Skript zur Vorlesung

Online-Algorithmen

Prof. Dr. Heiko Röglin
Institut für Informatik

4. Juli 2016

Vorwort

Dieses Skript ist als Begleitmaterial für die Vorlesung „Online-Algorithmen“ an der Universität Bonn konzipiert. Es deckt den Vorlesungsinhalt ab, dennoch empfehle ich den Teilnehmern der Vorlesung, sich auch das Buch von Borodin und El-Yaniv [6] anzuschauen, welches als Standardwerk im Bereich der Online-Algorithmen gilt.

Ich danke Tobias Brunsch und Michael Etscheid für das Korrekturlesen des Skriptes. Ebenso danke ich den zahlreichen Studenten, die im Laufe der letzten Semester durch ihre Anmerkungen dazu beigetragen haben, das Skript zu verbessern. Für weitere Hinweise auf Fehler und Verbesserungsvorschläge bin ich stets dankbar. Bitte senden Sie diese an meine E-Mail-Adresse `roeglin@cs.uni-bonn.de` oder sprechen Sie mich in der Vorlesung an.

Heiko Röglin

Inhaltsverzeichnis

Kapitel 1

Einleitung

Bei der Lösung von Optimierungsproblemen sind wir in den einführenden Vorlesungen stets davon ausgegangen, dass die konkrete Eingabe dem Algorithmus von Anfang an komplett bekannt ist. In manchen Anwendungen ist das aber nicht der Fall und die Eingabe wird erst Schritt für Schritt aufgedeckt. Probleme mit dieser Eigenschaft nennt man *Online-Probleme*. Die Speicherverwaltung eines Rechners muss beispielsweise bei dem Ausführen eines Programms, ohne dessen zukünftiges Verhalten zu kennen, entscheiden, welche Seiten im CPU-Cache gehalten und welche in den Hauptspeicher ausgelagert werden. Auch im Wirtschaftsleben sind Online-Probleme keine Seltenheit. Autofahrer stehen beispielsweise oft vor der Entscheidung, wann sie eine Tankstelle anfahren, ohne die Entwicklung der Spritpreise in den nächsten Tagen zu kennen.

Man könnte meinen, dass man bei solchen Online-Problemen algorithmisch wenig ausrichten kann. Weiß man nicht, auf welche Seiten ein Programm als nächstes zugreifen wird, so kann man scheinbar keine sinnvolle Entscheidung treffen, welche Seiten in den Hauptspeicher ausgelagert werden sollten. Tatsächlich ist dies aber nicht der Fall und man kann durch geschicktes Verhalten bei Online-Problemen besser abschneiden als bei beliebigem Verhalten.

Um eine bessere Intuition für Online-Probleme und Algorithmen zu entwickeln, schauen wir uns zunächst ein Problem an, das in der Literatur als *Ski Rental* bekannt ist. Es lässt sich einfach beschreiben: Wir gehen zum ersten Mal in unserem Leben Skifahren und wissen noch nicht, ob es uns gefallen wird und wie lange wir Spaß daran haben werden. Es gibt die Möglichkeiten, sich für 800 € eine neue Skiausrüstung zu kaufen oder für 50 € pro Tag eine zu leihen. Solange man noch keine Ausrüstung gekauft hat, steht man jeden Morgen erneut vor der Entscheidung, eine neue Ausrüstung zu kaufen oder für den Tag wieder eine zu leihen.

Kaufen wir direkt am ersten Tag die Ausrüstung, stellen dann aber fest, dass uns Skifahren keinen Spaß macht, so haben wir Kosten in Höhe von 800 €, obwohl 50 € ausgereicht hätten. Wir haben dann also 16 mal so viel ausgegeben wie notwendig gewesen wäre. Die andere extreme Strategie, nie eine Ausrüstung zu kaufen, ist nicht gut, wenn wir oft Skifahren. Sei x die Anzahl an Tagen, an denen wir Skifahren gehen. Für $x \geq 16$ betragen die optimalen Kosten 800 €, die Strategie, stets eine Ausrüstung

zu leihen, verursacht jedoch Kosten in Höhe von $50x$ €. Der Quotient von $50x$ und 800 kann beliebig groß werden.

Betrachten wir die folgende Strategie: wir leihen die ersten 15 Tage eine Ausrüstung und kaufen eine am 16. Tag, wenn uns Skifahren dann immer noch Spaß macht. Ist $x \leq 15$, so ist diese Strategie optimal. Ist $x \geq 16$, so ist es optimal, am ersten Tag direkt eine Ausrüstung zu kaufen, wofür Kosten in Höhe von 800 € anfallen. Unsere Strategie verursacht für die ersten 15 Tage Kosten in Höhe von 750 € für das Ausleihen und an Tag 16 weitere Kosten in Höhe von 800 € für den Kauf der Ausrüstung. Somit sind die Gesamtkosten unserer Strategie 1550 € und damit weniger als doppelt so groß wie die Kosten der optimalen Lösung. Diese Strategie garantiert also, dass wir stets weniger als doppelt so viel ausgeben wie die optimale Strategie, die x von Anfang an kennt.

Natürlich sind die Annahmen in diesem Beispiel, dass man auch nach den ersten Tagen nicht abschätzen kann, ob und wie lange Skifahren einem noch gefallen wird, nicht besonders realistisch. Dieses Problem sollte den Leser deshalb auch nur mit der Denkweise bei Online-Problemen vertraut machen.

Um formal definieren zu können, was ein Online-Problem ist, wiederholen wir zunächst die Definition eines Optimierungsproblems. Ein *Optimierungsproblem* Π besteht aus einer Menge $\mathcal{I}_\Pi$ von *Instanzen* oder *Eingaben*. Zu jeder Instanz $\sigma \in \mathcal{I}_\Pi$ gehört eine Menge $\mathcal{S}_\sigma$ von *Lösungen* und eine *Zielfunktion* $f_\sigma : \mathcal{S}_\sigma \to \mathbb{R}_{\geq 0}$, die jeder Lösung einen reellen Wert zuweist. Zusätzlich ist bei einem Optimierungsproblem vorgegeben, ob wir eine Lösung x mit *minimalem* oder mit *maximalem Wert* $f_\sigma(x)$ suchen. Wir bezeichnen in jedem Falle mit $\mathrm{OPT}(\sigma)$ den Wert einer optimalen Lösung. Wenn die betrachtete Instanz σ klar aus dem Kontext hervorgeht, so schreiben wir oft einfach OPT statt $\mathrm{OPT}(\sigma)$. Für einen Algorithmus A für ein Optimierungsproblem Π bezeichnen wir mit $A(\sigma)$ die Lösung, die er bei Eingabe σ ausgibt, und mit $w_A(\sigma)$ ihren Wert, also $w_A(\sigma) = f_\sigma(A(\sigma))$.

Wir beschränken uns zunächst auf Minimierungsprobleme, man kann aber auch für Maximierungsprobleme eine analoge Theorie entwickeln. Ein Online-Problem unterscheidet sich von einem normalen Optimierungsproblem darin, dass jede Eingabe $\sigma \in \mathcal{I}_\Pi$ aus einer endlichen Folge von Ereignissen besteht. Jede Eingabe $\sigma \in \mathcal{I}_\Pi$ hat also die Form $\sigma = (\sigma_1, \ldots, \sigma_p)$, wobei p nicht fest ist, sondern von Eingabe zu Eingabe variieren kann. Ein *Online-Algorithmus* A muss auf jedes Ereignis σ_i reagieren, wobei er weder die zukünftigen Ereignisse $\sigma_{i+1}, \sigma_{i+2}, \ldots$ noch die Anzahl der zukünftigen Ereignisse kennt. Aus den Entscheidungen, die der Online-Algorithmus A trifft, muss sich am Ende eine Lösung $A(\sigma) \in \mathcal{S}_\sigma$ ergeben. Wie sich diese Lösung genau ergibt und was es überhaupt bedeutet, dass der Algorithmus auf ein Ereignis reagieren muss, hängt vom konkreten Problem ab. Algorithmen, die bei ihren Entscheidungen Wissen über die zukünftigen Ereignisse einsetzen, bezeichnen wir als *Offline-Algorithmen*.

Definition 1.1 (Kompetitiver Faktor)**.** *Ein Online-Algorithmus A für ein Minimierungsproblem Π erreicht einen* kompetitiven Faktor (competitive ratio) *von $r \geq 1$, wenn es eine Konstante $\tau \in \mathbb{R}$ gibt, sodass*

$$w_A(\sigma) = f_\sigma(A(\sigma)) \leq r \cdot \mathrm{OPT}(\sigma) + \tau$$

für alle Instanzen $\sigma \in \mathcal{I}_\Pi$ gilt. Wir sagen dann, dass A ein r-kompetitiver Online-Algorithmus *ist. Gilt sogar*

$$w_A(\sigma) \leq r \cdot \mathrm{OPT}(\sigma)$$

für alle Instanzen $\sigma \in \mathcal{I}_\Pi$, so ist A ein strikt r-kompetitiver Online-Algorithmus.

Ein Online-Algorithmus ist also dann strikt r-kompetitiv, wenn er auf jeder Eingabe höchstens r mal so viele Kosten verursacht wie ein *optimaler Offline-Algorithmus*, der die Eingabe bereits von Anfang an kennt und beliebig viel Zeit hat, sie optimal zu lösen. Die Konstante τ erlaubt man bei r-kompetitiven Algorithmen, damit Eingaben σ mit kleinem Optimum $\mathrm{OPT}(\sigma)$ keinen zu großen Einfluss auf den kompetitiven Faktor haben. Oft führen Online-Algorithmen vor dem ersten Ereignis einen Vorverarbeitungsschritt durch, der gewisse Kosten verursacht, die sich erst bei Eingaben mit vielen Ereignissen amortisieren. Um eine solche Vorgehensweise zu unterstützen und den Einfluss von Eingaben, die nur aus wenigen Ereignissen bestehen, zu begrenzen, haben wir die Konstante τ eingeführt, die die Kosten des Vorverarbeitungsschrittes absorbieren kann.

Die Definition eines strikt r-kompetitiven Algorithmus erinnert stark an die Definition eines Approximationsalgorithmus mit Güte r. Ein Unterschied ist, dass wir nicht gefordert haben, dass die Laufzeit eines Online-Algorithmus polynomiell in der Länge der Eingabe beschränkt ist. Prinzipiell sind auch exponentielle oder noch größere endliche Laufzeiten erlaubt. Die Online-Algorithmen, die wir besprechen werden, sind jedoch alle effizient. Ein effizienter strikt r-kompetitiver Online-Algorithmus für ein Problem Π ist insbesondere ein r-Approximationsalgorithmus für Π. Umgekehrt gilt dies jedoch nicht notwendigerweise, da Approximationsalgorithmen normalerweise nicht online arbeiten, sondern zu Beginn Kenntnis der kompletten Eingabe benötigen.

Kapitel 2

Paging

In modernen Rechnern gibt es eine sogenannte *Speicherhierarchie*, die aus den verschiedenen Speicherarten wie zum Beispiel dem CPU-Cache, dem Hauptspeicher und dem Festplattenspeicher besteht. Je schneller der Speicher ist, desto weniger steht zur Verfügung, weshalb ein sinnvolles Speichermanagement benötigt wird. Wir wollen unsere Betrachtungen hier auf zwei Ebenen beschränken, also beispielsweise den CPU-Cache als schnellen und den Hauptspeicher als langsamen Speicher. Der Speicher ist in *Seiten* eingeteilt, die in der x86-Architektur eine Größe von 4kB haben. Moderne Prozessoren haben einen CPU-Cache von mehreren Megabytes, in dem sie Seiten speichern können. Zugriffe auf diesen CPU-Cache sind deutlich schneller als Zugriffe auf den Hauptspeicher. Greift ein Programm auf eine Seite zu, die nicht im CPU-Cache verfügbar ist, so spricht man von einem *Seitenfehler*. Durch ein geschicktes Speichermanagement *(Paging)* versucht man, die Zahl der Seitenfehler so klein wie möglich zu halten.

Wir formulieren Paging wie folgt als Online-Problem. Gegeben sei eine Zahl $k \geq 2$, die die Anzahl an Seiten bezeichnet, die im Cache gespeichert werden können. Eine Eingabe $\sigma = (\sigma_1, \ldots, \sigma_p)$ besteht aus einer endlichen Folge von Seitenzugriffen. Das heißt, jedes Ereignis σ_i ist ein Element aus $\mathbb{N}$ und entspricht der Nummer der angefragten Seite. Bei einem Ereignis σ_i entstehen dem Algorithmus keine Kosten, falls die Seite σ_i bereits im Cache gespeichert ist. Ansonsten entstehen Kosten 1 und der Algorithmus muss die Seite σ_i in den Cache laden. Ist dieser bereits voll, so muss er eine Seite aus dem Cache auswählen, die verdrängt (d. h. aus dem Cache entfernt) wird. Da es sich um ein Online-Problem handelt, muss die Entscheidung, welche Seite aus dem Cache verdrängt wird, unabhängig von den folgenden Seitenzugriffen erfolgen.

2.1 Deterministische Algorithmen

Wir betrachten nun einige mehr oder weniger natürliche deterministische Algorithmen für das Paging-Problem.

- **LRU (least-recently-used)**
  Verdränge die Seite, deren letzter Zugriff am längsten zurückliegt.
- **LFU (least-frequently-used)**
  Verdränge die Seite, auf die bisher am seltensten zugegriffen wurde.
- **FIFO (first-in-first-out)**
  Verdränge die Seite, die sich am längsten im Cache befindet.
- **LIFO (last-in-first-out)**
  Verdränge die Seite, die als letztes in den Cache geladen wurde.
- **FWF (flush-when-full)**
  Leere den Cache komplett, sobald bei vollem Cache ein Seitenfehler auftritt.
- **LFD (longest-forward-distance)**
  Verdränge die Seite, deren nächster Zugriff am weitesten in der Zukunft liegt.

Bei den ersten fünf dieser Algorithmen handelt es sich um Online-Algorithmen. Die Strategie LFD kann jedoch nur mit Wissen über die Zukunft ausgeführt werden und ist deshalb kein Online-Algorithmus. In Abschnitt 2.1.3 werden wir zeigen, dass es sich bei LFD sogar um einen optimalen Offline-Algorithmus handelt.

2.1.1 Markierungsalgorithmen

Anstatt alle Algorithmen separat zu analysieren, führen wir *Markierungsalgorithmen* ein. Dabei handelt es sich um eine spezielle Klasse von Online-Algorithmen für das Paging-Problem, zu der insbesondere LRU und FWF gehören. Wir zeigen dann, dass jeder Markierungsalgorithmus k-kompetitiv ist. Um definieren zu können, welche Algorithmen zu dieser Klasse gehören, zerlegen wir die Eingabe $\sigma = (\sigma_1, \ldots, \sigma_p)$ zunächst in Phasen.

- Phase 1 ist die maximale Teilsequenz von σ, die mit der ersten Anfrage beginnt und in der auf höchstens k verschiedene Seiten zugegriffen wird.
- Phase $i \geq 2$ ist die maximale Teilsequenz von σ, die direkt im Anschluss an Phase $i-1$ beginnt und in der auf höchstens k verschiedene Seiten zugegriffen wird.

Beispiel: Phaseneinteilung

Für $k = 3$ ergibt sich die folgende beispielhafte Phaseneinteilung.

$$\sigma = (\overbrace{1, 2, 4, 2, 1}^{\text{Phase 1}}, \overbrace{3, 5, 2, 3, 5}^{\text{Phase 2}}, \overbrace{1, 2, 3}^{\text{Phase 3}}, \overbrace{4}^{\text{Phase 4}})$$

Ein Markierungsalgorithmus ist ein Algorithmus, der für jede Seite implizit oder explizit verwaltet, ob sie markiert ist oder nicht. Zu Beginn jeder Phase werden alle Markierungen gelöscht und alle Seiten sind unmarkiert. Wird im Laufe einer Phase auf eine Seite zugegriffen, so wird diese Seite markiert. Ein Algorithmus ist genau

dann ein Markierungsalgorithmus, wenn er für keine Eingabe jemals eine markierte Seite aus dem Cache verdrängt. Ein Markierungsalgorithmus verdrängt demnach bei keiner Eingabe jemals eine Seite, auf die in der aktuellen Phase schon zugegriffen wurde.

Theorem 2.1. *LRU und FWF sind Markierungsalgorithmen.*

Beweis. Nehmen wir an, dass LRU kein Markierungsalgorithmus ist. Dann gibt es eine Eingabe σ, bei der LRU eine markierte Seite x in einer Phase i verdrängt. Sei σ_t der Seitenzugriff in Phase i, bei dem Seite x verdrängt wird. Da Seite x zu diesem Zeitpunkt markiert ist, muss es in Phase i bereits einen Zugriff auf Seite x gegeben haben. Es sei $\sigma_{t'}$ mit $t' < t$ der erste Seitenzugriff auf x in Phase i. Direkt nach dem Seitenzugriff $\sigma_{t'}$ ist Seite x diejenige, die zuletzt angefragt wurde. Somit kann sie von LRU zum Zeitpunkt σ_t nur dann verdrängt werden, wenn in der Teilsequenz $\sigma_{t'+1}, \ldots, \sigma_t$ auf mindestens k Seiten zugegriffen wird, die von x verschieden sind. Läge der Seitenzugriff σ_t in derselben Phase wie der Seitenzugriff $\sigma_{t'}$, so wäre in der Phase also auf mindestens $k+1$ verschiedene Seiten zugegriffen worden, was der Definition einer Phase widerspricht.

Betrachtet man die Arbeitsweise von FWF, so stellt man fest, dass dieser Algorithmus zu Beginn jeder Phase (mit Ausnahme der ersten) einen Seitenfehler verursacht, der dazu führt, dass der Cache komplett geleert wird. Da innerhalb einer Phase auf maximal k verschiedene Seiten zugegriffen wird, verdrängt FWF innerhalb einer Phase niemals Seiten aus dem Cache. Insgesamt können wir festhalten, dass FWF zu jedem Zeitpunkt exakt die markierten Seiten im Cache hält und somit ein Markierungsalgorithmus ist. □

Theorem 2.2. *Jeder Markierungsalgorithmus ist strikt k-kompetitiv.*

Beweis. Sei σ eine beliebige Eingabe für das Paging-Problem und bezeichne ℓ die Anzahl an Phasen, in die σ gemäß obiger Definition zerlegt wird. Gilt $\ell = 1$, so wird in der Eingabe σ nur auf höchstens k verschiedene Seiten zugegriffen. In diesem Fall berechnen Markierungsalgorithmen sogar die optimale Lösung, da sie wie der optimale Offline-Algorithmus genauso viele Seitenfehler verursachen, wie es verschiedene Seiten in der Eingabe σ gibt.

Wir können also davon ausgehen, dass $\ell \geq 2$ gilt. Die Kosten eines beliebigen Markierungsalgorithmus auf der Eingabe σ sind durch ℓk nach oben beschränkt. Der Grund ist, dass in jeder Phase auf höchstens k verschiedene Seiten zugegriffen wird. Da eine Seite beim ersten Zugriff in einer Phase markiert wird und der Algorithmus markierte Seiten niemals verdrängt, verursacht er für jede Seite maximal einen Seitenfehler pro Phase.

Wir behaupten, dass auch der optimale Offline-Algorithmus, der die Eingabe σ zu Beginn komplett kennt, mindestens $k + \ell - 2$ Seitenfehler auf σ verursacht, nämlich k in der ersten Phase und mindestens einen für jede weitere Phase bis auf die letzte. Zum Beweis identifizieren wir $\ell-2$ disjunkte Teilsequenzen in der Eingabe σ. Teilsequenz $i \in \{1, \ldots, \ell - 2\}$ fängt mit dem zweiten Zugriff von Phase $i+1$ an und hört mit dem ersten Zugriff von Phase $i+2$ auf (enthält diesen aber noch).

Beispiel: Einteilung in Teilsequenzen

Für obiges Beispiel mit $k = 3$ ergibt sich die folgende Einteilung in Teilsequenzen.

$$\sigma = (\overbrace{1, 2, 4, 2, 1}^{\text{Phase 1}}, \overbrace{3, \underbrace{5, 2, 3, 5}_{\text{Teilseq. 1}}, \underbrace{1, 2, 3}_{\text{Teilseq. 2}}}^{\text{Phase 2}}, \overbrace{4}^{\text{Phase 3}}, \overbrace{\ }^{\text{Phase 4}})$$

Sei x die erste Seite, auf die in Phase $i + 1$ zugegriffen wird. Dann sind zu Beginn der Teilsequenz i die Seite x und höchstens $k - 1$ weitere Seiten im Cache des optimalen Offline-Algorithmus. In Teilsequenz i wird aber auf k unterschiedliche von x verschiedene Seiten zugegriffen. Also verursacht der optimale Offline-Algorithmus für jede Teilsequenz mindestens einen Seitenfehler. Da zu Beginn von Phase 1 der Cache noch leer ist, verursacht er in der ersten Phase k Seitenfehler.

Damit haben wir gezeigt, dass

$$w_A(\sigma) \leq \ell k \leq (k + \ell - 2)k \leq \mathrm{OPT}(\sigma) \cdot k$$

für jeden Markierungsalgorithmus A und alle Eingaben σ gilt. Somit ist jeder Markierungsalgorithmus strikt k-kompetitiv. □

Die beiden vorangegangenen Theoreme ergeben direkt das folgende Korollar.

Korollar 2.3. *LRU und FWF sind strikt k-kompetitiv.*

Der Leser sollte als Übung beweisen, dass FIFO zwar kein Markierungsalgorithmus aber dennoch k-kompetitiv ist.

Der Beweis von Theorem 2.2 war nicht schwer. Dennoch sollten wir uns etwas Zeit nehmen, die wesentlichen Komponenten noch einmal anzuschauen. Um zu beweisen, dass ein Algorithmus einen bestimmten kompetitiven Faktor erreicht, sind zwei Dinge notwendig: Erstens müssen wir eine obere Schranke für den Wert der Lösung angeben, die der Algorithmus berechnet (in obigem Beweis ℓk). Zweitens müssen wir eine untere Schranke für den Wert der optimalen Lösung angeben (in obigem Beweis $k + \ell - 2$). Oftmals liegt die Schwierigkeit bei der Analyse von Online-Algorithmen genau darin, eine nützliche untere Schranke für den Wert der optimalen Lösung zu finden.

2.1.2 Untere Schranken

Gibt es für einen Online-Algorithmus A keine Konstante r, für die er r-kompetitiv ist, so sagen wir, dass der Algorithmus *nicht kompetitiv* ist.

Theorem 2.4. *LFU und LIFO sind nicht kompetitiv.*

Beweis. Sei $\ell \geq 2$ eine beliebige Konstante und $k \geq 2$. Wir betrachten das Verhalten von LFU und LIFO auf der folgenden Sequenz:

$$\sigma = (1^\ell, 2^\ell, \ldots, (k-1)^\ell, (k, k+1)^{\ell-1}).$$

Es erfolgen zunächst ℓ Zugriffe auf Seite 1, dann ℓ Zugriffe auf Seite 2 usw. Am Ende erfolgen dann jeweils $\ell - 1$ abwechselnde Zugriffe auf Seite k und Seite $k + 1$.

Es ist nicht schwierig eine Strategie anzugeben, die bei Eingabe σ genau $k + 1$ Seitenfehler verursacht. LFU und LIFO füllen bei Eingabe σ zunächst den Cache mit den Seiten 1 bis k. Wird das erste Mal auf Seite $k + 1$ zugegriffen, so wird von beiden Algorithmen Seite k aus dem Cache verdrängt. Danach wird auf Seite k zugegriffen und Seite $k + 1$ wird aus dem Cache verdrängt usw. Das bedeutet, dass bei jedem Seitenzugriff in der Teilsequenz $(k, k + 1)^{\ell-1}$ ein Seitenfehler erfolgt. Zusätzlich wird beim erstmaligen Zugriff auf die Seiten $1, \ldots, k - 1$ jeweils ein Seitenfehler verursacht. Damit erzeugen LFU und LIFO insgesamt genau $k - 1 + 2(\ell - 1)$ Seitenfehler.

Wir müssen zeigen, dass für jede Konstante $\tau \in \mathbb{R}$ und jede Konstante $r \geq 1$ eine Wahl von ℓ existiert, für die

$$w_{\text{LFU}}(\sigma) = w_{\text{LIFO}}(\sigma) > r \cdot \text{OPT}(\sigma) + \tau$$

gilt. Diese Ungleichung ist äquivalent zu

$$\begin{aligned} & k - 1 + 2(\ell - 1) > r \cdot (k + 1) + \tau \\ \iff \; & \ell \geq 1 + \frac{r \cdot (k + 1) + \tau - k + 1}{2}. \end{aligned}$$

Um diese Ungleichung zu erfüllen, müssen wir ℓ für gegebene r, k und τ nur hinreichend groß wählen. Somit gibt es keine Konstante r, für die LFU oder LIFO r-kompetitiv sind. □

Wir haben gesehen, dass LRU und FWF k-kompetitiv sind. Eine natürliche Frage ist, ob wir einen besseren Online-Algorithmus finden oder die Analyse dieser Algorithmen verbessern können. Das folgende Theorem beantwortet diese Frage negativ. Es zeigt, dass jeder *deterministische* Online-Algorithmus bestenfalls k-kompetitiv ist. Deterministisch bedeutet, dass der Algorithmus keine zufälligen Entscheidungen trifft. Das Gegenstück dazu sind *randomisierte* Online-Algorithmen, mit denen wir uns in Abschnitt 2.2 ausführlich beschäftigen werden.

Theorem 2.5. *Es gibt für kein $r < k$ einen deterministischen r-kompetitiven Online-Algorithmus für das Paging-Problem.*

Beweis. Sei A ein beliebiger deterministischer Online-Algorithmus für das Paging-Problem. Wir wollen zeigen, dass für jede beliebige Konstante $\tau \in \mathbb{R}$ und jede Konstante $r < k$ eine Eingabe σ existiert, für die

$$w_A(\sigma) > r \cdot \text{OPT}(\sigma) + \tau$$

gilt. Wir konstruieren dazu eine Eingabe σ mit $k + \ell$ Seitenzugriffen für ein beliebiges $\ell \in \mathbb{N}$, das wir später wählen werden. In dieser Eingabe wird auf lediglich $k + 1$ verschiedene Seiten zugegriffen. Die ersten k Seitenzugriffe $\sigma_1, \ldots, \sigma_k$ erfolgen auf k verschiedene Seiten. Das heißt, nach der Teilsequenz $\sigma_1, \ldots, \sigma_k$ haben sowohl der Algorithmus A als auch der optimale Offline-Algorithmus genau diese k Seiten im Cache.

Bei den folgenden Seitenzugriffen $\sigma_{k+1}, \ldots, \sigma_{k+\ell}$ wird stets auf die Seite zugegriffen, die Algorithmus A nicht im Cache hat. Das bedeutet, Algorithmus A verursacht auf der so konstruierten Sequenz σ genau $k + \ell$ Seitenfehler.

Wir analysieren nun, wie viele Seitenfehler LFD auf der soeben konstruierten Eingabe σ erzeugt. Zwar haben wir noch nicht bewiesen, dass LFD ein optimaler Offline-Algorithmus ist, aber die Anzahl an Seitenfehlern von LFD ist in jedem Falle eine obere Abschätzung für die Anzahl an Seitenfehlern eines optimalen Offline-Algorithmus. LFD verdrängt stets die Seite, deren nächster Zugriff am weitesten in der Zukunft liegt. Somit verursacht LFD einen Seitenfehler bei Seitenzugriff σ_{k+1} und dann frühestens wieder bei Seitenzugriff σ_{2k+1}. Dann verdrängt LFD wieder die Seite, deren nächster Zugriff am weitesten in der Zukunft liegt, und damit erfolgt der nächsten Seitenfehler frühestens bei Seitenzugriff σ_{3k+1} usw. Hierbei haben wir ausgenutzt, dass es insgesamt nur $k + 1$ viele verschiedene Seiten in der konstruierten Eingabe gibt.

Insgesamt können wir die Anzahl an Seitenfehlern, die LFD auf der Eingabe σ verursacht, durch

$$w_{\mathrm{LFD}}(\sigma) \leq k + \left\lceil \frac{\ell}{k} \right\rceil \leq k + 1 + \frac{\ell}{k}$$

nach oben abschätzen. Für jedes $k \geq 2$, jedes $r < k$ und jede Konstante $\tau \in \mathbb{R}$ können wir ℓ so wählen, dass

$$\begin{aligned} w_A(\sigma) &= k + \ell \\ &> r \cdot \left(k + 1 + \frac{\ell}{k} \right) + \tau \\ &\geq r \cdot w_{\mathrm{LFD}}(\sigma) + \tau \end{aligned}$$

gilt. Man rechnet leicht nach, dass dafür nur

$$\ell > \frac{k}{k - r} \cdot (r(k + 1) - k + \tau)$$

gelten muss. Da die Anzahl an Seitenfehlern von LFD mindestens so groß ist wie die Anzahl an Seitenfehlern des optimalen Offline-Algorithmus, gilt für solche ℓ insbesondere

$$w_A(\sigma) > r \cdot \mathrm{OPT}(\sigma) + \tau.$$

Somit ist Algorithmus A nicht r-kompetitiv. □

In vorangegangenen Vorlesungen haben wir oft untere Schranken für die Komplexität von Optimierungsproblemen bewiesen, die auf der Annahme P≠NP beruhen. Bei Online-Problemen ist das nicht so. Theorem 2.5 beruht auf keiner unbewiesenen Annahme.

2.1.3 Optimaler Offline-Algorithmus

Um ein besseres Verständnis für das Paging-Problems zu entwickeln, beweisen wir nun noch, dass LFD ein optimaler Offline-Algorithmus ist. Das bedeutet insbesondere, dass

die Offline-Variante des Paging-Problems effizient gelöst werden kann. Dies steht im Gegensatz zu vielen anderen Online-Problemen, deren Offline-Varianten NP-schwer sind. Ein Grund, sich für effiziente optimale Offline-Algorithmen zu interessieren, ist die experimentelle Evaluation von Online-Algorithmen. Möchte man nämlich experimentell den kompetitiven Faktor eines Online-Algorithmus bestimmen, so benötigt man als Vergleich die Kosten eines optimalen Offline-Algorithmus.

Theorem 2.6. *LFD ist ein optimaler Offline-Algorithmus für das Paging-Problem.*

Beweis. Um das Theorem zu beweisen, betrachten wir einen beliebigen optimalen Offline-Algorithmus OPT für das Paging-Problem. Wir werden das Verhalten von OPT Schritt für Schritt so modifizieren, dass sich seine Kosten durch die Modifikation nicht vergrößern und dass sich der modifizierte Algorithmus am Ende wie LFD verhält. Gelingt uns eine solche Modifikation, dann haben wir gezeigt, dass die Kosten von LFD höchstens so groß sind wie die eines optimalen Offline-Algorithmus. Das bedeutet, dass LFD ein optimaler Offline-Algorithmus ist.

Die Modifikation von OPT beruht auf folgendem Lemma.

