt zunächst die Wahl des Modells für Problemstellung und Algorithmus eine große Rolle. Wir stellen hier zwei Modelle vor, Task Systeme und Anfrage-Antwort-Spiele.

2.2.1 Task Systeme

Das Modell des *Task Systems* wurde zunächst von Borodin, Linial und Saks [BLS87] eingeführt. Ein Task System wird definiert durch zwei ganze Zahlen N und m, die Anzahl der Zustände und die Anzahl der Tasks (Anfragen), sowie einer $N \times N$ Übergangskostenmatrix $D = (d_{ij})$ und einer $N \times m$ Taskkostenmatrix $C = (c_{ik})$. Alle Kosten sind nicht-negativ, können jedoch auch den Wert ∞ annehmen. Für die Übergangsmatrix D gilt die Dreiecksungleichung, d.h. $d_{ij} \leq d_{ik} + d_{kj} \;\; \forall i, j, k$.

Die Anfragefolge ist in diesem Modell eine Folge von n Tasks. Ein Online-Algorithmus A startet im Zustand $s(0) = 1$ und entscheidet nun nach dem Erhalt jeder Anfrage $\sigma(t)$, welchen neuen Zustand $s(t) \in 1, \dots, N$ er nach Bearbeitung der Anfrage wählt. Die Kosten $c_A(\sigma)$ für den Algorithmus bei Bearbeitung einer Folge σ berechnen sich dann wie folgt:

$$c_A(\sigma) = \sum_{1 \leq t \leq n} d_{s(t-1),s(t)} + \sum_{1 \leq t \leq n} c_{s(t),\sigma(t)}$$

Der erste Term bezeichnet die Kosten für die Zustandsübergänge, der zweite die Kosten für die Bearbeitung der Tasks.

Task Systeme im Beispiel

Ein gutes Beispiel für ein Task System ist das Paging Problem[1]. Hierbei wird ein schneller Hauptspeicherbereich verwaltet, in dem k Seiten konstanter Länge aus einem langsameren Massenspeicher gepuffert werden können. Die Tasks entsprechen dann den Seiten im Massenspeicher, die in den Hauptspeicher gelesen werden müssen. Die Zustände sind alle k-Tupel $(b_1, \dots, b_k)$ von Seitennummern, die sich gerade im Hauptspeicher befinden. Die Übergangskosten zwischen zwei Zuständen a und b sind gegeben durch

$$d(a, b) = \sum_{i=1}^{k} \delta_i \quad \text{mit} \quad \delta_i = \begin{cases} 0 & \text{falls } a_i = b_i \\ 1 & \text{sonst} \end{cases}$$

[1] siehe auch Abschnitt 3.3.

Die Taskkosten für einen Zustand b und eine Anfrageseite m sind 0, falls es ein i mit $b_i = m$ gibt und ∞ sonst, denn der Zustandsübergang muß eben so gewählt werden, daß die Seite im Speicher liegt.

Die Grenzen dieses Modells sieht man jedoch deutlich, wenn man versucht, das eingangs erwähnte Bin Packing Problem durch Task Systeme zu modellieren. Dies scheitert daran, daß die Anzahl der Tasks, also der Paketgrößen, nicht endlich und daher auch die Zustandsmenge nicht endlich ist. Läßt man nur Paketgrößen zu, die Vielfaches einer kleinen Konstanten $\epsilon > 0$ sind, und gibt man eine obere Schranke für die Anzahl der benutzten Behälter vor, so läßt sich auch das Bin Packing Problem als Task System formulieren.

Ein Algorithmus für Task Systeme

Borodin *et al.* [BLS87] geben einen allgemeinen $(2N-1)$-kompetitiven deterministischen Online-Algorithmus für Task Systeme an. Die Autoren zeigen auch eine untere Schranke von $(2N-1)$ für den deterministischen Fall, so daß der Algorithmus optimal ist.

Daß die Kompetitivität hier von der Anzahl N der Zustände abhängt, ist natürlich ein Nachteil. Die untere Schranke zeigt jedoch, daß die Allgemeinheit des Modells keinen besseren kompetitiven Faktor zuläßt. Um bessere Algorithmen zu finden, muß man somit speziell das untersuchte Problem analysieren — eine Tatsache, die nicht weiter verwundert.

2.2.2 Anfrage-Antwort-Spiele

Ein noch allgemeineres Modell für die Bearbeitung von Problemen durch Online-Algorithmen definieren Ben-David *et al.* [BDBK+90]. Ein *Anfrage-Antwort-Spiel* zwischen einem Online-Algorithmus und einem Gegenspieler besteht aus einer Anfrage-Menge R und einer endlichen Antwort-Menge S, sowie einer Familie von Kostenfunktionen f mit

$$f_n : R^n \times S^n \to \mathbb{R} \cup \{\infty\} \quad \forall n \in 0, 1, \ldots.$$

Es wird somit, anders als bei den Task Systemen, kein *Zustand* des Systems definiert, dieser ist allein aus den Antworten zu ersehen.

Ein *deterministischer Online-Algorithmus* A ist in diesem Modell eine Folge von Funktionen $a_i : R^i \to S$ für $i = 1, 2, \ldots$. Für jede Folge von Anfragen $\sigma = (r_1, \ldots, r_n)$ ist $A(\sigma) = (s_1, \ldots, s_n)$ definiert durch $s_i = a_i(r_1, \ldots, r_i)$. Die *Kosten* von A bei Bearbeitung der Anfragen σ sind dann $c_A(\sigma) = f_n(\sigma, A(\sigma))$.

Der Online-Algorithmus A heißt c-kompetitiv, wenn

$$c_A(\sigma) \le c \cdot c_{opt}(\sigma) := \min\{f_n(\sigma, s) | s \in S^n\}.$$

Auch der Begriff des Gegenspielers wird in diesem Modell genauer definiert. Ein *unwissender Gegenspieler* ist einfach gegeben durch eine Folge σ von Anfragen. Seine Kosten sind die optimalen Kosten $c_{opt}(\sigma)$ zur Beantwortung von σ.

Ein *adaptiver Offline-Gegenspieler* Q ist eine Folge von Funktionen $q_n : S^n \to R \cup \{\texttt{stop}\}$ für $n = 0, 1, \ldots, d_Q$, wobei q_{d_Q} immer den Wert `stop` annimmt. Die Funktionen modellieren die Auswahl einer neuen Anfrage aufgrund der vorherigen Antworten des

Algorithmus. d_Q ist eine Obergrenze für die Anzahl der Anfragen, der Gegenspieler kann jedoch auch weniger Anfragen liefern.

Für einen Online-Algorithmus A und einen adaptiven Offline-Gegenspieler Q sind die Anfragefolge $\sigma(A,Q) = (r_1,\ldots,r_n)$ und die Antwortfolge $s(A,Q) = (s_1,\ldots,s_n)$ rekursiv definiert durch $r_{i+1} = q_i(s_1,\ldots,s_i)$ und $s_i = a_i(r_1,\ldots,r_i)$. Dabei ist $n = \min\{\, l \,|\, q_l(s_1,\ldots,s_l) = \texttt{stop}\}$. Die Kosten $c_A(Q)$ von A gegen Q sind $f_n(\sigma(A,Q), s(A,Q))$ und die Kosten $c_Q(A)$ des Gegenspielers Q gegen A sind $c_{opt}(\sigma(A,Q))$, d.h. die optimalen Kosten.

Ein *adaptiver Online-Gegenspieler* $G = (Q,g)$ ist nun ein adaptiver Offline-Gegenspieler mit einer zusätzlichen Folge g von Funktionen $g_i : S^i \to S$ für $n = 0,1,\ldots,d_Q$, die die Antworten des Gegenspielers auf die Anfrage $r_{i+1} = q_i(s_1,\ldots,s_i)$ modellieren. Die Antwortfolge $b = (b_1,\ldots,b_n)$ des Gegenspielers ist also gegeben durch $b_{i+1} = g_i(s_1,\ldots,s_i)$. Da $\sigma(A,G)$ unabhängig von g ist, definiert man $\sigma(A,G) := \sigma(A,Q)$, $a(A,G) := a(A,Q)$ und $c_A(G) := c_A(Q)$. Die Kosten $c_G(A)$ des Online-Gegenspielers G gegen A unterscheiden sich aber von den Kosten $c_Q(A)$ für Q gegen A. Sie sind durch $f_n(\sigma(A,G), b(A,G))$ gegeben.

Diese Definitionen sind vor allem im Fall eines randomisierten Online-Algorithmus von Bedeutung. Dann werden alle Werte, die von den Entscheidungen des Algorithmus abhängen, wie z.B. s_i und $c_Q(A)$, ersetzt durch Zufallsvariablen.

Ein randomisierter Online-Algorithmus A heißt c-kompetitiv gegen einen adaptiven Online- oder Offline-Gegenspieler Q, wenn

$$E[c_A(Q)] \leq c \cdot E[c_Q(A)].$$

Anfrage-Antwort-Spiele im Beispiel

Das Bin Packing Problem läßt sich leicht als Anfrage-Antwort-Spiel darstellen. Die Anfragemenge R entspricht dem halboffenen Intervall $(0,1]$, die Antworten sind natürliche Zahlen, die die Nummer des Behälters angeben, in den das Paket gepackt wurde. Dabei muß die Anzahl der benutzten Behälter wieder beschränkt sein. Jede Kostenfunktion f_i benutzt nur die Antworten aus S^i und zählt, wieviele Behälter tatsächlich benutzt wurden.

2.2.3 Ein Entscheidungsverfahren

Manasse, McGeoch und Sleator [MMS88] beschreiben ein Verfahren, mit dem allgemein entschieden werden kann, ob es für ein Online-Problem einen c-kompetitiven deterministischen Algorithmus für gegebenes c gibt. Betrachtet werden dabei nur Online-Probleme, die sich im Modell der Task Systeme formulieren lassen. Zusätzlich wird gefordert, daß die Kostenmatrizen C und D beschränkte, rationale Koeffizienten haben.

Die Bedeutung dieses Verfahrens liegt weniger in der praktischen Anwendung, da die Laufzeit exponentiell in N, der Anzahl der Zustände, ist, sondern in seiner theoretischen Aussage.

Zunächst wird der Begriff der *Residuen* eingeführt. Diese vergleichen die bisherigen Kosten eines c-kompetitiven Algorithmus A mit den bisher optimalen Kosten. Dazu bezeichne

$c_{opt}(t,s)$ die minimalen Kosten, die zur Bearbeitung der Folge der ersten t Tasks benötigt werden, wenn der Algorithmus im Zustand s endet. $c_{opt}(t,s)$ läßt sich für alle s leicht berechnen, wenn $c_{opt}(t-1,s)$ für alle s bekannt ist. Seien weiterhin $c_A(t)$ die tatsächlichen Kosten, die A bei Bearbeitung der ersten t Tasks verursacht hat. Der Residuenvektor $RES(t) = (RES(t,s_1),\ldots,RES(t,s_n))$ zur Zeit t wird dann gegeben durch

$$RES(t,s) = c \cdot c_{opt}(t,s) - c_A(t),$$

d.h. insbesondere muß stets gelten $RES(t,s) \geq -a$, wobei a die additive Konstante aus Abschnitt 2.1 darstellt. Falls notwendig schreiben wir auch $RES(\sigma_t, s)$ für $RES(t,s)$ um zu zeigen, auf welche Folge σ wir uns beziehen.

Während der Bearbeitung der Taskfolge kann der Algorithmus A ständig den Residuenvektor berechnen und mitführen. Nennen wir einen Algorithmus, der zwischen zwei Tasks nur den Residuenvektor $RES(t)$ und den aktuellen Zustand $s(t)$ speichert, einen *residuen-basierten* Algorithmus. Ein residuen-basierter Algorithmus speichert also insbesondere keine Information über schon bearbeitete Tasks, d.h. die Entscheidung, welcher neue Zustand nach einem Task gewählt wird, hängt nur von der Residue ab, nicht vom Task selbst.

Satz 2.2.1 *Wenn es für ein Task System T einen c-kompetitiven Algorithmus gibt, dann existiert für T auch ein c-kompetitiver residuen-basierter Algorithmus.*

Beweis: Sei A der gegebene c-kompetitive Algorithmus und A' der gesuchte residuen-basierte. Erhält A' einen Task $\sigma(i)$, so simuliert A' den Algorithmus A auf jeder Folge von Tasks mit Längen $0,1,2,\ldots,i-1$. Stimmt der Residuenvektor und Zustand von A nach Simulation der Taskfolge mit dem Residuenvektor und dem Zustand von A' überein, so wechselt A' nach Erhalt des Tasks $\sigma(i)$ in den Zustand, in den auch A wechseln würde, wenn er $\sigma(i)$ als nächsten Task nach der simulierten Folge erhielte.