Lemma 2.7. *Sei A ein beliebiger von LFD verschiedener Offline-Algorithmus für das Paging-Problem und sei σ eine beliebige Eingabe, auf der er sich nicht wie LFD verhält. Sei ferner σ_t der erste Seitenzugriff, bei dem sich Algorithmus A und LFD unterschiedlich verhalten. Dann existiert ein Offline-Algorithmus B, der die folgenden Eigenschaften erfüllt.*

1. *Algorithmus B verhält sich auf der Teilsequenz $\sigma_1, \ldots, \sigma_{t-1}$ genauso wie Algorithmus A.*
2. *Bei Ereignis σ_t verdrängt Algorithmus B diejenige Seite aus dem Cache, deren nächster Zugriff am weitesten in der Zukunft liegt.*
3. *Auf Eingabe σ verursacht Algorithmus B höchstens so viele Seitenfehler wie Algorithmus A.*

Aus diesem Lemma folgt, dass LFD ein optimaler Offline-Algorithmus ist. Um das einzusehen, betrachten wir einen optimalen Offline-Algorithmus OPT und eine beliebige Eingabe $\sigma = (\sigma_1, \ldots, \sigma_n)$. Wir wenden Lemma 2.7 nun so oft nacheinander an, bis wir einen Algorithmus erhalten, der sich auf der Eingabe σ genauso wie LFD verhält. Konkret erhalten wir nach der ersten Anwendung des Lemmas mit $A = \text{OPT}$ einen Algorithmus $A_1 := B$. Dann wenden wir das Lemma mit $A = A_1$ an und erhalten einen Algorithmus $A_2 := B$ und so weiter. Da sich Algorithmus A_i mindestens auf der Teilsequenz $\sigma_1, \ldots, \sigma_i$ genauso wie LFD verhält, müssen wir die Anwendung des Lemmas nur höchstens n mal wiederholen, bis wir einen Algorithmus A_n erhalten, der sich auf der kompletten Eingabe σ wie LFD verhält. Lemma 2.7 besagt, dass

$$w_{\text{LFD}}(\sigma) = w_{A_n}(\sigma) \leq w_{A_{n-1}}(\sigma) \leq \ldots \leq w_{A_1}(\sigma) \leq w_{\text{OPT}}(\sigma)$$

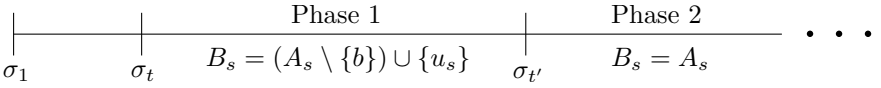
gilt. Damit ist gezeigt, dass LFD auf beliebigen Eingaben σ die gleiche Anzahl an Seitenfehlern verursacht wie ein optimaler Offline-Algorithmus. Mit anderen Worten, LFD ist ein optimaler Offline-Algorithmus. □

Beweis von Lemma 2.7. Wir konstruieren nun einen Algorithmus B mit den geforderten Eigenschaften. Auf der Teilsequenz $\sigma_1, \ldots, \sigma_{t-1}$ verhalten sich die Algorithmen A und B identisch. Bei Ereignis σ_t verursachen beide einen Seitenfehler, verdrängen jedoch unterschiedliche Seiten aus dem Cache. Es sei b die Seite, die Algorithmus B verdrängt, also die Seite, deren nächster Zugriff am weitesten in der Zukunft liegt, und es sei $a \neq b$ die Seite, die Algorithmus A verdrängt. Dann hat der Cache von Algorithmus A nach dem Seitenzugriff σ_t die Form $X \cup \{b\}$ und der Cache von Algorithmus B hat die Form $X \cup \{a\}$, wobei X mit $|X| = k-1$ die Menge der gemeinsamen Seiten im Cache bezeichnet.

Wir beschreiben nun, wie sich Algorithmus B bei den Seitenzugriffen $\sigma_{t+1}, \sigma_{t+2}, \ldots$ verhält. Dazu bezeichnen wir mit A_s und B_s die Cacheinhalte der Algorithmen A und B vor Seitenzugriff σ_s. Gemäß dieser Cacheinhalte teilen wir die Sequenz $\sigma_{t+1}, \sigma_{t+2}, \ldots$ der noch ausstehenden Seitenzugriffe in zwei Phasen ein.

- Phase 1 umfasst alle Zeitpunkte $s \geq t+1$, für die $B_s = (A_s \setminus \{b\}) \cup \{u_s\}$ für ein $u_s \notin A_s$ gilt. Dies ist insbesondere für $s = t+1$ mit $u_{t+1} = a$ der Fall.
- Phase 2 umfasst alle Zeitpunkte $s \geq t+1$, für die $B_s = A_s$ gilt.

Wir definieren das Verhalten von Algorithmus B nun so, dass es einen Zeitpunkt $t' \geq t+1$ gibt, sodass Phase 1 aus den Seitenzugriffen $\sigma_{t+1}, \ldots, \sigma_{t'}$ besteht und Phase 2 aus den Seitenzugriffen $\sigma_{t'+1}, \sigma_{t'+2}, \ldots$. Das heißt insbesondere, dass alle noch ausstehenden Seitenzugriffe zu einer der beiden Phasen gehören. Diese Situation ist in der folgenden Abbildung dargestellt.



Die folgende Fallunterscheidung beschreibt das Verhalten von Algorithmus B in Phase 1. Sei also $B_s = (A_s \setminus \{b\}) \cup \{u_s\}$ mit $u_s \notin A_s$.

1. Fall: Zugriff auf $u \in A_s \cap B_s$. (Es erfolgt kein Seitenfehler.)

2. Fall: Zugriff auf $u \notin A_s \cup B_s$. (A und B verursachen Seitenfehler.)
- **Fall 2.1:** A verdrängt die Seite b.
  Dann verdrängt B die Seite u_s.
  Anschließend gilt $A_{s+1} = B_{s+1}$ ($\rightarrow$ Phase 2).
- **Fall 2.2:** A verdrängt die Seite $v \neq b$.
  Dann verdrängt B die Seite v.
  Anschließend gilt $B_{s+1} = (A_{s+1} \setminus \{b\}) \cup \{u_s\}$.

3. Fall: Zugriff auf u_s. (Nur A verursacht einen Seitenfehler.)
- **Fall 3.1:** A verdrängt die Seite b.
  Anschließend gilt $A_{s+1} = B_{s+1}$ ($\rightarrow$ Phase 2).
- **Fall 3.2:** A verdrängt die Seite $v \neq b$.
  Anschließend gilt $B_{s+1} = (A_{s+1} \setminus \{b\}) \cup \{v\}$.

4. Fall: Zugriff auf b. (Nur B verursacht einen Seitenfehler.)
Dann verdrängt B die Seite u_s.
Anschließend gilt $A_{s+1} = B_{s+1}$ ($\rightarrow$ Phase 2).

Sobald einmal Phase 2 erreicht ist, verhalten sich die Algorithmen A und B auf den noch ausstehenden Seitenzugriffen identisch. Damit wird Phase 2 nicht mehr verlassen, sobald sie einmal erreicht ist. In der obigen Fallunterscheidung sieht man auch, dass in der ersten Phase nur Konfigurationen erreicht werden, die entweder zu Phase 1 oder zu Phase 2 gehören. Somit ist gezeigt, dass die beiden Phasen zusammen alle noch ausstehenden Seitenzugriffe enthalten.

Es bleibt zu zeigen, dass Algorithmus B nicht mehr Seitenfehler verursacht als Algorithmus A. Dazu halten wir zunächst fest, dass es in der obigen Fallunterscheidung nur einen Fall gibt, der dazu führt, dass Algorithmus B einen Seitenfehler verursacht, nicht jedoch Algorithmus A. Dies ist der vierte Fall, in dem auf Seite b zugegriffen wird. Da dieser Fall zu Phase 2 führt, kann er höchstens einmal auftreten. Damit haben wir zunächst gezeigt, dass Algorithmus B höchstens einen Seitenfehler mehr als Algorithmus A verursacht.

Da es sich bei b aber um die Seite handelt, deren Zugriff zum Zeitpunkt t am weitesten in der Zukunft liegt, muss es zuvor einen Zugriff auf Seite a gegeben haben. Solange der dritte Fall nicht aufgetreten ist, gilt $u_s = a$. Das bedeutet, dass der erste Zugriff auf Seite a zu einem Zeitpunkt s mit $u_s = a$ und somit im dritten Fall erfolgt sein muss. In diesem Fall verursacht Algorithmus A einen Seitenfehler, nicht jedoch Algorithmus B. Das heißt, wenn später Fall 4 erreicht wird, hat Algorithmus B verglichen mit Algorithmus A bereits einen Seitenfehler gespart. Insgesamt folgt daraus, dass Algorithmus B nicht mehr Seitenfehler als Algorithmus A verursacht. □

2.1.4 Zusammenfassung der Ergebnisse

Wir fassen nun noch einmal die Ergebnisse über die diskutierten deterministischen Algorithmen für das Paging-Problem zusammen.

LRU (least-recently-used) **FWF** (flush-when-full)	strikt k-kompetitiv da Markierungsalgorithmen
FIFO (first-in-first-out)	kein Markierungsalgorithmus, aber dennoch k-kompetitiv (Übung)
LFU (least-frequently-used) **LIFO** (last-in-first-out)	nicht kompetitiv
LFD (longest-forward-distance)	optimaler Offline-Algorithmus

2.2 Randomisierte Algorithmen

Wir haben in Theorem 2.5 explizit von *deterministischen* Algorithmen gesprochen und im Beweis auch ausgenutzt, dass wir einen solchen Algorithmus vorliegen haben. Nur weil wir angenommen haben, dass der Algorithmus deterministisch ist, konnten wir sicher zu jedem Zeitpunkt sagen, welche Seite er nicht im Cache hat, und auf diese zugreifen. Dies legt sofort die Frage nahe, ob *randomisierte Algorithmen*, die zufällige Entscheidungen treffen dürfen, einen besseren kompetitiven Faktor als k erreichen können, weil sie weniger vorhersehbar sind.

Bevor wir genauer diskutieren, wie der kompetitive Faktor eines randomisierten Online-Algorithmus überhaupt definiert ist, beschreiben wir zunächst zwei einfache solche Algorithmen für das Paging-Problem.

- **RANDOM**
  Verdränge, sobald der Cache voll ist, bei jedem Seitenfehler eine uniform zufällige Seite aus dem Cache.
- **MARK**
  Verwalte wie bei Markierungsalgorithmen ein Bit für jede Seite, das angibt, ob sie markiert ist oder nicht. Wird auf eine Seite zugegriffen, die im Cache ist, so wird diese markiert, falls sie das nicht ohnehin schon ist. Tritt ein Seitenfehler auf und gibt es zu diesem Zeitpunkt eine nicht markierte Seite im Cache, so wähle eine uniform zufällige nicht markierte Seite aus und verdränge diese. Die neue Seite wird dann in den Cache geladen und markiert. Tritt ein Seitenfehler auf und alle Seiten im Cache sind markiert, so lösche alle Markierungen, verdränge eine uniform zufällige Seite aus dem Cache, lade die neue Seite in den Cache und markiere sie.

  Dieser Algorithmus trägt seinen Namen, da er ein naheliegender randomisierter Markierungsalgorithmus ist.

Im Gegensatz zu deterministischen Online-Algorithmen stehen die Kosten $w_A(\sigma)$ eines randomisierten Online-Algorithmus A auf einer Eingabe σ im Allgemeinen nicht fest, sondern es handelt sich dabei um eine Zufallsvariable, die von den zufälligen Entscheidungen des Algorithmus abhängt. Uns interessieren bei der Bestimmung des kompetitiven Faktors insbesondere die *erwarteten Kosten* $\mathbf{E}[w_A(\sigma)]$. Der Erwartungswert einer Zufallsvariable ist das mit der Wahrscheinlichkeit gewichtete Mittel der möglichen Werte. In fast allen Problemen, die wir besprechen werden, sind die Kosten ganze Zahlen. In diesem Fall gilt

$$\mathbf{E}[w_A(\sigma)] = \sum_{i=-\infty}^{\infty} i \cdot \mathbf{Pr}[w_A(\sigma) = i],$$

wobei $\mathbf{Pr}[w_A(\sigma) = i]$ die Wahrscheinlichkeit bezeichnet, dass die Kosten von Algorithmus A auf der Eingabe σ genau gleich i sind.

Definition 2.8 (Kompetitiver Faktor für randomisierte Algorithmen)**.** *Ein randomisierter Online-Algorithmus A für ein Minimierungsproblem Π erreicht einen kompetitiven Faktor von $r \geq 1$, wenn es eine Konstante $\tau \in \mathbb{R}$ gibt, sodass*

$$\mathbf{E}[w_A(\sigma)] \leq r \cdot \mathrm{OPT}(\sigma) + \tau$$

für alle Instanzen $\sigma \in \mathcal{I}_\Pi$ gilt. Gilt diese Ungleichung sogar für $\tau = 0$, so ist A strikt r-kompetitiv.

Diese Definition unterscheidet sich von Definition 1.1 nur darin, dass die Kosten von Algorithmus A auf der Eingabe σ durch die entsprechenden erwarteten Kosten ersetzt

wurden. In der Literatur wird manchmal zwischen verschiedenen Möglichkeiten unterschieden, den kompetitiven Faktor von randomisierten Online-Algorithmen zu definieren. Diese Möglichkeiten werden anhand von verschiedenen Gegenspielern motiviert. Es sei hier nur erwähnt, dass die obige Definition dem blinden (oblivious) Gegenspieler entspricht. Wir gehen in dieser Vorlesung nicht auf die anderen Möglichkeiten ein, den kompetitiven Faktor zu definieren, da sie in der aktuellen Forschung nur von geringer Bedeutung sind.

Lesern, die ihre Kenntnisse in Wahrscheinlichkeitsrechnung auffrischen möchten, sei das Buch von Ulrich Krengel [14] empfohlen. Das Buch von Michael Mitzenmacher und Eli Upfal [17] sowie das Skript zur Vorlesung randomisierte Algorithmen [20] enthalten ebenfalls Einführungen in Wahrscheinlichkeitsrechnung, die insbesondere auf Aspekte eingehen, die für die Informatik wichtig sind.

2.2.1 Potentialfunktionen

Bevor wir zur Analyse der beiden Algorithmen RANDOM und MARK kommen, lernen wir mit *Potentialfunktionen* ein wichtiges Hilfsmittel zur Analyse von Online-Algorithmen kennen. Wir setzen es in diesem Abschnitt zur Analyse von randomisierten Algorithmen ein, aber es findet auch oft bei der Analyse von deterministischen Algorithmen Anwendung. Haben wir einen deterministischen Online-Algorithmus vorliegen, so können im Folgenden alle Erwartungswerte einfach durch die entsprechenden deterministischen Werte ersetzt werden.

Können wir für einen Online-Algorithmus zeigen, dass er bei jedem Ereignis höchstens r-mal so viele Kosten verursacht wie ein optimaler Offline-Algorithmus, so folgt daraus, dass der Algorithmus r-kompetitiv ist. Für viele r-kompetitive Algorithmen gilt diese Aussage aber nicht und stattdessen sind lediglich die durchschnittlichen Kosten pro Schritt höchstens r-mal so groß wie die durchschnittlichen Kosten des optimalen Offline-Algorithmus pro Schritt. Um dies zu zeigen, verwendet man häufig Potentialfunktionen.

Für einen Online-Algorithmus A bezeichnen wir mit $\mathcal{S}_A$ die Menge seiner möglichen Konfigurationen und mit $\mathcal{S}_{\text{OPT}}$ die Menge aller möglichen Konfigurationen eines optimalen Offline-Algorithmus OPT. Was genau eine Konfiguration ist, hängt von dem konkreten Online-Problem und dem Algorithmus ab. Beim Paging ist es naheliegend, als Konfiguration den aktuellen Cacheinhalt zu wählen. Man könnte also $\mathcal{S}_A = \mathcal{S}_{\text{OPT}}$ als die Menge aller Teilmengen von Seiten mit Kardinalität höchstens k wählen. Wir bezeichnen eine Funktion $\Phi : \mathcal{S}_A \times \mathcal{S}_{\text{OPT}} \to \mathbb{R}$, die jedem Paar von Konfigurationen einen reellen Wert zuweist, als Potentialfunktion.

Ist eine Potentialfunktion Φ für einen Online-Algorithmus A gegeben, so erzeugt jede Eingabe $\sigma = (\sigma_1, \ldots, \sigma_n)$ eine Sequenz von Potentialen $\Phi_0, \Phi_1, \ldots, \Phi_n$. Dabei ist Φ_0 das Potential, bevor das erste Ereignis σ_1 von A und OPT verarbeitet wird, und Φ_i für $i \geq 1$ ist das Potential, nachdem das i-te Ereignis σ_i von A und OPT verarbeitet wurde. Ist A ein randomisierter Algorithmus, so ist jedes Potential Φ_i eine Zufallsvariable. Wir können den Erwartungswert $\mathbf{E}[\Phi_i]$ als den Kontostand von Algorithmus A zum

Zeitpunkt i betrachten. Bei jedem Ereignis σ_i ist es das Ziel von Algorithmus A, höchstens r-mal so viele Kosten wie der optimale Offline-Algorithmus zu erzeugen. Gelingt ihm dies nicht und sind seine Kosten höher, so muss er die Differenz von seinem Konto bezahlen, d. h. das Potential sinkt. Gelingt es Algorithmus A hingegen sogar, weniger als die r-fachen Kosten des optimalen Offline-Algorithmus zu erzeugen, so wird die Differenz auf sein Konto eingezahlt, d. h. das Potential steigt. Gibt es eine Konstante τ, für die garantiert werden kann, dass der Kontostand Φ_i zu jedem Zeitpunkt i um höchstens τ kleiner ist als der initiale Kontostand Φ_0, so folgt daraus, dass der Algorithmus r-kompetitiv ist.

Um dies zu formalisieren, bezeichnen wir für $i \geq 1$ mit A_i die Kosten, die Algorithmus A bei Ereignis σ_i entstehen, und mit OPT_i die Kosten, die dem optimalen Offline-Algorithmus bei Ereignis σ_i entstehen. Wir definieren die *amortisierten Kosten von Algorithmus A bei Ereignis σ_i* als

$$a_i = A_i + \Phi_i - \Phi_{i-1}.$$

Ist A ein randomisierter Algorithmus, so handelt es sich auch bei den amortisierten Kosten a_i um Zufallsvariablen.

Theorem 2.9. *Es sei A ein Online-Algorithmus und $r \geq 1$. Gibt es eine Konstante $b \geq 0$ und eine Potentialfunktion Φ, die die folgenden drei Bedingungen für jede Eingabe σ erfüllt, so erreicht Algorithmus A einen kompetitiven Faktor von r.*

1. *Für jedes $i \geq 1$ gilt $\mathbf{E}[a_i] \leq r \cdot \mathrm{OPT}_i$.*
2. *Es gilt $\mathbf{E}[\Phi_0] \leq b$.*
3. *Für jedes $i \geq 1$ gilt $\mathbf{E}[\Phi_i] \geq -b$.*

Beweis. Wir können die erwarteten Kosten von Algorithmus A auf der Eingabe σ als

$$\begin{aligned}\mathbf{E}[w_A(\sigma)] = \mathbf{E}\left[\sum_{i=1}^{n} A_i\right] = \sum_{i=1}^{n} \mathbf{E}[A_i] = \sum_{i=1}^{n} \mathbf{E}[a_i - \Phi_i + \Phi_{i-1}] \\ = \sum_{i=1}^{n} \Big(\mathbf{E}[a_i] - \mathbf{E}[\Phi_i] + \mathbf{E}[\Phi_{i-1}]\Big)\end{aligned}$$

schreiben. Dabei haben wir im zweiten und vierten Schritt die Linearität des Erwartungswertes ausgenutzt[1] und im zweiten Schritt haben wir die Definition der amortisierten Kosten eingesetzt. Nutzt man nun noch aus, dass es sich bei dem Term auf der rechten Seite um eine Teleskopsumme handelt, bei der sich alle Potentiale bis auf $\mathbf{E}[\Phi_0]$ und $-\mathbf{E}[\Phi_n]$ aufheben, so erhält man

$$\mathbf{E}[w_A(\sigma)] = \sum_{i=1}^{n} \mathbf{E}[a_i] + \mathbf{E}[\Phi_0] - \mathbf{E}[\Phi_n] \leq r \cdot \sum_{i=1}^{n} \mathrm{OPT}_i + 2b = r \cdot \mathrm{OPT}(\sigma) + 2b.$$

Da b eine Konstante ist, folgt daraus, dass Algorithmus A einen kompetitiven Faktor von r erreicht. □

[1] Bei der Linearität des Erwartungswertes handelt es sich um eine sehr nützliche Eigenschaft von Erwartungswerten. Sind zwei Zufallsvariablen X und Y gegeben, die beliebig voneinander abhängen dürfen, so gilt $\mathbf{E}[X + Y] = \mathbf{E}[X] + \mathbf{E}[Y]$.

Ist A ein deterministischer Online-Algorithmus, so sind alle Größen, die in dem Theorem vorkommen, fest und dementsprechend können die Erwartungswerte durch die entsprechenden deterministischen Werte ersetzt werden. Dieses Theorem entspricht genau der Intuition, die wir oben formuliert haben. Sind die erwarteten Kosten $\mathbf{E}[A_i]$ von Algorithmus A in einem Schritt größer als die r-fachen Kosten $r \cdot \mathrm{OPT}_i$ des optimalen Offline-Algorithmus, so muss das Potential (d. h. der Kontostand) in diesem Schritt um mindestens diese Differenz fallen, damit die erste Bedingung erfüllt ist. Ist $\mathbf{E}[A_i]$ hingegen kleiner als $r \cdot \mathrm{OPT}_i$, so darf das Potential um die Differenz ansteigen. Die zweite und dritte Bedingung stellen sicher, dass zu Beginn nicht mehr als ein konstanter Betrag auf dem Konto ist und dass das Konto zu keinem Zeitpunkt um mehr als einen konstanten Betrag überzogen ist.

2.2.2 Analyse von RANDOM

Mithilfe der oben eingeführten Methode der Potentialfunktionen können wir den Algorithmus RANDOM analysieren.

Theorem 2.10. *Der Algorithmus RANDOM ist k-kompetitiv.*

Beweis. Es sei OPT ein beliebiger optimaler Offline-Algorithmus und σ eine beliebige Eingabe. Eine Konfiguration, in der es z Seiten gibt, die sowohl RANDOM als auch der optimale Offline-Algorithmus im Cache gespeichert haben, bildet die Potentialfunktion Φ, die wir in diesem Beweis einsetzen, auf $k(k-z)$ ab. Besteht die Eingabe σ aus n Seitenzugriffen, so ergibt sich daraus eine Sequenz $\Phi_0, \Phi_1, \ldots, \Phi_n$ von Potentialen. Bezeichnen wir mit z_i die Anzahl an Seiten, die sowohl RANDOM als auch der optimale Offline-Algorithmus nach dem Seitenzugriff σ_i im Cache gespeichert haben, so gilt $\Phi_i = k(k - z_i)$ für $i \geq 1$. Außerdem gilt $\Phi_0 = k^2$, da zu Beginn die Caches leer sind. Das Potential ist durch Null nach unten und durch k^2 nach oben beschränkt und es ist genau dann gleich Null, wenn beide Algorithmen dieselben k Seiten im Cache gespeichert haben.

Die Intuition hinter dieser Potentialfunktion ist, dass es gut ist, wenn sich die Cacheinhalte des optimalen Offline-Algorithmus und von RANDOM nur wenig unterscheiden, da es dann nur wenige Seiten gibt, bei deren Anfrage RANDOM einen Seitenfehler verursacht, der optimale Offline-Algorithmus aber nicht. Im Idealfall, dass die Cacheinhalte übereinstimmen, kann es überhaupt nicht passieren, dass RANDOM mehr Kosten als der optimale Offline-Algorithmus verursacht. Man kann die Potentialfunktion nun wie folgt interpretieren: Je größer das Potential von RANDOM ist, d. h. je mehr Kosten er verglichen mit den k-fachen Kosten des optimalen Offline-Algorithmus eingespart hat, desto größer darf sein Cache von dem Cache des optimalen Offline-Algorithmus abweichen. Hat er bisher wenig Kosten eingespart, so müssen zumindest die Cacheinhalte ähnlich aussehen. Anders ausgedrückt, kann es Schritte geben, in denen RANDOM mehr als die k-fachen Kosten des optimalen Offline-Algorithmus verursacht, aber nach diesen Schritten muss es mehr gemeinsame Seiten geben, die in beiden Caches vorhanden sind.

Wir bezeichnen mit Rand_i und OPT_i die Kosten von RANDOM bzw. die des optimalen Offline-Algorithmus bei Seitenzugriff σ_i. Diese sind entweder eins oder null, je nachdem ob ein Seitenfehler auftritt oder nicht. Die amortisierten Kosten definieren wir für $i \geq 1$ als

$$a_i = \text{Rand}_i + \Phi_i - \Phi_{i-1}.$$

Sowohl bei den Kosten Rand_i als auch bei den amortisierten Kosten a_i handelt es sich um Zufallsvariablen. Damit wir Theorem 2.9 anwenden können, weisen wir nach, dass für alle $i \geq 1$

$$\mathbf{E}[a_i] \leq k \cdot \text{OPT}_i \tag{2.1}$$

gilt. Ist diese Ungleichung bewiesen, so können wir Theorem 2.9 mit der Konstante $b = k^2$ anwenden und erhalten, dass RANDOM k-kompetitiv ist. Wir weisen die zu (2.1) äquivalente Ungleichung

$$\mathbf{E}[\Phi_i - \Phi_{i-1}] \leq k \cdot \text{OPT}_i - \mathbf{E}[\text{Rand}_i] \tag{2.2}$$

nun mithilfe einer Fallunterscheidung nach. Solange insgesamt auf nur maximal k verschiedene Seiten zugegriffen wird, verhalten sich RANDOM und der optimale Offline-Algorithmus identisch, da sie keine Seiten aus dem Cache verdrängen müssen. Somit verursachen sie auch exakt die gleichen Kosten. Ab dem Zugriff auf die k-te Seite haben RANDOM und der optimale Offline-Algorithmus für den Rest der Sequenz jeweils genau k Seiten im Cache gespeichert. Wir gehen in der Fallunterscheidung davon aus, dass diese Situation bereits erreicht wurde.

Sei $i \geq 1$ beliebig und bezeichne P mit $|P| = z_{i-1}$ die Menge der Seiten, die RANDOM und OPT unmittelbar vor dem Seitenzugriff σ_i gemeinsam im Cache haben. Außerdem bezeichne $p = \sigma_i$ die Seite, auf die als nächstes zugegriffen wird. Zwar handelt es sich bei P um eine Zufallsvariable, aber wir zeigen, dass (2.2) für jede Wahl von P gilt. Das heißt, wir können in der folgenden Fallunterscheidung davon ausgehen, dass die ersten $i-1$ Schritte bereits abgeschlossen sind, und wir wissen, wie die Cacheinhalte unmittelbar vor dem Zugriff σ_i aussehen.

1. p ist im Cache von RANDOM.
   In diesem Fall gilt $\text{Rand}_i = 0$. Ist p auch im Cache des optimalen Offline-Algorithmus, so ändern sich die Cacheinhalte nicht und es gilt $\Phi_i - \Phi_{i-1} = 0$. Ist p nicht im Cache des optimalen Offline-Algorithmus, so gilt $\Phi_i - \Phi_{i-1} \in \{0, -k\}$, je nachdem, ob der optimale Offline-Algorithmus eine Seite aus P oder eine andere Seite verdrängt. Damit ist die linke Seite von (2.2) stets kleiner oder gleich null, während die rechte Seite stets größer oder gleich null ist. Damit ist (2.2) bewiesen.

2. p ist nicht im Cache von RANDOM, aber im Cache von OPT.
   In diesem Fall gilt $\text{Rand}_i = 1$ und $\text{OPT}_i = 0$. Verdrängt RANDOM eine Seite aus P aus dem Cache, so gilt $\Phi_i - \Phi_{i-1} = 0$. Verdrängt RANDOM eine Seite, die nicht zu P gehört, so gilt $\Phi_i - \Phi_{i-1} = -k$. Die Wahrscheinlichkeit, eine Seite zu verdrängen, die nicht zu P gehört, beträgt $(k - z_{i-1})/k$. Damit gilt

$$\mathbf{E}[\Phi_i - \Phi_{i-1}] = \frac{k - z_{i-1}}{k} \cdot (-k) = z_{i-1} - k \leq -1. \tag{2.3}$$

Dabei haben wir $z_{i-1} \leq k-1$ ausgenutzt, was gilt, da p nur im Cache von OPT enthalten ist. Wegen $k \cdot \text{OPT}_i - \text{Rand}_i = -1$ ist damit (2.2) bewiesen.

3. p ist weder im Cache von RANDOM noch im Cache von OPT.
   In diesem Fall nimmt die rechte Seite von (2.2) den Wert $k-1$ an.

   (a) OPT verdrängt eine Seite, die nicht zu P gehört.
   In diesem Fall gilt $\Phi_i - \Phi_{i-1} \in \{0, -k\}$, je nachdem, ob RANDOM eine Seite aus P oder eine andere Seite verdrängt. Somit ist die linke Seite von (2.2) stets kleiner als $k-1$.

   (b) OPT verdrängt eine Seite aus P.
   Verdrängt RANDOM dieselbe Seite, so ändert sich das Potential nicht. Das ist auch der Fall, wenn RANDOM eine Seite verdrängt, die nicht zu P gehört. Nur in dem Fall, dass RANDOM eine andere Seite aus P als der optimale Offline-Algorithmus verdrängt, erhöht sich das Potential um k. Die Wahrscheinlichkeit, dass dies eintritt, beträgt $(z_{i-1}-1)/k$. Daraus folgt

$$\mathbf{E}\left[\Phi_i - \Phi_{i-1}\right] = \frac{z_{i-1}-1}{k} \cdot k \leq k-1. \tag{2.4}$$

Damit ist gezeigt, dass (2.2) für jede Wahl von P gilt. □

Wir haben im vorangegangenen Beweis nachgerechnet, dass die gewählte Potentialfunktion das gewünschte Ergebnis liefert. Wie aber findet man eine geeignete Potentialfunktion? Zunächst muss man eine Idee entwickeln, von welchen Faktoren ein geeignetes Potential abhängen sollte. Das ist problemspezifisch und es gibt kein allgemeines Rezept dafür. Aber selbst, wenn man wie oben bereits die Ähnlichkeit der Cacheinhalte als geeigneten Faktor identifiziert hat, ist die Wahl der Potentialfunktion nicht eindeutig. Warum haben wir bei z gemeinsamen Seiten nicht einfach $k-z$ anstatt $k(k-z)$ als Potential definiert? Auch hierfür ist es schwierig ein Patentrezept anzugeben, aber hat man zumindest schon $a(k-z)$ für ein $a > 0$ als Möglichkeit ins Auge gefasst, so ergibt sich die Wahl $a = k$ aus dem Beweis. Es handelt sich dabei nämlich um die einzige Wahl, für die (2.3) und (2.4) gleichzeitig erfüllt sind. Ist $a < k$, so gilt die zu zeigende Ungleichung (2.2) in Fall 2 im Allgemeinen nicht. Ist $a > k$, so gilt diese Ungleichung in Fall 3b im Allgemeinen nicht.

Auf den ersten Blick erscheint es keinen Grund zu geben, RANDOM dem deterministischen Algorithmus LRU vorzuziehen. Beide erreichen denselben kompetitiven Faktor und LRU erreicht diesen sogar deterministisch. Ein großer Vorteil von RANDOM ist aber, dass er keinen Speicher benötigt, wohingegen LRU für jede Seite speichern muss, wann auf sie das letzte Mal zugegriffen wurde. Da es uns ja gerade darum geht, den kleinen Cache möglichst sinnvoll zu nutzen, ist dieser Vorteil nicht zu unterschätzen. Außerdem benötigt RANDOM pro Anfrage nur eine konstante Laufzeit, wohingegen bei LRU eine Priority Queue verwaltet werden muss, um die Seite im Cache zu finden, deren letzter Zugriff am längsten zurückliegt.

Untere Schranke für den kompetitiven Faktor von RANDOM

Wir zeigen nun eine zu Theorem 2.10 passende untere Schranke für den kompetitiven Faktor von RANDOM. Zunächst beschäftigen wir uns kurz mit *geometrischen Zufallsvariablen*. Solche Zufallsvariablen treten immer dann auf, wenn wir ein Experiment, das entweder erfolgreich sein kann oder nicht, solange unabhängig wiederholen, bis der erste Erfolg eintritt. Wir werfen also beispielsweise solange einen Würfel, bis das erste Mal die Augenzahl sechs erscheint. Die Zufallsvariable X, die in einem solchen Szenario die Anzahl an Wiederholungen beschreibt, ist geometrisch verteilt. Ist $p \in (0, 1]$ die Wahrscheinlichkeit, dass ein einzelnes Experiment erfolgreich ist, so nimmt X nur natürliche Zahlen als Werte an und für jedes $i \in \mathbb{N}$ gilt

$$\mathbf{Pr}[X = i] = (1 - p)^{i-1} p.$$

Eine kurze Rechnung zeigt $\mathbf{E}[X] = 1/p$. Wir sagen, dass X *geometrisch mit Parameter p* verteilt ist. Im Erwartungswert müssen wir den Würfel also sechsmal werfen, bis wir das erste Mal eine Sechs sehen. In der unteren Schranke für RANDOM werden abgeschnittene geometrische Zufallsvariablen vorkommen. Eine bei $n \in \mathbb{N}$ abgeschnittene geometrische Zufallsvariable X ist die Zufallsvariable $\min\{X, n\}$. Übertragen auf obige Anschauung bedeutet das, dass wir nach $n - 1$ erfolglosen Versuchen, den n-ten Versuch unabhängig von seinem tatsächlichen Ausgang als erfolgreich definieren.

Lemma 2.11. *Es sei X eine geometrisch verteilte Zufallsvariable mit Parameter p und es sei $n \in \mathbb{N}$. Für die Zufallsvariable $Y = \min\{X, n\}$ gilt*

$$\mathbf{E}[Y] = \frac{1 - (1 - p)^n}{p}.$$

Beweis. Mit $q = 1 - p$ erhalten wir

$$\begin{aligned}
\mathbf{E}[Y] &= \sum_{i=1}^{n} i \cdot \mathbf{Pr}[\min\{X, n\} = i] = \sum_{i=1}^{n-1} i \cdot \mathbf{Pr}[X = i] + \sum_{i=n}^{\infty} n \cdot \mathbf{Pr}[X = i] \\
&= \sum_{i=1}^{\infty} \min\{i, n\} \cdot \mathbf{Pr}[X = i] = \sum_{i=1}^{\infty} \min\{i, n\} \cdot p \cdot q^{i-1} \\
&= \sum_{i=1}^{\infty} i \cdot p \cdot q^{i-1} - \sum_{i=n+1}^{\infty} (i - n) \cdot p \cdot q^{i-1} \\
&= \sum_{i=1}^{\infty} i \cdot \mathbf{Pr}[X = i] - \sum_{i=1}^{\infty} i \cdot p \cdot q^{i+n-1} \\
&= \mathbf{E}[X] - q^n \cdot \sum_{i=1}^{\infty} i \cdot p \cdot q^{i-1} = (1 - q^n) \cdot \mathbf{E}[X] = \frac{1 - q^n}{p}.
\end{aligned}$$

□

Nun haben wir alle Hilfsmittel beisammen, um die untere Schranke für den kompetitiven Faktor von RANDOM zu beweisen.

Theorem 2.12. *Der kompetitive Faktor von RANDOM beträgt mindestens k.*

Beweis. Wir betrachten die Sequenz

$$\sigma = ((a_1, \ldots, a_k), (b_1, a_2, \ldots, a_k)^\ell, (b_2, a_2, \ldots, a_k)^\ell, \ldots, (b_m, a_2, \ldots, a_k)^\ell).$$

Zunächst wird auf die Seiten $a_1, \ldots, a_k$ jeweils einmal zugegriffen und anschließend gibt es m Blöcke der Form $(b_i, a_2, \ldots, a_k)$, die jeweils ℓ mal wiederholt werden. Dabei bezeichnen $a_1, \ldots, a_k, b_1, \ldots, b_m$ paarweise verschiedene Seiten und die Werte ℓ und m werden später gewählt.

Der optimale Offline-Algorithmus verursacht $\mathrm{OPT}(\sigma) = k + m$ Seitenfehler auf der Sequenz σ. Zunächst verursacht er bei den Zugriffen auf die Seiten $a_1, \ldots, a_k$ jeweils einen Seitenfehler und anschließend verursacht er für jede Teilsequenz $(b_i, a_2, \ldots, a_k)^\ell$ genau einen Seitenfehler.

Wir bestimmen nun, wie viele Seitenfehler RANDOM im Erwartungswert auf einer Teilsequenz der Form $(b_i, a_2, \ldots, a_k)^\ell$ verursacht. Zunächst können wir festhalten, dass RANDOM unmittelbar bevor er diese Teilsequenz erreicht, höchstens $k-1$ der Seiten $b_i, a_2, \ldots, a_k$ im Cache hat. Dies liegt daran, dass die Seite b_i vorher noch nicht angefragt wurde. Das bedeutet, dass es in der Teilsequenz mindestens einen Seitenfehler geben wird. Wir nennen einen Seitenfehler erfolgreich, wenn davor genau $k-1$ der Seiten $b_i, a_2, \ldots, a_k$ im Cache sind und RANDOM die einzige Seite aus dem Cache verdrängt, die nicht zu $b_i, a_2, \ldots, a_k$ gehört. Ist ein Seitenfehler erfolgreich, so treten in der Teilsequenz keine weiteren Seitenfehler mehr auf. Selbst wenn wir optimistisch davon ausgehen, dass genau $k-1$ der Seiten $b_i, a_2, \ldots, a_k$ im Cache sind, beträgt die Wahrscheinlichkeit, dass RANDOM die richtige Seite aus dem Cache verdrängt, genau $1/k$. Das bedeutet, die Wahrscheinlichkeit, dass ein Seitenfehler erfolgreich ist, beträgt höchstens $1/k$.

Würden wir den Block $(b_i, a_2, \ldots, a_k)$ unendlich oft wiederholen, so könnten wir die Anzahl an Seitenfehlern bis zum ersten erfolgreichen Seitenfehler durch eine geometrische Zufallsvariable mit Parameter $1/k$ nach unten abschätzen. Dadurch, dass der Block aber nur ℓ-mal wiederholt wird, können wir die Anzahl an Seitenfehlern nur durch eine abgeschnittene geometrische Zufallsvariable abschätzen. Da es bis zum ersten erfolgreichen Seitenfehler in jedem Durchlauf des Blockes $(b_i, a_2, \ldots, a_k)$ mindestens einen Seitenfehler gibt, treten unter der Annahme, dass alle Seitenfehler erfolglos sind, in der Teilsequenz $(b_i, a_2, \ldots, a_k)^\ell$ mindestens ℓ Seitenfehler auf. Demzufolge können wir die Anzahl an Seitenfehlern in der Teilsequenz durch eine bei ℓ abgeschnittene geometrische Zufallsvariable mit Parameter $1/k$ nach unten abschätzen. Gemäß Lemma 2.11 beträgt die erwartete Anzahl an Seitenfehlern in dieser Teilsequenz somit mindestens

$$k \cdot \left(1 - \left(1 - \frac{1}{k}\right)^\ell\right).$$

Mit der Linearität des Erwartungswertes folgt, dass wir die erwarteten Kosten von RANDOM auf σ wie folgt nach unten abschätzen können:

$$\mathbf{E}\left[w_{\mathrm{RANDOM}}(\sigma)\right] \geq k + mk \cdot \left(1 - \left(1 - \frac{1}{k}\right)^\ell\right) \geq mk \cdot \left(1 - \left(1 - \frac{1}{k}\right)^\ell\right).$$

Seien nun beliebige Konstanten $r < k$ und τ gegeben. Wir müssen zeigen, dass wir die Parameter m und ℓ so wählen können, dass

$$\mathbf{E}\left[w_{\mathrm{RANDOM}}(\sigma)\right] > r \cdot \mathrm{OPT}(\sigma) + \tau$$

gilt. Auf Grund unserer Vorüberlegungen genügt es, die Parameter so zu wählen, dass die Ungleichung

$$mk \cdot \left(1 - \left(1 - \frac{1}{k}\right)^{\ell}\right) > r(k+m) + \tau \tag{2.5}$$

erfüllt ist. Da $\lim_{\ell \to \infty}(1 - 1/k)^{\ell} = 0$ und $r < k$ gelten, existiert ein ℓ, für das

$$r' := k\left(1 - \left(1 - \frac{1}{k}\right)^{\ell}\right) > r$$

gilt. Damit vereinfacht sich (2.5) zu

$$mr' > r(k+m) + \tau,$$

was für $m = 1 + (rk + \tau)/(r' - r)$ erfüllt ist. Damit ist gezeigt, dass RANDOM für kein $r < k$ einen kompetitiven Faktor von r erreicht. □

2.2.3 Analyse von MARK

Wir zeigen in diesem Abschnitt, dass der Algorithmus MARK einen deutlich besseren kompetitiven Faktor als RANDOM erreicht. Für $k \geq 1$ bezeichnen wir im Folgenden mit H_k die *k-te harmonische Zahl*, also

$$H_k = \sum_{i=1}^{k} \frac{1}{i}.$$

Aus früheren Vorlesungen ist dem Leser wahrscheinlich bekannt, dass $H_k = \Theta(\log k)$ gilt. Es gilt sogar die schärfere Abschätzung

$$\ln k < H_k \leq 1 + \ln k.$$

Theorem 2.13. *Der Algorithmus MARK ist $2H_k$-kompetitiv.*

Beweis. Es sei σ eine beliebige Eingabe mit n Phasen bezüglich derselben Phaseneinteilung wie bei deterministischen Paging-Algorithmen. In der ersten Phase verursachen MARK und der optimale Offline-Algorithmus jeweils genau k Seitenfehler. In jeder weiteren Phase $i \geq 2$ hängt die Anzahl an erwarteten Seitenfehlern, die MARK verursacht, davon ab, wie viele neue Seiten es in Phase i verglichen mit Phase $i-1$ gibt. Formal teilen wir die Menge der Seiten, auf die in Phase i zugegriffen wird, in neue und alte Seiten ein. Dabei nennen wir eine Seite alt, wenn auf sie auch in Phase $i-1$ zugegriffen wird, und sonst neu. Es sei m_i die Anzahl an neuen Seiten in Phase i. Dann beträgt die Anzahl an alten Seiten in jeder Phase i bis auf die letzte genau $k - m_i$. In der letzten Phase n beträgt die Anzahl an alten Seiten höchstens $k - m_n$.

Neue Seiten verursachen in Phase i jeweils genau einen Seitenfehler, egal wann sie in der Phase vorkommen. Alte Seiten hingegen sind zu Beginn von Phase i im Cache. Sie verursachen nur dann einen Seitenfehler, wenn sie in Phase i vor ihrem ersten Zugriff verdrängt werden. Die Wahrscheinlichkeit, dass eine alte Seite bei ihrem ersten Zugriff in Phase i noch im Cache ist, sinkt mit der Anzahl an neuen Seiten, auf die vorher

in Phase i zugegriffen wird. Demzufolge ist die erwartete Anzahl an Seitenfehlern von MARK in Phase i am größten, wenn zunächst auf alle neuen Seiten mindestens einmal zugegriffen wird und erst danach auf die alten. Wir gehen im Folgenden davon aus, dass in der Sequenz σ die Reihenfolge der Zugriffe so gewählt ist. Dadurch kann für jede alte Seite die Wahrscheinlichkeit, dass sie bei ihrem ersten Zugriff in einer Phase noch im Cache ist, höchstens sinken, und somit können die erwarteten Kosten von MARK höchstens steigen.

Prinzipiell kann eine solche Umordnung der Zugriffe in σ jedoch auch die Kosten des optimalen Offline-Algorithmus vergrößern, weshalb wir a priori nicht ohne Beschränkung der Allgemeinheit davon ausgehen können, dass in jeder Phase zunächst auf die neuen Seiten zugegriffen wird. Die Abschätzung für die Kosten eines optimalen Offline-Algorithmus, die wir unten herleiten werden, hängt jedoch nur von den m_i und nicht von der Reihenfolge der Zugriffe innerhalb einer Phase ab. Aus diesem Grunde können wir im Folgenden tatsächlich ohne Beschränkung der Allgemeinheit davon ausgehen, dass in jeder Phase zunächst auf die neuen Seiten zugegriffen wird.

Außerdem füllen wir gegebenenfalls die letzte Phase n auf, sodass auch dort auf genau $k - m_n$ alte Seiten zugegriffen wird. Da wir dafür weniger als k Zugriffe zu der Sequenz σ hinzufügen müssen, ändert dies die Kosten von MARK und die des optimalen Offline-Algorithmus nur um höchstens eine Konstante, was wegen der Konstante τ in Definition 2.8 den kompetitiven Faktor nicht beeinflusst.

Wir sortieren die alten Seiten nun nach ihrem ersten Zugriff in Phase i und bezeichnen für $j \in \{1, \ldots, k - m_i\}$ mit p_j die Wahrscheinlichkeit, dass die j-te alte Seite bei ihrem ersten Zugriff in Phase i noch im Cache enthalten ist. Es gilt dann

$$p_1 = \frac{k - m_i}{k},$$

denn von den k Seiten, die zu Beginn von Phase i im Cache sind, werden vor dem Zugriff auf die erste alte Seite m_i uniform zufällig ausgewählte Seiten verdrängt. Allgemein gilt

$$p_j = \frac{k - m_i - (j-1)}{k - (j-1)},$$

für jedes $j \in \{1, \ldots, k - m_i\}$. Um dies einzusehen, betrachten wir Zähler und Nenner getrennt. Der Zähler $k - m_i - (j-1)$ entspricht der Anzahl an unmarkierten alten Seiten im Cache von MARK direkt vor dem ersten Zugriff auf die j-te alte Seite. Der Nenner entspricht der Anzahl an unmarkierten alten Seiten insgesamt (d. h. hier werden auch die unmarkierten alten Seiten gezählt, die MARK nicht mehr im Cache hat). Da die unmarkierten alten Seiten bisher in Phase i noch nicht vorgekommen sind, besitzt jede davon die gleiche Wahrscheinlichkeit, noch im Cache von MARK zu sein. Deshalb ist von den $k - (j-1)$ in Frage kommenden Seiten eine uniform zufällig gewählte Teilmenge der Größe $k - m_i - (j-1)$ im Cache. Da die j-te alte Seite unmittelbar vor ihrem ersten Zugriff nicht markiert ist, entspricht die Wahrscheinlichkeit p_j, dass sie im Cache ist, genau dem Quotienten dieser beiden Terme.

Die erwartete Anzahl an Seitenfehlern, die die j-te alte Seite verursacht, beträgt

$$p_j \cdot 0 + (1 - p_j) \cdot 1 = 1 - p_j.$$

Nutzt man die Linearität des Erwartungswertes und berücksichtigt man, dass jede der $m_i \geq 1$ neuen Seiten einen Seitenfehler verursacht, so kann man die erwartete Anzahl an Seitenfehlern von MARK in Phase $i \geq 2$ durch

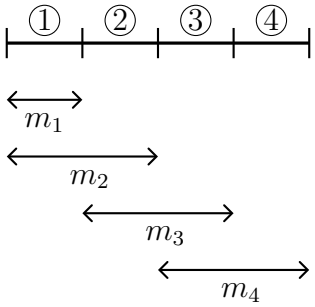
$$m_i + \sum_{j=1}^{k-m_i} (1 - p_j) = m_i + \sum_{j=1}^{k-m_i} \frac{m_i}{k-(j-1)} \leq m_i \cdot \sum_{j=1}^{k} \frac{1}{k-(j-1)} = m_i \cdot H_k$$

nach oben beschränken. Definieren wir außerdem $m_1 = k$, so ist diese Abschätzung auch für die erste Phase gültig, in der MARK genau k Seitenfehler verursacht. Wir können wieder mithilfe der Linearität des Erwartungswertes die erwarteten Kosten von MARK auf der Eingabe σ durch

$$\mathbf{E}\left[w_{\text{MARK}}(\sigma)\right] \leq H_k \cdot \sum_{i=1}^{n} m_i$$

nach oben abschätzen.

Nun müssen wir nur noch die Kosten des optimalen Offline-Algorithmus nach unten beschränken. Dazu betrachten wir für $i \in \{2, \ldots n\}$ die zwei aufeinanderfolgenden Phasen $i-1$ und i. Insgesamt wird in der Teilsequenz, die aus diesen beiden Phasen besteht, auf $k + m_i$ viele verschiedene Seiten zugegriffen. Da maximal k von diesen zu Beginn von Phase $i-1$ im Cache enthalten sein können, verursacht jeder Algorithmus auf dieser Teilsequenz mindestens m_i Seitenfehler. Außerdem verursacht jeder Algorithmus in der ersten Phase genau $m_1 = k$ Seitenfehler. Diese Situation ist in der folgenden Abbildung für eine Sequenz mit vier Phasen beispielhaft dargestellt.



Betrachten wir die erste Phase und alle Teilsequenzen von zwei aufeinanderfolgenden Phasen und addieren die Seitenfehler auf, so erhalten wir $\sum_{i=1}^{n} m_i$. In dieser Summe haben wir jeden Seitenfehler maximal zweimal gezählt. Deshalb erhalten wir

$$\text{OPT}(\sigma) \geq \frac{1}{2} \sum_{i=1}^{n} m_i$$

als untere Schranke für die Kosten des optimalen Offline-Algorithmus. Insgesamt folgt, wie gewünscht,

$$\mathbf{E}\left[w_{\text{MARK}}(\sigma)\right] \leq H_k \cdot \sum_{i=1}^{n} m_i \leq 2H_k \cdot \text{OPT}(\sigma). \qquad \square$$

Das Theorem zeigt, dass randomisierte Algorithmen einen deutlich besseren kompetitiven Faktor erreichen können, als dies mit deterministischen Algorithmen möglich ist. Dies ist nicht nur für das Paging-Problem der Fall, sondern es ist eine Eigenschaft von sehr vielen Online-Problemen.

2.2.4 Untere Schranke für randomisierte Online-Algorithmen

Zum Abschluss zeigen wir nun noch, dass es sich bei MARK um einen asymptotisch optimalen Online-Algorithmus handelt.

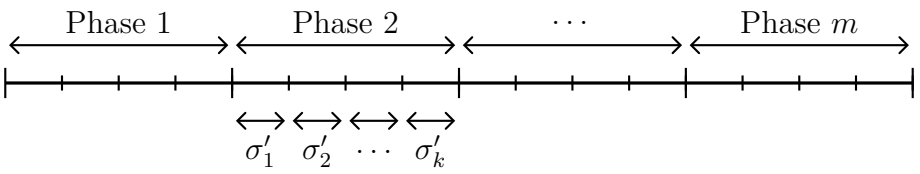
Theorem 2.14. *Es existiert kein randomisierter Online-Algorithmus für das Paging-Problem mit einem kleineren kompetitiven Faktor als H_k.*

Beweis. Es sei A ein beliebiger randomisierter Paging-Algorithmus. Wir konstruieren eine Eingabe, in der auf $k+1$ verschiedene Seiten zugegriffen wird. Sobald in einer Sequenz insgesamt auf mindestens k von diesen Seiten zugegriffen wurde, gibt es genau eine der $k+1$ Seiten, die Algorithmus A nicht im Cache hat. Für jede Sequenz σ von Seitenzugriffen und jede Seite i können wir die Wahrscheinlichkeit p_i berechnen, mit der Algorithmus A nach der Bearbeitung der Sequenz σ die Seite i nicht im Cache hat. Dies ergibt eine Wahrscheinlichkeitsverteilung $(p_1, \ldots, p_{k+1})$ mit $p_i \in [0,1]$ und $\sum_{i=1}^{k+1} p_i = 1$.

Wir konstruieren die Eingabe nach und nach basierend auf dieser Wahrscheinlichkeitsverteilung. In Anlehnung an die untere Schranke für deterministische Algorithmen, die wir in Theorem 2.5 gezeigt haben, läge es nahe, stets auf die Seite i zuzugreifen, für die p_i am größten ist. Das alleine reicht allerdings nicht aus. Es gibt einen Online-Algorithmus, der auf Eingaben, die so konstruiert werden, einen konstanten kompetitiven Faktor erreicht, der nicht von k abhängt. Dieser Algorithmus basiert auf der Idee, zu jedem Zeitpunkt jede Seite (abgesehen von der, die zuletzt angefragt wurde) mit einer Wahrscheinlichkeit von ungefähr $1/k$ nicht im Cache zu haben. Die erwarteten Kosten eines solchen Algorithmus betragen pro Schritt nur ungefähr $1/k$ und mit der richtigen Strategie kann der Algorithmus erzwingen, dass auch der optimale Offline-Algorithmus bis auf einen konstanten Faktor Kosten in der gleichen Größenordnung verursacht. Die Details sollte der Leser sich als Übung überlegen.

Die Sequenz σ, die wir tatsächlich konstruieren, besteht aus einer noch zu wählenden Anzahl m an Phasen. In der ersten Phase wird auf alle $k+1$ Seiten jeweils einmal zugegriffen. In jeder weiteren Phase wird auf genau k verschiedene Seiten zugegriffen, aber gegebenenfalls mehrfach. Genauso wie bei Markierungsalgorithmen sagen wir, dass eine Seite markiert ist, wenn auf sie bereits in der entsprechenden Phase zugegriffen wurde. Im Unterschied zu Markierungsalgorithmen ist aber zu Beginn jeder Phase zusätzlich bereits die letzte Seite, auf die in der vorangegangenen Phase zugegriffen wurde, markiert.

Jede Phase σ' ist in k Teilphasen $\sigma'_1, \ldots, \sigma'_k$ unterteilt. Diese haben die Eigenschaft, dass in jeder Teilphase genau eine Seite neu markiert wird. Das bedeutet, dass am Ende der j-ten Teilphase genau $j+1$ Seiten markiert sind. Jede Teilphase besteht entweder nur aus einem Zugriff auf eine noch nicht markierte Seite oder es wird zunächst auf markierte Seiten zugegriffen und danach erfolgt genau ein Zugriff auf eine noch nicht markierte Seite. Diese Konstruktion ist in der folgenden Abbildung illustriert.



Um die Kosten eines optimalen Offline-Algorithmus auf der Sequenz σ nach oben abzuschätzen, betrachten wir einen Algorithmus, der beim letzten Zugriff in einer Phase immer genau die Seite verdrängt, die in der darauffolgenden Phase als letztes markiert wird. Dies stellt sicher, dass erst der letzte Zugriff der darauffolgenden Phase den nächsten Seitenfehler verursacht. Somit entsteht in jeder Phase (mit Ausnahme der ersten) genau ein Seitenfehler. Berücksichtigt man die $k+1$ Seitenfehler, die in der ersten Phase entstehen, so folgt insgesamt

$$\mathrm{OPT}(\sigma) \leq (k+1) + (m-1) = k + m.$$

Die j-te Teilphase σ'_j wird mit dem folgenden Algorithmus konstruiert. Es sei M die Menge der zu Beginn von σ'_j markierten Seiten, es sei $u = k + 1 - j$ die Anzahl an unmarkierten Seiten zu Beginn von σ'_j und es sei $\gamma = \sum_{i \in M} p_i$ die Wahrscheinlichkeit, dass die Seite, die sich nicht im Cache befindet, markiert ist.

Das Ziel ist es, die Teilphase σ'_j so zu konstruieren, dass Algorithmus A im Erwartungswert in dieser Teilphase Kosten von mindestens $1/u$ entstehen. Dazu unterschieden wir zwei Fälle.

1. Falls $\gamma = 0$ gilt, muss es eine Seite $a \notin M$ geben, für die $p_a \geq 1/u$ gilt. In diesem Fall besteht die j-te Teilphase nur aus einem Zugriff auf Seite a.

2. Falls $\gamma > 0$ gilt, sei $a \in M$ eine beliebige Seite mit $p_a > 0$ und es sei $\varepsilon = p_a$. Der erste Zugriff in der j-ten Teilphase erfolgt auf Seite a. Anschließend werden an diesen Seitenzugriff weitere Seitenzugriffe gemäß der folgenden Regel angehängt.

   $E := p_a$;
   while $((E < 1/u)$ **and** $(\gamma > u\varepsilon))$ {
   Hänge Zugriff $b \in M$ mit $b = \arg\max_{i \in M} p_i$ an.
   $E := E + p_b$;
   Berechne $p_1, \ldots, p_{k+1}$ und $\gamma = \sum_{i \in M} p_i$ neu.
   }
   Hänge Zugriff auf $b' \notin M$ mit $b' = \arg\max_{i \notin M} p_i$ an.

Zunächst halten wir fest, dass die while-Schleife stets terminiert, denn solange $\gamma > u\varepsilon$ gilt, ist $p_b \geq \gamma/|M| > u\varepsilon/|M|$. Somit erhöht sich E mit jedem Durchlauf der Schleife um mindestens $u\varepsilon/|M|$. Wir argumentieren nun, dass die erwarteten Kosten von A auf der so konstruierten j-ten Teilsequenz mindestens $1/u$ betragen. Dazu ist zunächst die Beobachtung wichtig, dass E per Konstruktion zu jedem Zeitpunkt den erwarteten Kosten von Algorithmus A auf dem bisher konstruierten Teil der Teilsequenz σ'_j entspricht. Wenn die Schleife terminiert, weil $E \geq 1/u$ gilt, ist also nichts mehr zu zeigen. Ansonsten gilt $\gamma \leq u\varepsilon$ und damit gilt für die im letzten Schritt angehängte Seite

$$p_{b'} \geq \frac{1-\gamma}{u} \geq \frac{1-u\varepsilon}{u} = \frac{1}{u} - \varepsilon.$$

Insgesamt betragen die erwarteten Kosten von A auf der Teilsequenz σ'_j in diesem Fall mindestens

$$p_a + p_{b'} \geq \varepsilon + \frac{1}{u} - \varepsilon = \frac{1}{u}.$$

Die Idee, die der Konstruktion der Teilphase σ'_j zugrunde liegt, kann man grob wie folgt zusammenfassen: Gibt es keine unmarkierte Seite, die Algorithmus A mit einer genügend hohen Wahrscheinlichkeit von $1/u$ nicht im Cache hat, so greifen wir zunächst auf markierte Seiten zu. Dies generiert Kosten für A, nicht jedoch für den optimalen Offline-Algorithmus, der die markierten Seiten im Cache halten kann. Wir stoppen, sobald wir durch die Zugriffe auf markierte Seiten erwartete Kosten von mindestens $1/u$ für Algorithmus A erzeugt haben oder es eine unmarkierte Seite gibt, die mit genügend hoher Wahrscheinlichkeit nicht im Cache ist.

Am Ende der k-ten Teilsequenz σ'_k definieren wir die aktuelle Phase als abgeschlossen. Dann werden die Markierungen von allen Seiten, bis auf die, auf die zuletzt zugegriffen wurde, gelöscht und die nächste Phase beginnt.

Die erwarteten Kosten von Algorithmus A in der Phase σ' betragen gemäß Konstruktion mindestens

$$\sum_{j=1}^{k} \frac{1}{k+1-j} = H_k.$$

In der ersten Phase verursacht Algorithmus A genau $k+1$ Seitenfehler. Es gilt also insgesamt

$$\mathrm{OPT}(\sigma) \leq k+m \qquad \text{und} \qquad \mathbf{E}\left[w_A(\sigma)\right] \geq k+1+(m-1)H_k.$$

Da m beliebig groß gewählt werden kann, zeigt das, dass es keinen Online-Algorithmus gibt, der einen kompetitiven Faktor $r < H_k$ erreicht. □

Zwischen dem kompetitiven Faktor von MARK, den wir bewiesen haben, und der vorangegangenen unteren Schranke liegt ein Faktor von zwei. Tatsächlich kann man zeigen, dass MARK nur genau $(2H_k-1)$-kompetitiv ist [1]. Es existieren andere randomisierte Online-Algorithmen für das Paging-Problem, die einen kompetitiven Faktor von H_k erreichen [16].

Kapitel 3

Das k-Server-Problem

In diesem Kapitel beschäftigen wir uns mit dem *k-Server-Problem*. Dabei handelt es sich um ein abstraktes Online-Problem, mit dem man eine Reihe von verschiedenen Problemen modellieren kann. Um es zu beschreiben, gehen wir davon aus, dass eine ganze Zahl $k \geq 2$ und ein *metrischer Raum* $\mathcal{M} = (M, d)$ mit $|M| > k$ gegeben sind. Das bedeutet, dass M eine beliebige Menge ist und $d : M \times M \to \mathbb{R}_{\geq 0}$ eine Funktion, die jedem Paar von Elementen aus M einen Abstand zuweist. Dabei müssen die folgenden drei Eigenschaften für alle $x, y, z \in M$ erfüllt sein:

1. $d(x, y) = 0 \iff x = y$,
2. $d(x, y) = d(y, x)$ (Symmetrie),
3. $d(x, z) \leq d(x, y) + d(y, z)$ (Dreiecksungleichung).

Die Funktion d nennt man eine *Metrik* auf M, wobei wir im Folgenden der Einfachheit halber auch oft $\mathcal{M}$ selbst als Metrik statt als metrischen Raum bezeichnen werden. Außerdem werden wir statt von Elementen aus M auch einfach von *Punkten* sprechen. Wählt man $M = \mathbb{R}^2$ und d als die Funktion, die jedem Paar von Punkten seinen euklidischen Abstand zuweist, so erhält man ein anschauliches Beispiel für eine Metrik. Um das Modell so allgemein wie möglich zu halten, betrachten wir in diesem Kapitel aber allgemeine metrische Räume.

Ist die Menge M endlich, so sagen wir, dass die Metrik $\mathcal{M}$ endlich ist. In diesem Fall kann man $\mathcal{M}$ als einen vollständigen gewichteten Graphen mit ungerichteten Kanten darstellen, dessen Knoten die Elemente aus M darstellen. Das Gewicht einer Kante zwischen zwei Knoten entspricht dann genau dem Abstand der beiden entsprechenden Punkte. Der Leser sollte sich als Übung überlegen, dass auch umgekehrt jeder (nicht unbedingt vollständige) gewichtete Graph mit ungerichteten Kanten eine Metrik definiert, wenn alle Kantengewichte positiv sind und man die Länge des kürzesten Weges zwischen zwei Knoten als ihren Abstand definiert.

Bei dem k-Server-Problem muss ein Online-Algorithmus k Server kontrollieren, die sich unabhängig durch den metrischen Raum bewegen können. Jeder Server steht zu jedem

Zeitpunkt auf einem Punkt des metrischen Raumes. Eine Eingabe $\sigma = (\sigma_1, \ldots, \sigma_n)$ ist eine Folge von Anfragen $\sigma_i \in M$, auf die der Online-Algorithmus reagieren muss. Falls sich zu dem Zeitpunkt, zu dem eine Anfrage σ_i gestellt wird, kein Server an dem Punkt σ_i befindet, so muss der Algorithmus einen der Server auf diesen Punkt verschieben, damit dieser die Anfrage bedienen kann. Der Algorithmus kann einen beliebigen Server auswählen und die Kosten, die entstehen, entsprechen dem Abstand zwischen σ_i und dem Punkt, auf dem sich dieser Server gerade befindet. Genauso wie beim Paging kennt der Algorithmus bei Anfrage σ_i weder die weiteren Anfragen noch weiß er, wie viele weitere Anfragen es noch geben wird.