Zu zeigen ist nun zunächst, daß der Algorithmus A' terminiert, daß es also immer eine Folge gibt, so daß A nach Bearbeitung der Folge den gleichen Residuenvektor wie A' hat. Nach dem Erhalt des ersten Tasks $\sigma(1)$ beendet die Taskfolge τ_0 mit Länge 0 bereits die Suche, also hält A'. Angenommen, es gibt eine Taskfolge τ_{i-1}, die die Suche nach Erhalt des i-ten Tasks beendet. Dann gleicht der Zustand $s(i)$ von A' bei Erhalt des $(i+1)$-ten Tasks $\sigma(i+1)$ dem Zustand von A nach Bearbeitung der Taskfolge $\tau_i = \tau_{i-1} + \sigma(i)$. Daß auch die Residuen von A nach Bearbeitung von τ_i den Residuen von A' nach Bearbeitung von σ_i gleichen, ist nicht unmittelbar klar. Denn die Folge τ_{i-1} muß ja nicht mit der Folge σ_{i-1} der Tasks, die A bereits bearbeitet hat, übereinstimmen. Vor Bearbeitung des i-ten Tasks gilt folgendes:

$$c \cdot c_{opt}(\tau_{i-1}, t) - c_A(\tau_{i-1})$$

$$= RES_A(i-1,t) = RES_{A'}(i-1,t)$$

$$= c \cdot c_{opt}(\sigma_{i-1}, t) - c_{A'}(\sigma_{i-1})$$

für alle Zustände t. Die tatsächlichen Kosten und die optimalen Kosten für A und A' müssen also nicht übereinstimmen, nur deren Differenz. Für die optimalen Kosten gilt jedoch:

$$c_{opt}(\sigma_i, t) = \min\{c_{opt}(\sigma_{i-1}, u) + d_{u,t} + c_{u,\sigma(i)} \mid u \text{ Zustand}\}$$
$$c_{opt}(\tau_i, t) = \min\{c_{opt}(\tau_{i-1}, u) + d_{u,t} + c_{u,\sigma(i)} \mid u \text{ Zustand}\}$$

Gilt nun $c_{opt}(\tau_{i-1}, u) - c_{opt}(\sigma_{i-1}, u) = \Delta$ für alle Zustände u, so wird für jeden Zustand t jeweils der gleiche neue Zustand u minimale Kosten ergeben. In unserem Fall ist $\Delta := (c_A(\tau_{i-1}) - c_{A'}(\sigma_{i-1}))/c$. Es ergibt sich somit

$$c_{opt}(\tau_i, t) - c_{opt}(\sigma_i, t) = \Delta \quad \text{für alle Zustände t.} \tag{2.1}$$

Das heißt, für alle Zustände t gilt

$$\begin{aligned} RES_A(i,t) = c \cdot c_{opt}(\tau_i, t) - c_A(\tau_i) &\overset{(2.1)}{=} c \cdot c_{opt}(\sigma_i, t) + c \cdot \Delta - c_A(\tau_i) \\ &= c \cdot c_{opt}(\sigma_i, t) + c \cdot \Delta - c_A(\tau_{i-1}) + d_{s(i-1),s(i)} + c_{s(i),\sigma(i)} \\ &= c \cdot c_{opt}(\sigma_i, t) - c_{A'}(\sigma_{i-1}) + d_{s(i-1),s(i)} + c_{s(i),\sigma(i)} \\ &= c \cdot c_{opt}(\sigma_i, t) - c_{A'}(\sigma_i) = RES_{A'}(i,t). \end{aligned}$$

Die Residuen gleichen sich und die Suche wird daher auch bei Erhalt des $(i+1)$-ten Tasks beendet.

Jeder Residuenvektor von A' wird auch von A nach Bearbeitung irgendeiner Folge erhalten. Da die Residuen von A nach unten beschränkt sind, sind es auch die von A', d.h. A' ist ebenfalls c-kompetitiv. □

Diese Tatsache schränkt nun umgekehrt die Suche nach Algorithmen, die c-kompetitiv sind, schon etwas ein. Noch weiter führt der Begriff der *Pseudo-Residuen* oder kurz *p-Residuen*. Diese bilden jeweils eine untere Schranke für die entsprechenden Residuen, da bestimmte zu hohe Residuenwerte verkleinert werden. Ein Residuenvektor könnte zwei Elemente $RES(t,s)$ und $RES(t,u)$ haben, so daß die Ungleichung

$$RES(t,s) > RES(t,u) + c \cdot d_{su} \tag{2.2}$$

gilt. In diesem Fall ist $RES(t,s)$ zu groß, um nachfolgende Residuen überhaupt noch zu beeinflussen. Denn für alle Zustände v gilt

$$RES(t+1, v) = \min\{RES(t,w) + c \cdot d_{wv} - c_A(t+1) \mid w \text{ Zustand}\}.$$

Aber aufgrund von Ungleichung (2.2) und der Dreiecksungleichung erhalten wir

$$RES(t,s) + c \cdot d_{sv} > RES(t,u) + c \cdot d_{su} + c \cdot d_{sv} \geq RES(t,u) + c \cdot d_{uv}.$$

p-Residuen entsprechen nun den Residuen, wobei man jedoch bestimmte Werte gegebenenfalls verkleinert, so daß immer die Ungleichung

$$RES(t,s) \leq RES(t,u) + c \cdot d_{su} \tag{2.3}$$

gilt. Für *p-residuen-basierte* Algorithmen, die Residuen mit solchen *abgeschnittenen* Werten mitführen, gilt der folgende Satz.

Satz 2.2.2 *Wenn es für ein Task System T einen c-kompetitiven Algorithmus gibt, dann existiert auch ein p-residuen-basierter c-kompetitiver Algorithmus für T, der zusätzlich die Ungleichung*

$$-(c+1)B \leq PRES(t,s) < (2c+1)B$$

für alle t und s erfüllt. Dabei bildet B eine obere Schranke für die Elemente d_{ij} der Matrix D, die als beschränkt vorausgesetzt ist.

Beweis: Angenommen, für jeden c-kompetitiven Algorithmus würde eine p-Residue irgendwann unter den Wert $-(c+1)B$ sinken. In diesem Fall wären alle abgeschnittenen Residuen aufgrund von Ungleichung (2.3) kleiner als $-B$. Der Algorithmus befindet sich also in einer schlechteren Situation als ganz zu Beginn, denn dort waren alle p-Residuen positiv. Wiederholt man die Folge von Anfragen, die zu dieser schlechten Situation geführt hat, so sinken die p-Residuen weiter, denn der einzige Vorteil für den Algorithmus ist, daß er sich beim zweiten Durchgang in einem günstigeren Anfangszustand befinden könnte. Die Kosten, um diesen günstigen Zustand zu erreichen, wären im ersten Durchgang $\leq B$ gewesen, nun sind sie gleich Null. Die Kosten für jeden weiteren Durchgang sind somit um höchstens B kleiner als die für den ersten Durchgang. Damit sinken die Residuen jedoch auch in allen weiteren Durchgängen um einen Wert größer als Null. Wird der Vorgang genügend oft wiederholt, sinken die p-Residuen beliebig tief, der Algorithmus ist also nicht c-kompetitiv, im Widerspruch zur Annahme. Somit gibt es einen c-kompetitiven Algorithmus A, dessen p-Residuen nie kleiner als $-(c+1)B$ werden.

Analog zum Beweis von Satz 2.2.1 konstruiert man nun einen p-residuen-basierten Algorithmus A''. Dieser startet mit den abgeschnittenen Residuen von A und simuliert dann zunächst A auf länger werdenden Taskfolgen, bis die Residuen von A nach Abschneiden der zu hohen Werte mit den Residuen von A'' übereinstimmen und sich A und A'' im gleichen Zustand befinden. A'' wechselt in den Zustand, in den auch A nach Bearbeitung von $\sigma(i)$ wechseln würde, und schneidet dann seine Residuen ab. Ist daraufhin eine p-Residue von A'' größer als $(2c+1)B$, dann wechselt A'' in den Anfangszustand zurück und setzt seine Residuen auf die abgeschnittenen Anfangsresiduen von A.

Zu zeigen ist noch, daß dieses Zurückgehen zum Anfangszustand legal ist, d.h. die p-Residuen dadurch nur verkleinert werden und somit immer noch eine untere Schranke für die tatsächlichen Residuen bilden. Angenommen also, eine der p-Residuen ist größer als $(2c+1)B$. Der Wechsel aus dem Zustand von A'' in den Anfangszustand von A verringert die p-Residuen um höchstens B, also ist dann immer noch eine der p-Residuen mindestens $2cB$. Die Differenz zweier p-Residuen kann nie größer als cB werden, da die p-Residuen sonst abgeschnitten werden. Somit sind alle Residuen größer gleich cB. Im Anfangszustand von A ist jedoch eine Residue gleich Null und somit sind die Anfangsresiduen alle kleiner gleich cB, woraus die Behauptung folgt.

Die p-Residuen von A'' liegen dann innerhalb der geforderten oberen und unteren Schranke. Der Beweis, daß A'' terminiert, verläuft analog zum Beweis von Satz 2.2.1. □

Da die p-Residuen $PRES(t,s)$ Linearkombinationen der Übergangskosten d_{ij} und der Taskkosten c_{ik} sind und diese Werte rational sind, gibt es nur endlich viele mögliche

Werte von p-Residuen und somit nur endlich viele Algorithmen, die die Bedingungen aus Satz 2.2.2 erfüllen.

Diese endliche Untermenge von Algorithmen läßt sich durchsuchen, und somit kann festgestellt werden, ob ein c-kompetitiver Algorithmus für ein gegebenes Task System T existiert. Das Entscheidungsverfahren benötigt dafür allerdings exponentielle Zeit in N, der Anzahl der Zustände.

2.2.4 Vergleich der Gegenspieler

Da Online-Algorithmen gegen verschiedene Gegenspieler analysiert werden können, ergibt sich die Frage nach einem Zusammenhang der verschiedenen Arten von Kompetitivität, die in Abschnitt 2.1 definiert wurden. Einige Ergebnisse dazu finden sich in der Arbeit von Ben-David *et al.* [BDBK+90]. Als Modell für die Untersuchungen wird das Anfrage-Antwort-Spiel verwendet. Das erste Resultat zeigt die Stärke eines adaptiven Offline-Gegenspielers.

Satz 2.2.3 *Falls es für ein Problem einen randomisierten Online-Algorithmus gibt, der c-kompetitiv gegen jeden adaptiven Offline-Gegenspieler ist, dann gibt es auch einen c-kompetitiven deterministischen Online-Algorithmus.*

Beweis: Man betrachtet das Anfrage-Antwort-Spiel als Spiel zwischen zwei Personen Q, dem Gegenspieler, und A, dem Algorithmus. In jeder Runde liefert Q eine Anfrage, die A beantworten muß. Eine Position im Spiel ist ein Paar (σ, s) der bisherigen Anfragen und Antworten. f_n bezeichne wie in Abschnitt 2.2.2 die Kostenfunktion des Anfrage-Antwort-Spiels. Eine Position heißt *sofort gewinnend* für Q, wenn $f_n(\sigma, s) > c \cdot c_{opt}(\sigma) + a$. Eine Position heißt *gewinnend* für Q, wenn Q in einer endlichen Anzahl t von Schritten eine sofort gewinnende Position erreichen kann, wie immer A auch spielt. Insbesondere ist die Anfangsposition genau dann gewinnend, wenn es einen adaptiven Offline-Gegenspieler Q gibt, gegen den kein Algorithmus A c-kompetitiv ist.

Angenommen nun, die Anfangsposition sei gewinnend für Q. Ist A ein c-kompetitiver randomisierter Algorithmus, verteilt über deterministische Algorithmen A_x, so gilt für jeden Algorithmus A_x, daß $c_{A_x}(Q) > c \cdot c_Q(A_x) + a$ in einer endlichen Anzahl von Schritten t_x. Diese Anzahl ist jedoch für alle x beschränkt durch die maximale Anzahl d_Q von Anfragen, die der Gegenspieler ausführen darf. Durch Bilden des Erwartungswertes erhält man

$$E[c_{A_x}(Q)] > c \cdot E[c_Q(A_x)] + a,$$

also ist A nicht c-kompetitiv, im Widerspruch zur Annahme.

Zu zeigen bleibt nun, daß, wenn Q keine Gewinnstrategie hat, es eine Gewinnstrategie für A gibt und somit auch einen deterministischen c-kompetitiven Online-Algorithmus. (σ, s) ist eine gewinnende Position für Q genau dann, wenn es eine Anfrage r_{i+1} gibt, so daß für alle Antworten s_{i+1} die Position $(\sigma r_{i+1}, s s_{i+1})$ wieder gewinnend ist. In diese Aussage geht die Endlichkeit der Menge der Antworten ein. Ist also (σ, s) keine gewinnende Position für Q, dann gibt es für jede Anfrage r_{i+1} eine Antwort s_{i+1}, so daß $(\sigma r_{i+1}, s s_{i+1})$

erneut keine gewinnende Position für Q ist. Wählt A nun immer genau diese Antwort aus, so kann Q nie eine gewinnende Position erreichen, und A gewinnt. □

Mit anderen Worten ausgedrückt, bedeutet dies, daß ein adaptiver Offline-Gegenspieler so stark ist, daß Randomisierung nicht weiterhilft. Für adaptive Online-Gegenspieler läßt sich ein ähnlicher Satz formulieren.

Satz 2.2.4 *Gibt es für ein Problem einen randomisierten Online-Algorithmus A, der c-kompetitiv gegen jeden adaptiven Online-Gegenspieler ist, so gibt es auch einen c^2-kompetitiven deterministischen Online-Algorithmus.*

Dieser Satz folgt aus dem folgenden allgemeineren Satz unter Benutzung von Satz 2.2.3 und der Tatsache, daß aus Kompetitivät gegen adaptive Gegenspieler immer auch Kompetitivität gegen unwissende Gegenspieler folgt.

Satz 2.2.5 *Ist ein randomisierter Online-Algorithmus A c-kompetitiv gegen jeden adaptiven Online-Gegenspieler und gibt es einen weiteren randomisierten Online-Algorithmus H, der d-kompetitiv gegen jeden unwissenden Gegenspieler ist, dann ist A $(c \cdot d)$-kompetitiv gegen jeden adaptiven Offline-Gegenspieler.*

Beweis: Sei ein adaptiver Offline-Gegenspieler Q gegeben und sei A verteilt über deterministische Algorithmen A_x. Wir müssen zeigen, daß

$$E_x[c_{A_x}(Q)] \leq E_x[c \cdot d \cdot c_Q(A_x)].$$

Sei H der randomisierte Algorithmus, der d-kompetitiv gegen jeden unwissenden Gegenspieler ist, verteilt über H_y. Es gilt also

$$E_y[c_{H_y}(\sigma)] \leq d \cdot c_{opt}(\sigma) \tag{2.4}$$

für alle n und $\sigma \in R^n$.