Prinzipiell erlauben wir einem Online-Algorithmus auch, dass er bei einer Anfrage mehrere Server bewegt. Die Summe der zurückgelegten Wege entspricht dann den Kosten, die dem Algorithmus entstehen. Wir nennen einen Algorithmus *faul*, falls er nur dann Server bewegt, wenn eine Anfrage an einem Punkt auftritt, auf dem sich im Moment kein Server befindet, und falls er in diesem Fall immer nur genau einen Server bewegt. Der Leser sollte sich überlegen, dass jeder nicht faule Algorithmus A leicht durch einen faulen Algorithmus ersetzt werden kann, dessen Kosten höchstens so groß sind wie die von A. Aus diesem Grunde genügt es, bei unteren Schranken ausschließlich faule Algorithmen zu betrachten.

Das k-Server-Problem ist nicht nur deswegen auf großes Interesse gestoßen, weil es ein einfach zu formulierendes und elegantes Online-Problem ist, sondern auch deswegen, weil mit ihm viele andere Probleme modelliert werden können.

- Wir können das Paging-Problem, mit dem wir uns im vorangegangenen Kapitel beschäftigt haben, als Spezialfall des k-Server-Problems auffassen. Dazu wählen wir M als die Menge der Seiten und definieren den Abstand zwischen jedem Paar von Seiten als 1. Die Positionen der k Server entsprechen dann den Seiten, die der Algorithmus im Cache hat. Bei jedem Seitenfehler muss der Algorithmus entscheiden, welchen Server er bewegt, d. h. welche Seite er aus dem Cache verdrängt, und es entstehen ihm Kosten 1 für das Verschieben des Servers.
- Dem *k-Headed Disk Problem* liegt eine Festplatte mit k Lese-/Schreibköpfen zu Grunde, die sich unabhängig entlang einer Achse bewegen können, unter der die Scheibe rotiert. Soll nun von einer Spur gelesen oder auf eine Spur geschrieben werden, so muss entschieden werden, welcher Kopf zu dieser Spur gefahren wird. Natürlich ist dabei nicht bekannt, auf welche Spuren in Zukunft zugegriffen wird. Dieses Problem entspricht dem k-Server-Problem auf einer Linie. Es kann also beispielsweise modelliert werden, indem man $M = [0, 1]$ und $d(x, y) = |x - y|$ wählt.

3.1 Einführende Bemerkungen

Wir werden in diesem Abschnitt als Vorbereitung auf den Rest des Kapitels zunächst diskutieren, warum ein einfacher Greedy-Algorithmus nicht kompetitiv ist, wir werden den aktuellen Stand der Forschung zum k-Server-Problem beschreiben und anschließend einen optimalen Offline-Algorithmus angeben.

3.1.1 Der Greedy-Algorithmus

Der natürliche Greedy-Algorithmus für das k-Server-Problem bewegt bei jeder Anfrage σ_i an einem Punkt, auf dem im Moment kein Server steht, einfach den nächstgelegenen Server zu σ_i. Dass dies keine gute Idee ist, zeigt bereits ein einfaches Beispiel mit $k = 2$ und $|M| = 3$. Wir ordnen die drei Punkte a, b und c, aus denen M besteht, auf einer Linie wie folgt an.

Die genauen Abstände sind bei dem Beispiel unwichtig, da wir im Folgenden nur ausnutzen, dass der Abstand zwischen a und b kleiner ist als der zwischen b und c. Wir betrachten nun die Eingabe $\sigma = (c, (a, b)^\ell)$, die aus einer Anfrage an c besteht, auf die jeweils ℓ alternierende Anfragen an a und b folgen.

Unabhängig von den Startpositionen der Server steht nach den ersten beiden Anfragen ein Server an Punkt a und der andere steht an Punkt c. Während der nun folgenden alternierenden Anfragen an a und b lässt der Greedy-Algorithmus stets einen Server auf c und bewegt den anderen zwischen a und b hin und her. Da wir ℓ beliebig groß wählen können, können wir somit auch die Kosten, die dem Greedy-Algorithmus auf der Eingabe σ entstehen, beliebig groß machen.

Auf der anderen Seite sind die Kosten des optimalen Offline-Algorithmus unabhängig von ℓ durch eine Konstante beschränkt, da der optimale Offline-Algorithmus während der alternierenden Aufrufe stets einen Server auf a und einen auf b lassen kann. Somit ist klar, dass der Greedy-Algorithmus nicht kompetitiv ist.

3.1.2 Die *k*-Server-Vermutung

Die Erkenntnis, dass der Greedy-Algorithmus nicht kompetitiv ist, legt natürlich die Frage nahe, ob es überhaupt einen kompetitiven Online-Algorithmus für das k-Server-Problem gibt und was der beste kompetitive Faktor ist, der erreicht werden kann. Diese Frage konnte bis heute nicht abschließend beantwortet werden, obwohl die folgende Vermutung, die 1988 erstmalig von Manasse, McGeoch und Sleater [15] aufgestellt wurde, viel Forschung am k-Server-Problem nach sich gezogen hat.

Vermutung 3.1 (k-Server-Vermutung)**.** *Für jeden metrischen Raum gibt es einen deterministischen k-kompetitiven Online-Algorithmus für das k-Server-Problem.*

Heute hat man diese Vermutung zumindest bis auf einen Faktor von zwei beweisen können. Wir werden sehen, dass es für keinen metrischen Raum mit mindestens $k + 1$ Punkten einen deterministischen Online-Algorithmus gibt, der r-kompetitiv für ein $r < k$ ist. Außerdem ist bekannt, dass es einen deterministischen Online-Algorithmus gibt, der $(2k - 1)$-kompetitiv ist. Trotz intensiver Bemühungen ist es bis heute noch nicht gelungen, die k-Server-Vermutung zu beweisen oder zu widerlegen.

Bei randomisierten Algorithmen ist die Lücke zwischen den besten bekannten oberen und unteren Schranken noch weitaus größer. Man vermutet, dass randomisierte Algorithmen genauso wie beim Paging einen deutlich besseren kompetitiven Faktor als deterministische Algorithmen erreichen können.

Vermutung 3.2 (Randomisierte k-Server-Vermutung)**.** *Für jeden metrischen Raum gibt es einen randomisierten $O(\log k)$-kompetitiven Online-Algorithmus für das k-Server-Problem.*

Für den Spezialfall Paging haben wir bereits in Theorem 2.14 eine untere Schranke von $\Omega(\log k)$ für den kompetitiven Faktor eines jeden randomisierten Online-Algorithmus bewiesen. Die beste bekannte untere Schranke für das k-Server-Problem, die für jede Metrik mit mindestens $k+1$ Punkten gilt, beträgt $\Omega(\log k / \log\log k)$ [4]. Die beste obere Schranke für allgemeine metrische Räume ist im Moment $2k-1$. Das heißt, trotz der deutlich schwächeren unteren Schranken gibt es momentan keinen Beweis dafür, dass mit randomisierten Algorithmen ein besserer kompetitiver Faktor erreicht werden kann als mit deterministischen. Nur wenn man erlaubt, dass der kompetitive Faktor auch von der Größe der Metrik abhängen darf, sind bessere Schranken bekannt. Bansal et al. [3] haben 2011 einen randomisierten Online-Algorithmus mit einem kompetitiven Faktor von $O(\log^2(k) \cdot \log^3(N) \cdot \log\log(N))$ für $N = |M|$ präsentiert.

3.1.3 Optimale Offline-Algorithmen

A priori ist nicht klar, ob man das k-Server-Problem effizient lösen kann, wenn man die gesamte Eingabe bereits von Anfang an kennt. Bevor wir uns dieser algorithmischen Frage nähern, beschreiben wir die Offline-Variante des k-Server-Problems genauer. Oft wird bei der ersten Betrachtung der Fehler gemacht, dass man dem optimalen Offline-Algorithmus zugesteht, die Anfragen in einer anderen Reihenfolge zu bearbeiten, als sie auftreten. Zur Verdeutlichung betrachten wir die Metrik $\mathcal{M} = (\mathbb{N}, d)$ mit $d(x, y) = |x - y|$. Sei $k = 2$ und $\sigma = (1, 2, 4, 3)$. Nach der zweiten Anfrage dieser Sequenz steht ein Server auf 2 und die nächste zu bearbeitende Anfrage erfolgt an 4. Zwar weiß der Offline-Algorithmus, dass danach eine Anfrage an 3 erfolgen wird, dennoch kann er diese nicht unterwegs bearbeiten, während er den Server von 2 auf 4 schiebt. Die Anfrage an 3 kann auch im Offline-Szenario erst dann bearbeitet werden, wenn die Anfrage an 4 abgearbeitet ist.

Betrachtet man k als eine Konstante, so erhält man mittels dynamischer Programmierung einen Polynomialzeitalgorithmus. Dies ist eine gute Gelegenheit für den Leser, diese wichtige Methode des Algorithmenentwurfs noch einmal zu wiederholen und auf das k-Server-Problem anzuwenden. Das Problem bei diesem Ansatz ist allerdings, dass die Laufzeit exponentiell von k abhängt. Somit ist dieser Ansatz schon für relativ kleine k nicht praktikabel.

Tatsächlich kann man die Offline-Variante des k-Server-Problems viel effizienter lösen, indem man sie auf das Min-Cost-Flow-Problem, ein bekanntes kombinatorisches Optimierungsproblem, reduziert. Die Eingabe für dieses Problem besteht aus einem gerichteten Graphen $G = (V, E)$ mit einer Quelle $s \in V$, einer Senke $t \in V$, einer

Kapazitätsfunktion $u : E \to \mathbb{R}_{\geq 0}$ und einer Kostenfunktion $c : E \to \mathbb{R}$. Dabei seien die Kosten c so gewählt, dass es keine Kreise mit insgesamt negativen Kosten im Graphen G gibt.

Genauso wie bei herkömmlichen Flussproblemen geht es darum, in dem Graphen G einen maximalen Fluss von der Quelle zur Senke zu berechnen. Ein Fluss ist eine Funktion $f : E \to \mathbb{R}_{\geq 0}$, die die Kapazitätsbeschränkungen und die Flusserhaltung einhält. Das heißt, es muss $0 \leq f(e) \leq u(e)$ für jede Kante $e \in E$ gelten und es muss $\sum_{e=(u,v)\in E} f(e) = \sum_{e=(v,u)\in E} f(e)$ für alle Knoten $v \in V \setminus \{s, t\}$ gelten. Der Wert $|f|$ eines Flusses f gibt an, wie viele Einheiten von der Quelle zur Senke transportiert werden. Formal gilt $|f| = \sum_{e=(s,v)\in E} f(e) - \sum_{e=(v,s)\in E} f(e)$.

Ein Fluss heißt maximal, wenn er unter allen gültigen Flüssen den größtmöglichen Wert besitzt. In den einführenden Grundvorlesungen haben wir gesehen, wie ein maximaler Fluss effizient berechnet werden kann. Im Allgemeinen ist der maximale Fluss aber nicht eindeutig, was das erweiterte Problem motiviert, einen maximalen Fluss mit minimalen Kosten zu berechnen. Formal ist es im Min-Cost-Flow-Problem die Aufgabe, einen maximalen Fluss f zu berechnen, der unter allen maximalen Flüssen die geringsten Kosten $c(f) = \sum_{e\in E} c(e) \cdot f(e)$ besitzt.

Es gibt einen Algorithmus (Successive-Shortest-Path-Algorithmus), der das Min-Cost-Flow-Problem für ganzzahlige Kapazitäten $u : E \to \mathbb{N}$ in Zeit $O(n^3 F)$ löst, wobei F der Wert des maximalen Flusses ist. Im Allgemeinen ist dies zwar keine polynomielle sondern nur eine pseudopolynomielle Laufzeit, für unsere Zwecke ist es jedoch ausreichend, wie wir gleich sehen werden. Es sei nur erwähnt, dass es für das Min-Cost-Flow-Problem auch Algorithmen mit einer polynomiellen Laufzeit gibt. Da das Min-Cost-Flow-Problem aber nicht Thema dieser Vorlesung ist, verweisen wir den interessierten Leser z. B. auf das Buch von Korte und Vygen [13].

Eine wichtige Eigenschaft, die wir bei der folgenden Konstruktion ausnutzen werden, ist, dass es für ganzzahlige Kapazitäten $u : E \to \mathbb{N}$ stets einen ganzzahligen optimalen Fluss $f : E \to \mathbb{N}$ gibt, den der oben erwähnte Algorithmus auch berechnet. Wir können also ohne Beschränkung der Allgemeinheit davon ausgehen, dass wir einen optimalen ganzzahligen Fluss vorliegen haben.

Das Min-Cost-Flow-Problem ist wichtig, da man viele andere Probleme darauf reduzieren kann. So ist es auch bei dem k-Server-Problem. Sei eine Metrik $\mathcal{M}$ und eine Eingabe $\sigma = (\sigma_1, \ldots, \sigma_n)$ für das k-Server-Problem gegeben, die wir optimal lösen möchten. Dabei gehen wir der Einfachheit halber davon aus, dass sich zu Beginn alle Server am selben Punkt $o \in M$ befinden. Außerdem können wir ohne Beschränkung der Allgemeinheit annehmen, dass $n \geq k$ gilt, da man für Eingaben mit $n < k$ einfach $k - n$ Server weglassen und dann das resultierende n-Server-Problem lösen kann.

Wir konstruieren einen Graphen $G = (V, E)$, der zusätzlich zur Quelle s und zur Senke t jeweils einen Knoten s_i für jeden Server i enthält und zwei Knoten σ_j und σ'_j für jede Anfrage σ_j. Es gilt also

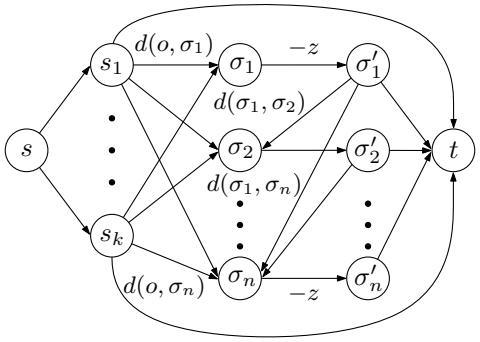
$$V = \{s, t\} \cup \{s_1, \ldots, s_k\} \cup \{\sigma_1, \ldots, \sigma_n\} \cup \{\sigma'_1, \ldots, \sigma'_n\}$$

und es sei

$$E = \{(s, s_i) \mid i \in \{1, \ldots, k\}\}$$

$$
\begin{aligned}
&\cup \{(s_i, \sigma_j) \mid i \in \{1, \ldots, k\}, j \in \{1, \ldots, n\}\} \\
&\cup \{(s_i, t) \mid i \in \{1, \ldots, k\}\} \\
&\cup \{(\sigma_j, \sigma'_j) \mid j \in \{1, \ldots, n\}\} \\
&\cup \{(\sigma'_j, \sigma_\ell) \mid j, \ell \in \{1, \ldots, n\}, \ell > j\} \\
&\cup \{(\sigma'_j, t) \mid j \in \{1, \ldots, n\}\}.
\end{aligned}
$$

Diese Konstruktion ist in der folgenden Abbildung dargestellt.



Wir setzen alle Kapazitäten auf 1, also $u(e) = 1$ für alle $e \in E$. Ferner definieren wir die Kosten so, wie es in der vorangegangenen Abbildung angedeutet ist. Es gelte für alle $i \in \{1, \ldots, k\}$ alle $j \in \{1, \ldots, n\}$ und alle $\ell \in \{1, \ldots, n\}$ mit $\ell > j$

$$
\begin{aligned}
c(s, s_i) &= 0, \\
c(s_i, \sigma_j) &= d(o, \sigma_j), \\
c(s_i, t) &= 0, \\
c(\sigma_j, \sigma'_j) &= -z, \\
c(\sigma'_j, \sigma_\ell) &= d(\sigma_j, \sigma_\ell), \\
c(\sigma'_j, t) &= 0.
\end{aligned}
$$

Dabei sei z so gewählt, dass es größer als $2 \max_{x,y \in M} d(x, y)$ ist.

Zunächst können wir festhalten, dass es in dem Graphen G keinen Kreis und somit natürlich insbesondere auch keinen Kreis mit insgesamt negativen Kosten gibt und wir deshalb eine gültige Instanz des Min-Cost-Flow-Problems konstruiert haben. Außerdem hat der maximale Fluss den Wert k, da die Quelle k ausgehende Kanten mit jeweils Kapazität 1 besitzt. Der oben erwähnte Algorithmus löst diese Instanz des Min-Cost-Flow-Problems somit in Zeit $O(n^3 k)$ (mit einer genaueren Analyse des Algorithmus kann man sogar eine Laufzeit von $O(n^2 k)$ erreichen).

Nun müssen wir nur noch zeigen, dass wir aus dem Fluss f mit minimalen Kosten effizient eine optimale Offline-Lösung für das k-Server-Problem ablesen können. Zunächst beobachten wir, dass $f(e) = 1$ für alle Kanten $e = (\sigma_j, \sigma'_j)$ gilt. Der maximale Fluss mit minimalen Kosten lastet diese Kanten also vollständig aus. Dies zu zeigen, überlassen wir dem Leser als Übung. Zur Vereinfachung sollte man dabei ausnutzen, dass der optimale Fluss f ganzzahlig ist.

Da alle Kapazitäten 1 sind, impliziert die Ganzzahligkeit von f, dass jede Kante entweder Fluss 0 oder Fluss 1 hat. Deshalb entspricht ein maximaler Fluss mit Wert k

einer Auswahl von k kantendisjunkten Wegen von s nach t. Aus der Struktur des Graphen G folgt, dass diese Wege nicht nur kantendisjunkt sondern sogar knotendisjunkt sind. Jeder dieser Wege kann eindeutig einem Server zugeordnet werden, da jeder Weg genau einen Knoten s_i enthält und jeder solche Knoten in genau einem der Pfade vorkommt. Man überlegt sich leicht anhand der Struktur des Graphen G, dass der Pfad P_i, der den Knoten s_i enthält, die Gestalt $P_i = (s, s_i, \sigma_{j_1}, \sigma'_{j_1}, \ldots, \sigma_{j_\ell}, \sigma'_{j_\ell}, t)$ für ein $\ell \geq 0$ und $j_1 < \ldots < j_\ell$ haben muss. Die Länge dieses Weges beträgt $d(o, \sigma_{j_1}) + d(\sigma_{j_1}, \sigma_{j_2}) + \ldots + d(\sigma_{j_{\ell-1}}, \sigma_{j_\ell}) - \ell z$. Dies sind abgesehen von dem letzten Term $-\ell z$ genau die Kosten, die einem Server entstehen, der an Punkt o startet, und die Anfragen $\sigma_{j_1}, \ldots, \sigma_{j_\ell}$ in dieser Reihenfolge bearbeitet.

Aus unseren Vorüberlegungen folgt, dass jede Kante $e = (\sigma_j, \sigma'_j)$ in genau einem der Wege P_i enthalten ist. Wir erhalten somit aus dem maximalen Fluss eine Lösung L für das k-Server-Problem, indem wir Server i genau die Anfragen abarbeiten lassen, die auf seinem Weg P_i legen. Die Kosten der Lösung L entsprechen genau der Summe der Weglängen plus nz, da jede der n Kanten $e = (\sigma_j, \sigma'_j)$ einen Term von $-z$ zu der Länge des sie beinhaltenden Weges beiträgt.

Insgesamt haben wir damit gezeigt, dass wir aus einer Lösung f für die oben beschriebene Instanz des Min-Cost-Flow-Problems eine Lösung für das k-Server-Problem mit Kosten genau $c(f) + nz$ konstruieren können. Das zeigt noch nicht, dass dies auch eine optimale Lösung für das k-Server-Problem ist. Mit ähnlichen Argumenten wie oben kann man aber jede Lösung L für das k-Server-Problem mit Kosten $w(L)$ in einen Fluss mit Kosten $w(L) - nz$ übersetzen. Dieser Fluss besteht aus k kantendisjunkten Wegen, wobei der Weg P_i genau wie oben die Anfragen enthält, die Server i in der Lösung für das k-Server-Problem abarbeitet. Damit ist gezeigt, dass es eine Bijektion zwischen den maximalen ganzzahligen Flüssen, die alle Kanten der Form (σ_j, σ'_j) benutzen, und den Lösungen für das k-Server-Problem gibt, bei der sich insbesondere die Kosten bis auf einen festen additiven Term nz übertragen. Somit entspricht der maximale Fluss mit minimalen Kosten einer Lösung für das k-Server-Problem mit minimalen Kosten.

3.2 Untere Schranke für deterministische Algorithmen

Theorem 2.5 besagt, dass es für das Paging-Problem keinen deterministischen Online-Algorithmus gibt, der einen besseren kompetitiven Faktor als k erreicht. Übertragen auf das k-Server-Problem bedeutet das, dass es eine spezielle Metrik gibt (nämlich die uniforme Metrik, die jedem Paar von Punkten den Abstand 1 zuweist), für die es keinen Algorithmus für das k-Server-Problem gibt, der besser als k-kompetitiv ist. Dieses Ergebnis schließt nicht aus, dass das k-Server-Problem auf anderen Metriken wie beispielsweise der, die wir zur Modellierung des k-Headed Disk Problem eingesetzt haben, deutlich besser gelöst werden kann. Wir zeigen in diesem Abschnitt, dass dies nicht so ist, und es für keine Metrik mit mindestens $k + 1$ Punkten einen Algorithmus gibt, der besser als k-kompetitiv ist.

Theorem 3.3. *Es sei $\mathcal{M} = (M, d)$ ein beliebiger metrischer Raum mit $|M| \geq k + 1$. Dann gibt es für kein $r < k$ einen deterministischen r-kompetitiven Online-Algorithmus für das k-Server-Problem auf $\mathcal{M}$.*

Beweis. Es sei $B = \{b_1, \ldots, b_{k+1}\} \subseteq M$ eine beliebige Teilmenge von M mit $k+1$ Elementen und es sei A ein beliebiger fauler deterministischer Algorithmus. Wir haben bereits am Anfang dieses Kapitels diskutiert, dass wir uns bei unteren Schranken ohne Beschränkung der Allgemeinheit auf faule Algorithmen beschränken können, da jeder andere Online-Algorithmus durch einen faulen Algorithmus mit höchstens genauso großen Kosten ersetzt werden kann.

Ähnlich wie im Beweis von Theorem 2.5 konstruieren wir eine Eingabe σ, in der nur auf die $k + 1$ Elemente aus B zugegriffen wird. Wir gehen davon aus, dass die Server so initialisiert werden, dass zu Beginn auf jedem Punkt höchstens ein Server steht. Außerdem gehen wir davon aus, dass alle Server zu Beginn an Punkten aus B stehen. Da der Algorithmus A faul ist, bleiben diese beiden Eigenschaften im weiteren Verlauf erhalten. Die zweite Annahme ist keine Einschränkung, da wir sie durch die Wahl von B sicherstellen können. Die erste Annahme ist ebenfalls keine Einschränkung, da wir alle Kosten, die durch unterschiedliche Initialisierungen entstehen, in der additiven Konstante τ in Definition 1.1 verbergen können. Der Leser sollte sich dies als Übung noch einmal genau überlegen.

In der Menge B gibt es genau einen Punkt b_1, auf dem zu Beginn kein Server steht. Es sei $\sigma_1 = b_1$. Um diese erste Anfrage zu bearbeiten, muss Algorithmus A einen Server von einem Punkt $a \in B \setminus \{b_1\}$ zu Punkt b_1 bewegen. Das bedeutet, dass sich nach der Anfrage σ_1 kein Server an Punkt a befindet. Wir wählen $\sigma_2 = a$, das heißt, die nächste Anfrage erfolgt an einem Punkt aus B, an dem Algorithmus A keinen Server platziert hat. Algorithmus A muss also wieder einen Server von einem Punkt $a' \in B \setminus \{a\}$ zur Anfrage $\sigma_2 = a$ bewegen. Die Anfrage σ_3 erfolgt dann an Punkt a' usw. Diese Konstruktion setzen wir bis σ_n fort.

Lemma 3.4. *Es gilt*

$$w_A(\sigma) \geq \sum_{i=1}^{n-1} d(\sigma_i, \sigma_{i+1}).$$

Beweis. Aus der Konstruktion von σ ergibt sich, dass Algorithmus A bei jeder Anfrage einen Server verschieben muss. Das ist analog zu dem Beweis von Theorem 2.5, in dem die Eingabe so gewählt wurde, dass dem Online-Algorithmus bei jedem Seitenzugriff ein Seitenfehler entsteht. Auf den ersten Blick scheint es aber nicht trivial zu sein, die Kosten von Algorithmus A auf der konstruierten Eingabe σ anzugeben, da wir nicht wissen, welche Server Algorithmus A bewegt. Unsere Konstruktion von σ erlaubt es aber, die Kosten von A präzise zu beschreiben. Wir wissen nämlich für jedes $i < n$, dass sich der Server, der die Anfrage σ_i bearbeitet, vorher an Position σ_{i+1} befunden hat. Dementsprechend entstehen dem Algorithmus A bei der Bearbeitung der Anfrage σ_i Kosten in Höhe von $d(\sigma_{i+1}, \sigma_i)$. Damit können wir die Kosten von Algorithmus A auf der Eingabe σ insgesamt durch

$$w_A(\sigma) \geq \sum_{i=1}^{n-1} d(\sigma_{i+1}, \sigma_i) = \sum_{i=1}^{n-1} d(\sigma_i, \sigma_{i+1})$$

nach unten abschätzen. □

Nun müssen wir noch die Kosten des optimalen Offline-Algorithmus auf der Eingabe σ nach oben abschätzen. Im Beweis von Theorem 2.5 war das relativ einfach. Wir haben dort argumentiert, dass LFD nur bei jedem k-ten Seitenzugriff ein Seitenfehler entsteht. Dieses Argument lässt sich auch auf das k-Server-Problem übertragen und man kann argumentieren, dass nur bei jeder k-ten Anfrage ein Server bewegt werden muss. Das hilft uns aber nicht weiter, da nicht mehr alle Abstände in der Metrik gleich sind. Es könnte also sein, dass der optimale Offline-Algorithmus zwar nur selten einen Server bewegt, dafür dann aber über eine große Entfernung. Deshalb benötigen wir ein etwas komplizierteres Argument.

Lemma 3.5. *Es gilt*

$$\mathrm{OPT}(\sigma) \leq \frac{1}{k} \sum_{i=1}^{n-1} d(\sigma_i, \sigma_{i+1}).$$

Beweis. Wir geben einen indirekten Beweis dafür, dass es einen Offline-Algorithmus gibt, der die behauptete Kostenschranke einhält. Dazu definieren wir zunächst eine Klasse $\mathcal{C}$ von Algorithmen. Jeder Algorithmus aus dieser Klasse ist durch eine Menge $S \subseteq B$ mit $b_1 \in S$ und $|S| = k$ bestimmt. Den zu einer solchen Menge gehörenden Algorithmus nennen wir C_S und wir definieren sein Verhalten wie folgt: Zu Beginn platziert Algorithmus C_S die Server auf genau den k Punkten aus S. Wegen $b_1 \in S$ muss der Algorithmus bei der ersten Anfrage σ_1 keinen Server bewegen. Erfolgt für $i \geq 2$ eine Anfrage σ_i an einem Punkt, auf dem kein Server steht, so bewegt Algorithmus C_S den Server, der auf σ_{i-1} steht, zu der Anfrage σ_i. Wir können sicher sein, dass es einen solchen Server gibt, da im Schritt davor die Anfrage σ_{i-1} von dem Algorithmus bedient wurde.

Da es k verschiedene Mengen S mit den oben geforderten Eigenschaften gibt, haben wir auf diese Weise k verschiedene Algorithmen beschrieben. Wir bezeichnen mit $S^i \subseteq B$ die Menge der Punkte, auf denen sich die Server von Algorithmus C_S nach der Anfrage σ_i befinden. Wir beweisen nun durch vollständige Induktion, dass $S_1^i \neq S_2^i$ für alle $S_1 \neq S_2$ und für alle $i \geq 0$ gilt. Der Induktionsanfang folgt direkt aus der Definition, denn $S_1^0 = S_1 \neq S_2 = S_2^0$. Für den Induktionsschritt gehen wir davon aus, dass $S_1^i \neq S_2^i$ für ein i gilt. Abhängig von der nächsten Anfrage σ_{i+1} unterscheiden wir die folgenden Fälle.

- $\sigma_{i+1} \in S_1^i$ und $\sigma_{i+1} \in S_2^i$: In diesem Fall bewegt keiner der Algorithmen einen Server. Es gilt also $S_1^{i+1} = S_1^i \neq S_2^i = S_2^{i+1}$.
- $\sigma_{i+1} \in S_1^i$ und $\sigma_{i+1} \notin S_2^i$: Da S_1^i und S_2^i die Positionen der Server direkt nach der Bearbeitung von Anfrage σ_i sind, gilt $\sigma_i \in S_1^i$ und $\sigma_i \in S_2^i$. Da Algorithmus C_{S_1} bei Anfrage σ_{i+1} keinen Server bewegt, gilt auch $\sigma_i \in S_1^{i+1}$. Auf der anderen Seite bewegt aber Algorithmus C_{S_2} den Server von σ_i zu σ_{i+1}. Es gilt also $\sigma_i \notin S_2^{i+1}$. Damit ist $S_1^{i+1} \neq S_2^{i+1}$.
- $\sigma_{i+1} \notin S_1^i$ und $\sigma_{i+1} \in S_2^i$: Dieser Fall ist symmetrisch zum vorangegangenen.

- $\sigma_{i+1} \notin S_1^i$ und $\sigma_{i+1} \notin S_2^i$: Dieser Fall kann nicht eintreten, da aus $|S_1^i| = |S_2^i| = k$, $|B| = k+1$ und $S_1^i \neq S_2^i$ direkt $S_1^i \cup S_2^i = B$ folgt.

Wir haben somit bewiesen, dass zwei Algorithmen C_{S_1} und C_{S_2} zu keinem Zeitpunkt ihre Server an exakt den gleichen Positionen haben, falls $S_1 \neq S_2$ gilt. Da es insgesamt k Algorithmen der oben beschriebenen Form gibt und alle Algorithmen direkt nach einer Anfrage σ_i einen Server auf Position σ_i haben, gibt es für jede Position $b \in B \setminus \{\sigma_i\}$ genau einen Algorithmus C_S mit $b \notin S^i$. Bei jeder Anfrage σ_{i+1} gibt es also genau einen Algorithmus, der keinen Server auf σ_{i+1} hat und deshalb den Server, der auf σ_i steht, dorthin verschiebt. Diesem Algorithmus entstehen in diesem Schritt Kosten $d(\sigma_i, \sigma_{i+1})$, während allen anderen Algorithmen keine Kosten entstehen. Das bedeutet, dass wir die Summe der Kosten aller Algorithmen genau angeben können. Es gilt

$$\sum_S w_{C_S}(\sigma) = \sum_{i=1}^{n-1} d(\sigma_i, \sigma_{i+1}).$$

Da wir über k verschiedene Algorithmen summieren, betragen die durchschnittlichen Kosten dieser Algorithmen genau

$$\frac{1}{k} \sum_{i=1}^{n-1} d(\sigma_i, \sigma_{i+1}).$$

Da nicht alle Algorithmen C_S mehr Kosten als die durchschnittlichen Kosten verursachen können, muss es einen Algorithmus C_S geben, dessen Kosten $w_{C_S}(\sigma)$ höchstens so groß sind wie dieser Durchschnitt. Dies gilt dann natürlich insbesondere für den optimalen Offline-Algorithmus und damit ist das Lemma bewiesen. □

Aus den beiden vorangegangenen Lemmas 3.4 und 3.5 folgt direkt das Theorem. Für gegebenes $r < k$ und τ muss man dafür n lediglich so groß wählen, dass

$$\sum_{i=1}^{n-1} d(\sigma_i, \sigma_{i+1}) > \frac{r}{k} \sum_{i=1}^{n-1} d(\sigma_i, \sigma_{i+1}) + \tau$$

gilt. □

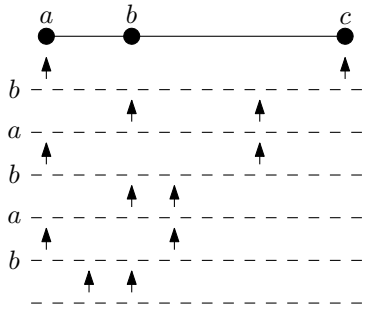
3.3 Das k-Server-Problem auf Linien und Bäumen

Bevor wir uns dem k-Server-Problem auf allgemeinen Metriken widmen, betrachten wir zunächst einige interessante Spezialfälle. Der erste Spezialfall ist durch das k-Headed Disk Problem motiviert, dem die Metrik $\mathcal{M} = ([0,1], d)$ mit $d(x,y) = |x-y|$ zugrunde liegt. Diese Metrik nennen wir im Folgenden auch einfach *Linie*. In Abschnitt 3.1.1 haben wir gesehen, dass der Greedy-Algorithmus, der stets den nächstgelegenen Server zu einer Anfrage bewegt, bereits für diesen Spezialfall nicht kompetitiv ist.