Für jedes y definieren wir nun einen adaptiven Online-Gegenspieler $G_y = (Q, g_y)$, so daß für jeden deterministischen Online-Algorithmus F für die Antwortfolge des Gegenspielers G_y gilt $b(F, G_y) = H_y(\sigma(F, Q))$. Dieser Gegenspieler unterscheidet sich vom Offline-Gegenspieler Q also nur dadurch, daß er die selbst erzeugten Anfragen online beantwortet, und zwar genau so, wie H_y es tun würde.

Da A c-kompetitiv ist gegen jeden Online-Gegenspieler, ist A auch c-kompetitiv gegen G_y für alle y, d.h.

$$E_x[c_{A_x}(G_y)] \leq E_x[c \cdot c_{G_y}(A_x)].$$

Nach Bilden des Erwartungswertes erhält man

$$E_y[E_x[c_{A_x}(G_y)]] \leq E_y[E_x[c \cdot c_{G_y}(A_x)]]. \tag{2.5}$$

Für jedes y gilt weiterhin

$$\sigma(A_x, G_y) = \sigma(A_x, Q) =: \sigma_x, \tag{2.6}$$

da G_y die gleiche Anfragefolge wie Q erzeugt. Somit gilt

$$E_x[c_{A_x}(Q)] \stackrel{(2.6)}{=} E_y E_x[c_{A_x}(G_y)] \stackrel{(2.5)}{\leq} E_y[c \cdot E_x[c_{G_y}(\sigma_x)]]$$

$$= c \cdot E_x E_y[c_{H_y}(\sigma_x)] \stackrel{(2.4)}{\leq} c \cdot E_x[d \cdot c_{opt}(\sigma_x)]$$

$$= E_x[c \cdot d \cdot c_Q(A_x)]$$

□

Für Satz 2.2.4 geben die Autoren ein explizites Konstruktionsverfahren an, nicht jedoch für Satz 2.2.3. Hier konstruieren sie ein Verfahren, das $(1+\epsilon)c$-kompetitiv ist für beliebiges ϵ und sich somit dem theoretisch möglichen Wert beliebig nähert.

2.2.5 Offene Probleme

Läßt sich das Entscheidungsverfahren von Manasse, McGeoch und Slator auf eine größere Klasse von Online-Problemen übertragen?

Kann die Laufzeit des Verfahrens verbessert werden?

2.2.6 Literatur — Theoretische Aussagen

[BDBK+90] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the power of randomization in online algorithms. In *Proc. 22nd ACM Symp. on Theory of Computing*, pages 379–386, 1990.

[BLS87] A. Borodin, N. Linial, and M. Saks. An optimal online algorithm for metrical task systems. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 373–382, 1987.

[CDRS90] D. Coppersmith, P. Doyle, P. Raghavan, and M. Snir. Random walks on weighted graphs, and application to on-line algorithms (preliminary version). In *Proc. 22nd ACM Symp. on Theory of Computing*, pages 369–378, 1990.

[DM91] X. Deng and S. Mahajan. Infinite games: Randomization, computability, and applications to online problems (preliminary version). In *Proc. 24th ACM Symp. on Theory of Computing*, pages 289–298, 1991.

[MMS88] M. Manasse, L. A. McGeoch, and D. Sleator. Competitive algorithms for online problems. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 322–333, 1988.

[RS89] P. Raghavan and M. Snir. Memory vs. randomization in on-line algorithms. In *Proc. 16th Internat. Colloq. Autom. Lang. Program.*, 1989.

Kapitel 3

Das k-Server Problem und Varianten

In diesem Kapitel geben wir einige Anwendungen der kompetitiven Analyse an. Dabei beginnen wir mit dem sehr intensiv untersuchten k-Server Problem, das aufgrund seiner Allgemeinheit auch als Modell für andere Online-Probleme gewählt werden kann. Im Anschluß daran diskutieren wir verwandte Problemstellungen, die mehr oder weniger eng mit dem k-Server Problem zusammenhängen. Im zweiten Teil dieses Bandes folgt eine Untersuchung von graphentheoretischen oder geometrischen Online-Problemen.

3.1 Das k-Server Problem

Lancelot [...] *If I serve not him* [Bassanio],
I will run as far as God has any ground.
— William Shakespeare *The Merchant of Venice*

Im *k-Server Problem* hat der Online-Algorithmus die Aufgabe, k Server auf einem beliebigen metrischen Raum M entsprechend einer Folge von Anfragen zu bewegen. Die Anfragen entsprechen dabei einem Punkt $p \in M$, zu dem der Algorithmus dann einen Server bewegen muß. Die Kosten entsprechen der Distanz, um die der Server bewegt wird. Die Entscheidung, welcher Server bewegt wird, muß ohne Wissen über zukünftige Anfragen getroffen werden.

Das k-Server Problem läßt sich auch auf einem Graphen G betrachten. Hier ist die Menge der Punkte endlich, der Fall des Graphen ist jedoch insofern allgemeiner, als hier die Dreiecksungleichung nicht gelten muß und man auch asymmetrische Graphen betrachten kann.

3.1.1 Anwendungen

Das k-Server Problem ist aufgrund der frei wählbaren Metrik ein sehr allgemeines Problem, das verschiedene andere Probleme als Spezialfälle enthält.

Das *Paging Problem*, die Verwaltung eines schnellen Hauptspeicherbereichs, in dem Seiten aus einem langsamen Massenspeicher gepuffert werden, ist ein solcher Spezialfall.

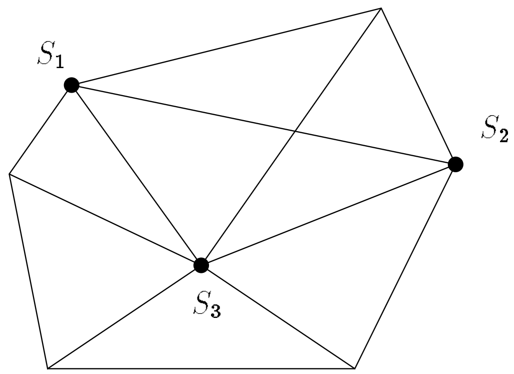

Abbildung 3.1: Das k Server Problem auf einem Graphen.

Man wählt hier die diskrete Metrik, d.h. gleiche Punkte haben den Abstand 0, und alle anderen Abstände sind gleich 1. Die Server entsprechen den Seitenrahmen, die im schnellen Speicher liegen, und die Punkte des metrischen Raums den Seiten im langsamen Speicher. Das Bewegen eines Servers zu einem Punkt entspricht dem Laden einer Seite in einen Seitenrahmen.

Ein weiteres Beispiel bildet die *Steuerung von Festplatten* mit zwei oder mehr Lese- und Schreibköpfen [MV91, CNV92]. Hier ist die Zuordnung noch natürlicher. Die Köpfe bilden die Server, die Platte den metrischen Raum.

3.1.2 Untere Schranken

Für den Fall eines deterministischen Online-Algorithmus auf einem symmetrischen Graphen mit mindestens $k+1$ Knoten bewiesen Manasse *et al.* [MMS88] eine untere Schranke von k für die Kompetitivität des Algorithmus.

Diese Schranke folgt aus der folgenden allgemeineren Aussage, die einen k-Server Online-Algorithmus mit einem h-Server Offline-Algorithmus vergleicht, wobei $h \leq k$.

Satz 3.1.1 *Sei A ein Online-Algorithmus für das symmetrische k-Server Problem auf einem Graphen mit mindestens $k+1$ Knoten. Dann gibt es für alle $i = 1, 2, \ldots$ eine Folge σ_i der Länge i und einen h-Server Offline-Algorithmus B, so daß*

$$c_A(\sigma_i) > \frac{k}{k-h+1} \cdot c_B(\sigma_i).$$

Beweis: O.B.d.A nehmen wir an, daß A jeweils nur höchstens einen Server pro Zug bewegt und die Server von A auf verschiedenen Punkten beginnen. Sei H der Teilgraph von G, der diese Punkte und einen weiteren Punkt enthält. Da A deterministisch ist, läßt sich eine Folge σ dadurch definieren, daß $\sigma(i)$ gerade der Punkt ist, der zum Zeitpunkt i nicht von einem Server von A belegt ist. Dann gilt für die Kosten von A

$$C_A(\sigma, t) = \sum_{i=1}^{t-1} d_{\sigma(i+1), \sigma(i)}.$$

Sei S eine Teilmenge der Knoten aus H mit h Knoten, die auch $\sigma(1)$ enthält. Wir definieren einen h-Server Offline-Algorithmus $B(S)$: Zu Beginn befinden sich die Server von B auf den Knoten in S. Eine Anfrage auf Punkt $\sigma(i)$ wird wie folgt bearbeitet. In jedem Fall befindet sich zu diesem Zeitpunkt ein Server auf $\sigma(i-1)$, dem Punkt der letzten Anfrage. Dieser Server wird nun zum Punkt $\sigma(i)$ bewegt, falls sich dort nicht schon ein Server befindet. Andernfalls wird kein Server bewegt.

Es gilt dann die folgende wichtige Aussage: Wird dieser Algorithmus auf verschiedenen Teilmengen S und T von H gestartet, so werden die Teilmengen $S(i)$ und $T(i)$ der Punkte, auf denen sich vor Bearbeitung der Anfrage $\sigma(i)$ Server befinden, nie gleich. Nehmen wir an, $S(i) \neq T(i)$. Falls $\sigma(i) \in S(i)$ und $\sigma(i) \in T(i)$, dann gilt $T(i+1) = T(i)$ und $S(i+1) = S(i)$, die Mengen bleiben verschieden. Liegt $\sigma(i)$ in keiner der beiden Mengen, so wird sowohl in $T(i+1)$ als auch in $S(i+1)$ $\sigma(i)$ weggelassen und $\sigma(i+1)$ hinzugefügt, die nicht-leere symmetrische Differenz ändert sich also nicht. Liegt schließlich $\sigma(i)$ nur in einer der beiden Mengen, so liegt $\sigma(i+1)$ nach Bearbeitung der Anfrage nur in einer der beiden Mengen, womit die Aussage gezeigt ist.

Nehmen wir an, man startet $B(S)$ auf allen Teilmengen S von H mit h Elementen, die $\sigma(1)$ enthalten. Es gibt $\binom{k}{h-1}$ solcher Mengen und die Anzahl der Mengen $S(i)$ bleibt konstant. Die Mengen $S(i)$ sind weiterhin gerade alle h-elementigen Teilmengen von H, die $\sigma(i)$ enthalten. Die erste Anfrage kostet alle Algorithmen nichts. Jede weitere Anfrage $i+1$ mit $i \geq 1$ kostet die Algorithmen, die auf $\sigma(i+1)$ keinen Server haben $d_{\sigma(i),\sigma(i+1)}$ und die anderen nichts. Es gibt also $\binom{k-1}{h-1}$ Algorithmen, die Kosten verursachen und die Gesamtkosten bis zum Zeitpunkt t sind

$$\sum_{i=1}^{t-1} \binom{k-1}{h-1} \cdot d_{\sigma(i),\sigma(i+1)}.$$

Der Erwartungswert für einen zufällig gewählten Algorithmus $B(S)$ ist daher

$$\frac{\binom{k-1}{h-1}}{\binom{k}{h-1}} \cdot \sum_{i=1}^{t-1} d_{\sigma(i),\sigma(i+1)} = \frac{k-h+1}{k} \sum_{i=1}^{t-1} d_{\sigma(i+1),\sigma(i)}.$$

Für eine Wahl von S muß der Algorithmus $B(S)$ die Eigenschaft haben, daß seine Kosten unter dem Erwartungswert liegen. Durch Vergleich der Kosten dieses Algorithmus $B(S)$ und der Kosten des Algorithmus A folgt die Behauptung. $\square$

Dieser Satz gilt jedoch nicht für randomisierte Online-Algorithmen. Hier zeigten Karloff, Rabani und Ravid [KRR91] eine untere Schranke von $\Omega(\log \log k)$ für alle metrischen Räume mit mindestens $k+1$ Punkten und eine untere Schranke von $\Omega(\log k)$ für genügend große metrische Räume.

Die bisherigen unteren Schranken gelten alle für symmetrische Graphen oder metrische Räume. Der nächste Satz zeigt, daß der asymmetrische Fall noch deutlich schwieriger ist [CKPV90].

Satz 3.1.2 *Es gibt keinen deterministischen kompetitiven 2-Server Algorithmus für asymmetrische metrische Räume.*

Beweis: Für $m > 1$ betrachten wir den gerichteten Graphen H_m auf $3m$ Knoten $\{x_0, y_0, z_0, \ldots, x_{m-1}, y_{m-1}, z_{m-1}\}$. Gerichtete Verbindungen bestehen zwischen folgenden Punktepaaren:

$$(x_i, z_i), (y_i, z_i), (z_i, x_{i+1}), (z_i, y_{i+1}) \quad \forall i = 0, \ldots, m-1.$$

Dabei rechnet man, wie auch im folgenden, falls nötig modulo m.

Sei M_m der asymmetrische metrische Raum auf der gleichen Knotenmenge, bei dem der Abstand zweier Knoten x und y gerade die Länge des kürzesten Pfades von x nach y in H_m ist. Es gilt dann $d(x_i, y_i) = d(y_i, x_i) = 2m$, aber $d(x_i, z_i) = 1 \neq 2m - 1 = d(z_i, x_i)$.

Für gegebenes c sei m so groß, daß $(2m-1)/4 > c$. Zu Anfang befinden sich alle Server am Punkt z_0. Der Gegenspieler G durchläuft nun eine Reihe von Phasen. Zu Beginn der Phase i befinden sich beide Server von G, sowie auch die Server des Online-Algorithmus A auf z_{i-1}. G erzeugt zuerst Anfragen auf den Punkten x_i und y_i und bearbeitet sie mit seinen beiden Servern. Dann erzeugt er solange weiter abwechselnd Anfragen auf x_i und y_i, bis A ebenfalls seine beiden Server auf x_i und y_i positioniert hat (oder bis dessen Kosten c_A ohnehin zu groß geworden sind).

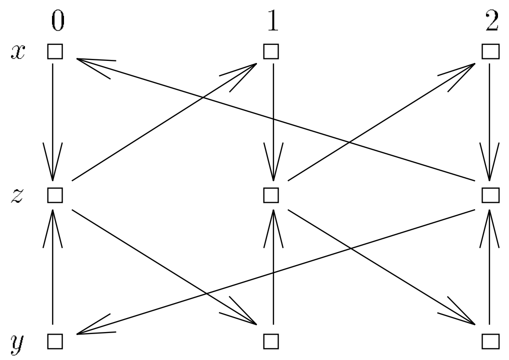

Abbildung 3.2: Der Graph H_m für $m = 2$.

Nun erzeugt G eine Anfrage auf z_i. Bewegt A den Server von x_i nach z_i, so benutzt G seinen Server auf y_i und umgekehrt. Danach kann G eine Anfrage auf x_i, bzw. y_i erzeugen, die ihn nichts, den Online-Algorithmus A jedoch mindestens $2m - 1$ kostet. Um die Phase abzuschließen, erzeugt der G erneut eine Anfrage auf z_i und bewegt auch seinen zweiten Server dorthin.

Insgesamt entstehen für den Gegenspieler Kosten in Höhe von 4, während der Online-Algorithmus mindestens $2m - 1$ an Kosten erzeugt. Nach genügend vielen Phasen gilt also

$$c_A > c \cdot c_G + a,$$

d.h. der Algorithmus A ist nicht c-kompetitiv. $\square$

Der gleiche Beweis zeigt den folgenden Satz, der jedoch auch aufgrund von Satz 2.2.4 klar ist.

Satz 3.1.3 *Es gibt keinen randomisierten 2-Server Algorithmus auf asymmetrischen metrischen Räumen, der kompetitiv gegen einen adaptiven Online-Gegenspieler ist.*

Korollar 3.1.4 *Für asymmetrische metrische Räume mit m Punkten gibt es keinen deterministischen c-kompetitiven Online-Algorithmus mit $c < (2m-1)/4$.*

3.1.3 Algorithmen

Es wurden bereits einige Algorithmen für das k-Server Problem vorgeschlagen, jedoch noch keine, die den gefundenen unteren Schranken für beliebige Räume und alle k wirklich nahe kommen.

Die Algorithmen Bal und Res

Der Fall $k = 1$ ist trivial, im Fall $k = 2$ fanden Manasse, McGeoch und Sleator [MMS88] einen 2-kompetitiven deterministischen Algorithmus **Res**, der somit optimal ist. Einen weiteren optimalen k-kompetitiven deterministischen Algorithmus **Bal** gaben sie für den Fall $k = m - 1$ an, wobei m die Anzahl der Knoten im Graphen bezeichnet.

Res basiert auf p-Residuen, wie sie in Abschnitt 2.2.3 eingeführt wurden. Der Algorithmus hält die beiden Server immer auf unterschiedlichen Knoten 1 und 2, dabei sei 1 der Knoten der letzten Anfrage. Auch der Gegenspieler muß einen Server auf Knoten 1 haben, sein anderer Server habe Position i. Es gibt nun $(m-1)$ endliche p-Residuen, die den möglichen Positionen des zweiten Servers entsprechen. Sei $R_i(t)$ die Residue, die zum Zeitpunkt t die Kosten von **Res** mit den optimalen Kosten eines Algorithmus, der zwei Server auf Knoten 1 und i hat, vergleicht. Anfangs gilt offenbar $R_i(0) = 2d_{\mathbf{2},i}$ für alle i. Eine Anfrage am Knoten j bearbeitet **Res** mit dem Server auf Knoten 1, falls

$$\min_k \{R_k + 2d_{k,j}\} \;\geq\; 2d_{\mathbf{1},j} + d_{\mathbf{1},\mathbf{2}},$$

und mit dem Server auf Knoten 2 sonst.

Bal ist ein einfacher Algorithmus, der versucht, die Anfragen möglichst gleich auf alle Server zu verteilen. Dazu führt er die gesamte bisher zurückgelegte Distanz D_i für jeden Server auf Knoten i mit. Bei einer Anfrage am Knoten j wird immer der Server bewegt, für den der Ausdruck $D_i + d_{ij}$ minimal ist.

Der Beweis der Kompetitivität findet sich für beide Algorithmen in der oben angegebenen Arbeit [MMS88].

Der harmonische Algorithmus Harmonic

Harmonic ist ein randomisierter Online-Algorithmus für das k-Server Problem, vorgeschlagen von Raghavan und Snir [RS89]. Ist d_i der Abstand von Server i zur Anfrage, wird dieser Server mit Wahrscheinlichkeit

$$p_i = (1/d_i) \,/\, (\sum_{j=1}^{k} 1/d_j)$$

zum Anfragepunkt bewegt.

Berman *et al.* [BKT90] zeigen, daß **Harmonic** für den Fall $k = 3$ kompetitiv ist. Die gezeigte Kompetitivität ist hier jedoch sehr hoch, der Wert liegt bei 3^{17000} [Rag90]. Grove gelang es, dieses Resultat zu verallgemeinern und zu zeigen, daß **Harmonic** $(\frac{5}{4}k \cdot 2^k - 2k)$-kompetitiv gegen jeden adaptiven Online-Gegenspieler ist für beliebiges $k \geq 2$ [Gro91]. Daraus folgt aufgrund von Satz 2.2.4 die Existenz eines deterministischen k-Server Algorithmus mit Kompetitivität $(\frac{5}{4}k \cdot 2^k - 2k)^2$.

Schon früher beschrieben Fiat, Rabani und Ravid [FRR90] einen allgemeinen randomisierten kompetitiven Algorithmus für beliebiges k. Die Kompetitivät dieses Algorithmus ist ebenfalls exponentiell in k.

Widerstandsgraphen

Für eine Klasse von Graphen, für die sich ein Widerstandsgraph konstruieren läßt, beschrieben Coppersmith *et al.* [CDRS90] den randomisierten k-Server Algorithmus **RWalk**, der k-kompetitiv gegen einen adaptiven Online-Gegenspieler ist für beliebiges k. Dies ist derzeit die größte Klasse von Graphen, für die ein k-kompetitiver k-Server Algorithmus bekannt ist.

Einen gewichteten symmetrischen Graphen G mit Gewichten c_{ij} nennt man *Widerstandsgraphen*, wenn es für G einen Graphen G' mit positiven Gewichten σ_{ij} gibt, der sich von G nur durch die Kantengewichte unterscheidet und für den folgendes gilt:

Sieht man G' als Stromkreis mit Widerständen der Stärke σ_{ij} an den Kanten, dann entspricht c_{ij} dem effektiven Widerstand zwischen i und j, wenn man dort eine Spannung von 1 V anlegt, d.h. es fließt ein Strom von $1/c_{ij}$ A.

In der mathematische Formulierung heißt ein Graph G mit m Knoten und Kantengewichten c_{ij} *Widerstandsgraph* oder *resistiv*, wenn es eine nicht-negative Lösung für σ_{ij} in den folgenden Gleichungen gibt:

$$\bar{\sigma}_{ii} = \sum_{j=1, j\neq i}^{m} \sigma_{ij}$$

$$\bar{\sigma}_{ij} = \sigma_{ji}, \qquad j \neq i,\ 1 \leq i, j \leq m-1$$

$$\bar{c}_{ij} = (c_{im} + c_{jm} - c_{ij})/2$$

$$\sum_{j=1}^{m-1} \bar{\sigma}_{ij} \cdot \bar{c}_{jl} = \delta_{il}, \qquad 1 \leq i, l \leq m-1$$

Den Graphen G' nennt man dann den *resistiv inversen* Graph zu G. Bei der Berechnung von G' können negative Widerstände $\sigma_{ij} < 0$ auftreten. In diesem Fall existiert kein resistiv inverser Graph. Genauere Informationen finden sich im Buch von Weinberg [Wei62].

Für folgende Klassen von Graphen läßt sich der resistiv inverse Graph jedoch berechnen:

- Alle Graphen mit drei Punkten, die die Dreiecksungleichung erfüllen.
- Punkte auf einer Linie, $c_{ij} = |r_i - r_j|$, $r_i, r_j \in \mathbb{R}$, dabei bildet die gesamte Linie den metrischen Raum.
- Die uniforme Kostenmatrix $c_{ij} = d$ für $i \neq j$, dies entspricht dem Paging Problem.
- Alle Graphen G, die aus einem spannenden Baum T für G dadurch hervorgehen, daß man die Distanz zweier Punkte in G durch die minimale Distanz über Kanten von T festlegt.

Der Algorithmus **RWalk** arbeitet nun wie folgt. Ist der Anfragepunkt a_{k+1} von einem Server belegt, wird keiner der Server bewegt. Ansonsten betrachtet man den Teilgraph G', der a_{k+1} und alle Punkte a_i mit $i \in \{1, \ldots, l\}$, die von Servern belegt sind, enthält. G' ist resistiv und **RWalk** wählt mit Wahrscheinlichkeit

$$p_i = \frac{\sigma_{i,k+1}}{\sum_{j=1}^{k} \sigma_{j,k+1}}$$

den Server auf Punkt a_i, um die Anfrage zu bearbeiten, wobei σ_{ij} die Kantengewichte des resistiv inversen Graphen von G' bezeichnet.

Deng und Mahajan [DM91] zeigen, daß die Widerstandsgraphen genau den Graphen entsprechen, auf denen der von Coppersmith *et al.* angegebene Algorithmus **RWalk** k-kompetitiv ist.

Spezielle Räume

Einige Algorithmen für spezielle metrische Räume oder Graphen wurden veröffentlicht.

Chrobak *et al.* [CKPV90] stellen den deterministischen k-Server Algorithmus **Double-Coverage** für den Fall einer Linie $M = \mathbb{R}$, die alle Punkte enthält, vor. Dieser Algorithmus zieht den Server, der am nächsten beim Anfragepunkt p steht, zunächst auf p und bearbeitet so die Anfrage. Weiterhin zieht er jedoch den Server, der auf der anderen Seite von p am nächsten zu p steht, ebenfalls um die gleiche Distanz auf p zu. Es werden also immer zwei Server bewegt, außer in dem Fall, daß alle Server auf einer Seite von p liegen. Chrobak *et al.* zeigen, daß **Double-Coverage** k-kompetitiv ist für alle k. Der Algorithmus ist weiterhin *gedächtnislos* in dem Sinne, daß zur Entscheidung nur die aktuellen Positionen der Server und der neue Anfragepunkt benötigt werden.

Auch für Bäume geben Chrobak und Larmore [CL91] einen k-kompetitiven deterministischen k-Server Algorithmus an. Dabei wird der Baum als Einbettung eines graphtheoretischen Baums in die Ebene gesehen, d.h. der Baum ist kein diskreter metrischer Raum mehr, da die Punkte auf den Kanten mit zum metrischen Raum gehören. Sei p der Anfragepunkt und $(p, q]$ die Menge der Punkte auf dem eindeutigen kürzesten Pfad von p nach q, dabei ist p ausgeschlossen. s_i bezeichne die Position des i-ten Servers. Ein Server heißt *aktiv*, wenn kein weiterer Server auf dem Intervall $(s_i, p]$ liegt. Haben mehrere Server die gleiche Position, so soll höchstens einer von ihnen aktiv werden. Der Algorithmus arbeitet wie folgt:

Algorithmus TREE
while Kein Server hat p erreicht **do**
  Sei $d = \min\{|s_i, q_i|\}$, wobei $q_i \in (s_i, p]$ entweder
  ein Knoten oder p ist.
  Bewege jeden aktiven Server um d in Richtung p.
end while;

Der Beweis der Kompetitivität benutzt eine amortisierte Analyse.

Für den Kreis beschrieben Fiat *et al.* einen $O(k^3)$-kompetitiven deterministischen k-Server Algorithmus [FRRS].

3.1.4 Erweiterte Problemstellungen

Das k-Server Problem kann auf verschiedene Art und Weise modifiziert werden. Einige Varianten und Resultate dazu werden hier angegeben.

Das k-Taxi Problem

Im *k-Taxi Problem* besteht eine Anfrage nicht nur aus einem Punkt p, an den ein Server bewegt werden muß, sondern aus einem Paar (s, t) von Punkten. Zunächst erscheint nur der Punkt s als Anfrage, ist dann ein Server nach s bewegt worden, wird auch t bekannt, und der Server wird dorthin bewegt. Das Problem entspricht also der Verwaltung von k Taxen, wobei Passagiere ihre Fahrtziele nicht im voraus bekanntgeben.

Fiat *et al.* konstruieren einen kompetitiven k-Server Algorithmus, der nach kleinen Änderungen auch kompetitiv für das k-Taxi Problem ist [FRR90]. Der kompetitive Faktor ist jedoch exponentiell in k.

Das Mintime k-Server Problem

Bezieht man die Zeit, die ein Server braucht, um sich von einem Punkt zum anderen zu bewegen, mit ein, erhält man das *Mintime k-Server Problem.* Unter der Annahme, daß alle Server sich mit der gleichen Geschwindigkeit bewegen, sind die Kosten des Algorithmus nun die Zeit und nicht die Distanz. Die nächste Anfrage erscheint jedoch wie beim k-Server Problem erst dann, wenn die letzte Anfrage bearbeitet wurde. Der Unterschied zwischen den beiden Problemen besteht also nur darin, daß, während ein Server sich zu einem Anfragepunkt bewegt, die anderen Server sich kostenlos bewegen können.