Wir modifizieren den Greedy-Algorithmus nun für die Linie zu einem Algorithmus namens *Double Coverage (DC)*. Liegt die nächste Anfrage σ_i links oder rechts von

allen Servern, so bewegt der DC-Algorithmus den Server, der sich am weitesten links bzw. rechts befindet, zu der Anfrage σ_i. Ansonsten bewegt der DC-Algorithmus den nächsten Server links von σ_i und gleichzeitig den nächsten Server rechts von σ_i mit derselben Geschwindigkeit in Richtung der Anfrage σ_i. Er stoppt beide Server, sobald der erste von ihnen die Anfrage σ_i erreicht. Der DC-Algorithmus ist somit kein fauler Algorithmus, aber er kann durch einen faulen Algorithmus simuliert werden, dessen Kosten nicht größer sind. Wir haben den DC-Algorithmus als nicht faulen Algorithmus beschrieben, da dies natürlicher ist und die Analyse erleichtert.

Bevor wir den DC-Algorithmus analysieren, betrachten wir, wie er sich auf dem Beispiel verhält, das zeigt, dass der Greedy-Algorithmus auf der Linie nicht kompetitiv ist. Dazu gehen wir davon aus, dass wir in der Situation sind, dass sich ein Server an Punkt a und der andere an Punkt c befindet. Die folgende Abbildung zeigt, wie der DC-Algorithmus startend von dieser Ausgangssituation bei der Sequenz b, a, b, a, b die Server verschiebt.



3.3.1 Analyse des DC-Algorithmus auf der Linie

Theorem 3.6. *Der DC-Algorithmus ist k-kompetitiv für das k-Server-Problem auf der Linie.*

Beweis. Wir können nicht argumentieren, dass der DC-Algorithmus in jedem Schritt höchstens k mal so viele Kosten verursacht wie der optimale Offline-Algorithmus, und wenden deshalb die Methode der Potentialfunktionen an, die wir in Abschnitt 2.2.1 eingeführt haben. Mithilfe dieser Methode können wir argumentieren, dass der DC-Algorithmus im Durchschnitt höchstens k mal so viele Kosten wie der optimale Offline-Algorithmus verursacht. Zunächst müssen wir eine Potentialfunktion Φ definieren, die jedes Paar von Konfigurationen des DC-Algorithmus und des optimalen Offline-Algorithmus auf eine reelle Zahl abbildet.

Unter der Konfiguration eines Algorithmus zu einem bestimmten Zeitpunkt verstehen wir beim k-Server-Problem die Positionen der Server. Es seien $s_1, \ldots, s_k \in [0, 1]$ die aktuellen Positionen der Server des DC-Algorithmus und es seien $o_1, \ldots, o_k \in [0, 1]$ die aktuellen Positionen der Server eines optimalen Offline-Algorithmus, wobei wir im Rest des Beweises davon ausgehen, dass ein beliebiger optimaler Offline-Algorithmus fixiert ist. Das Potential setzt sich aus zwei Termen zusammen. Für den ersten Term betrachten wir eine Zuordnung zwischen den Servern des DC-Algorithmus und denen

des optimalen Offline-Algorithmus mit minimalen Kosten $M_{\min}$. Formal sei $\mathcal{S}_k$ die Menge aller Permutationen der Menge $\{1, \ldots, k\}$ und es sei

$$M_{\min} = \min_{\pi \in \mathcal{S}_k} \sum_{i=1}^{k} d(s_i, o_{\pi(i)}).$$

Als zweiten Term benötigen wir noch die Summe der paarweisen Abstände der Server des DC-Algorithmus, also

$$\Sigma_{DC} = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} d(s_i, s_j).$$

Das Potential Φ ergibt sich dann aus diesen beiden Größen als

$$\Phi = k \cdot M_{\min} + \Sigma_{DC}.$$

Für eine Eingabe $\sigma = (\sigma_1, \ldots, \sigma_n)$ bezeichnen wir mit $DC_i(\sigma)$ und $\mathrm{OPT}_i(\sigma)$ die Kosten, die dem DC-Algorithmus bzw. dem optimalen Offline-Algorithmus bei der Bearbeitung der Eingabe σ bei Anfrage σ_i entstehen. Außerdem bezeichnen wir mit Φ_0 das Potential vor der ersten Anfrage und für $i \geq 1$ mit Φ_i das Potential direkt nach der Bearbeitung der i-ten Anfrage σ_i. Genau wie in Abschnitt 2.2.1 definieren wir die amortisierten Kosten des DC-Algorithmus als

$$a_i = DC_i(\sigma) + \Phi_i - \Phi_{i-1}.$$

Können wir zeigen, dass es eine Konstante b gibt, sodass für alle Eingaben σ die folgenden Eigenschaften gelten, so folgt aus Theorem 2.9, dass der DC-Algorithmus k-kompetitiv ist.

1. Für jedes $i \geq 1$ gilt $a_i \leq k \cdot \mathrm{OPT}_i(\sigma)$.
2. Es gilt $\Phi_0 \leq b$.
3. Für jedes $i \geq 1$ gilt $\Phi_i \geq -b$.

Wir erinnern den Leser noch einmal daran, dass der Wert der Potentialfunktion als Kontostand des DC-Algorithmus interpretiert werden kann. Je größer er ist, desto mehr Kosten hat der DC-Algorithmus bisher verglichen mit den k-fachen Kosten des optimalen Offline-Algorithmus gespart.

Die zweite und dritte Eigenschaft gelten für $b = 2k^2$, denn aus der Definition der Potentialfunktion und der Beschränkung der Funktion d durch 1 nach oben folgt für alle i

$$0 \leq \Phi_i \leq k^2 + \binom{k}{2} \leq 2k^2.$$

Um die erste Eigenschaft zu zeigen, betrachten wir in einem beliebigen Schritt $i \geq 1$ die Veränderung des Potentials. Wir müssen zeigen, dass

$$\Phi_i - \Phi_{i-1} \leq k \cdot \mathrm{OPT}_i(\sigma) - DC_i(\sigma) \tag{3.1}$$

gilt. In dem betrachteten Schritt i können sowohl der DC-Algorithmus als auch der optimale Offline-Algorithmus Server bewegen. Dies kann beides zu einer Veränderung des Potentials führen. Um die Analyse übersichtlicher zu gestalten, betrachten wir diese beiden Bewegungen nacheinander. Dazu sei Φ'_{i-1} das Potential nachdem OPT die Anfrage σ_i bearbeitet hat, aber bevor der DC-Algorithmus die Anfrage σ_i bearbeitet hat.

Lemma 3.7. *Es gilt*

$$\Phi'_{i-1} \leq \Phi_{i-1} + k \cdot \mathrm{OPT}_i(\sigma).$$

Beweis. Da wir ohne Beschränkung der Allgemeinheit davon ausgehen können, dass der optimale Offline-Algorithmus OPT faul ist, bewegt er bei Anfrage σ_i einen Server um eine Strecke von genau $\mathrm{OPT}_i(\sigma)$. Dies hat keinen Einfluss auf den Term Σ_{DC}. Der Term $k \cdot M_{\min}$ kann durch die Bewegung des Servers um maximal $k \cdot \mathrm{OPT}_i(\sigma)$ steigen, denn der Wert der optimalen Zuordnung π nach Schritt σ_{i-1} (die nach der Bewegung des Servers i. A. nicht mehr optimal ist) steigt um maximal $\mathrm{OPT}_i(\sigma)$. Damit folgt das Lemma. □

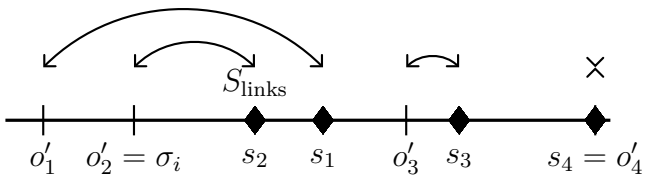
Lemma 3.8. *Es gilt*

$$\Phi_i \leq \Phi'_{i-1} - DC_i(\sigma).$$

Beweis. Bei dem DC-Algorithmus unterscheiden wir zwei Fälle, je nachdem, ob er einen oder zwei Server bewegt. Zunächst betrachten wir den Fall, dass der DC-Algorithmus einen Server bewegt. Dieser Fall tritt ein, wenn die Anfrage σ_i links von allen oder rechts von allen Servern liegt. Sei ohne Beschränkung der Allgemeinheit ersteres der Fall und sei S_{links} der Server des DC-Algorithmus, der sich am weitesten links befindet. Es seien $o'_1, \ldots, o'_k \in [0,1]$ die Positionen der Server von OPT, nachdem er Anfrage σ_i bearbeitet hat, und es seien $s_1, \ldots, s_k \in [0,1]$ die Positionen der Server des DC-Algorithmus, direkt bevor er Anfrage σ_i bearbeitet. Außerdem sei $M'_{\min}$ der Wert der besten Zuordnung zu diesem Zeitpunkt, also

$$M'_{\min} = \min_{\pi \in \mathcal{S}_k} \sum_{j=1}^{k} d(s_j, o'_{\pi(j)}).$$

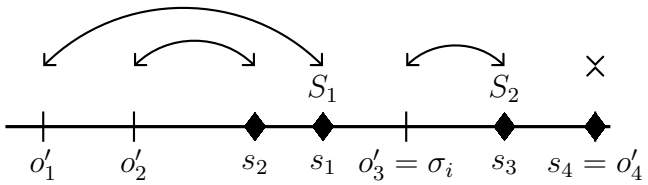
Wir wissen, dass es ein j geben muss, für das $o'_j = \sigma_i$ gilt. Der Server j des optimalen Offline-Algorithmus befindet sich auf Grund der obigen Annahme links von allen Servern des DC-Algorithmus. Wir überlassen es dem Leser als Übung, zu zeigen, dass es eine optimale Zuordnung π gibt, in der der Server j von OPT und der Server S_{links} einander zugeordnet sind. Die folgende Abbildung stellt die beschriebene Situation noch einmal für ein Beispiel mit $k = 4$ dar. Die Pfeile symbolisieren eine optimale Zuordnung π.



Da sich der Server S_{links} zu dem Punkt o'_j bewegt, verringert sich der Wert der optimalen Zuordnung um genau die Strecke, die dieser Server zurücklegt, also um $DC_i(\sigma)$. Da der Wert der Zuordnung mit dem Faktor k im Potential gewichtet ist, sinkt der erste Summand im Potential somit um $k \cdot DC_i(\sigma)$. Die paarweisen Distanzen der Server des DC-Algorithmus steigen jedoch an. Da sich der Server S_{links} von allen $k-1$ anderen Server wegbewegt, steigt die Summe der paarweisen Distanzen um genau $(k-1) \cdot DC_i(\sigma)$. Insgesamt gilt somit

$$\Phi_i \leq \Phi'_{i-1} - k \cdot DC_i(\sigma) + (k-1) \cdot DC_i(\sigma) = \Phi'_{i-1} - DC_i(\sigma).$$

Nun betrachten wir noch den Fall, dass der DC-Algorithmus zwei Server S_1 und S_2 bewegt. Diese bewegen sich mit der gleichen Geschwindigkeit aufeinander zu, bis der erste die Anfrage σ_i erreicht. Beide Server legen dabei eine Entfernung von $DC_i(\sigma)/2$ zurück. Wir überlassen es wieder dem Leser als Übung, zu zeigen, dass es vor der Bewegung der Server S_1 und S_2 eine optimale Zuordnung π gibt, in der der Server von OPT, der sich an Position $o'_j = \sigma_i$ befindet, entweder Server S_1 oder Server S_2 zugeordnet ist. Durch die Bewegung dieses Servers reduziert sich der Wert der Zuordnung π um $DC_i(\sigma)/2$. Selbst wenn der andere Server sich von seinem zugeordneten Server von OPT wegbewegt, kann sich der Wert der Zuordnung π für diesen um maximal $DC_i(\sigma)/2$ erhöhen. Somit steigt durch das Bewegen der Server S_1 und S_2 der Wert der Zuweisung π insgesamt nicht an. Die folgende Abbildung stellt die beschriebene Situation noch einmal für ein Beispiel mit $k = 4$ dar. Die Pfeile symbolisieren eine optimale Zuordnung π.



Nun betrachten wir noch den Term Σ_{DC}. Für jeden Server S' des DC-Algorithmus außer S_1 und S_2 bewegt sich genau einer dieser beiden Server auf S' zu und einer bewegt sich von S' um dieselbe Entfernung weg. Deshalb ändert sich der Wert, den S' zu Σ_{DC} beiträgt, nicht. Die einzige Entfernung, die sich ändert und deren Änderung nicht durch eine andere kompensiert wird, ist die Entfernung zwischen S_1 und S_2, die um genau $DC_i(\sigma)$ fällt. Insgesamt folgt damit wie gewünscht

$$\Phi_i \leq \Phi'_{i-1} - DC_i(\sigma). \qquad \square$$

Aus den vorangegangenen Lemmas 3.7 und 3.8 folgt nun direkt die zu zeigende Ungleichung (3.1), denn

$$\Phi_i \leq \Phi'_{i-1} - DC_i(\sigma) \leq \Phi_{i-1} + k \cdot \text{OPT}_i(\sigma) - DC_i(\sigma).$$

Damit ist der Beweis abgeschlossen und es ist gezeigt, dass der DC-Algorithmus auf der Linie k-kompetitiv ist. $\square$

Die obige Analyse des DC-Algorithmus ist bestmöglich, da aus Theorem 3.3 direkt folgt, dass der DC-Algorithmus keinen kleineren kompetitiven Faktor als k erreichen kann.

3.3.2 Analyse des DC-Algorithmus auf Bäumen

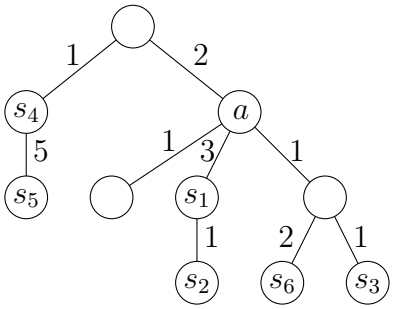
Ein weiterer wichtiger Spezialfall sind Metriken, die durch gewichtete Bäume dargestellt werden können. Wir nennen eine Metrik $\mathcal{M} = (M, d)$ eine *Baummetrik* oder einfach nur einen *Baum*, wenn es einen Baum $G = (V, E)$ mit $V = M$ und Kantengewichten $w : E \to \mathbb{R}_{\geq 0}$ gibt, sodass der Abstand $d(x, y)$ zwischen jedem Paar von Punkten x und y genau dem Gewicht des Weges zwischen x und y in dem gewichteten Baum G entspricht. Wir haben dabei implizit ausgenutzt, dass es in einem Baum zwischen jedem Paar von Knoten einen eindeutig bestimmten Weg gibt.

Es seien $s_1, \ldots, s_k \in M$ die aktuellen Positionen der Server des DC-Algorithmus. Dabei ist es erlaubt, dass sich mehrere Server an derselben Position befinden. Um den DC-Algorithmus auf Bäume erweitern zu können, definieren wir zunächst, wann ein Server s_i zu einer Anfrage an einem Punkt $a \in M$ *benachbart* ist. Dies ist dann der Fall, wenn sich auf dem eindeutigen Weg von s_i zu a in dem Baum G kein anderer Server befindet. Befinden sich an einem Punkt $b \in M$ mehrere Server und befindet sich auf dem Weg von b zu a kein weiterer Server, so definieren wir einen beliebig ausgewählten Server auf Punkt b als zu a benachbart. Alle anderen Server auf Punkt b betrachten wir als nicht zu a benachbart.

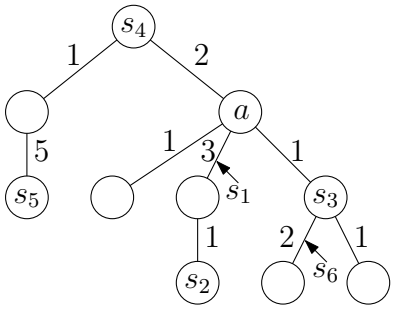
Der DC-Algorithmus bewegt alle zu a benachbarten Server solange mit derselben Geschwindigkeit in Richtung der Anfrage a, bis der erste von diesen Anfrage a erreicht. Die zurückgelegte Wegstrecke ist dabei immer auf das Gewicht der Kanten zu beziehen. Sind also beispielsweise zwei Server genau eine Kante von der Anfrage a entfernt, bewegt sich aber der erste Server über eine Kante mit Gewicht eins und der zweite über eine Kante mit Gewicht zwei, so befindet sich der zweite Server genau in der Mitte der Kante mit Gewicht zwei, wenn der erste Server die Kante mit Gewicht eins vollständig passiert und die Anfrage a erreicht hat. Wir definieren den Schritt des DC-Algorithmus damit als abgeschlossen. Formal haben wir zwar gesagt, dass sich jeder Server zu jedem Zeitpunkt an einem Punkt der Metrik befindet, was in dem obigen Beispiel für den zweiten Server nicht der Fall ist, aber da man den so beschriebenen DC-Algorithmus ohnehin wieder durch einen faulen Algorithmus simulieren kann, dessen Kosten nicht größer sind, stellt dies kein Problem dar.

Eine weitere wichtige Eigenschaft des DC-Algorithmus ist, dass sich die Menge der zu der Anfrage a benachbarten Server im Laufe der Bearbeitung einer Anfrage ändern kann. Es kann nämlich die Situation eintreten, dass ein Server S_1 zu Beginn zu der Anfrage a benachbart ist und dass ein anderer Server S_2 nach einer gewissen Zeit den Pfad von S_1 an einem Knoten erreicht, der zwischen der aktuellen Position von S_1 und der Anfrage a liegt. Ab diesem Zeitpunkt ist der Server S_1 nicht mehr zu der Anfrage a benachbart und wird auch nicht mehr weiterbewegt. Es kann hingegen nicht passieren, dass während der Bearbeitung der Anfrage neue benachbarte Server hinzukommen. Das Verhalten des DC-Algorithmus ist an einem Beispiel in Abbildung 3.1 illustriert.

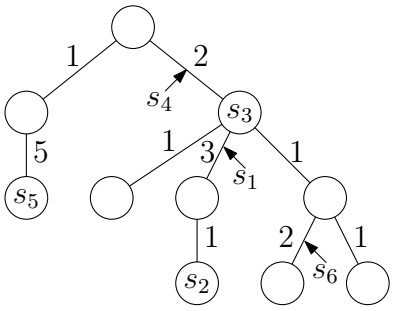
Der soeben beschriebene DC-Algorithmus auf Bäumen ist tatsächlich eine Erweiterung des DC-Algorithmus, den wir für Linien kennengelernt haben. Für diesen Spezialfall haben wir bereits einen kompetitiven Faktor von k nachgewiesen. Wir zeigen nun, dass der DC-Algorithmus diesen kompetitiven Faktor sogar für allgemeine Bäume erreicht.



(a) Die Server s_1, s_3, s_4 und s_6 sind zur Anfrage a benachbart und bewegen sich in Richtung a.



(b) Nach einer Zeiteinheit haben die Server s_3 und s_4 den nächsten Knoten auf ihrem jeweiligen Pfad zu a erreicht. Die Server s_1 und s_6 haben zu diesem Zeitpunkt erst ein Drittel bzw. die Hälfte ihres Weges zum nächsten Knoten zurückgelegt. Ab diesem Zeitpunkt ist Server s_6 nicht mehr zu der Anfrage a benachbart und bewegt sich deshalb nicht mehr weiter.



(c) Nach einer weiteren Zeiteinheit hat Server s_3 die Anfrage a erreicht. Zu diesem Zeitpunkt haben die Server s_4 und s_1 die Hälfte bzw. zwei Drittel ihrer aktuellen Kante zurückgelegt.

Abbildung 3.1: Beispiel für die Ausführung des DC-Algorithmus

Theorem 3.9. *Der DC-Algorithmus ist k-kompetitiv für das k-Server-Problem auf jeder Baummetrik $\mathcal{M} = (V, d)$.*

Beweis. Wir beziehen uns in diesem Beweis stark auf den Beweis von Theorem 3.6 und wir benutzen dieselbe Notation und sogar dieselbe Potentialfunktion. Statt durch $2k^2$ können wir den Wert der Potentialfunktion für die Metrik $\mathcal{M}$ nur noch durch $2k^2 \cdot \max_{x,y \in V} d(x, y)$ nach oben abschätzen. Da wir eine feste Metrik betrachten, handelt es sich hierbei um eine Konstante, die nicht von der Eingabe abhängt. Der Leser sollte sich davon überzeugen, dass die Aussage und der Beweis von Lemma 3.7 weiterhin Bestand haben. Lediglich der Beweis von Lemma 3.8 ändert sich, nicht jedoch die Aussage. Wir weisen die entsprechende Ungleichung

$$\Phi_i \leq \Phi'_{i-1} - DC_i(\sigma) \tag{3.2}$$

nun für die Baummetrik $\mathcal{M}$ nach. Dazu teilen wir die Bearbeitung der Anfrage an Position σ_i in Phasen abhängig davon ein, wie viele Server der DC-Algorithmus bewegt. Dies können je nach Grad des Baumes am Anfang durchaus alle k Server sein. Eine

Phase ist abgeschlossen, wenn entweder ein Server die Anfrage erreicht oder wenn sich die Anzahl an Servern, die zu σ_i benachbart sind, reduziert.

Wir betrachten eine Phase, in der sich m Server bewegen, und definieren d als die Wegstrecke, die jeder dieser Server in der Phase zurücklegt. Wieder schätzen wir die Auswirkung dieser Serverbewegungen auf die beiden Terme $M_{\min}$ und Σ_{DC} getrennt voneinander ab. Wir beginnen mit $M_{\min}$ und nutzen genauso wie im Falle der Linie aus, dass es eine optimale Zuordnung π gibt, in der ein zu σ_i benachbarter Server des DC-Algorithmus dem Server des optimalen Offline-Algorithmus an Position σ_i zugewiesen ist. Da sich dieser Server auf den ihm zugeordneten Server des optimalen Offline-Algorithmus zubewegt und alle anderen $m-1$ aktiven Server des DC-Algorithmus sich im schlimmsten Fall von ihren zugeordneten Servern wegbewegen, steigt der Wert der optimalen Zuordnung um höchstens $(m-2)d$. Da dieser Term im Potential Φ mit k gewichtet ist, bewirkt die Veränderung des Terms $M_{\min}$ einen Anstieg des Potentials um höchstens

$$k(m-2)d = kmd - 2kd. \tag{3.3}$$

Nun betrachten wir noch, wie sich der Term Σ_{DC} durch die Bewegung der m Server in dieser Phase ändert. Für jeden Server, der nicht zu der Anfrage σ_i benachbart ist, gibt es einen aktiven Server, der sich von ihm wegbewegt, und $m-1$ Server, die sich auf ihn zubewegen. Da es insgesamt $k-m$ Server gibt, die sich nicht bewegen, führt dies zu einer Verringerung des Terms Σ_{DC} um

$$(k-m)(m-2)d = kmd - 2kd - m^2d + 2md.$$

Jedes Paar von aktiven Servern bewegt sich aufeinander zu und reduziert seinen Abstand um $2d$. Da es insgesamt $\binom{m}{2}$ solcher Paare gibt, führt dies zu einer Verringerung des Terms Σ_{DC} um

$$\binom{m}{2}2d = dm(m-1) = m^2d - md.$$

Insgesamt verringert sich Σ_{DC} somit um

$$kmd - 2kd + md.$$

Gemeinsam mit (3.3) impliziert dies, dass sich das Potential um mindestens

$$kmd - 2kd + md - (kmd - 2kd) = md$$

verringert. Dieser Term entspricht genau den Kosten, die dem DC-Algorithmus für das Verschieben der Server in dieser Phase anfallen. Summieren wir dies über alle Phasen auf, so ist (3.2) gezeigt. □

3.3.3 Anwendungen des DC-Algorithmus

Bis heute ist keine Erweiterung des DC-Algorithmus bekannt, die auf allgemeinen Metriken einen kleinen kompetitiven Faktor erreicht, der nur von k abhängt. Man kann den DC-Algorithmus allerdings so auf allgemeine endliche Metriken verallgemeinern,

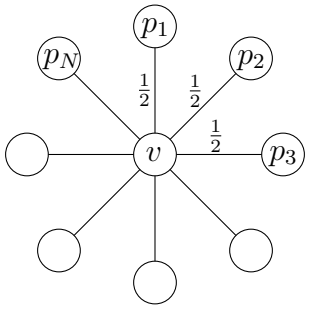
dass er einen endlichen kompetitiven Faktor erreicht, der von der Größe der Metrik abhängt. Sei dazu $G = (V, E)$ ein beliebiger gewichteter Graph mit N Knoten, der eine Metrik $\mathcal{M}$ darstellt. Um das k-Server-Problem auf $\mathcal{M}$ mithilfe des DC-Algorithmus zu lösen, berechnen wir zunächst einen minimalen Spannbaum $T = (V, E_T)$ von G und betrachten anschließend das k-Server-Problem auf der Baummetrik $\mathcal{M}_T$, die durch T induziert wird. Verglichen mit der Metrik $\mathcal{M}$ sind in der Baummetrik einige Distanzen möglicherweise größer geworden.

Wir bezeichnen mit $\mathrm{OPT}(\sigma)$ und $\mathrm{OPT}_T(\sigma)$ den Wert einer optimalen Offline-Lösung des k-Server-Problems auf der Metrik $\mathcal{M}$ bzw. $\mathcal{M}_T$ bei Eingabe σ. Wenden wir den DC-Algorithmus bei der Eingabe σ auf die Baummetrik $\mathcal{M}_T$ an, so garantiert uns Theorem 3.9, dass wir eine Lösung mit Wert $w_{DC}(\sigma) \leq k \cdot \mathrm{OPT}_T(\sigma) + \tau$ erhalten, wobei τ eine Konstante ist, die nicht von σ (wohl aber von $\mathcal{M}_T$) abhängt.

Um die Kosten $\mathrm{OPT}_T(\sigma)$ zu den Kosten $\mathrm{OPT}(\sigma)$, die uns eigentlich interessieren, in Beziehung zu setzen, nutzen wir die folgende Eigenschaft von minimalen Spannbäumen: Für jede Kante $e = \{x, y\} \in E$ mit Gewicht $w(e)$ beträgt die Länge des Weges zwischen den Knoten x und y im minimalen Spannbaum T höchstens $(N-1)w(e)$. Dem Leser sei der Beweis dieser Aussage als Übung empfohlen. Mit dieser Aussage folgt direkt, dass $\mathrm{OPT}_T(\sigma) \leq (N-1) \cdot \mathrm{OPT}(\sigma)$ gilt. Insgesamt erhalten wir das folgende Korollar.

Korollar 3.10. *Der DC-Algorithmus ist $(N-1)k$-kompetitiv für das k-Server-Problem auf beliebigen Metriken mit N Punkten.*

Wir haben am Anfang des Kapitels bereits diskutiert, dass Paging als der Spezialfall des k-Server-Problems aufgefasst werden kann, in dem der paarweise Abstand zwischen jedem Paar von Punkten eins ist. Wir können Paging aber auch durch eine sternförmige Metrik modellieren, in der es einen zentralen Punkt v und für jede Seite i einen Punkt p_i gibt. Dabei sei der Abstand zwischen v und jedem Punkt p_i genau $1/2$. Der Abstand zwischen zwei verschiedenen Punkten p_i und p_j ergibt sich aus dem transitiven Abschluss des Graphen zu 1. Die Metrik kann also durch den folgenden Graphen beschrieben werden.



Jeder Algorithmus für das k-Server-Problem auf dieser Metrik kann als Paging-Algorithmus aufgefasst werden, wenn man die Knoten p_i, auf denen sich ein Server befindet, als die Seiten betrachtet, die momentan im Cache gespeichert sind. Wir initialisieren den Algorithmus für das k-Server-Problem so, dass sich zu Beginn alle Server an Knoten v befinden. Dies entspricht der Initialisierung für das Paging-Problem mit leerem Cache. Der einzige Unterschied in den Kosten ist, dass die erste Bewegung eines Servers

von v zu einem Knoten p_i nur Kosten 1/2 statt 1 verursacht. Alle weiteren Übergänge von p_i zu einem anderen p_j verursachen Kosten 1. Dadurch kann zwischen den Kosten eines Algorithmus für das k-Server-Problem auf dieser Metrik und der Anzahl an Seitenfehlern des entsprechenden Paging-Algorithmus eine Abweichung von höchstens $k/2$ entstehen. Da es sich hierbei um eine Konstante handelt, ist diese Diskrepanz für den kompetitiven Faktor nicht relevant.

Da es sich bei obiger Metrik um einen Baum handelt, kann der DC-Algorithmus direkt ausgeführt werden und erreicht gemäß Theorem 3.9 einen kompetitiven Faktor von k. Solange sich noch mindestens ein Server an Knoten v befindet, wird bei jedem Seitenfehler ein beliebiger davon auf den der Seite entsprechenden Knoten verschoben. Ist kein Server mehr an Position v und tritt ein Seitenfehler ein, so werden alle Server vom DC-Algorithmus wieder zurück auf den Knoten v gefahren. Von dort aus wird dann ein beliebiger Server zu der Anfrage gefahren während die anderen an Knoten v bleiben. Demzufolge entspricht der DC-Algorithmus auf dieser Metrik genau dem Algorithmus FWF (flush-when-full).

Wir können mithilfe dieses Ansatzes sogar das gewichtete Paging-Problem lösen, bei dem verschiedene Seiten beim Laden in den Cache unterschiedlich große Kosten verursachen können. Jede Seite i hat also ein bestimmtes Gewicht w_i. Dies können wir dadurch modellieren, dass wir der Kante zwischen den Knoten v und p_i das Gewicht $w_i/2$ anstatt 1/2 zuweisen. Auch für dieses gewichtete Paging-Problem erreicht der DC-Algorithmus einen kompetitiven Faktor von k. Allerdings stimmen die Kosten des DC-Algorithmus nicht exakt mit den Kosten des entsprechenden Paging-Algorithmus überein, da die Gewichte der Seiten, auf denen sich gerade Server befinden, erst zur Hälfte gezählt wurden. Diese Diskrepanz beträgt aber höchstens $k \cdot \max_i w_i/2$, was wiederum eine Konstante ist und keine Auswirkungen auf den kompetitiven Faktor hat.