Der Algorithmus von Fiat, Rabani und Ravid für das allgemeine k-Server Problem [FRR90] kann so modifiziert werden, daß er kompetitiv für das Mintime k-Server Problem ist.

Das verteilte k-Server Problem

Bartal und Rosén betrachten den Fall, daß die Server innerhalb eines Computernetzwerkes bewegt werden [BR92]. Jeder Punkt im Graphen entspricht also einem Prozessor. Anfragen tauchen nun nicht *global* auf, sondern nur *lokal*, d.h. nur in dem Prozessor, der dem Anfragepunkt entspricht. Zusätzlich zu den Kosten, die die Bewegung der Server verursacht, entstehen nun Kommunikationskosten, denn um einen Server zum Anfragepunkt zu bewegen, muß der entsprechende Prozessor Nachrichten verschicken. Die Kommunikationskosten entsprechen dabei den Entfernungen der Prozessoren mal einer Konstante $D > 0$. Die Schwierigkeit liegt hier also darin, daß nicht nur die zukünftigen Anfragen unbekannt sind, sondern auch Anfragen, von denen ein Prozessor keine Nachricht erhalten hat. Denn jeden Prozessor von einer Anfrage zu informieren, ist im allgemeinen zu teuer. Daraus folgt auch, daß nicht jeder Prozessor ständig über die Positionen aller Server informiert sein kann.

Kompetitivität wird in diesem Fall so ausgelegt, daß die Kosten des lokalen Online-Algorithmus innerhalb einer Konstante der Kosten eines globalen Offline-Algorithmus liegen müssen. Um so erstaunlicher ist es, daß es dennoch gelingt, hier kompetitive Algorithmen anzugegeben [BR92].

Server mit Exkursionen

Erlaubt man, daß die Server eine Anfrage an einem Punkt auch dann bearbeiten können, wenn sie nicht dorthin bewegt werden, so spricht man vom *k-Server Problem mit Exkursionen*. Man stellt sich vor, daß die Server sich zum Anfragepunkt und wieder zurück bewegen. Dies ist sinnvoll, falls die Kosten für eine solche Exkursion kleiner sind als die Kosten, um den Server zum Anfragepunkt zu bewegen und dort zu belassen.

Ein Beispiel hierfür ist das Problem der Verteilung von Feuerwachen. Eine Feuerwache kann einen Brand auch aus einer gewissen Entfernung löschen, treten an einer Stelle jedoch gehäuft Brände auf, so kann es billiger sein, die Feuerwache zu bewegen. Das Bewegen einer ganzen Feuerwache ist aber natürlich teurer als die Fahrt zum Brandort.

Dieses Problem ist schwieriger zu bearbeiten als das k-Server Problem. Die einzigen bisher erzielten Resultate betreffen $(m-1)$ Server auf einem Graphen G mit m Punkten. Hier zeigten Borodin *et al.* [BLS87] eine untere Schranke von $(2m-1)$. Manasse *et al.* stellen den folgenden Algorithmus Bale vor, der $(2m-1)$-kompetitiv ist [MMS88].

Offenbar sind die Kosten für eine Exkursion für den Fall von $n-1$ Servern nur vom Anfragepunkt abhängig, da alle anderen Punkte dann von Servern belegt sind und immer die billigste Exkursion gewählt werden kann. Seien r_i die Kosten für eine Exkursion zum Punkt i. Der Algorithmus Bale bildet das $(m-1)$-Server Problem auf dem Graphen G mit m Knoten auf ein $(2m-1)$-Server Problem auf einem Graphen G' mit $2m$ Knoten ab. Auf G' führt Bale den Algorithmus Bal aus Abschnitt 3.1.3 aus.

G' erhält man aus G durch Verdopplung aller Punkte. Die Distanz zweier Knoten x und y in G' wird dann wie folgt definiert: Entsprechen x und y dem gleichen Knoten i in G, ist ihre Distanz r_i. Sonst enspricht ihre Distanz der Distanz der beiden ensprechenden

Punkte in G. BALE bewegt seine Server so, daß von den beiden Knoten in G', die dem nicht von einem Server belegten Knoten in G entsprechen, genau einer belegt sind.

Auf eine Anfrage am Knoten i antwortet BALE wie folgt. Ist i von einem Server belegt, geschieht nichts weiter. Ist i der nicht belegte Knoten, so erzeugt BALE eine Anfrage am nicht belegten entsprechenden Knoten in G', die vom Algorithmus BAL beantwortet wird. Bewegt BAL nun den Server von dem anderen i entsprechenden Knoten, so führt BALE eine Exkursion aus, bewegt BAL aber irgendeinen anderen Server von einem Knoten j in G', so bewegt BALE den Server, der den Knoten in G, der j entspricht, belegt.

Die Kosten von BALE entsprechen dann genau den Kosten von BAL in der Simulation. Manasse *et al.* zeigen, daß BALE $(2m-1)$-kompetitiv ist [MMS88]. Die wichtigste Beobachtung dabei ist, daß jedem Online-Algorithmus A für Server mit Exkursionen auf G ein Online-Algorithmus A' auf G' entspricht, dessen Kosten durch die Kosten von A beschränkt sind.

Caching mit Gewichten

Hängen die Kosten für das Bewegen eines Servers von p nach q nur vom Punkt p ab, spricht man vom *Caching mit Gewichten*, denn dieser Fall entspricht dem Caching oder Paging Problem, siehe Abschnitt 3.1.1, wobei jedoch das Einlesen einer Seite p vom langsamen Massenspeicher nicht für jede Seite gleich teuer ist, sondern von der Länge $w(p)$ der Seite abhängt. Jeder Seite wird also eine Länge, bzw. jedem Punkt ein Gewicht zugeordnet, k entspricht der Anzahl der Seiten, die im schnellen Speicher gehalten werden können.

Chrobak *et al.* zeigen, daß der Algorithmus BAL, der in Abschnitt 3.1.3 beschrieben ist, auch für Caching mit Gewichten k-kompetitiv ist [CKPV90]. Damit ist BAL optimal aufgrund der unteren Schranken für Caching ohne Gewichte [MMS88].

Weiterhin zeigen die Autoren, daß es für Caching mit Gewichten keinen k-kompetitiven gedächtnislosen Algorithmus geben kann.

Die `fork` Operation

Sieht man die Server im k-Server Problem als Zeiger innerhalb einer Suchstruktur, die die Suchzeit verkürzen sollen, so ist es natürlich, außer dem Bewegen eines Servers auch das Gabeln eines Servers zuzulassen, d.h. der Server springt auf eine Stelle, die schon von einem anderen Server besetzt ist. Dies entspricht dem Kopieren eines Zeigers. Diese Operation wurde zuerst von Cole und Raghunathan [CR90] vorgeschlagen und wird dort als `fork` bezeichnet.

Die Autoren untersuchen diese Variante des k-Server Problems nur auf einem Liniensegment der Länge l. Hier zeigen sie, daß für das k-Server Problem mit `fork` ab $k=3$ ein Online-Algorithmus, der die Operation `fork` nicht benutzt, höchstens $\sqrt{l}$-kompetitiv sein kann. Weiterhin geben sie einen Algorithmus für $k=3$ an, der 184-kompetitiv ist.

3.1.5 Offene Probleme

Das wichtigste offene Problem auf dem Gebiet der k-Server Algorithmen ist natürlich, ob es einen allgemeinen k-kompetitiven k-Server Algorithmus für beliebige Räume gibt. Auch für Spezialfälle oder eingeschränkte Fragestellungen gibt es jedoch offene Fragen.

Gibt es beispielsweise randomisierte Online-Algorithmen für das k-Server Problem auf asymmetrischen Graphen, die kompetitiv gegen unwissende Gegenspieler sind?

3.1.6 Literatur — k-Server Problem

[BKT90] P. Berman, H. J. Karloff, and G. Tardos. A competitive three-server algorithm. In *Proc. 1st SIAM Symp. on Discrete Algorithms*, pages 280–290, 1990.

[BLS87] A. Borodin, N. Linial, and M. Saks. An optimal online algorithm for metrical task systems. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 373–382, 1987.

[BR92] Y. Bartal and A. Rosén. The distributed k-server problem – a competitive distributed translator for k-server algorithms. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, pages 344–353, 1992.

[CDRS90] D. Coppersmith, P. Doyle, P. Raghavan, and M. Snir. Random walks on weighted graphs, and application to on-line algorithms (preliminary version). In *Proc. 22nd ACM Symp. on Theory of Computing*, pages 369–378, 1990.

[CKPV90] M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan. New results on server problems. In *Proc. 1st SIAM Symp. on Discrete Algorithms*, pages 291–300, 1990.

[CL91] M. Chrobak and L. L. Larmore. An optimal on-line algorithm for k-servers on trees. *SIAM J. of Computing*, 20(1):144–148, 1991.

[CNV92] S. D. Carson, V. Nirkhe, and P. Vongsathorn. A discrete-state model of the two-headed disk. *Information Processing Letters*, 41(6):341–345, 1992.

[CR90] R. Cole and A. Raghunathan. Online algorithms for finger searching. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 480–489, 1990.

[DM91] X. Deng and S. Mahajan. Server problems and resistive spaces. *Information Processing Letters*, 37(4):193–196, 1991.

[FRR90] A. Fiat, Y. Rabani, and Y. Ravid. Competitive k-server algorithms (extended abstract). In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 454–469, 1990.

[FRRS] A. Fiat, Y. Rabani, Y. Ravid, and B. Schieber. A deterministic $O(k^3)$-competitive k-server algorithm for the circle.

[Gro91] E. F. Grove. The harmonic online k-server algorithm is competitive. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 260–266, 1991.

[IR91] S. Irani and R. Rubinfeld. A competitive 2-server algorithm. *Information Processing Letters*, 39(2):85–91, 1991.

[KRR91] H. Karloff, Y. Rabani, and Y. Ravid. Lower bounds for randomized k-server and motion-planning algorithms. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 278–288, 1991.

[MMS88] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for online problems. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 322–333, 1988.

[MV91] Y. Manolopoulos and A. Vakali. Seek distances in disks with two independent heads per surface. *Information Processing Letters*, 37(1):37–42, 1991.

[Rag90] P. Raghavan. Lecture notes on randomized algorithms. Technical report, IBM Yorktown, 1990.

[RS89] P. Raghavan and M. Snir. Memory vs. randomization in on-line algorithms. In *Proc. 16th Internat. Colloq. Autom. Lang. Program.*, volume 372 of *LNCS*, 1989.

[Wei62] L. Weinberg. *Network Analysis and Synthesis*. McGraw-Hill, New York, 1962.

3.2 Selbstanordnende Listen

Aber viele, die da sind die Ersten,
werden die Letzten, und die Letzten
werden die Ersten sein.
— MATTHÄUS *19,30*

Im *List Update Problem* betrachtet man eine lineare Liste $L = (a_1, \ldots, a_m)$ von Elementen, auf die eine Folge von Zugriffen $\sigma = (\sigma_1, \ldots, \sigma_n)$ stattfindet, wobei $\sigma_i \in \{a_1, \ldots, a_m\}$ ist.

Die Kosten für jeden Zugriff σ_i entsprechen der Position von σ_i in der Liste. Nach jedem Zugriff kann das Element, auf das zugegriffen wurde, ohne zusätzliche Kosten an eine Stelle weiter am Anfang der Liste getauscht werden. Die Liste ist also *selbstanordnend* im Gegensatz zu *statischen* linearen Listen.

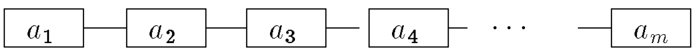

Abbildung 3.3: Die lineare Liste L.

Das List Update Problem ist eines der klassischen Online-Probleme, es wurde bereits sehr intensiv untersucht, bevor die kompetitive Analyse von Sleator und Tarjan eingeführt wurde [Bit79, BM85, HH85, Riv76].

3.2.1 Untere Schranken

Die Kompetitivität deterministischer Online-Algorithmen für das List Update Problem läßt sich auf einfache Weise nach unten beschränken.

Satz 3.2.1 *Für alle deterministischen Online-Algorithmen A zur Selbstanordnung von Listen der Länge m gibt es Anfragefolgen σ, so daß gilt*

$$c_A(\sigma) \geq 2m/(m+1) \cdot c_{opt}(\sigma)$$

Beweis: Man betrachtet den Gegenspieler, der stets eine Anfrage für das letzte Element in der Liste des Algorithmus A erzeugt. Die Kosten von A sind also m für jede Anfrage. Der Gegenspieler sortiert die Folge nun nach absteigender Zugriffshäufigkeit. Dies erzeugt Kosten, die nur von der Länge m der Liste abhängen. Während der Gegenspieler die Anfragen bearbeitet, bleibt die Liste statisch. Somit entstehen für den Gegenspieler Kosten von durchschnittlich höchstens $(m+1)/2$ pro Anfrage. □

Diese untere Schranke gilt auch für die Kompetitivität randomisierter Online-Algorithmen, jedoch nur gegen adaptive Gegenspieler. Dies zeigen Irani *et al.* [IRWS91]. Ein Gegenbeispiel für den Fall von Kompetitivität gegen unwissende Gegenspieler ist der Algorithmus SPLIT [Ira91], den wir im Abschnitt 3.2.3 kurz beschreiben.

3.2.2 Deterministische Algorithmen

Der wichtigste Algorithmus für die Selbstanordnung von Listen, der auch in der Literatur am ausgiebigsten untersucht wurde, ist die MOVE-TO-FRONT Strategie, kurz **MTF** genannt. Diese tauscht das Element, auf das zugegriffen wurde, in jedem Fall an den Anfang der Liste. Daß dieses einfache Verfahren dennoch die für deterministische Algorithmen optimale Kompetitivität erreicht, zeigt der folgende Satz von Irani [Ira91].

Satz 3.2.2 *Für alle Zugriffsfolgen σ auf Folgen der Länge m gilt*

$$c_{MTF}(\sigma) \leq 2m/(m+1) \cdot c_{opt}(\sigma),$$

d.h. **MTF** *ist $2m/(m+1)$-kompetitiv und somit insbesondere auch 2-kompetitiv.*

Daß **MTF** 2-kompetitiv ist, zeigten Sleator und Tarjan bereits früher in einer der ersten Arbeiten, die die kompetitive Analyse benutzt [ST85].

Ein anderes einfaches Verfahren ist TRANSPOSE, hier wird das Element, auf das zugegriffen wurde, nur um eine Stelle weiter nach vorne gerückt. Wird jedoch auf die letzten beiden Elemente der Liste abwechselnd zugegriffen, so benötigt TRANSPOSE für jeden Zugriff lineare Kosten gegenüber konstanten optimalen Kosten. Es gibt also keine von m unabhängige Konstante c, so daß TRANSPOSE c-kompetitiv für alle Zugriffsfolgen auf allen Folgen der Länge m ist.

3.2.3 Randomisierte Algorithmen

Da Move-To-Front eine für den deterministischen Fall optimale Strategie darstellt, gilt in neuerer Zeit das Interesse vermehrt randomisierten Algorithmen zur Lösung des List Update Problems.

Irani [Ira91] schlägt den folgenden Algorithmus Split vor, der für jedes Element a_i ein Splitelement $split(a_i)$ speichert. Anfangs gilt $split(a_i) = a_i$ für alle Elemente der Folge. Nach einem Zugriff auf ein Element a_i bewegt Split dieses mit Wahrscheinlichkeit 1/2 direkt vor das Element $split(a_i)$ und setzt $split(a_i)$ auf das erste Element der Liste. Mit gleicher Wahrscheinlichkeit wird a_i an den Anfang der Liste bewegt und $split(a_i)$ bleibt unverändert.

Satz 3.2.3 *Der Algorithmus* Split *ist 31/16-kompetitiv gegen jeden unwissenden Gegenspieler.*

Irani *et al.* [IRWS91] geben einen weiteren randomisierten Algorithmus an, der 7/4-kompetitiv ist.

Satz 3.2.4 *Kein Online-Algorithmus für das List Update Problem kann c-kompetitiv gegen unwissende Gegenspieler sein mit $c < 9/8$.*

Diese untere Schranke wird bewiesen, indem man eine Liste der Länge zwei und eine zufällige Anfragefolge betrachtet. Die Kosten jedes Online-Algorithmus sind dann 3/2 pro Anfrage. Die Kosten des Offline-Algorithmus sind komplizierter und werden mit Hilfe einer Markov-Kette berechnet.

Betrachtet man eine Liste der Länge drei, so erhält man eine verbesserte untere Schranke von $1,27$, die unabhängig von Reingold und Westbrook und von Chrobak und Larmore gezeigt wurde. Die Schwierigkeit bei der Betrachtung längerer Listen ist, daß das Verfahren für den Offline-Algorithmus zu kompliziert wird.

3.2.4 Das gewichtete List Update Problem

Ordnet man jedem Element der Folge L ein positives Gewicht w_i zu und definiert man die Kosten eines Zugriffs als die Summe der Gewichte der Elemente, die vor dem Element stehen, auf das zugegriffen wird, so erhält man das *gewichtete* List Update Problem.

In der Arbeit von d'Amore *et al.* [dMSN91] finden sich einige Ergebnisse zu dieser geänderten Problemstellung. Zunächst stellt sich heraus, daß Move-To-Front hier ungünstig ist, da es die Gewichte nicht berücksichtigt. Genauer läßt sich zeigen, daß es keine Konstante c gibt, so daß **MTF** c-kompetitiv ist.

Die Autoren schlagen daraufhin die Verfahren Counting-Move-To-Front **CMTF** und Randomized-Move-To-Front **RMTF** vor. Der Algorithmus **CMTF** aktualisiert für jedes Element einen Zähler, der bei jedem Zugriff um w_{min}/w_i erhöht wird. Ist der Zähler größer als 1, wird das Element an den Anfang der Liste getauscht und der Zähler um 1 verringert. Bei **RMTF** wird jedes Element mit der Wahrscheinlichkeit $p_i = w_{min}/w_i$ an den Anfang der Liste gesetzt.

Satz 3.2.5 *Die Online-Algorithmen* CMTF *und* RMTF *sind 2-kompetitiv.*

Für den Fall $w_i \equiv 1$ entsprechen beide Algorithmen der üblichen Strategie MOVE-TO-FRONT.

3.2.5 Offene Probleme

- Was ist die untere Schranke für das List Update Problem für randomisierte Algorithmen?

- Reicht es aus, dazu den Gegenspieler jeweils nur Paare aufeinanderfolgenden Anfragen betrachten zu lassen? Die Beweise der bisherigen Resultate benutzen diese Methode.

3.2.6 Literatur — Selbstanordnende Listen

[Bit79] J. R. Bitner. Heuristics that dynamically organize data structures. *SIAM J. of Computing*, 8(1):82–110, 1979.

[BM85] J. L. Bentley and C. McGeogh. Amortized analyses of self-organizing sequential search heuristics. *Communications of the ACM*, 28(4):404–411, 1985.

[dMSN91] F. d'Amore, A. Marchetti-Spaccamela, and U. Nanni. Competitive algorithms for the weighted list update problem. In *Proc. 2nd Workshop on Algorithms and Data Structures*, pages 240–248, 1991.

[HH85] J. H. Hester and D. S. Hirschberg. Self-organizing linear search. *ACM Computing Surveys*, 17(3):295–311, 1985.

[Ira91] S. Irani. Two results on the list update problem. *Information Processing Letters*, 38(6):301–306, 1991.

[IRWS91] S. Irani, N. Reingold, J. Westbrook, and D. D. Sleator. Randomized competitive algorithms for the list update problem. In *Proc. 2nd SIAM Symp. on Discrete Algorithms*, pages 251–260, 1991.

[Riv76] R. Rivest. On self-organizing sequential search heuristics. *Communications of the ACM*, 19(2):63–67, 1976.

[ST85] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

3.3 Paging und Caching

Paging and swapping [...] *can't solve*
the basic problem: they can't make more memory
appear out of nowhere.
— Mike Loukides *System Performance Tuning*

Betrachtet man ein Speichersystem mit k Seiten schnellem Speicher, etwa einem Cache, und einer großen Anzahl m von Seiten im langsameren Massenspeicher, so entsteht das Problem, während einer Folge von Speicheranfragen zu entscheiden, welche der Speicherseiten im Cache gehalten werden sollen und welche durch neu geladene Seiten überschrieben werden. Dieses Problem bezeichnet man als *Paging Problem.* Wird jeder Seite zusätzlich ein Gewicht w_i zugeordnet, das die Kosten beim Laden der Seite in das Cache darstellt, so spricht man vom (gewichteten) *Caching Problem.*

Die Kosten, die zu minimieren sind, sind die Anzahl bzw. das Gewicht der Lesevorgänge in das Cache, wobei jeder Online-Algorithmus im worst case natürlich für jede Anfrage neu lesen muß. Es zeigt sich, daß insbesondere die wichtigen Algorithmen **LRU** und **FIFO** auf Schleifen einer Länge größer gleich $(k + 1)$ schlechte Resultate liefern. Dies ist ein für die Praxis wichtiger Fall, es zeigt sich jedoch, daß die Algorithmen dennoch die beste mögliche Kompetitivität erreichen. Verbesserungen der Algorithmen für solche Spezialfälle können also nicht zu einer allgemein erhöhten Kompetitivität führen.

In der Vergangenheit wurde das Paging Problem daher hauptsächlich unter Annahmen über die Verteilung der Anfragen analysiert. In neuerer Zeit wird jedoch auch hier vermehrt die Methode der kompetitiven Analyse benutzt.

Das Paging Problem ist ein Spezialfall das k-Server Problems, siehe dazu Abschnitt 3.1.1.

3.3.1 Untere Schranken

Im deterministischen Fall läßt sich recht einfach eine untere Schranke für das Paging Problem angeben [ST85].

Satz 3.3.1 *Kein deterministischer Online-Algorithmus für das Paging Problem kann c-kompetitiv sein, für $c < k$.*

Randomisierte Algorithmen können einen wesentlich besseren kompetitiven Faktor erreichen [FKL$^+$91].

Satz 3.3.2 *Eine untere Schranke für die Kompetitivität randomisierter Online-Algorithmen gegen unwissende Gegenspieler ist H_k.*

Dabei bezeichnet H_k die k-te harmonische Zahl. Es gilt $\ln(k + 1) \leq H_k \leq \ln(k) + 1$, d.h. $H_k \in \Theta(\log k)$.

3.3.2 Deterministische Algorithmen

Sleator und Tarjan [ST85] untersuchen die Algorithmen LEAST-RECENTLY-USED **LRU** (entferne die Seite, deren Zugriff am weitesten zurückliegt) und FIRST-IN-FIRST-OUT **FIFO** (die Seiten werden in der Reihenfolge entfernt, wie sie gelesen wurden). Beide Verfahren sind deterministisch und als solche optimal, denn es gilt:

Satz 3.3.3 *Die Algorithmen* **LRU** *und* **FIFO** *sind k-kompetitiv.*

Karlin *et al.* [KMRS88] zeigen, daß auch die FLUSH-WHEN-FULL **FWF** Strategie k-kompetitiv ist. Hier wird das gesamte Cache geleert, sobald es voll ist und eine weitere Seite gelesen werden soll.

3.3.3 Randomisierte Algorithmen

Fiat *et al.* [FKL+91] geben die zwei randomisierten Algorithmen MARK und **EATR** für das Paging Problem an.

Der Algorithmus MARK fügt den Seiten, die im Cache liegen, eine Markierung hinzu. Zu Beginn sind alle Seiten im Cache markiert. Erfolgt eine Anfrage, so geschieht nichts, falls die angefragte Seite bereits im Cache liegt. Andernfalls wird die Seite an Stelle einer zufällig ausgewählten nicht markierten Seite eingelesen und markiert. Waren bereits alle Seiten im Cache markiert, werden vor dem Einlesen der neuen Seite alle Markierungen rückgängig gemacht.

Satz 3.3.4 *Der Algorithmus* MARK *ist $2H_k$-kompetitiv für das allgemeine Paging Problem.*

In einem Spezialfall erreicht MARK sogar die untere Schranke.

Satz 3.3.5 MARK *ist H_k-kompetitiv für den Fall $k = m - 1$.*

Dabei bezeichnet m die Gesamtanzahl der Speicherseiten. Diese bessere Schranke gilt nicht für den allgemeinen Fall, dies zeigt schon das Beispiel $k = 2$, $m = 4$. Enthält das Cache des Algorithmus MARK nämlich anfangs die Seiten 1 und 2 und das des Gegenspielers die Seiten 1 und 3, so erzeugt der Gegenspieler Anfragen auf den Seiten 3 und 4. Danach enthält das Cache von MARK diese beiden Seiten, wobei Kosten von 2 entstanden sind. Der Gegenspieler hat nur eine Seite eingelesen, dies ist die Seite 4, die nun anstelle von Seite 3 im Cache steht. Nach der Phase haben die beiden Algorithmen somit wieder genau eine gemeinsame Seite im Cache. Für den kompetitiven Faktor $c_{2,4}$ gilt also $1{,}5 = H_2 < c_{2,4} = 2 < 2 \cdot H_2 = 3$.

Für einen weiteren Spezialfall $k = 2$ konstruieren die Autoren den folgenden Algorithmus END-AFTER-TWICE-REQUESTED **EATR**, der dort ebenfalls H_k-kompetitiv ist.

EATR unterteilt die Bearbeitung der Anfragefolge in *Phasen*. Die erste Phase beginnt mit der ersten Anfrage auf eine Seite, die nicht im Cache liegt. Eine Seite heißt *frei*, wenn sie am Ende der letzten Phase nicht im Cache lag und in dieser Phase noch nicht angefragt wurde. Eine Seite heißt *verbraucht*, wenn sie nicht frei und nicht die Seite der letzten Anfrage ist.

Eine Seite im Cache von **EATR** ist nun stets die Seite der letzten Anfrage, die andere Seite im Cache wird zufällig unter den verbrauchten Seiten gewählt. Wird eine verbrauchte Seite angefragt, befinden sich die beiden zuletzt angefragten Seiten im Cache. In letzterem Fall endet die Phase, sobald wieder eine Seite angefragt wird, die nicht im Cache liegt.

Satz 3.3.6 *Der Algorithmus* **EATR** *ist für den Speziallfall $k = 2$ des Paging Problems 3/2-kompetitiv gegen unwissende Gegenspieler.*

Einen Algorithmus für das Paging Problem, der auch im allgemeinen Fall H_k-kompetitiv ist, gaben zuerst McGeoch und Sleator [MS89] an. Ihr *Partitionierungs-Algorithmus* erreicht somit genau die untere Schranke. Damit ist hier, anders als für das allgemeine k-Server Problem, ein optimaler Algorithmus bekannt. Er ist jedoch deutlich komplizierter als der Algorithmus MARK und kann hier aus Platzgründen nicht wiedergegeben werden.

3.3.4 Ein optimaler Offline-Algorithmus

Schon 1966 gab Belady einen optimalen Offline-Algorithmus für das Paging Problem an [Bel66]. Der Algorithmus MIN entfernt bei vollem Cache immer die Speicherseite, bei der die Wartezeit für den nächsten Zugriff am größten wäre.