Korollar 3.11. *Der DC-Algorithmus ist k-kompetitiv für das Paging-Problem und das gewichtete Paging-Problem.*

3.4 Das 2-Server-Problem im euklidischen Raum

Wir betrachten nun als weiteren Spezialfall das 2-Server-Problem im euklidischen Raum. Zunächst beschränken wir uns auf das Einheitsquadrat $M = [0,1]^2$ und die Metrik $\mathcal{M} = (M, d)$, wobei der Abstand zweier Punkte $x = (x_1, x_2) \in M$ und $y = (y_1, y_2) \in M$ als

$$d(x,y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

gegeben ist. Für diese Metrik $\mathcal{M}$, bei der es sich nicht um eine Baummetrik handelt, entwerfen wir einen Algorithmus, der einen kompetitiven Faktor von 3 erreicht und auf ähnlichen Ideen wie der DC-Algorithmus beruht.

Um diesen Algorithmus zu beschreiben, definieren wir den *Slack* dreier Punkte $x, y, r \in M$ als

$$\text{slack}(x,y,r) = d(x,y) + d(x,r) - d(y,r).$$

Wegen der Dreiecksungleichung ist der Slack niemals negativ. Für jedes $\gamma \in [0,1]$ betrachten wir einen Algorithmus *Slack Coverage$_\gamma$ (SC$_\gamma$)*, dessen Verhalten wie folgt beschrieben werden kann. Es seien $x \in M$ und $y \in M$ die Positionen, an denen sich momentan die Server des Algorithmus SC_γ befinden. Außerdem sei $r \in M$ die Position der nächsten Anfrage und ohne Beschränkung der Allgemeinheit sei $d(x,r) \leq d(y,r)$. Der Algorithmus SC_γ bewegt zunächst den Server an Position y um $\gamma \cdot \mathrm{slack}(x,y,r)$ in Richtung x und bewegt anschließend den Server an Position x zu der Anfrage an Position r. Da wir $d(x,r) \leq d(y,r)$ angenommen haben, ist der Slack durch $d(x,y)$ nach oben beschränkt, d. h. der Server an Position y wird tatsächlich in Richtung x, nicht aber darüber hinaus verschoben.

Der Leser sollte sich als Übung zwei Dinge überlegen. Zum einen sollte er sich klar machen, dass der Algorithmus $\mathrm{SC}_{1/2}$ angewendet auf die Linie genau dem DC-Algorithmus mit zwei Servern entspricht. Zum anderen sollte er beweisen, dass $d(y',r) \leq d(y,r)$ für jedes $\gamma \in [0,1]$ gilt, wobei y' die neue Position des Servers ist, der sich an Position y befand. Das bedeutet, die Entfernung dieses Servers zu der Anfrage r vergrößert sich durch die Verschiebung nicht.

Theorem 3.12. *Der Algorithmus $SC_{1/2}$ ist 3-kompetitiv für das 2-Server-Problem im euklidischen Einheitsquadrat.*

Beweis. Wir greifen in dem Beweis auf eine Potentialfunktion zurück, die große Ähnlichkeit zu der Potentialfunktion aus Theorem 3.6 besitzt. Es seien x und y die aktuellen Positionen der Server des Algorithmus SC_γ mit $\gamma = 1/2$ und es seien o_1 und o_2 die Positionen der Server des optimalen Offline-Algorithmus. Dann ist das Potential Φ für zwei noch zu wählende Parameter $a, b \in \mathbb{R}_{\geq 0}$ als

$$\Phi = a \cdot M_{\min} + b \cdot d(x,y)$$

definiert, wobei $M_{\min}$ wie im Beweis von Theorem 3.6 den Wert einer optimalen Zuordnung der Positionen x und y zu den Positionen o_1 und o_2 angibt. Auch der zweite Term $d(x,y)$ kommt in der Potentialfunktion in jenem Beweis als Σ_{DC} vor. Da es hier nur zwei Server gibt, degeneriert dieser Term zu $d(x,y)$.

Es sei eine Eingabe $\sigma = (\sigma_1, \ldots, \sigma_n)$ gegeben und es bezeichne wie üblich $\Phi_0, \Phi_1, \ldots, \Phi_n$ die Sequenz der Potentiale, die sich für diese Eingabe ergeben. Unser Ziel ist es, mithilfe der Potentialfunktion Φ zu zeigen, dass der SC_γ-Algorithmus a-kompetitiv ist. Da die Potentialfunktion nur Werte zwischen 0 und der Konstante $\sqrt{2}(2a+b)$ annehmen kann, muss dazu nur wie im Beweis von Theorem 3.6 gezeigt werden, dass für jedes $i \geq 1$ die Ungleichung

$$\Phi_i - \Phi_{i-1} \leq a \cdot \mathrm{OPT}_i(\sigma) - SC_i(\sigma)$$

gilt, wobei $\mathrm{OPT}_i(\sigma)$ und $SC_i(\sigma)$ die Kosten des optimalen Offline-Algorithmus bzw. die Kosten des Algorithmus $\mathrm{SC}_{1/2}$ für die Bearbeitung der Anfrage σ_i bezeichnen.

Es seien o_1' und o_2' die Positionen der Server des optimalen Offline-Algorithmus, nachdem dieser die Anfrage $r := \sigma_i$ bearbeitet hat. Wir betrachten wieder das Potential Φ_{i-1}' bezüglich der Positionen x und y der Server des SC_γ-Algorithmus und der Positionen o_1' und o_2' der Server des optimalen Offline-Algorithmus. Dieses Potential bezieht sich also auf die Konfiguration, die erreicht ist, nachdem der SC_γ-Algorithmus die

Anfragen $\sigma_1, \ldots, \sigma_{i-1}$ bearbeitet hat und nachdem der optimale Offline-Algorithmus die Anfragen $\sigma_1, \ldots, \sigma_i$ bearbeitet hat.

Wir gehen ohne Beschränkung der Allgemeinheit davon aus, dass der optimale Offline-Algorithmus faul ist und somit bei der Bearbeitung der Anfrage höchstens einen Server um eine Strecke der Länge $\mathrm{OPT}_i(\sigma)$ bewegt. Diese Bewegung hat keinen Einfluss auf den Term $d(x, y)$ in der Potentialfunktion. Der Wert einer optimalen Zuordnung der Server kann um maximal $\mathrm{OPT}_i(\sigma)$ steigen, da sich der Server, den der optimale Offline-Algorithmus bewegt, schlimmstenfalls von dem ihm zugewiesenen Server des SC_γ-Algorithmus wegbewegt. Da der Wert der optimalen Zuordnung in der Potentialfunktion mit dem Faktor a gewichtet ist, gilt somit (vergleiche Lemma 3.7)

$$\Phi'_{i-1} \leq \Phi_{i-1} + a \cdot \mathrm{OPT}_i(\sigma).$$

Nun müssen wir den Einfluss der Bewegung der Server des SC_γ-Algorithmus auf das Potential abschätzen. Aus der Definition des Algorithmus folgt, dass seine Kosten $SC_i(\sigma)$ bei der Bearbeitung der Anfrage σ_i genau $d(x, r)+\gamma\cdot\mathrm{slack}(x, y, r)$ betragen. Wir müssen argumentieren, dass sich das Potential um mindestens diesen Wert reduziert (vergleiche Lemma 3.8). Wir betrachten zunächst die Änderung $\Delta d(x, y)$ des zweiten Terms und erhalten

$$\Delta d(x, y) := d(x', y') - d(x, y) = d(r, y') - d(x, y) \leq d(r, y) - d(x, y),$$

wobei wir in der letzten Abschätzung die Eigenschaft ausgenutzt haben, dass der Server, der vor Bearbeitung der Anfrage σ_i an Position y steht, nach dem Schritt nicht weiter von der Anfrage r entfernt ist als vorher.

Um die Änderung des Wertes der optimalen Zuordnung abschätzen zu können, unterscheiden wir zwei Fälle abhängig davon, wie die optimale Zuordnung vor der Bearbeitung der Anfrage σ_i durch den SC_γ-Algorithmus aussieht. Es sei π die optimale Zuordnung der Positionen x und y zu den Positionen o'_1 und o'_2. Da der optimale Offline-Algorithmus gerade die Anfrage $\sigma_i = r$ bearbeitet hat, steht mindestens einer seiner Server an dieser Position. Es sei ohne Beschränkung der Allgemeinheit $o'_1 = r$.

- **1. Fall:** x und o'_1 sind in π einander zugeordnet.
  Der Wert der Zuweisung π verringert sich durch die Bewegung des Servers von x zu r um $d(x, r)$ und erhöht sich durch die Bewegung des Servers, der zu Beginn an Position y steht, um maximal $\gamma \cdot \mathrm{slack}(x, y, r)$. Insgesamt gilt somit

$$\begin{aligned} \Phi_i - \Phi'_{i-1} &\leq a \cdot [\gamma \cdot \mathrm{slack}(x, y, r) - d(x, r)] + b \cdot \Delta d(x, y) \\ &\leq a \cdot [\gamma \cdot \mathrm{slack}(x, y, r) - d(x, r)] + b \cdot [d(r, y) - d(x, y)]. \end{aligned} \tag{3.4}$$

- **2. Fall:** y und o'_1 sind in π einander zugeordnet.
  Nachdem der SC_γ-Algorithmus den Server von x zu $r = o'_1$ bewegt hat, gibt es ein optimales Matching, in dem dieser Server und der Server des optimalen Offline-Algorithmus an derselben Position einander zugewiesen sind. In diesem Matching sind dementsprechend auch die beiden verbleibenden Server an den

Positionen y' und o'_2 einander zugewiesen. Für die Änderung $\Delta M_{\min}$ des Wertes der optimalen Zuweisung ergibt sich somit

$$\begin{aligned}\Delta M_{\min} &= [d(x', o'_1) + d(y', o'_2)] - [d(x, o'_2) + d(y, o'_1)] \\ &= d(y', o'_2) - d(x, o'_2) - d(y, r) \\ &\leq d(y', x) - d(y, r) \\ &= d(y, x) - \gamma \cdot \text{slack}(x, y, r) - d(y, r),\end{aligned}$$

wobei wir bei der Ungleichung die Dreiecksungleichung ausgenutzt haben und in der letzten Zeile, dass $\text{slack}(x, y, r) \leq d(x, y)$ gilt, und die Position y' deshalb zwischen y und x liegt. Insgesamt gilt in diesem Fall

$$\begin{aligned}\Phi_i - \Phi'_{i-1} &\leq a \cdot [d(y, x) - \gamma \cdot \text{slack}(x, y, r) - d(y, r)] + b \cdot \Delta d(x, y) \\ &\leq a \cdot [d(y, x) - \gamma \cdot \text{slack}(x, y, r) - d(y, r)] + b \cdot [d(r, y) - d(x, y)].\end{aligned} \tag{3.5}$$

Wir müssen nun nachweisen, dass in beiden Fällen

$$\Phi_i - \Phi'_{i-1} \leq -SC_i(\sigma) = -[d(x, r) + \gamma \cdot \text{slack}(x, y, r)]$$

gilt. Eine hinreichende Bedingung hierfür, die sich aus (3.4) und (3.5) ergibt, ist, dass die beiden Ungleichungen

$$\begin{aligned}&a \cdot [\gamma \cdot \text{slack}(x, y, r) - d(x, r)] + b \cdot [d(r, y) - d(x, y)] \\ &\leq -[d(x, r) + \gamma \cdot \text{slack}(x, y, r)]\end{aligned}$$

und

$$\begin{aligned}&a \cdot [d(y, x) - \gamma \cdot \text{slack}(x, y, r) - d(y, r)] + b \cdot [d(r, y) - d(x, y)] \\ &\leq -[d(x, r) + \gamma \cdot \text{slack}(x, y, r)]\end{aligned}$$

erfüllt sind. Diese Ungleichungen sind äquivalent zu

$$d(x, y) \cdot [\gamma(a + 1) - b] + d(x, r) \cdot [\gamma(a + 1) + 1 - a] + d(y, r) \cdot [b - \gamma(a + 1)] \leq 0$$

und

$$d(x, y) \cdot [\gamma(1 - a) + a - b] + d(x, r) \cdot [\gamma(1 - a) + 1] + d(y, r) \cdot [b - a - \gamma(1 - a)] \leq 0.$$

Können wir Parameter a, b und γ finden, für die diese beiden Ungleichungen erfüllt sind, so haben wir bewiesen, dass der SC_γ-Algorithmus a-kompetitiv ist. Man kann leicht nachrechnen, dass beide Ungleichungen für $\gamma = 1/2$, $a = 3$ und $b = 2$ erfüllt sind. Damit ist das Theorem bewiesen. □

Der Leser sollte als Übung zeigen, dass der $\text{SC}_{1/2}$-Algorithmus auch in höherdimensionalen euklidischen Räumen einen kompetitiven Faktor von 3 erreicht. Dies ist deswegen von Interesse, da man in einem gewissem Sinne jeden metrischen Raum in einen euklidischen Raum einbetten kann. Ist ein beliebiger metrischer Raum $\mathcal{M}$ mit N Punkten

gegeben, so können wir dazu N Punkte in einem hochdimensionalen euklidischen Raum finden, die den Punkten des metrischen Raumes entsprechen. Dabei gilt, dass die Entfernung zweier Punkte im euklidischen Raum nicht kleiner ist als die Entfernung der entsprechenden Punkte in $\mathcal{M}$ und dass sie um höchstens einen Faktor $O(\log N)$ größer ist.

Mit demselben Argument, mit dem wir Korollar 3.10 bewiesen haben, folgt daraus, dass wir das 2-Server-Problem in beliebigen Metriken mit einem kompetitiven Faktor von $O(\log N)$ lösen können. Dazu betten wir die gegebene Metrik zunächst in eine euklidische Metrik ein und wenden dann den $\mathrm{SC}_{1/2}$-Algorithmus auf dieser Metrik an.

Kapitel 4

Approximation von Metriken

Wir haben beim k-Server-Problem gesehen, dass es hilfreich sein kann, allgemeine Metriken durch spezielle Metriken wie Baummetriken oder euklidische Metriken zu approximieren. Beispielsweise haben wir erst bewiesen, dass der DC-Algorithmus für das k-Server-Problem k-kompetitiv für beliebige Baummetriken ist, und anschließend argumentiert, dass man zu jeder Metrik $\mathcal{M}$ mit N Punkten eine Baummetrik finden kann, in der die Abstände verglichen mit $\mathcal{M}$ nicht kürzer sind und um maximal den Faktor $N-1$ gestreckt werden. Dadurch haben wir einen $(N-1)k$-kompetitiven Algorithmus für das k-Server-Problem auf beliebigen Metriken erhalten.

In diesem Kapitel beschäftigen wir uns intensiver mit der Approximation beliebiger Metriken durch Baummetriken, da dies nicht nur für das k-Server-Problem, sondern auch für viele andere Online-Probleme von großer Bedeutung ist. Wir werden sehen, dass man die maximale Streckung $N-1$ deutlich verringern kann, wenn man statt einer einzigen Baummetrik eine Wahrscheinlichkeitsverteilung auf Baummetriken einsetzt. Um dies zu präzisieren, benötigen wir zunächst einige Definitionen.

Definition 4.1. *Es sei $\mathcal{M} = (M, d)$ eine beliebige Metrik. Wir sagen, dass eine Metrik $\mathcal{M}' = (M', d')$ mit $M \subseteq M'$ die Metrik $\mathcal{M}$* dominiert*, wenn $d(x, y) \leq d'(x, y)$ für alle $x, y \in M$ gilt. Es sei $\mathcal{S}$ eine Menge von Metriken, die $\mathcal{M}$ dominieren, und es sei $\mathcal{D}$ eine Wahrscheinlichkeitsverteilung auf der Menge $\mathcal{S}$. Wir sagen, dass $(\mathcal{S}, \mathcal{D})$ eine* α-Approximation *der Metrik $\mathcal{M}$ ist, wenn für alle $x, y \in M$*

$$\mathop{\mathbf{E}}_{(M',d')\sim\mathcal{D}}[d'(x, y)] \leq \alpha \cdot d(x, y)$$

gilt. Das bedeutet, für jedes feste Paar x und y von Punkten aus M darf die erwartete Entfernung in einer zufällig gemäß der Verteilung $\mathcal{D}$ gewählten Metrik höchstens $\alpha \cdot d(x, y)$ betragen.

Ist $(\mathcal{S}, \mathcal{D})$ eine α-Approximation einer Metrik $\mathcal{M}$, so sagen wir auch, dass die Metrik $\mathcal{M}$ in die Menge $\mathcal{S}$ *eingebettet* ist. Wir nennen α dann den *Streckungsfaktor der Einbettung.*

4.1 Approximation durch Baummetriken

Deterministisch können beliebige Metriken nicht besser als mit einem Streckungsfaktor von $\Omega(N)$ in Baummetriken eingebettet werden. Die Einbettung mithilfe des minimalen Spannbaums, die wir in Abschnitt 3.3.3 beschrieben haben, ist also asymptotisch optimal. Ein Beispiel, bei dem keine asymptotisch bessere Einbettung möglich ist, ist ein Kreis auf N Knoten, in dem jede Kante das Gewicht 1 hat. Man sieht schnell ein, dass jeder Baum, der aus dem Kreis entsteht, indem eine Kante entfernt wird, eine Streckung von $N - 1$ hat. Es ist aber nicht trivial zu argumentieren, dass es keine asymptotisch bessere Einbettung gibt, da die Metriken, die in der Einbettung benutzt werden, zusätzliche Knoten und Kanten enthalten dürfen.

Wir werden die Diskussion dieser unteren Schranke nicht weiter vertiefen und uns stattdessen mit oberen Schranken für randomisierte Einbettungen beschäftigen. Das folgende Theorem wurde 2004 von Fakcharoenphol, Rao und Talwar bewiesen [9].

Theorem 4.2. *Für jede Metrik $\mathcal{M}$ mit N Punkten gibt es eine Menge $\mathcal{S}$ von Baummetriken, die $\mathcal{M}$ dominieren, und eine Wahrscheinlichkeitsverteilung $\mathcal{D}$ auf $\mathcal{S}$, sodass $(\mathcal{S}, \mathcal{D})$ eine $O(\log N)$-Approximation von $\mathcal{M}$ ist. Es gibt ferner einen effizienten randomisierten Algorithmus, der zu jeder Metrik $\mathcal{M}$ eine Metrik aus $\mathcal{S}$ zufällig gemäß der Verteilung $\mathcal{D}$ auswählt.*

Im Folgenden sei $\mathcal{M} = (V, d)$ eine beliebige Metrik mit $N = |V|$ Punkten. Wir gehen davon aus, dass der Abstand jedes Paares von unterschiedlichen Punkten aus V echt größer als 1 ist. Dies ist keine Einschränkung, da es durch eine geeignete Skalierung der Metrik stets erreicht werden kann. Außerdem bezeichnen wir mit Δ den maximalen Abstand zweier Punkte aus V und $\delta \in \mathbb{N}$ sei so gewählt, dass $2^{\delta-1} < \Delta \leq 2^{\delta}$ gilt.

Der Beweis des Theorems besteht aus zwei Teilen. Zunächst zeigen wir, wie man aus einer rekursiven Partition der Menge V mit gewissen Eigenschaften eine Baummetrik generieren kann, die die Metrik $\mathcal{M}$ dominiert. Im zweiten Schritt zeigen wir, dass man eine solche rekursive Partition randomisiert so erzeugen kann, dass die erwartete Streckung der Einbettung durch $O(\log N)$ beschränkt ist.

4.1.1 Hierarchische Partitionen und Baummetriken

Zunächst benötigen wir einige grundlegende Definitionen.

Definition 4.3. *Eine* Partition der Metrik $\mathcal{M} = (V, d)$ mit Radius $r \geq 1$ *ist eine Partition der Menge V in Klassen $V_1, \ldots, V_\ell$, sodass für jede Menge V_i ein Zentrum $c_i \in V$ existiert, sodass $d(c_i, v) \leq r$ für alle Punkte $v \in V_i$ gilt.*

Wir weisen explizit darauf hin, dass die Zentren nicht zu den ihnen entsprechenden Klassen gehören müssen, sondern beliebig aus V gewählt sein dürfen. Außerdem beobachten wir, dass aufgrund der Dreiecksungleichung der *Durchmesser* $\max_{x,y \in V_i} d(x, y)$ jeder Klasse V_i in einer Partition mit Radius r höchstens $2r$ beträgt.

Definition 4.4. *Eine* hierarchische Partition *der Metrik $\mathcal{M} = (V, d)$ ist eine Sequenz $D_0, D_1, \ldots, D_\delta$ von $\delta + 1$ Partitionen von V mit den folgenden Eigenschaften.*

1. *Es gilt $D_\delta = \{V\}$, d. h. D_δ ist die triviale Partition der Metrik $\mathcal{M}$ mit Radius 2^δ, in der alle Punkte aus V zu derselben Klasse gehören.*
2. *Für jedes $i < \delta$ ist D_i eine Partition der Metrik $\mathcal{M}$ mit Radius 2^i, die die Partition D_{i+1} verfeinert. Das heißt, jede Klasse von D_i ist Teilmenge einer Klasse von D_{i+1}.*

Aus einer hierarchischen Partition $D_0, \ldots, D_\delta$ der Metrik $\mathcal{M} = (V, d)$ erzeugen wir nun eine Baummetrik auf V. Dazu konstruieren wir einen Baum T, dessen Knoten den Klassen der Partitionen D_i entsprechen. Die Wurzel von T entspricht der einzigen Klasse V der Partition D_δ. Die Knoten mit Tiefe 1 sind die Klassen der Partition $D_{\delta-1}$, die Knoten mit Tiefe 2 sind die Klassen der Partition $D_{\delta-2}$ und so weiter. Die Blätter des Baumes sind die Klassen der Partition D_0. Da wir davon ausgehen, dass jedes Paar von Punkten aus V einen Abstand echt größer als 1 besitzt, und da jede Klasse in D_0 einen Radius von 1 hat, besteht die Partition D_0 aus N Klassen mit jeweils einem Element. Es gibt somit eine bijektive Abbildung zwischen den Blättern von T und den Elementen aus V.

Die Kanten des Baumes T wählen wir auf die naheliegende Art und Weise. Für jedes $i < \delta$ und jede Klasse X der Partition D_i gibt es per Definition eine Klasse Y der Partition D_{i+1} mit $X \subseteq Y$. Der Knoten in T, der X repräsentiert, ist der Sohn des Knotens, der Y repräsentiert. Der Kante zwischen diesen beiden Knoten weisen wir das Gewicht 2^{i+1} zu. Abbildung 4.1 zeigt ein Beispiel für eine hierarchische Partition und den daraus resultierenden Baum.

Da es eine Bijektion zwischen der Menge V und den Blättern des Baumes T gibt, können wir den Baum T als eine Baummetrik (V_T, d_T) auf einer Menge $V_T \supseteq V$ auffassen. Dabei ist der Abstand $d_T(x, y)$ zweier Punkte $x \in V$ und $y \in V$ als das Gewicht des Weges zwischen den zu x und y gehörenden Blättern in T definiert.

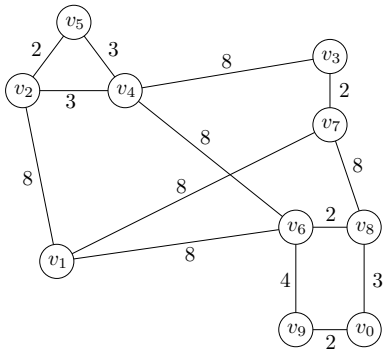
Lemma 4.5. *Für jede hierarchische Partition einer Metrik $\mathcal{M} = (V, d)$ dominiert die resultierende Baummetrik d_T die Metrik d.*

Beweis. Es seien $x, y \in V$ beliebig. Der Durchmesser der Klassen in der Partition D_i beträgt höchstens 2^{i+1}. In allen Partitionen D_i mit $2^{i+1} < d(x, y)$ gehören x und y demnach zu verschiedenen Klassen. Insbesondere in der Partition D_j für $j = \lceil \log_2 d(x, y) \rceil - 2$ ist dies der Fall, denn es gilt

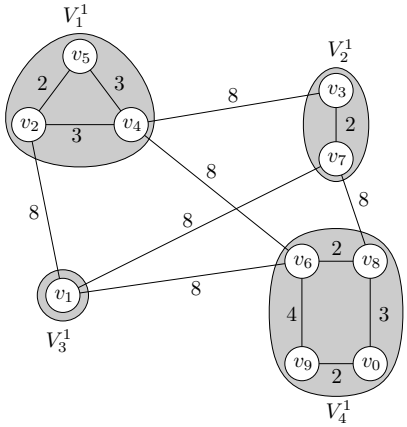
$$2^{j+1} = 2^{\lceil \log_2 d(x,y) \rceil - 1} < 2^{\log_2 d(x,y)} = d(x, y).$$

Auf dem Weg zwischen den beiden Blättern von T, die den Punkten x und y entsprechen, gibt es somit auf jeden Fall zwei Kanten zwischen den Partitionen D_j und D_{j+1} mit Gewicht 2^{j+1}. Also gilt

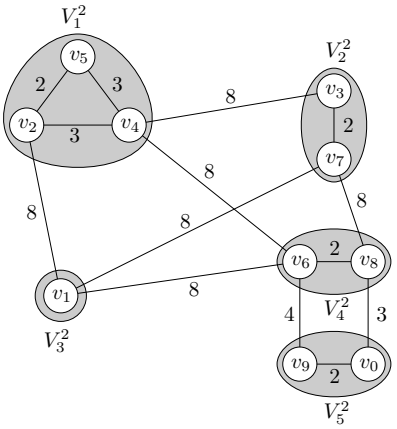
$$d_T(x, y) \geq 2 \cdot 2^{j+1} = 2^{j+2} = 2^{\lceil \log_2 d(x,y) \rceil} \geq d(x, y). \qquad \square$$



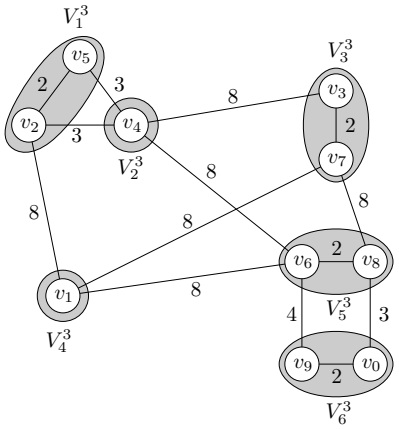
(a) Die Metrik d auf $V = \{v_0, \ldots, v_9\}$ wird durch die kürzesten Wege in diesem Graphen induziert.



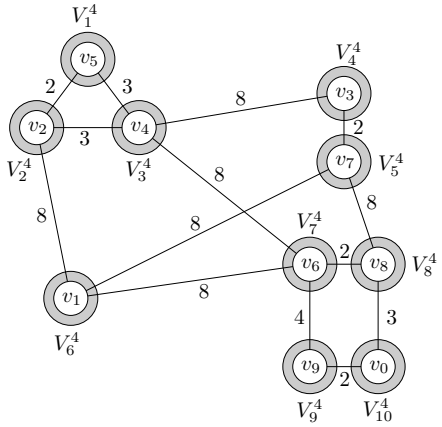
(b) Es gilt $\Delta = 16$ und $\delta = 4$. Das Bild zeigt die Partition D_3.



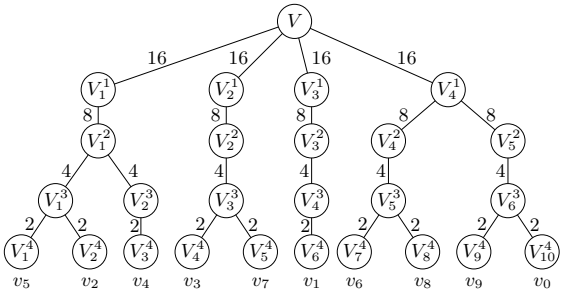
(c) die Partition D_2



(d) die Partition D_1



(e) die Partition D_0



(f) Der Baum, der zu der hierarchischen Partition aus den Abbildungen (a) bis (e) gehört.

Abbildung 4.1: Beispiel für eine hierarchische Partition

4.1.2 Erzeugung von hierarchischen Partitionen

Wir geben nun einen randomisierten Algorithmus an, der eine hierarchische Partition einer gegebenen Metrik $\mathcal{M} = (V, d)$ berechnet. Für eine Menge $X \subseteq V$, einen Knoten $v \in V$ und einen Radius $r \geq 0$ bezeichnen wir im Folgenden mit $\mathcal{B}(X, v, r)$ die Kugel in X mit Radius r und Zentrum v, d. h.

$$\mathcal{B}(X, v, r) = \{x \in X \mid d(v, x) \leq r\}.$$

Der folgende Algorithmus greift auf die Funktion PARTITION zurück, die weiter unten angegeben ist. Diese Funktion erhält als Parameter neben der Metrik $\mathcal{M} = (V, d)$ eine Partition D von V, einen Radius α und eine Permutation π der Punkte aus V. Sie gibt eine Partition von V mit Radius α zurück, die D verfeinert.

HIERPART($\mathcal{M} = (V, d)$)

1. Wähle β uniform zufällig aus dem Intervall $[1, 2]$.
2. Wähle eine uniform zufällige Permutation π der Menge $\{1, \ldots, N\}$.
3. $D_\delta = \{V\}$;
4. **for** $(i = \delta - 1;\ i \geq 0;\ i{-}{-})$
5. **if** (D_{i+1} besitzt eine Klasse mit mehr als einem Element)
6. $\beta_i = 2^{i-1}\beta$;
7. $D_i =$PARTITION$(\mathcal{M}, D_{i+1}, \beta_i, \pi)$;
8. **else**
9. $D_i = D_{i+1}$;
10. **return** $D_0, D_1, \ldots, D_\delta$;

Da die Funktion PARTITION bei jedem Aufruf in Zeile 7 eine Partition von V mit Radius $\beta_i = 2^{i-1}\beta \leq 2^i$ berechnet, die die Partition D_{i+1} verfeinert, handelt es sich bei $D_0, D_1, \ldots, D_\delta$ um eine hierarchische Partition gemäß Definition 4.4.

PARTITION($\mathcal{M}, D, \alpha, \pi$)

1. $D' = \{\}$;
2. **for each** (Klasse X in Partition D)
3. **for** $(i = 1,\ i \leq N,\ i{+}{+})$
4. $B_{\pi(i)} := \mathcal{B}(X, v_{\pi(i)}, \alpha)$;
5. $X = X \setminus B_{\pi(i)}$;
6. **if** $(B_{\pi(i)} \neq \emptyset)$ { Füge $B_{\pi(i)}$ zu D' hinzu. }
7. **return** D';

PARTITION betrachtet die Klassen der übergebenen Partition D einzeln. Jede solche Klasse X wird gegebenenfalls in mehrere Klassen zerlegt. Dazu werden die Knoten

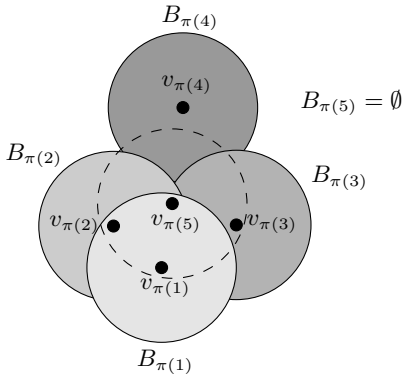


Abbildung 4.2: Beispiel für die Mengen B_i aus PARTITION: Man sieht, dass die Zentren, um die herum in Zeile 4 die Kugeln gebildet werden, im Allgemeinen nicht zu den entsprechenden Klassen gehören. Das ist in diesem Beispiel für $v_{\pi(2)}$ der Fall.

von V in der von der Permutation π bestimmten Reihenfolge betrachtet. Die erste Klasse besteht aus allen Punkten aus X, die einen Abstand von höchstens α zu $v_{\pi(1)}$ haben. Diese Punkte werden dann von der weiteren Betrachtung ausgeschlossen. Die zweite Klasse besteht aus allen übrig gebliebenen Punkten aus X, die einen Abstand von höchstens α zu $v_{\pi(2)}$ haben, und so weiter. Die Funktion PARTITION ist in Abbildung 4.2 an einem Beispiel veranschaulicht.