3.3.5 Snoopy Caching

Karlin *et al.* untersuchen die Autoren ein noch allgemeineres Modell für das Paging oder Caching Problem [KMRS88]. Sie betrachten ein bus-basiertes Multiprozessorsystem, wobei jeder Prozessor über ein eigenes Cache verfügt, das über den Bus an den gemeinsam benutzten langsameren Hauptspeicher angeschlossen ist. Jede Speicherseite muß mindestens einmal repräsentiert sein, sie muß jedoch nicht unbedingt im Hauptspeicher liegen. Ist die Speicherseite jedoch in mehreren Caches gespeichert, so muß jede Änderung über den Bus an die anderen Prozessoren weitergeleitet werden. Die Anfragen sind in diesem Modell Lese- oder Schreibzugriffe auf eine bestimmte Seite.

Man unterscheidet *direktes* und *assoziatives* Snoopy Caching. Beim direkten Snoopy Caching wird die Position einer Seite im Cache durch eine Hashfunktion bestimmt. Es kann dann jeweils nur *eine* Seite mit gleicher Hashfunktion im Cache gespeichert werden, auch wenn andere Plätze im Cache noch frei sind. Dagegen sind beim assoziativen Snoopy Caching die Plätze für die Speicherseiten frei wählbar.

Für direktes Snoopy Caching geben die Autoren den deterministischen Online-Algorithmus **DSC** an, der 2-kompetitiv ist. Sie zeigen auch, daß dies der optimale kompetitive Faktor ist.

Sie verallgemeinern weiterhin das Verfahren **FIFO** für das assoziative Snoopy Caching Modell zu einem Online-Algorithmus **SCWF** und zeigen, daß dieser Algorithmus kompetitiv ist.

Weitere Resultate für einige Spezialfälle des Snoopy Caching finden sich in der Arbeit von Karlin *et al.* [KMMO90].

3.3.6 Caching auf mehreren Ebenen

Das bisher untersuchte Modell für Paging und Caching geht von zwei Speicherebenen aus: langsamer Massenspeicher auf der einen Seite und schneller Hauptspeicher auf der anderen. Es ist möglich, dieses Modell auf eine größere Anzahl von Speicherebenen auszuweiten, die an Zugriffsgeschwindigkeit zu- und an Größe abnehmen.

Denkbar wäre zum Beispiel, Speicherseiten zunächst komprimiert in den Speicher zu laden, und diese nur im Bedarfsfalle zu dekomprimieren. Damit können mehr Speicherseiten eingelesen werden. Es entsteht eine Zwischenebene, die etwas schneller als der Massenspeicher und etwas größer als der Hauptspeicher ist.

Bern, Greene und Raghunathan [BGR93] untersuchen dieses erweiterte Modell und geben kompetitive Algorithmen an. Sie entwickeln auch Methoden für den Fall, daß sich die Grenze zwischen den Cachepuffern verschieben läßt. So ist es im Falle der Komprimierung von Speicherseiten möglich, nur *eine* dekomprimierte Seite im Hauptspeicher zu halten, dies erhöht die Kapazität an komprimierten Seiten, oder aber *mehrere*, wobei dann weniger komprimierte Seiten gespeichert werden können.

3.3.7 Offene Probleme

Gibt es Algorithmen für das gewichtete List Update Problem, die kompetitiv gegen adaptive Offline-Gegenspieler sind?

3.3.8 Literatur — Paging und Caching

[Bel66] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.

[BGR93] M. Bern, D. Greene, and A. Raghunathan. On-line algorithms for cache sharing. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 422–429, 1993.

[FKL+91] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *J. of Algorithms*, 12:685–699, 1991.

[KMMO90] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki. Competitive randomized algorithms for non-uniform problems. In *Proc. 1st SIAM Symp. on Discrete Algorithms*, pages 301–309, 1990.

[KMRS88] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[MS89] L. A. McGeoch and D. D. Sleator. A strongly competitive randomized paging algorithm. Technical report CMU-CS-89-122, School of Computer Science, Carnegie Mellon University, 1989.

[ST85] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[You91] N. Young. Competitive paging as cache-size varies. In *Proc. 2nd SIAM Symp. on Discrete Algorithms*, pages 241–250, 1991.

3.4 Load Balancing und Prozeßplanung

Was würde er wohl dazu sagen, wenn er erfahren würde, daß sein Prozeß noch gar nicht begonnen hat, daß noch nicht einmal das Glockenzeichen zu seinem Prozeß gegeben ist?
— FRANZ KAFKA *Der Prozeß*

Eine ganze Reihe von Problemen ergibt sich bei der Vergabe von Resourcen in parallelen Systemen mit mehreren gleichzeitig ablaufenden Prozessen. Allgemein bezeichnet man diese Klasse von Problemen als *Scheduling Probleme*. Viele der Probleme wurden zunächst offline untersucht, sie lassen sich jedoch alle auch auf den allgemeinen Online-Fall übertragen.

Die Literatur zu diesem Gebiet ist sehr umfangreich, insbesondere wird eine sehr große Zahl leicht unterschiedlicher Modelle untersucht.

3.4.1 Prozeßplanung

Das einfachste Modell für Prozeßplanung besteht aus m Prozessoren, hier *Maschinen* genannt, und n Prozessen, die auf den Maschinen ausgeführt werden sollen. Auf jeder Maschine kann jeweils nur ein Prozeß ausgeführt werden. Das Problem besteht darin, die Prozesse so auf die einzelnen Maschinen zu verteilen, daß die insgesamt benötigte Zeit möglichst kurz wird, d.h. der letzte Prozeß möglichst früh stoppt. Dieses Problem wird auch als *Machine Scheduling* bezeichnet.

Im Offline-Fall ist dabei die Dauer eines Prozesses schon vor dessen Ausführung bekannt, während ein Online-Algorithmus den Prozeß einer Maschine zuweisen muß, ohne dessen Dauer zu kennen. Interessant ist, daß das Offline-Problem NP-vollständig ist, dies gilt für die meisten in diesem Abschnitt erwähnten Offline-Probleme.

Den Online-Fall untersuchte Graham [Gra66] bereits 1966. Er zeigte, daß der deterministische Greedy-Algorithmus, der die Prozesse in einer beliebigen Reihenfolge ohne Wartezeit auf die Maschinen verteilt, bereits $(2 - \frac{1}{m})$-kompetitiv ist. Er zeigte auch, daß dies optimal ist für deterministische Online-Algorithmen.

Abbildung 3.4: Eine untere Schranke für deterministische Prozeßplanung, oben die optimale Planung, unten die Planung des Online-Algorithmus.

Die untere Schranke resultiert aus dem Fall von $m(m-1)$ Prozessen der Länge 1 und einem Prozeß mit Länge m. Die optimale Planung ordnet den langen Prozeß gleich zu Beginn einer der Maschinen zu und erzielt damit eine Gesamtdauer von m. Ein deterministischer Online-Algorithmus ordnet den langen Prozeß jedoch unter Umständen erst als letzten einer Maschine zu. Die Gesamtdauer beträgt in diesem Fall $(m-1)+m$. Beide Situationen sind in Abbildung 3.4 dargestellt. Indem man die Dauer der kurzen Prozesse halbiert und deren Anzahl verdoppelt, läßt sich das Beispiel auf beliebig viele Prozesse erweitern.

Erscheinen neuer Prozesse während der Bearbeitung

Dieses sehr einfache Modell für Prozeßplanung läßt sich schrittweise erweitern. Eine erste Möglichkeit besteht darin, daß zu Beginn der Bearbeitung noch nicht alle Prozesse bekannt sind, d.h. einzelne Prozesse erscheinen erst nach einer gewissen Zeit. Der Offline-Algorithmus kennt diese Zeitpunkte zwar bereits im Voraus, darf die Prozesse jedoch nicht schon vor ihrem Erscheinen starten.

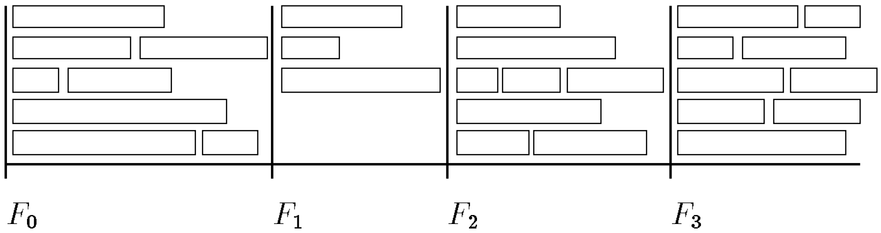

Abbildung 3.5: Während der Bearbeitung erscheinen neue Prozesse.

Shmoys *et al.* [SWW91] schlagen das folgende Online-Verfahren A' vor, das einen beliebigen Online-Algorithmus A für das einfache Modell auf das Modell mit später erscheinenden Prozessen erweitert. Zunächst werden alle zu Beginn bekannten Prozesse mit Hilfe von Algorithmus A bearbeitet, dies dauert bis zu einem Zeitpunkt F_1. Während der Bearbeitung werden alle neu hinzukommenden Prozesse in einer Liste gesammelt. Mit der

Bearbeitung dieser Liste wird dann zum Zeitpunkt F_1 begonnen, wiederum nach dem Verfahren A. Dies geschieht bis zu einem Zeitpunkt F_2, nach dem die im Zeitraum zwischen F_1 und F_2 erschienenen Prozesse bearbeitet werden, und so weiter. Die Bearbeitung wird also in einzelne Phasen unterteilt, innerhalb derer stets der Algorithmus A zum Einsatz kommt.

Die Bedeutung dieses Verfahrens liegt darin, daß es den kompetitiven Faktor nur unwesentlich verschlechtert.

Satz 3.4.1 *Ist der Algorithmus A c-kompetitiv, so ist das Verfahren A' für neu hinzukommende Prozesse immer noch $2c$-kompetitiv.*

Beweis: Sei σ die Folge der Prozesse, S_1 die Menge der bereits zu Beginn bekannten Prozesse und entsprechend S_{i+1} die Menge der Prozesse, die im Zeitintervall $(F_{i-1}, F_i]$ erscheinen. F_k sei der Endzeitpunkt. Betrachtet man die Folge σ', in der die Prozesse aus S_i bereits zum Zeitpunkt F_{i-2} bekannt sind, so ist klar, daß $c_{opt}(\sigma') \leq c_{opt}(\sigma)$. Weiterhin gilt $F_{k-2} + F_k - F_{k-1} \leq c \cdot c_{opt}(\sigma')$, da die Prozesse in S_k nicht vor F_{k-2} erscheinen und $F_k - F_{k-1} \leq c \cdot c_{opt}(S_k)$ wegen der Kompetitivität von A gilt. Analog folgt $F_{k-1} - F_{k-2} \leq c \cdot c_{opt}(\sigma')$ und somit $F_k \leq 2c \cdot c_{opt}(\sigma') \leq 2c \cdot c_{opt}(\sigma)$. □

Das dargestellte Verfahren kann auch auf die im folgenden genannten Modelle angewandt werden, so daß von nun an stets vorausgesetzt wird, daß alle Prozesse bereits zu Beginn bekannt sind.

Ähnliche Maschinen

Im obigen Modell hatten wir die Annahme gemacht, daß alle Maschinen mit der gleichen Geschwindigkeit laufen. Ein allgemeineres Modell, auch *related machines* genannt, erhält man, indem man jeder Maschine eine Geschwindigkeit s_i zuordnet. Ein Prozeß der Länge C benötigt dann zur Ausführung auf dieser Maschine die Zeit C/s_i. Diese Erweiterung ändert die Situation für Online-Algorithmen entscheidend. Davis und Jaffe [DJ81] zeigen hier eine untere Schranke von $\Omega(\sqrt{m})$ für den kompetitiven Faktor. Jaffe gab einen passenden $O(\sqrt{m})$-kompetitiven Online-Algorithmus an [Jaf80].

Unterbrechung von Prozessen

Eine Verbesserung der Kompetitivität ergibt sich, wenn man die realistische Annahme macht, daß die Prozesse abgebrochen und später auf einer anderen Maschine erneut gestartet werden können (*preemption*). Shmoys *et al.* [SWW91] geben einen Online-Algorithmus an, der diese Möglichkeit ausnutzt und eine Kompetitivität von $O(\log m)$ erreicht. Sie zeigen auch, daß dies optimal ist für deterministische Online-Algorithmen und beliebige Geschwindigkeiten. Die Autoren geben ein weiteres Verfahren an, dessen Kompetitivität von dem Verhältnis $R = s_{max}/s_{min}$ der minimalen und der maximalen Maschinengeschwindigkeit abhängt, damit erhalten sie insgesamt einen $O(\min\{\log R, \log m\})$-kompetitiven Algorithmus. Diese Erweiterung ist ebenfalls optimal, denn für den Fall $R < m$ zeigen die Autoren eine weitere untere Schranke von $\Omega(\log R)$.

Beliebige Maschinen

Erfordern die Prozesse unterschiedliche Resourcen, so ist es natürlich anzunehmen, daß ein Prozeß auf Maschine M_1 schneller als auf Maschine M_2 läuft, während dies für einen anderen Prozeß genau umgekehrt aussieht. Im Modell beliebiger Maschinen wird jedem Prozeß ein Kosten*vektor* zugeordnet, der die Dauer des Prozesses für jede Maschine angibt. Das Prozeßplanungsproblem wird dadurch noch weiter kompliziert.

Shmoys *et al.* [SWW91] geben einen $O(\log n)$-kompetitiven Algorithmus an für den Fall, daß für jeden Prozeß die relative Geschwindigkeit der einzelnen Maschinen bekannt ist und nur die Größe des Prozesses unbekannt. Dabei bezeichnet n die Anzahl der Prozesse.

Aspnes *et al.* [AAF+93] erweitern das Ergebnis und erhalten einen Algorithmus mit der gleichen Kompetitivität, der ohne diese zusätzliche Information auskommt.

Randomisierung

Im Gegensatz zu den meisten anderen Online-Problemen entsteht bei der Prozeßplanung durch die Einführung von Randomisierung keine wesentliche Verbesserung der Kompetitivität. Shmoys *et al.* [SWW91] zeigen, daß kein randomisierter Online-Algorithmus für das einfache Prozeßplanungsproblem besser als $(2 - O(\frac{1}{\sqrt{m}}))$-kompetitiv gegen unwissende Gegenspieler sein kann. Die untere Schranke $(2 - \frac{1}{m})$ von Graham verschiebt sich also nur unwesentlich nach unten.

3.4.2 Verallgemeinerung – Load Balancing

Im vorangehenden Abschnitt wurde stets davon ausgegangen, daß eine Maschine zu einem Zeitpunkt nur einen Prozeß bearbeiten kann. Das *Load Balancing* Problem verallgemeinert dieses Modell auf multitaskingfähige Maschinen. Dabei kann man sich die Maschinen natürlich auch als Kommunikationskanäle eines Netzwerks vorstellen oder als eine andere, gemeinsam benutzbare Resource.

Gegeben sind m Maschinen und eine Folge von Prozessen, die nach und nach erscheinen. Sofort bei Erscheinen eines Prozesses muß dieser einer der Maschinen zugeordnet werden, wobei sich die Belastung (*load*) der Maschine erhöht. Jedem Prozeß ist ein Kostenvektor zugeordnet, der für jede Maschine angibt, um welchen Betrag sich deren Belastung erhöhen würde.

Das Ziel ist es, die maximale Belastung einer Maschine zu minimieren.

Wie im Fall der Prozeßplanung unterscheidet man verschiedene Maschinenmodelle, die jeweils die Beschaffenheit des Kostenvektors festlegen. Weiterhin ist es wichtig, ob die Prozesse *permanent* sind, also nach einmaliger Zuordnung auf der Maschine bleiben, oder *temporär*, d.h. nach einer gewissen Dauer verringert sich die Belastung der Maschine wieder.

Identische Maschinen

Man spricht von identischen Maschinen, wenn alle Einträge des Kostenvektors gleich einer Zahl sind, die nur vom Prozeß abhängt. Alle Prozesse erhöhen also die Belastung jeder

Maschine um den gleichen Wert.

Wie man leicht sieht, überträgt sich Grahams Algorithmus für Prozeplanung auf identischen Maschinen auf diesen Fall. Die Kompetitivität von $2 - \frac{1}{m}$ bleibt dabei bestehen. Lange Zeit war unklar, ob sich dieses Ergebnis noch verbessern läßt. Erst 1992 gaben Bartal *et al.* [BFKV92] einen Online-Algorithmus an, dessen kompetitiver Faktor $(2 - \frac{1}{70}) < 1,986$ unabhängig von m ist. Ihr neuer $(2 - \frac{1}{70})$-kompetitiver Online-Algorithmus sortiert die Maschinen zunächst nach aufsteigender Belastung und teilt diese Liste dann ungefähr in der Mitte in zwei Teile L und R auf. Erscheint ein neuer Prozeß mit Kosten C, so wird dieser der ersten Maschine in R zugeordnet, falls

$$\min R + C \;\leq\; (2 - \frac{1}{70})\,\mathrm{avg} L,$$

und der ersten Maschine in L – dies ist die am wenigsten belastete Maschine – sonst. Dabei steht min für die kleinste Belastung und avg für die mittlere Belastung einer Liste.

Für $m \geq 6$ geben Karger *et al.* einen Algorithmus an, der eine noch etwas verbesserte Kompetitivität von 1,945 erzielt [KPT94].

Bartal *et al.* zeigen, daß auch in diesem Fall Randomisierung die Kompetitivität erhöhen kann, indem sie für den Fall $m = 2$ einen randomisierten 4/3-kompetitiven Online-Algorithmus konstruieren. Sie beweisen auch, daß dies optimal ist [BFKV92].

Ähnliche Maschinen

Wie bei der Prozeßplanung sprechen wir auch beim Load Balancing Problem von *ähnlichen Maschinen*, falls jeder Maschine i eine gewisse Geschwindigkeit s_i zugeordnet ist, so daß die Belastung der Maschine für einen Prozeß mit Kosten C um den Betrag C/s_i ansteigt.

Ein Greedy-Algorithmus für dieses Problem könnte Prozesse stets der am wenigsten belasteten Maschine zuordnen. Dieser Algorithmus ist jedoch nicht kompetitiv, wie man leicht an dem Beispiel von zwei Maschinen mit Geschwindigkeiten 1 und S sieht. Erscheinen zwei Prozesse mit Kosten 1 und S, so würde der teurere Prozeß der langsameren Maschine zugeordnet.

Ein anderer Ansatz ist, stets die Maschine auszuwählen, für die die *resultierende* Belastung, bei der der neue Prozeß also bereits mitgerechnet wird, minimiert wird. Aspnes *et al.* zeigen, daß dieser zweite Greedy-Algorithmus tatsächlich kompetitiv ist [AAF+93]. Die Kompetivität beträgt $\Theta(\log m)$.

Daß auch eine konstante Kompetitivität erreichbar ist, zeigen Azar *et al.* [AKP+93]. Sie konstruieren einen 20-kompetitiven Online-Algorithmus für das Modell ähnlicher Maschinen und zeigen eine untere Schranke von $3 - o(1)$. Den gleichen Algorithmus fanden unabhängig auch Kalyanasundaram und Pruhs.

Aspnes *et al.* konnten dieses Resultat durch die Beschreibung eines 8-kompetitiven Online-Algorithmus noch verbessern [AAF+93].

Maschinen mit Einschränkungen

Den Fall, daß die Prozesse nicht von allen Maschinen bearbeitet werden können, sondern nur von einer Teilmenge, untersuchten zuerst Azar, Naor und Rom [ANR92]. Man kann dieses Modell als Modell ähnlicher Maschinen deuten, wobei auch der Wert ∞ als Geschwindigkeit zugelassen ist. Für den Fall permanenter Prozesse erhalten die Autoren einen $O(\log m)$-kompetitiven Online-Algorithmus und eine passende untere Schranke.

Azar, Broder und Karlin untersuchen den allgemeinen Fall temporärer Prozesse mit Einschränkungen [ABK92]. Sie zeigen, daß sich der kompetitive Faktor von $O(\log m)$ nicht aufrechterhalten läßt, sondern daß in diesem Fall eine untere Schranke von $\Omega(\sqrt{m})$ gilt. Für den Greedy-Algorithmus, der stets die am wenigsten belasteten Maschine wählt, zeigen sie eine Kompetitivität von $\Theta(m^{2/3})$.

Einen Online-Algorithmus, der die gezeigte untere Schranke von $\Omega(\sqrt{m})$ erreicht, gaben zuerst Azar *et al.* [AKP+93] an. Die Kompetitivität des Algorithmus beträgt genauer $2\lceil\sqrt{m}\rceil$.

Bekannte Prozeßlänge

Den Fall von ähnlichen Maschinen und temporären Prozessen unter der zusätzlichen Voraussetzung, daß bei Ankunft eines Prozesses seine Dauer bekannt ist, untersuchen Azar *et al.* [AKP+93]. Sie zeigen, daß die Kenntnis dieser Dauer die Kompetitivität eines Online-Algorithmus auf $O(\log mT)$ verringert, wobei T das Verhältnis von maximaler und minimaler Prozeßdauer ist.

Der allgemeine Fall

Läßt man beliebige Kostenvektoren zu, erhält man das allgemeine Load Balancing Problem (*unrelated machines*).

Aspnes *et al.* [AAF+93] geben für diesen Fall einen $O(\log m)$-kompetitiven Online-Algorithmus an, wobei die Prozesse permanent sind. Weiterhin zeigen sie, daß der natürliche Greedy-Algorithmus nur eine Kompetitivität von $\Theta(m)$ erreicht.

Aus dem Fall von Maschinen mit Einschränkungen überträgt sich die untere Schranke von $\Omega(\log m)$ aus der Arbeit von Azar, Naor und Rom [ANR92] auf dieses Modell.

Ebenso überträgt sich die untere Schranke von $\Omega(\sqrt{m})$ [ABK92] auf den allgemeinen Fall mit temporären Prozessen.

Unterbrechung von Prozessen

Wie im Fall der Prozeßplanung läßt sich auch für das Load Balancing Problem untersuchen, wie die zusätzliche Möglichkeit, Prozesse unterbrechen und auf einer anderen Maschine fortführen zu können, die Situation verändert.

Der Online-Algorithmus kann in diesem Fall stets die optimale Zuordnung erreichen und somit eine Kompetitivität von 1. Der Preis dafür kann jedoch eine sehr hohe Zahl von Unterbrechungen sein, die in der Praxis natürlich Zeit kosten.

Philips und Westbrook [PW93] betrachten den Fall von Maschinen mit Einschränkungen und geben einen Online-Algorithmus an, der eine Kompetitivität von $O(\log m)$ erreicht und dabei nicht zu viele Unterbrechungen vornimmt.

3.4.3 Virtual Circuit Routing

Bei parallelen Rechnern tritt neben der Prozeßverwaltung ein zusätzliches Problem auf, daß eine weitere Verallgemeinerung des Load Balancing Problems darstellt, das Kommunikationsproblem oder *Virtual Circuit Routing* Problem.

Dieses betrifft verwandte Prozesse, die an einem Problem arbeiten und Daten austauschen wollen. Die Prozessoren, auf denen die Prozesse laufen, bilden die Knoten eines Graphen, der die Verbindungen der Prozessoren enthält. Jeder Kante ist dabei ein Gewicht zugeordnet, die Kapazität oder Bandbreite der Verbindung.

Online treffen nun Kommunikationsanfragen ein, die den Start- und den Zielknoten, sowie die gewünschte Kapazität der Verbindung enthalten. Die *Belastung* (*load*) ist nun für die Kanten des Netzwerkes definiert. Bei Zuordnung einer Verbindung nach einer Kommunikationsanfrage erhöht sich die Belastung aller benutzter Kanten um den entsprechenden Wert. Ziel des Online-Algorithmus ist es, eine Zuordnung vorzunehmen, die die maximale Belastung minimiert.

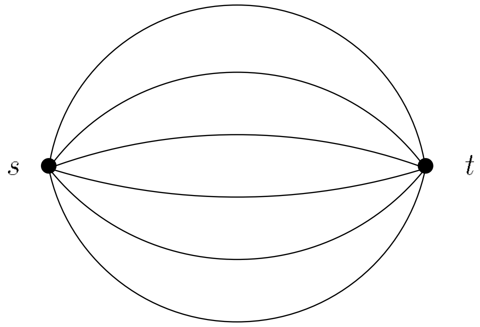

Abbildung 3.6: Ein einfacher Graph für Virtual Circuit Routing.

Das dieses Problem tatsächlich das Load Balancing Problem verallgemeinert, sieht man an dem einfachen Graphen, der in Abbildung 3.6 dargestellt ist. Jeder Maschine ist eine Kante zugeordnet. Der Bandbreite der Kante entspricht der Geschwindigkeit der Maschine. Statt Prozessen mit Kosten C erscheinen Kommunikationsanfragen von s nach t mit gewünschter Kapazität C.

Aspnes *et al.* [AAF+93] untersuchen das Virtual Circuit Routing Problem. Sie geben einen $O(\log m)$-kompetitiven Online-Algorithmus an und zeigen eine gleich lautende untere Schranke. Dabei bezeichnet m die Anzahl der Knoten im Netzwerk.

3.4.4 Offene Probleme

Gibt es einen $O(\sqrt{m})$-kompetitiven Online-Algorithmus für den allgemeinen Fall von Load Balancing mit temporären Prozessen?

3.4.5 Literatur — Load Balancing und Prozeßplanung

[AAF+93] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line load balancing with applications to machine scheduling and virtual circuit routing. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 623–631, 1993.

[ABK92] Y. Azar, A. Broder, and A. Karlin. On-line load balancing. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, pages 218–225, 1992.

[AKP+93] Y. Azar, B. Kalyanasundaram, S. Plotkin, K. R. Pruhs, and O. Waarts. Online load balancing of temporary tasks. In *Proc. Workshop on Algorithms and Data Structures*, pages 119–130, 1993.

[ANR92] Y. Azar, J. Naor, and R. Rom. The competitiveness of on-line assignment. In *Proc. 3rd SIAM Symp. on Discrete Algorithms*, pages 218–225, 1992.

[BFKV92] Y. Bartal, A. Fiat, H. Karloff, and R. Vohra. New algorithms for an ancient scheduling problem. In *Proc. 24th ACM Symp. on Theory of Computing*, 1992.

[DJ81] E. Davis and J. M. Jaffe. Algorithms for scheduling tasks on unrelated processors. *J. of the ACM*, 28:712–736, 1981.

[Gra66] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[Gra69] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. of Computing*, 17(2):416–429, 1969.

[Jaf80] J. M. Jaffe. Efficient scheduling of tasks without full use of processor resources. *Theoretical Computer Science*, 12:1–17, 1980.

[KPT94] D. R. Karger, S. J. Philips, and E. Torng. A better algorithm for an ancient scheduling problem. In *Proc. 5th SIAM Symp. on Discrete Algorithms*, pages 132–140, 1994.

[PW93] S. Philips and J. Westbrook. Online load balancing and network flow. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 402–411, 1993.

[SWW91] D. Shmoys, J. Wein, and D. P. Williamson. Scheduling parallel machines online. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 131–140, 1991.