4.1.3 Analyse des Streckungsfaktors

Theorem 4.2 folgt direkt aus dem folgenden Lemma.

Lemma 4.6. *Es sei d_T die Baummetrik, die sich aus der hierarchischen Partition ergibt, die der Algorithmus* HIERPART($\mathcal{M} = (V, d)$) *ausgibt. Für jedes Paar $x \in V$ und $y \in V$ gilt*

$$\mathbf{E}\left[d_T(x, y)\right] \leq 64 H_N \cdot d(x, y),$$

wobei H_N die N-te harmonische Zahl bezeichnet.

Beweis. Es seien $x, y \in V$ beliebig. Wir betrachten in dem Baum T, der sich aus der hierarchischen Partition $D_0, \ldots, D_\delta$ ergibt, die der Algorithmus HIERPART($\mathcal{M} = (V, d)$) ausgibt, den Weg zwischen den Blättern, die x und y repräsentieren. Uns interessiert insbesondere, bis zu welcher Ebene des Baumes dieser Weg hinaufführt, denn dies entscheidet über die Länge $d_T(x, y)$ des Weges. Entspricht diese Ebene der Partition D_i, so sind die Punkte x und y in den Partitionen $D_i, \ldots, D_\delta$ jeweils in derselben Klasse und in den Partitionen $D_0, \ldots, D_{i-1}$ getrennt. Wir sagen dann, dass die Trennung des Paares $\{x, y\}$ auf Ebene $i - 1$ erfolgt. Es seien $z_x \in V$ und $z_y \in V$ die Zentren, um die herum in Zeile 4 der Funktion PARTITION die Klassen der Partition D_{i-1} gebildet werden, die x bzw. y enthalten. Kommt z_x in der Permutation π vor z_y, so sagen wir, dass *der Punkt z_x das Paar $\{x, y\}$ auf Ebene $i - 1$ trennt.* Ansonsten sagen wir entsprechend, dass der Punkt z_y das Paar $\{x, y\}$ auf Ebene $i - 1$ trennt.

Für $z \in V$ und $j \in \{0, 1, \ldots, \delta - 1\}$ bezeichnen wir mit $A(z, j)$ das Ereignis, dass der Punkt z das Paar $\{x, y\}$ auf Ebene j trennt. Aus der obigen Definition ergibt sich

direkt, dass es genau einen Punkt $z \in V$ und eine Ebene $j \in \{0, 1, \ldots, \delta - 1\}$ gibt, für die das Ereignis $A(z, j)$ eintritt. Tritt das Ereignis $A(z, j)$ ein, so gilt

$$d_T(x, y) = 2 \sum_{i=1}^{j+1} 2^i \leq 2^{j+3},$$

da der Weg zwischen den Blättern, die x und y repräsentieren, bis zum Vaterknoten von z aus der Partition D_{j+1} läuft, und somit jeweils zwei Kanten mit den Gewichten $2^1, 2^2, \ldots, 2^{j+1}$ enthält.

Für den erwarteten Abstand zwischen x und y in der Baummetrik d_T gilt somit

$$\mathbf{E}\left[d_T(x, y)\right] \leq \sum_{z \in V} \sum_{j=0}^{\delta-1} 2^{j+3} \cdot \mathbf{Pr}\left[A(z, j)\right]. \tag{4.1}$$

Um die Wahrscheinlichkeit für das Ereignis $A(z, j)$ abzuschätzen, sortieren wir die Punkte aus V zunächst nach ihrem Abstand zu x und y. Für $z \in V$ definieren wir dazu $d(z, \{x, y\}) = \min\{d(z, x), d(z, y)\}$. Es sei $V = \{v_1, \ldots, v_N\}$ mit $d(v_1, \{x, y\}) \leq d(v_2, \{x, y\}) \leq \ldots \leq d(v_N, \{x, y\})$.

Lemma 4.7. *Für jeden Punkt $v_\ell \in V$ und jede Ebene $j \in \{0, 1, \ldots, \delta - 1\}$ gilt*

$$\mathbf{Pr}\left[A(v_\ell, j)\right] \leq \frac{d(x, y)}{\ell \cdot 2^{j-1}}.$$

Beweis. Sei ohne Beschränkung der Allgemeinheit $d(v_\ell, x) \leq d(v_\ell, y)$. Damit der Knoten v_ℓ das Paar $\{x, y\}$ auf Ebene j trennen kann, müssen die folgenden beiden Bedingungen erfüllt sein.

1. Bei der Erstellung der Partition D_j ist die Kugel um v_ℓ, die in Zeile 4 der Funktion Partition gebildet wird, die erste Kugel, die x oder y enthält.
2. Der Radius $\beta_j = 2^{j-1}\beta$ liegt in dem Intervall $[d(v_\ell, x), d(v_\ell, y))$, da die Kugel um v_ℓ ansonsten entweder keinen der Punkte x und y oder beide enthält.

Die Wahrscheinlichkeit für die zweite Bedingung können wir leicht abschätzen:

$$\begin{aligned}\mathbf{Pr}\left[\beta_j \in [d(v_\ell, x), d(v_\ell, y))\right] &= \mathbf{Pr}\left[\beta \in \left[\frac{d(v_\ell, x)}{2^{j-1}}, \frac{d(v_\ell, y)}{2^{j-1}}\right)\right] \\ &\leq \frac{d(v_\ell, y)}{2^{j-1}} - \frac{d(v_\ell, x)}{2^{j-1}} \leq \frac{d(x, y)}{2^{j-1}}.\end{aligned}$$

Für die erste Ungleichung haben wir ausgenutzt, dass β uniform zufällig aus dem Intervall $[1, 2]$ gewählt wird. Das bedeutet, für ein beliebiges Intervall I beträgt die Wahrscheinlichkeit für das Ereignis $\beta \in I$ genau $|I \cap [1, 2]|$. Insbesondere ist diese Wahrscheinlichkeit für jedes Intervall $[a, b)$ durch $b - a$ nach oben beschränkt. Die zweite Ungleichung folgt direkt aus der Dreiecksungleichung.

Liegt β_j in dem Intervall, das in der zweiten Bedingung gegeben ist, so kann die erste Bedingung nur dann eintreten, wenn der Knoten v_ℓ in der Permutation π vor allen Knoten $v_1, \ldots, v_{\ell-1}$ kommt. Der Grund hierfür ist, dass die Kugeln mit Radius β_j um diese

Knoten herum alle mindestens einen der Punkte x und y enthalten. Die Wahrscheinlichkeit, dass der Knoten v_ℓ von allen Knoten aus der Menge $\{v_1, \ldots, v_\ell\}$ derjenige ist, der in der Permutation π zuerst kommt, beträgt aus Symmetriegründen genau $1/\ell$. Da die Permutation π unabhängig von β gewählt wird, können wir die Wahrscheinlichkeit, dass beide Bedingungen eintreten, durch $\frac{d(x,y)}{\ell \cdot 2^{j-1}}$ nach oben abschätzen. □

Mithilfe von Lemma 4.7 und der Formel (4.1) erhalten wir die Abschätzung

$$\mathbf{E}\left[d_T(x,y)\right] \leq \sum_{\ell=1}^{N} \sum_{j=0}^{\delta-1} 2^{j+3} \cdot \mathbf{Pr}\left[A(v_\ell, j)\right] \leq \sum_{\ell=1}^{N} \sum_{j=0}^{\delta-1} 2^{j+3} \cdot \frac{d(x,y)}{\ell \cdot 2^{j-1}} = 16\delta H_N \cdot d(x,y). \quad (4.2)$$

Dies zeigt, dass der erwartete Streckungsfaktor der Einbettung durch $16\delta H_N$ beschränkt ist. Wir argumentieren nun, dass der Term δ durch 4 ersetzt werden kann.

Lemma 4.8. *Für jeden Knoten v_ℓ gibt es maximal vier Ebenen $j \in \{0, 1, \ldots, \delta - 1\}$, für die es Realisierungen von β und π gibt, für die das Ereignis $A(v_\ell, j)$ eintritt.*

Beweis. Sei ohne Beschränkung der Allgemeinheit $d(v_\ell, x) \leq d(v_\ell, y)$.

- Wir betrachten zunächst den Fall, dass $d(x,y) \leq d(v_\ell, x)$ gilt. Mit dieser Ungleichung und der Dreiecksungleichung folgt

  $$d(v_\ell, x) \geq d(v_\ell, y) - d(x,y) \geq d(v_\ell, y) - d(v_\ell, x),$$

  was $d(v_\ell, x) \geq d(v_\ell, y)/2$ impliziert.

  Es sei j der größte Wert aus der Menge $\{0, 1, \ldots, \delta - 1\}$, für den das Intervall $[2^{j-1}, 2^j]$, aus dem β_j gewählt wird, einen nichtleeren Schnitt mit dem Intervall $[d(v_\ell, x), d(v_\ell, y))$ besitzt, in das β_j fallen muss, damit das Ereignis $A(v_\ell, j)$ eintreten kann. Dann gilt insbesondere $d(v_\ell, y) > 2^{j-1}$ und damit

  $$d(v_\ell, x) \geq \frac{d(v_\ell, y)}{2} > 2^{j-2}.$$

  Das bedeutet, in der Partition D_{j-2} kann der Knoten v_ℓ das Paar $\{x, y\}$ nicht trennen, da die Kugel um v_ℓ herum weder x noch y enthält. Da wir j größtmöglich gewählt haben, folgt insgesamt, dass das Ereignis $A(v_\ell, i)$ nur für $i \in \{j-1, j\}$ eintreten kann.

- Nun betrachten wir noch den Fall, dass $d(x,y) > d(v_\ell, x)$ gilt. Mit dieser Ungleichung und der Dreiecksungleichung folgt

  $$d(x,y) \geq d(v_\ell, y) - d(v_\ell, x) > d(v_\ell, y) - d(x,y),$$

  was $d(x,y) > d(v_\ell, y)/2$ impliziert.

  Sei j so wie im ersten Fall gewählt. Dann gilt $d(v_\ell, y) > 2^{j-1}$ und damit

  $$d(x,y) > \frac{d(v_\ell, y)}{2} > 2^{j-2}.$$

Das bedeutet, in der Partition D_{j-3} gehören x und y zu verschiedenen Klassen, da in dieser Partition jede Klasse einen Durchmesser von höchstens 2^{j-2} besitzt. Somit kann der Knoten v_ℓ das Paar $\{x, y\}$ auf keiner Ebene i mit $i \leq j-4$ trennen. Da wir j größtmöglich gewählt haben, folgt insgesamt, dass das Ereignis $A(v_\ell, i)$ nur für $i \in \{j-3, j-2, j-1, j\}$ eintreten kann. □

Mithilfe von Lemma 4.8 können wir die Summe in Formel (4.2) besser abschätzen. Für jedes ℓ gibt es nämlich nur maximal vier Werte von j, für die $\mathbf{Pr}[A(v_\ell, j)] > 0$ gilt. Für jedes ℓ genügt es also, vier Summanden der inneren Summe zu betrachten. Dies ergibt

$$\mathbf{E}[d_T(x,y)] \leq \sum_{\ell=1}^{N} \sum_{j=0}^{\delta-1} 2^{j+3} \cdot \mathbf{Pr}[A(v_\ell, j)] \leq \sum_{\ell=1}^{N} 4 \cdot \frac{16 d(x,y)}{\ell} = 64 H_N \cdot d(x,y). \qquad \square$$

Wir haben nun insgesamt gezeigt, dass sich jede Metrik mit einem Streckungsfaktor von $O(\log N)$ in Baummetriken einbetten lässt. Für die Anwendung dieses Ergebnisses auf das k-Server-Problem halten wir noch eine Beobachtung über den Durchmesser der Baummetriken fest, die von dem oben beschriebenen Algorithmus erzeugt werden.

Beobachtung 4.9. *Für jede Baummetrik $\mathcal{M}_T = (V_T, d_T)$ aus der Menge $\mathcal{S}$ aus Theorem 4.2 gilt*

$$\max_{x,y \in V_T} d_T(x,y) \leq 8 \cdot \max_{x,y \in V} d(x,y).$$

Beweis. Wir haben $\Delta = \max_{x,y \in V} d(x,y)$ definiert und $\delta \in \mathbb{N}$ so gewählt, dass $2^{\delta-1} < \Delta \leq 2^\delta$ gilt. In jeder Baummetrik, die einer hierarchischen Partition $D_0, D_1, \ldots, D_\delta$ der Metrik $\mathcal{M}$ entspricht, beträgt die Länge des längsten Weges $2 \sum_{i=1}^{\delta} 2^i \leq 2^{\delta+2} < 8\Delta$. □

4.2 Anwendung auf das k-Server-Problem

Wie bereits oben angekündigt, ergibt sich aus Theorem 4.2 das folgende Ergebnis.

Theorem 4.10. *Es gibt einen randomisierten Online-Algorithmus für das k-Server-Problem, der für jede Metrik mit N Punkten $O(k \cdot \log N)$-kompetitiv ist.*

Beweis. Sei $\mathcal{M} = (M, d)$ eine beliebige Metrik, für die das k-Server-Problem gelöst werden soll, und sei $(\mathcal{S}, \mathcal{D})$ die $O(\log N)$-Approximation von $\mathcal{M}$ aus Theorem 4.2. Der Algorithmus, den wir entwerfen, nutzt vor der ersten Anfrage den randomisierten Algorithmus aus Theorem 4.2, um eine zufällige Baummetrik $\mathcal{M}_T = (M_T, d_T)$ aus $\mathcal{S}$ gemäß der Verteilung $\mathcal{D}$ auszuwählen, und interpretiert die Eingabe σ für $\mathcal{M}$ als Eingabe für $\mathcal{M}_T$. Dies ist möglich, da $M \subseteq M_T$ gilt. Auf die Eingabe σ und die Metrik $\mathcal{M}_T$ wendet er nun den DC-Algorithmus an, der gemäß Theorem 3.9 einen kompetitiven Faktor von k auf der Baummetrik $\mathcal{M}_T$ erreicht.

Wir fixieren einen beliebigen optimalen Offline-Algorithmus für das k-Server-Problem und bezeichnen mit $\mathrm{OPT}(\sigma)$ und $\mathrm{OPT}_T(\sigma)$ die Lösungen, die er für die Metriken $\mathcal{M}$

bzw. $\mathcal{M}_T$ bei Eingabe σ berechnet. Eine Lösung L für eine Eingabe σ ist eine Folge von Bewegungen der Server, um die Anfragen in σ abzuarbeiten. Wir bezeichnen mit $d(L)$ die Kosten, die diese Serverbewegungen insgesamt erzeugen, wenn die Abstände auf M durch die Metrik d gegeben sind. Analog bezeichnen wir mit $d_T(L)$ die Kosten, die die Serverbewegungen in der Lösung L insgesamt erzeugen, wenn die Abstände auf M durch die Metrik d_T gegeben sind. Mit diesen Notationen können wir die Kosten eines optimalen Offline-Algorithmus auf Eingabe σ bezüglich der Metriken $\mathcal{M}$ und $\mathcal{M}_T$ als

$$d(\mathrm{OPT}(\sigma)) \quad \text{bzw.} \quad d_T(\mathrm{OPT}_T(\sigma))$$

schreiben. Außerdem folgt aus der jeweiligen Optimalität von $\mathrm{OPT}(\sigma)$ und $\mathrm{OPT}_T(\sigma)$, dass

$$d(\mathrm{OPT}(\sigma)) \leq d(\mathrm{OPT}_T(\sigma)) \quad \text{und} \quad d_T(\mathrm{OPT}_T(\sigma)) \leq d_T(\mathrm{OPT}(\sigma)) \tag{4.3}$$

gilt.

Da $\mathcal{M}_T$ in unserem Algorithmus eine zufällige Baummetrik ist, handelt es sich bei den Abständen $d_T(x, y)$ für $x, y \in M$ um Zufallsvariablen. Ebenso ist $\mathrm{OPT}_T(\sigma)$ eine zufällige Lösung der Eingabe σ. Theorem 4.2 garantiert, dass

$$d(x, y) \leq d_T(x, y) \quad \text{und} \quad \mathbf{E}\left[d_T(x, y)\right] \leq O(\log N) \cdot d(x, y) \tag{4.4}$$

für alle $x, y \in M$ gilt. Während die zweite Ungleichung nur im Erwartungswert gilt, gilt die erste Ungleichung für jede Wahl von $\mathcal{M}_T$, da jede Metrik aus $\mathcal{S}$ die Metrik $\mathcal{M}$ dominiert.

Wenden wir den DC-Algorithmus bei der Eingabe σ auf eine Baummetrik $\mathcal{M}_T$ an, so garantieren Theorem 3.9 und Beobachtung 4.9, dass wir eine Lösung $DC_T(\sigma)$ mit einem Wert $d_T(DC_T(\sigma)) \leq k \cdot d_T(\mathrm{OPT}_T(\sigma)) + \tau$ erhalten, wobei $\tau = 2k^2 \cdot 8 \max_{x,y \in M} d(x, y)$ eine Konstante ist, die von k und der Metrik $\mathcal{M}$ abhängt, nicht jedoch von σ oder $\mathcal{M}_T$. Insbesondere gilt

$$\mathbf{E}\left[d_T(DC_T(\sigma))\right] \leq \mathbf{E}\left[k \cdot d_T(\mathrm{OPT}_T(\sigma)) + \tau\right] = k \cdot \mathbf{E}\left[d_T(\mathrm{OPT}_T(\sigma))\right] + \tau,$$

wobei sich der Erwartungswert auf die zufällig gewählte Baummetrik $\mathcal{M}_T$ bezieht. Der Erwartungswert von $d_T(\mathrm{OPT}_T(\sigma))$ ist nicht einfach zu analysieren, denn die zufällige Metrik bestimmt sowohl die Abstände d_T als auch die Lösung $\mathrm{OPT}_T(\sigma)$. Die Abhängigkeiten zwischen diesen beiden zufälligen Größen machen eine Analyse des Erwartungswertes kompliziert. Zum Glück können wir aber auf die zweite Ungleichung von (4.3) zurückgreifen, die für jede Baummetrik $\mathcal{M}_T$ gilt. Damit folgt

$$\mathbf{E}\left[d_T(\mathrm{OPT}_T(\sigma))\right] \leq \mathbf{E}\left[d_T(\mathrm{OPT}(\sigma))\right].$$

Der Vorteil dieser Abschätzung ist, dass $\mathrm{OPT}(\sigma)$ eine feste Lösung ist, die nicht von der zufälligen Wahl von $\mathcal{M}_T$ abhängt. Wir können die Lösung $\mathrm{OPT}(\sigma)$ als eine Folge von Serverbewegungen $(x_1, y_1), \ldots, (x_\ell, y_\ell)$ mit $x_i, y_i \in M$ darstellen. Es gilt dann

$$\mathbf{E}\left[d_T(\mathrm{OPT}(\sigma))\right] = \mathbf{E}\left[\sum_{i=1}^{\ell} d_T(x_i, y_i)\right] = \sum_{i=1}^{\ell} \mathbf{E}\left[d_T(x_i, y_i)\right]$$

$$\leq O(\log N) \cdot \sum_{i=1}^{\ell} d(x_i, y_i) = O(\log N) \cdot d(\mathrm{OPT}(\sigma)),$$

wobei wir für die Ungleichung den zweiten Teil von (4.4) ausgenutzt haben.

Kombiniert man die letzten drei Ungleichungen, so erhält man

$$\mathbf{E}\left[d_T(DC_T(\sigma))\right] \leq k \cdot O(\log N) \cdot d(\mathrm{OPT}(\sigma)) + \tau.$$

Genauso wie oben ist es nicht einfach, den Erwartungswert von $d_T(DC_T(\sigma))$ zu analysieren, da der Zufall sowohl die Abstände d_T als auch die Lösung $DC_T(\sigma)$ beeinflusst. Mit der ersten Ungleichung von (4.4) folgt jedoch

$$\mathbf{E}\left[d(DC_T(\sigma))\right] \leq \mathbf{E}\left[d_T(DC_T(\sigma))\right]$$

und damit gilt insgesamt

$$\mathbf{E}\left[d(DC_T(\sigma))\right] \leq O(k \cdot \log N) \cdot d(\mathrm{OPT}(\sigma)) + \tau.$$

Damit ist der Beweis abgeschlossen, denn wir haben gezeigt, dass unser Algorithmus eine zufällige Lösung $DC_T(\sigma)$ berechnet, deren erwartete Kosten (gemessen in der Metrik $\mathcal{M}$) bis auf einen konstanten additiven Term höchstens um einen Faktor von $O(k \cdot \log N)$ größer sind als die Kosten einer optimalen Lösung. □

Als Übung sei es dem Leser empfohlen, der Frage nachzugehen, warum es wichtig ist, dass jede Baummetrik aus $\mathcal{S}$ die Metrik $\mathcal{M}$ dominiert. Es erscheint naheliegend, diese Einschränkung aufzugeben und statt der Bedingung (4.4) nur

$$d(x, y) \leq \mathbf{E}\left[d_T(x, y)\right] \leq O(\log N) \cdot d(x, y)$$

für alle $x, y \in M$ zu fordern. Der Leser sollte sich überlegen, an welcher Stelle der obige Beweis zusammenbricht, wenn wir (4.4) durch diese abgeschwächte Version ersetzen. Außerdem sollte er sich überlegen, dass diese Aussage tatsächlich zu schwach ist, um Theorem 4.10 zu zeigen.

Der oben beschriebene randomisierte Algorithmus erreicht einen schlechteren kompetitiven Faktor als der beste bekannte deterministische Algorithmus für das k-Server-Problem auf allgemeinen Metriken. Wir haben es dennoch vorgezogen, den randomisierten Algorithmus zu präsentieren, da Approximationen von Metriken durch Baummetriken auch für andere Probleme eine große Rolle spielen und ein wichtiges Hilfsmittel beim Entwurf von Online- und Approximationsalgorithmen sind.

Kapitel 5

Online-Probleme beim Handel

In diesem Kapitel beschäftigen wir uns mit Problemen, die beim Kaufen und Verkaufen von Aktien, Gütern und Devisen entstehen. Diese Fragestellungen eignen sich besonders gut dafür, als Online-Probleme modelliert zu werden, da man in der Regel Entscheidungen treffen muss, ohne beispielsweise die Entwicklung der Aktien- oder Wechselkurse in der Zukunft zu kennen.

5.1 Online-Suche und One-Way-Trading

Zunächst betrachten wir zwei einfache Probleme, die auftreten, wenn man Kapital von einer Anlageform in eine andere überführen möchte. Konkret gehen wir davon aus, dass wir eine bestimmte Menge Euro besitzen, die wir in Dollar tauschen möchten. In der ersten Variante dieses Problems, die wir *Online-Suche* nennen, betrachten wir jeden Morgen einmal den aktuellen Wechselkurs und entscheiden, ob wir diesen akzeptieren, d. h. alle unsere Euro zu diesem Kurs in Dollar tauschen, oder ob wir auf den nächsten Tag warten. Der Name Online-Suche rührt daher, dass wir bei diesem Problem einen möglichst guten Wechselkurs suchen, zu dem wir unser komplettes Kapital tauschen. In der zweiten Variante, die wir *One-Way-Trading* nennen, betrachten wir ebenfalls jeden Morgen einmal den aktuellen Wechselkurs und entscheiden, wie viele unserer Euro wir zu diesem Kurs in Dollar tauschen. Wir gehen der Einfachheit halber davon aus, dass beliebige reelle Beträge getauscht werden dürfen. Der Name One-Way-Trading rührt daher, dass wir nur in eine Richtung von Euro in Dollar tauschen und nicht umgekehrt.

Formal besteht eine Eingabe σ für eine der beiden Problemvarianten aus einer Folge $p_1, p_2, \ldots, p_n \in \mathbb{R}_{\geq 0}$ von Wechselkursen. Ein Online-Algorithmus muss dann zu jedem Zeitpunkt i basierend auf der Kenntnis des aktuellen und aller vergangenen Wechselkurse entscheiden, ob bzw. wie viele Euro in Dollar getauscht werden. Wir normieren das initiale Vermögen so, dass es ein Euro beträgt, und wir bezeichnen mit $x_i \in [0, 1]$ die Menge an Euro, die zum Zeitpunkt i zum Wechselkurs p_i in Dollar getauscht wird. Der Algorithmus muss die Werte x_i so wählen, dass $\sum_{i=1}^{n} x_i \leq 1$ gilt, und im Falle von Online-Suche muss zusätzlich $x_i \in \{0, 1\}$ gelten, d. h. es gibt höchstens ein i mit $x_i > 0$ und für dieses i gilt dann sogar $x_i = 1$. Die Anzahl Dollar, die der

Algorithmus am Ende erzielt, nennen wir seinen Gewinn und sie lässt sich als $\sum_{i=1}^{n} p_i x_i$ schreiben.

Das Ziel ist es, den Gewinn zu maximieren. Damit passen Online-Suche und One-Way-Trading nicht zu Definition 1.1, in der der kompetitive Faktor nur für Minimierungsprobleme definiert wird. Wir können die Definition aber kanonisch anpassen. Für einen Algorithmus A und eine Eingabe σ bezeichnen wir mit $p_A(\sigma)$ den Gewinn des Algorithmus A bei Eingabe σ und mit $\mathrm{OPT}(\sigma)$ bezeichnen wir den Gewinn eines optimalen Offline-Algorithmus bei Eingabe σ. Analog zu Definition 1.1 sagen wir, dass Algorithmus A einen kompetitiven Faktor von r erreicht, wenn es eine Konstante τ gibt, sodass

$$\mathrm{OPT}(\sigma) \leq r \cdot p_A(\sigma) + \tau$$

für alle Eingaben σ gilt. Gilt diese Ungleichung sogar für $\tau = 0$, so nennen wir Algorithmus A strikt r-kompetitiv. Bei randomisierten Algorithmen ersetzen wir in der obigen Ungleichung den Gewinn $p_A(\sigma)$ wieder durch den erwarteten Gewinn $\mathbf{E}[p_A(\sigma)]$.

5.1.1 Zusammenhang der beiden Problemvarianten

Auf den ersten Blick erscheinen Online-Suche und One-Way-Trading recht unterschiedliche Probleme zu sein. Tatsächlich kann man aber Algorithmen für diese beiden Problemvarianten zueinander in Beziehung setzen.

Theorem 5.1. *a) Zu jedem randomisierten Online-Algorithmus A_R für One-Way-Trading existiert ein deterministischer Online-Algorithmus A_D für One-Way-Trading, für den $p_{A_D}(\sigma) = \mathbf{E}[p_{A_R}(\sigma)]$ für alle Eingaben σ gilt.*

b) Für jeden deterministischen Online-Algorithmus A_D für One-Way-Trading existiert ein randomisierter Online-Algorithmus A_R für Online-Suche, für den $p_{A_D}(\sigma) = \mathbf{E}[p_{A_R}(\sigma)]$ für alle Eingaben σ gilt.

Der erste Teil des Theorems gilt insbesondere für randomisierte Algorithmen zur Online-Suche. Jeder solche Algorithmus kann also in einen deterministischen Algorithmus für One-Way-Trading mit demselben Gewinn überführt werden.

Beweis. Es sei $\sigma = (p_1, \ldots, p_n)$ eine beliebige Eingabe. Sei für Teil a) ein beliebiger randomisierter Online-Algorithmus A_R für One-Way-Trading gegeben. Wir bezeichnen mit X_i die Zufallsvariable, die angibt, wie viele Euro Algorithmus A_R im i-ten Schritt in Dollar tauscht. Der deterministische Algorithmus A_D tauscht im i-ten Schritt genau $\mathbf{E}[X_i]$ Euro in Dollar. Zunächst müssen wir argumentieren, dass dies möglich ist. Es gilt $\mathbf{E}[X_i] \in [0,1]$, denn wäre $\mathbf{E}[X_i] > 1$ so wäre auch $\mathbf{Pr}[X_i > 1] > 0$ und somit wäre bereits A_R kein zulässiger Algorithmus für One-Way-Trading. Ebenso gilt $\sum_{i=1}^{n} \mathbf{E}[X_i] \in [0,1]$, denn aus $\sum_{i=1}^{n} \mathbf{E}[X_i] > 1$ folgt $\mathbf{E}[\sum_{i=1}^{n} X_i] > 1$ und damit $\mathbf{Pr}[\sum_{i=1}^{n} X_i > 1] > 0$, was wiederum im Widerspruch dazu steht, dass A_R ein gültiger Algorithmus für One-Way-Trading ist. Es gilt außerdem

$$p_{A_D}(\sigma) = \sum_{i=1}^{n} p_i \mathbf{E}[X_i] = \mathbf{E}\left[\sum_{i=1}^{n} p_i X_i\right] = \mathbf{E}[p_{A_R}(\sigma)].$$

Sei für Teil b) ein beliebiger deterministischer Online-Algorithmus A_D für One-Way-Trading gegeben. Wir bezeichnen mit x_i den Betrag, den A_D im i-ten Schritt tauscht, und wir konstruieren einen randomisierten Algorithmus A_R mit der folgenden Eigenschaft: Für jedes i beträgt die Wahrscheinlichkeit, dass der Algorithmus A_R die ersten $i-1$ Wechselkurse nicht akzeptiert und den i-ten Wechselkurs akzeptiert, genau x_i. Wir bezeichnen mit y_i die Wahrscheinlichkeit, mit der A_R den i-ten Wechselkurs akzeptiert, wenn er keinen der ersten $i-1$ Wechselkurse akzeptiert hat. Dann lässt sich die oben genannte Eigenschaft in die Gleichungen $y_1 = x_1$ und

$$(1-y_1)\cdot\ldots\cdot(1-y_{i-1})\cdot y_i = x_i$$

für alle $i \in \{2,\ldots,n\}$ übersetzen. Man rechnet leicht nach, dass

$$y_i = \frac{x_i}{1-(x_1+\ldots+x_{i-1})}$$

für $i \in \{1,\ldots,n\}$ eine geeignete Wahl ist und dass für diese Wahl $y_i \in [0,1]$ gilt. Gilt $x_1+\ldots+x_{i-1}=1$, so folgt $x_i = 0$ und wir setzen $y_i = 0$. Somit handelt es sich bei A_R um einen wohldefinierten randomisierten Algorithmus.

Es sei $X_i \in \{0,1\}$ die Zufallsvariable, die angibt, wie viel der Algorithmus im i-ten Schritt tauscht. Es gilt

$$\mathbf{E}\left[p_{A_R}(\sigma)\right] = \mathbf{E}\left[\sum_{i=1}^{n} p_i X_i\right] = \sum_{i=1}^{n} p_i \mathbf{E}\left[X_i\right] = \sum_{i=1}^{n} p_i \mathbf{Pr}\left[X_i = 1\right] = \sum_{i=1}^{n} p_i x_i = p_{A_D}(\sigma). \quad \square$$

Aus dem vorangegangenen Theorem folgt, dass Randomisierung bei One-Way-Trading den kompetitiven Faktor nicht verbessern kann. Wir werden sehen, dass das bei Online-Suche anders ist und Randomisierung dort sehr wohl helfen kann, einen besseren kompetitiven Faktor zu erreichen.

5.1.2 Algorithmen für Online-Suche

Um bei Online-Suche und One-Way-Trading deterministisch einen endlichen kompetitiven Faktor erreichen zu können, müssen gewisse Informationen über die Eingabe von Anfang an vorliegen. Deshalb werden wir unter anderem Szenarien betrachten, in denen alle Wechselkurse in einem bekannten Intervall $[m, M]$ liegen. Außerdem werden wir Szenarien betrachten, in denen man statt m und M nur $\varphi = M/m$ kennt. An einigen Stellen gehen wir auch davon aus, dass die Länge n der Eingabe σ von Anfang an bekannt ist. Gibt es ein bekanntes Intervall $[m, M]$, aus dem die Wechselkurse stammen, so erlauben wir den Algorithmen nach dem letzten Schritt eventuell übrig gebliebene Euro zum Kurs m in Dollar zu tauschen.

Eine naheliegende Klasse von deterministischen Algorithmen für Online-Suche sind *Reservationspreisalgorithmen*, welche den ersten Wechselkurs oberhalb eines bestimmten Schwellenwerts akzeptieren. Der Reservationspreisalgorithmus RPA(z) akzeptiert den ersten Wechselkurs von mindestens z. Gibt es keinen solchen Wechselkurs in der Eingabe, so akzeptiert er keinen Wechselkurs und muss nach dem letzten Schritt den Wechselkurs m akzeptieren.

Theorem 5.2. *Der Algorithmus* RPA($p^\star$) *mit* $p^\star = \sqrt{Mm}$ *ist strikt* $\sqrt{\varphi}$*-kompetitiv.*

Beweis. Es sei $\sigma = (p_1, \ldots, p_n)$ eine beliebige Eingabe und es sei $p_{\max} = \max_i p_i$. Dann gilt $\mathrm{OPT}(\sigma) = p_{\max}$. Ist $p_{\max} < p^\star$, so gilt $p_{RPA(p^\star)}(\sigma) = m$, da RPA($p^\star$) keinen Wechselkurs akzeptiert und nach dem letzten Schritt zum Kurs m tauschen muss. In diesem Fall gilt also

$$\frac{\mathrm{OPT}(\sigma)}{p_{RPA(p^\star)}(\sigma)} < \frac{p^\star}{m} = \sqrt{\varphi}.$$

Ist $p_{\max} \geq p^\star$, so gilt $p_{RPA}(\sigma) \geq p^\star$, da RPA($p^\star$) den ersten Wechselkurs von mindestens $p^\star$ akzeptiert. In diesem Fall gilt also

$$\frac{\mathrm{OPT}(\sigma)}{p_{RPA(p^\star)}(\sigma)} \leq \frac{M}{p^\star} = \sqrt{\varphi}.$$

Damit ist gezeigt, dass der optimale Gewinn stets um höchstens einen Faktor von $\sqrt{\varphi}$ größer ist als der Gewinn von RPA($p^\star$). □

Der Leser sollte sich als Übung überlegen, dass es keinen deterministischen Algorithmus für Online-Suche gibt, der einen besseren strikten kompetitiven Faktor erreicht. Außerdem ist es eine Übung, zu zeigen, dass kein deterministischer Algorithmus einen strikt kompetitiven Faktor kleiner als den trivialen Faktor φ erreichen kann, wenn nur φ, nicht aber m und M bekannt sind.

Mit Randomisierung kann man einen deutlich besseren kompetitiven Faktor erreichen. Zunächst betrachten wir wieder das Szenario, dass m und M bekannt sind. Der Einfachheit halber nehmen wir an, dass $\varphi = M/m = 2^k$ für ein $k \in \mathbb{N}$ gilt. Der randomisierte Algorithmus EXPO wählt i uniform zufällig aus der Menge $\{0, \ldots, k-1\}$ und wendet den deterministischen Algorithmus RPA($m2^i$) an. Im Rest dieses Kapitels steht log stets für den binären Logarithmus $\log_2$.

Theorem 5.3. *Der Algorithmus EXPO ist strikt* $(c(\varphi)\log\varphi)$*-kompetitiv. Dabei ist* c *eine Funktion, die für große* φ *gegen 1 konvergiert und für die* $c(\varphi) \leq 2$ *für alle* $\varphi = 2^k$ *mit* $k \in \mathbb{N}$ *gilt.*

Beweis. Es sei $\sigma = (p_1, \ldots, p_n)$ eine beliebige Eingabe und wieder sei $p_{\max} = \max_i p_i$. Dann gilt $\mathrm{OPT}(\sigma) = p_{\max}$. Gilt $p_{\max} = m2^k$, so sei $j = k-1$. Ansonsten sei $j \in \{0, 1, \ldots, k-1\}$ so gewählt, dass $m2^j \leq p_{\max} < m2^{j+1}$ gilt. Ein solches j existiert wegen $M = m2^k$. Jeder Algorithmus RPA($m2^i$) mit $i \in \{0, \ldots, k-1\}$ und $i > j$ akzeptiert bei der Eingabe σ keinen Preis und erreicht somit nur einen Gewinn von m. Jeder Algorithmus RPA($m2^i$) mit $i \leq j$ erreicht auf der Eingabe σ mindestens einen Gewinn von $m2^i$. Da i uniform aus der Menge $\{0, \ldots, k-1\}$ gewählt wird, ergibt sich

$$\begin{aligned}\mathbf{E}\left[p_{\mathrm{EXPO}}(\sigma)\right] &= \sum_{i=0}^{k-1} \frac{p_{\mathrm{RPA}(m2^i)}(\sigma)}{k} \geq \frac{k-1-j}{k} \cdot m + \sum_{i=0}^{j} \frac{m2^i}{k} \\ &= \frac{m}{k}\Big(k - 1 - j + \sum_{i=0}^{j} 2^i\Big) = \frac{m}{k}(k - j + 2^{j+1} - 2).\end{aligned}$$

Außerdem gilt $\mathrm{OPT}(\sigma) = p_{\max} \leq m2^{j+1}$. Für den kompetitiven Faktor ergibt sich somit

$$\frac{\mathrm{OPT}(\sigma)}{\mathbf{E}\left[p_{\mathrm{EXPO}}(\sigma)\right]} \leq k \cdot \frac{2^{j+1}}{2^{j+1}+k-j-2}.$$

Eine einfache Kurvendiskussion zeigt, dass die Funktion

$$R(j) := \frac{2^{j+1}}{2^{j+1}+k-j-2}$$

für $j^\star = k-2+\frac{1}{\ln(2)}$ ihr Maximum annimmt. Es gilt

$$R(j^\star) = \frac{2^{k-1}e\ln 2}{2^{k-1}e\ln 2 - 1}.$$

Dieser Term ist für $k \geq 2$ (d. h. für $\varphi \geq 4$) durch 1,37 nach oben beschränkt. Außerdem konvergiert er für große k (d. h. für große φ) gegen 1. Für $k = 1$ (d. h. für $\varphi = 2$) gilt stets $j = 0$ und $R(0) = 2$. □

Die Idee, die dem Algorithmus EXPO zugrunde liegt, ist einfach. Liegt der Reservationspreis z nur um einen konstanten Faktor unterhalb von $p_{\max}$, so erreicht der entsprechende Algorithmus $\mathrm{RPA}(z)$ einen konstanten kompetitiven Faktor. Egal wie $p_{\max}$ gewählt ist, es existiert stets ein $i \in \{0,\ldots,k-1\}$, sodass $m2^i$ nur um höchstens den Faktor zwei unterhalb von $p_{\max}$ liegt. Mit einer Wahrscheinlichkeit von $1/\log\varphi$ wird dieses i gewählt und der Algorithmus $\mathrm{RPA}(m2^i)$ angewendet. Mit diesem Argument folgt ein kompetitiver Faktor von $2\log\varphi$. Dieser Ansatz ist stark auf Maximierungsprobleme zugeschnitten. Läge ein Minimierungsproblem vor, so würde es nicht genügen, zu zeigen, dass für ein i ein konstanter kompetitiver Faktor erreicht wird. Man müsste zusätzlich noch zeigen, dass durch falsche Wahlen von i nicht zu hohe Kosten erzeugt werden.

Der Leser überlege sich als Übung, dass der Algorithmus EXPO so angepasst werden kann, dass er den in Theorem 5.3 behaupteten kompetitiven Faktor auch dann erreicht, wenn er nur φ, nicht aber m und M kennt. Dazu beobachtet der Algorithmus zunächst den ersten Wechselkurs p_1, wählt $i \in \{0,\ldots,k-1\}$ uniform zufällig und wendet den Algorithmus $\mathrm{RPA}(p_1 2^i)$ auf die Eingabe an. Das heißt insbesondere, dass er für $i = 0$ direkt den ersten Wechselkurs p_1 akzeptiert.

Wir können den Algorithmus sogar auf das Szenario verallgemeinern, dass weder das Intervall $[m, M]$ noch φ bekannt sind. Sei dazu eine monoton fallende Wahrscheinlichkeitsverteilung $q = \{q(i)\}_{i=0}^{\infty}$ auf den natürlichen Zahlen gegeben. Eine solche Wahrscheinlichkeitsverteilung ist eine Folge von reellen Zahlen mit $q(i) \in [0,1]$, $\sum_{i=0}^{\infty} q(i) = 1$ und $q(0) \geq q(1) \geq q(2) \geq \ldots$. Der Algorithmus EXPO_q wählt i zufällig gemäß der Verteilung q, beobachtet den ersten Kurs p_1 und wendet dann den Algorithmus $\mathrm{RPA}(p_1 2^i)$ auf die Eingabe an.

Theorem 5.4. *Der Algorithmus* EXPO_q *ist strikt* $\frac{2}{q(\lfloor\log\varphi\rfloor)}$*-kompetitiv.*

Beweis. Es sei $\sigma = (p_1, p_2, \ldots, p_n)$ eine beliebige Eingabe. Wieder sei $p_{\max} = \max_i p_i$ und es sei $j \in \{0, 1, \ldots\}$ so gewählt, dass $p_1 2^j \leq p_{\max} < p_1 2^{j+1}$ gilt. Wegen $p_1 \geq m$ und $p_{\max} \leq M$ gilt

$$j = \left\lfloor \log\left(\frac{p_{\max}}{p_1}\right) \right\rfloor \leq \lfloor \log \varphi \rfloor.$$

Der Algorithmus RPA($p_1 2^j$) erreicht auf der Eingabe σ einen Gewinn von mindestens $p_1 2^j \geq \frac{p_{\max}}{2}$ und er wird mit einer Wahrscheinlichkeit von $q(j)$ ausgewählt. Damit folgt

$$\mathbf{E}\left[p_{\mathrm{EXPO}_q}\right] \geq q(j) \cdot p_1 2^j \geq p_{\max} \cdot \frac{q(\lfloor \log \varphi \rfloor)}{2} = \mathrm{OPT}(\sigma) \cdot \frac{q(\lfloor \log \varphi \rfloor)}{2}.$$

In der zweiten Abschätzung haben wir die Monotonie der Wahrscheinlichkeitsverteilung ausgenutzt. □

Es ist wichtig, dass zwar der kompetitive Faktor in Theorem 5.4 von φ abhängt, dass dem Algorithmus der Parameter φ aber nicht bekannt sein muss. Um einen möglichst guten kompetitiven Faktor zu erreichen, suchen wir nun nach einer Folge q, die unter den gegebenen Randbedingungen so lange wie möglich so groß wie möglich bleibt. Da die Reihe $\sum_{i=0}^{\infty} \frac{1}{(i+1)^{1+\varepsilon}}$ für jedes $\varepsilon > 0$ gegen eine Konstante c_ε konvergiert, ist es naheliegend, $q(i) = \frac{1}{c_\varepsilon (i+1)^{1+\varepsilon}}$ für ein $\varepsilon > 0$ zu wählen. Damit erhält man für jedes $\varepsilon > 0$ einen $O(\log^{1+\varepsilon} \varphi)$-kompetitiven Algorithmus. Man beachte, dass das gewählte ε nicht nur den Exponenten, sondern auch die Konstante in der O-Notation beeinflusst.

Um einen noch besseren kompetitiven Faktor zu erreichen, betrachten wir die Reihe $\sum_{i=0}^{\infty} \frac{1}{(i+2)\log^{1+\varepsilon}(i+2)}$, die ebenfalls für jedes $\varepsilon > 0$ gegen eine Konstante c'_ε konvergiert. Dementsprechend können wir auch $q(i) = \frac{1}{c'_\varepsilon (i+2)\log^{1+\varepsilon}(i+2)}$ setzen und erhalten damit für jedes $\varepsilon > 0$ einen $O(\log \varphi \log^{1+\varepsilon} \log \varphi)$-kompetitiven Algorithmus.

5.1.3 Optimaler Online-Algorithmus für One-Way-Trading

Wir haben im vorangegangenen Abschnitt mit EXPO einen $O(\log \varphi)$-kompetitiven Algorithmus für die Fälle kennengelernt, dass das Intervall $[m, M]$ oder der Faktor $\varphi = M/m$ bekannt sind. Diesen Algorithmus haben wir als randomisierten Algorithmus für Online-Suche präsentiert, mit Theorem 5.1 können wir ihn aber auch als deterministischen Algorithmus für One-Way-Trading auffassen. Wir werden in diesem Abschnitt sehen, dass der kompetitive Faktor $O(\log \varphi)$ zwar asymptotisch optimal ist, dass es aber einen Algorithmus gibt, dessen kompetitiver Faktor um einen konstanten Faktor besser als der von EXPO ist.

Um genau zu sein, werden wir in diesem Abschnitt einen optimalen deterministischen Online-Algorithmus für One-Way-Trading präsentieren, der den bestmöglichen kompetitiven Faktor erreicht. Dann werden wir anschließend argumentieren, dass dieser kompetitive Faktor in der Größenordnung $\Theta(\log \varphi)$ liegt und um einen konstanten Faktor besser als der von EXPO ist. Dabei gehen wir davon aus, dass das Intervall $[m, M]$ bekannt ist.

Der Algorithmus, den wir betrachten, heißt *Drohungs-Algorithmus (Threat)*. Er bekommt als Parameter einen Wert $r \geq 1$ übergeben und versucht, einen strikten kompetitiven Faktor von r zu erreichen. Dazu verhält sich Threat(r) auf einer Eingabe $\sigma = (p_1, \ldots, p_n)$ im i-ten Schritt wie folgt.

1. Falls p_i nicht größer ist als der bisher beste Kurs $\max_{1 \leq j < i} p_j$, so tauscht der Algorithmus Threat(r) nichts.
2. Ansonsten tauscht Threat(r) den kleinstmöglichen Betrag zum Kurs p_i, der garantiert, dass er strikt r-kompetitiv auf der Eingabe $\sigma^i = (p_1, \ldots, p_i)$ ist, die nach p_i abbricht. Der Betrag, den Threat(r) nach Schritt i noch nicht getauscht hat, muss auf dieser Eingabe zum Kurs m getauscht werden. Gibt es keinen Betrag mit der gewünschten Eigenschaft, so gibt Threat(r) aus, dass es keinen Algorithmus für One-Way-Trading gibt, der einen strikten kompetitiven Faktor von r erreicht.

Bevor wir uns überlegen, wie dieser Algorithmus implementiert werden kann, beweisen wir eine wesentliche Eigenschaft.

Lemma 5.5. *Es sei σ eine beliebige Eingabe der Länge n für One-Way-Trading. Gibt es einen Online-Algorithmus, der auf jeder Eingabe σ^i mit $i \in \{1, \ldots, n\}$ strikt r-kompetitiv ist, so ist auch der Algorithmus* Threat(r) *strikt r-kompetitiv auf σ und gibt insbesondere keine Fehlermeldung auf dieser Eingabe aus.*

Beweis. Sei $\sigma = (p_1, \ldots, p_n)$ eine beliebige Eingabe und sei A ein beliebiger Online-Algorithmus für One-Way-Trading, der auf jeder Eingabe σ^i mit $i \in \{1, \ldots, n\}$ strikt r-kompetitiv ist Wegen Theorem 5.1 können wir ohne Beschränkung der Allgemeinheit davon ausgehen, dass A deterministisch ist. Außerdem können wir ebenfalls ohne Beschränkung der Allgemeinheit davon ausgehen, dass die Wechselkurse streng monoton steigen, dass also $p_1 < p_2 < \ldots < p_n$ gilt. Enthält die Eingabe nämlich einen Wechselkurs p_i, der nicht besser ist als der bisher beste Kurs, so ändern sich durch das Löschen von p_i aus der Eingabe weder der Wert der optimalen Lösung noch das Verhalten von Threat(r). Solange es also solche Wechselkurse gibt, können sie aus der Eingabe entfernt werden, ohne dass dadurch der kompetitive Faktor von Threat(r) beeinflusst wird.

Da Algorithmus A ein Online-Algorithmus ist, muss er sich im ersten Schritt bei den Eingaben σ und σ^1 identisch verhalten. Da er strikt r-kompetitiv ist, tauscht er einen Betrag x_1, der garantiert, dass er strikt r-kompetitiv auf der Eingabe σ^1 ist. Insbesondere gibt es also einen solchen Betrag und der Algorithmus Threat(r) gibt keine Fehlermeldung aus, sondern bestimmt den kleinsten Betrag y_1, der garantiert, dass er strikt r-kompetitiv auf der Eingabe σ^1 ist. Es gilt somit $y_1 \leq x_1$.

Wir betrachten nun einen Schritt $i \geq 2$ und gehen davon aus, dass $\sum_{j=1}^{\ell}(x_j - y_j) \geq 0$ für alle $\ell \in \{1, \ldots, i-1\}$ gilt. In Schritt i tauscht Algorithmus A einen Betrag x_i, der garantiert, dass seine bisherigen Aktionen $(x_1, \ldots, x_i)$ eine strikt r-kompetitive Lösung auf der Sequenz σ^i sind. Der Algorithmus Threat(r) hat bisher unter Umständen weniger getauscht als Algorithmus A. Würde er in diesem Schritt $y_i' = x_i + \sum_{j=1}^{i-1}(x_j - y_j)$

tauschen, so hätte er insgesamt die gleiche Menge Euro wie Algorithmus A getauscht und hätte

$$
\begin{aligned}
& y'_i p_i + \sum_{j=1}^{i-1} y_j p_j \\
&= \Big(x_i + \sum_{j=1}^{i-1} (x_j - y_j) \Big) p_i + \sum_{j=1}^{i-1} y_j p_j \\
&\geq x_i p_i + \sum_{j=1}^{i-1} (x_j - y_j) p_{i-1} + \sum_{j=1}^{i-1} y_j p_j \\
&= x_i p_i + \sum_{j=1}^{i-2} (x_j - y_j) p_{i-1} + (x_{i-1} - y_{i-1}) p_{i-1} + \sum_{j=1}^{i-1} y_j p_j \\
&\geq x_i p_i + \sum_{j=1}^{i-2} (x_j - y_j) p_{i-2} + (x_{i-1} - y_{i-1}) p_{i-1} + \sum_{j=1}^{i-1} y_j p_j \\
&= x_i p_i + \sum_{j=1}^{i-3} (x_j - y_j) p_{i-2} + (x_{i-2} - y_{i-2}) p_{i-2} + (x_{i-1} - y_{i-1}) p_{i-1} + \sum_{j=1}^{i-1} y_j p_j \\
&\geq x_i p_i + \sum_{j=1}^{i-3} (x_j - y_j) p_{i-3} + (x_{i-2} - y_{i-2}) p_{i-2} + (x_{i-1} - y_{i-1}) p_{i-1} + \sum_{j=1}^{i-1} y_j p_j \\
&\geq \ldots \geq x_i p_i + \sum_{j=1}^{i-1} (x_j - y_j) p_j + \sum_{j=1}^{i-1} y_j p_j \\
&= \sum_{j=1}^{i} x_j p_j
\end{aligned}
$$

Dollar erwirtschaftet. Bei den Ungleichungen haben wir die Monotonie der Wechselkurse und die Induktionsvoraussetzung ausgenutzt. Damit wäre der Gewinn von Threat(r) auf der Eingabe σ^i mindestens so groß wie der von A und damit wäre Threat(r) strikt r-kompetitiv auf dieser Eingabe.

Es ist für Threat(r) möglich, den Betrag y'_i zu tauschen, da $y'_i + \sum_{j=1}^{i-1} y_j = \sum_{j=1}^{i} x_j \leq 1$ gilt. Damit haben wir gezeigt, dass es einen Betrag gibt, der garantiert, dass Threat(r) strikt r-kompetitiv auf σ^i ist. Somit gibt Threat(r) in Schritt i keine Fehlermeldung aus. Außerdem ist die Induktionsvoraussetzung im nächsten Schritt wieder gegeben, denn $y_i \leq y'_i$ impliziert $\sum_{j=1}^{i} (x_j - y_j) \geq 0$.

Es folgt induktiv, dass Threat(r) auf der Eingabe σ zu keinem Zeitpunkt eine Fehlermeldung ausgibt. Außerdem folgt, dass er im letzten Schritt so viel tauscht, dass er auf der Sequenz $\sigma^n = \sigma$ strikt r-kompetitiv ist. □

Aus dem vorangegangenen Lemma folgt direkt das folgende Theorem.

Theorem 5.6. *Gibt es einen strikt r-kompetitiven Online-Algorithmus für One-Way-Trading, so ist auch der Algorithmus* Threat(r) *strikt r-kompetitiv. Insbesondere gibt der Algorithmus* Threat(r) *dann auf keiner Eingabe eine Fehlermeldung aus.*

Wegen des vorangegangenen Theorems können wir sagen, dass der optimale strikte kompetitive Faktor, der für One-Way-Trading erreicht werden kann, die kleinste Zahl r ist, für die der Algorithmus Threat(r) auf keiner Eingabe eine Fehlermeldung ausgibt. Wir werden diese abstrakte Beschreibung nun expliziter machen. Dazu überlegen wir uns zunächst, wie der Algorithmus Threat(r) implementiert werden kann.

Sei $\sigma = (p_1, \ldots, p_n)$ mit $p_1 < \ldots < p_n$ eine Eingabe für One-Way-Trading. Zusätzlich gehen wir davon aus, dass $p_1 \geq rm$ gilt. Diese Einschränkung werden wir unten erklären. Wir bezeichnen nun mit x_i den Betrag, den Threat(r) in Schritt i tauscht. Außerdem bezeichnen wir mit E_i und D_i den Betrag an Euro bzw. Dollar, die Threat(r) nach Schritt i besitzt. Um auf der Eingabe σ^i einen strikten kompetitiven Faktor von r zu erreichen, muss

$$\frac{\mathrm{OPT}(\sigma^i)}{p_{\mathrm{Threat}(r)}(\sigma^i)} = \frac{p_i}{D_i + mE_i} \leq r \tag{5.1}$$

für alle i gelten. Der Nenner $D_i + mE_i$ entspricht genau dem Gewinn von Threat(r) auf der Eingabe σ^i, denn nach Schritt i besitzt der Algorithmus D_i Dollar und er kann zu einem Kurs von m die verbleibenden E_i Euro in Dollar tauschen. Definieren wir $E_0 = 1$ und $D_0 = 0$, so können wir diesen Gewinn für $i \geq 1$ als

$$D_i + mE_i = D_{i-1} + x_i p_i + m(E_{i-1} - x_i) \tag{5.2}$$

schreiben. Kombiniert man (5.1) und (5.2), so erhält man

$$\frac{p_i}{x_i(p_i - m) + D_{i-1} + mE_{i-1}} \leq r.$$

Löst man diese Ungleichung nach x_i auf, so erhält man

$$x_i \geq \frac{p_i - r(D_{i-1} + mE_{i-1})}{r(p_i - m)}.$$

Da Threat(r) das kleinste nichtnegative x_i mit dieser Eigenschaft wählt, können wir davon ausgehen, dass

$$x_i = \max\left\{0, \frac{p_i - r(D_{i-1} + mE_{i-1})}{r(p_i - m)}\right\} \tag{5.3}$$

gilt. Für solche i, für die das Maximum von dem zweiten Term angenommen wird, gilt (5.1) mit Gleichheit. Aus $p_1 \geq rm$ folgt mit $E_0 = 1$ und $D_0 = 0$, dass dies für $i = 1$ der Fall ist. Es gilt also

$$x_1 = \frac{p_1 - rm}{r(p_1 - m)}. \tag{5.4}$$

Ist (5.1) für $i - 1$ mit Gleichheit erfüllt, so gilt

$$D_{i-1} + mE_{i-1} = \frac{p_{i-1}}{r}. \tag{5.5}$$

Daraus ergibt sich

$$\frac{p_i - r(D_{i-1} + mE_{i-1})}{r(p_i - m)} = \frac{p_i - r \cdot \frac{p_{i-1}}{r}}{r(p_i - m)} = \frac{p_i - p_{i-1}}{r(p_i - m)} \geq 0, \tag{5.6}$$

wobei wir im letzten Schritt die Monotonie der Wechselkurse ausgenutzt haben. Das bedeutet, auch für i ist (5.1) mit Gleichheit erfüllt. Somit folgt induktiv, dass auf der betrachteten Eingabe das Maximum in (5.3) immer vom zweiten Term angenommen wird.

Aus (5.3) folgt auch, wie sich Threat(r) auf Eingaben verhält, die mit einem Kurs $p_1 < rm$ beginnen. Solange der Kurs in einer solchen Eingabe unterhalb von rm liegt, kauft Threat(r) nichts. Für alle Kurse danach verhält er sich so wie oben durch (5.4) und (5.6) beschrieben. Der einzige Unterschied ist, dass durch (5.4) im Allgemeinen nicht x_1 sondern $x_{i(r)}$ mit $i(r) = \min\{i \mid p_i \geq rm\}$ bestimmt wird.

Damit ist das Verhalten von Threat(r) explizit beschrieben. Wir wissen jedoch noch immer nicht, wie groß r mindestens sein muss, damit keine Fehlermeldung erzeugt wird. Bei einer Eingabe mit Kursen $m \leq p_1 < \ldots < p_n \leq M$ wird genau dann eine Fehlermeldung erzeugt, wenn die Summe $\sum_{i=1}^{n} x_i$ der umgetauschten Euro größer als 1 ist. Der Algorithmus Threat(r) ist auf der betrachteten Eingabe für jedes r ein strikt r-kompetitiver Algorithmus, für das

$$\sum_{i=1}^{n} x_i = \frac{p_{i(r)} - rm}{r(p_{i(r)} - m)} + \sum_{i=i(r)+1}^{n} \frac{p_i - p_{i-1}}{r(p_i - m)} \leq 1 \tag{5.7}$$

gilt. Da die linke Seite dieser Ungleichung monoton mit r fällt, können wir das kleinste r, für das Threat(r) keine Fehlermeldung ausgibt, dadurch bestimmen, dass wir aus der Ungleichung eine Gleichung machen und diese nach r auflösen. Den Wert, der sich dadurch ergibt, nennen wir $r^\star(n, m, p_1, \ldots, p_n)$.

Natürlich können wir keinen Online-Algorithmus entwerfen, der zunächst $r^\star$ berechnet und dann Threat$(r^\star)$ anwendet, da zur Berechnung von $r^\star$ nicht nur m und n, sondern auch alle Kurse $p_1, \ldots, p_n$ bekannt sein müssen. Ein weiteres Problem ist, dass wir uns auf Eingaben mit streng monoton steigenden Kursen beschränkt haben. Wir können allgemeine Eingaben der Länge n handhaben, indem wir alle Kurse streichen, die nicht besser sind als der beste vorangegangene Kurs. In obiger Formel für $r^\star$ dürfen wir dann aber nicht die Länge n der ursprünglichen Eingabe zugrunde legen, sondern die Länge k der Eingabe, die nach den Streichungen übrig bleibt. Wir können jedoch für allgemeine Eingaben der Länge n allein basierend auf der Kenntnis der Eingabelänge n und des Intervalls $[m, M]$ die folgende Größe $r_n^\star(m, M)$ berechnen:

$$r_n^\star(m, M) = \max_{\substack{k \leq n \\ m \leq p_1 < \ldots < p_k \leq M}} r^\star(k, m, p_1, \ldots, p_k).$$

Das bedeutet, dass ein strikter kompetitiver Faktor von $r_n^\star(m, M)$ für allgemeine Eingaben erreicht werden kann, wenn das Intervall $[m, M]$ und die Eingabelänge n bekannt sind. Dazu berechnet man zunächst $r_n^\star(m, M)$ und wendet dann Threat$(r_n^\star(m, M))$ an. Es gibt keine kurze explizite Formel für $r_n^\star(m, M)$. Mit einigen Rechnungen kann man $r_n^\star(m, M)$ aber implizit als die eindeutige Lösung der Gleichung

$$r = n \cdot \left(1 - \left(\frac{m(r-1)}{M - m}\right)^{1/n}\right)$$

beschreiben.

Korollar 5.7. *In dem Szenario, dass das Intervall $[m, M]$ und die Eingabelänge n bekannt sind, ist* Threat$(r_n^\star(m, M))$ *ein optimaler Online-Algorithmus.*

Beweis. Wir müssen zeigen, dass es keinen Online-Algorithmus A gibt, der auf allen Eingaben der Länge n mit Wechselkursen aus $[m, M]$ einen besseren kompetitiven Faktor als $r_n^\star(m, M)$ erreicht. Angenommen, es gäbe einen Online-Algorithmus A, der einen strikten kompetitiven Faktor von $r < r_n^\star(m, M)$ auf allen Eingaben der Länge n mit Wechselkursen aus $[m, M]$ erreicht. Da $r < r_n^\star(m, M)$ gilt, gibt es eine Eingabe $\sigma = (p_1, \ldots, p_k)$ mit $rm \leq p_1 < \ldots < p_k$ und $k \leq n$, auf der der Algorithmus Threat(r) eine Fehlermeldung ausgibt. Der Algorithmus A ist aber laut Annahme auf σ und auf jedem Präfix von σ strikt r-kompetitiv. Damit ergibt sich ein Widerspruch zu Lemma 5.5. □

Die Größe $r_n^\star(m, M)$ ist monoton wachsend in der Länge n der Eingabe. Dies folgt direkt aus der Definition. Ist die Eingabelänge n nicht bekannt, so muss mit beliebig großen Eingabelängen gerechnet werden und es kann nur ein kompetitiver Faktor von

$$r_\infty^\star(m, M) = \lim_{n \to \infty} r_n^\star(m, M)$$

erreicht werden. Diese Größe kann man mit einigen Rechnungen implizit als die eindeutige Lösung der Gleichung

$$r = \ln\left(\frac{M - m}{m(r - 1)}\right) \tag{5.8}$$

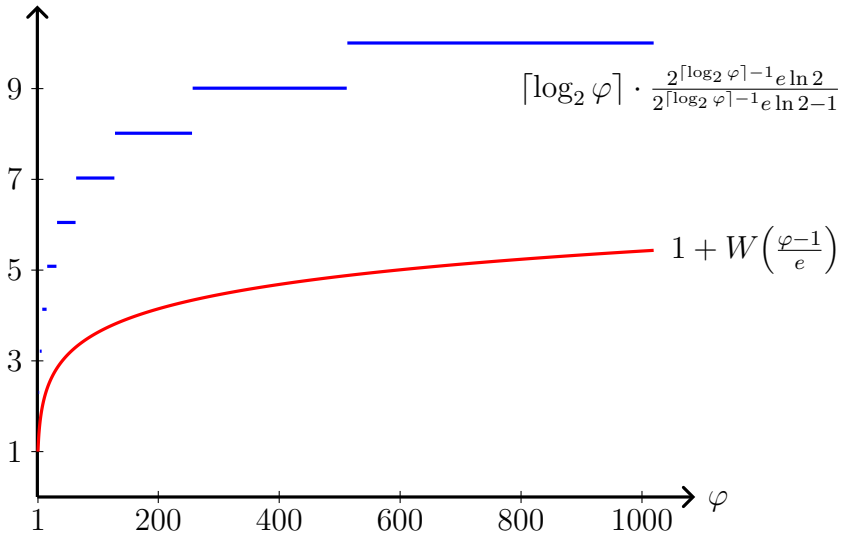
beschreiben. Man sieht, dass Gleichung (5.8) invariant gegenüber Skalierung von m und M ist. Das bedeutet, dass der kompetitive Faktor, der erreicht werden kann, nur von $\varphi = M/m$ abhängt. Auflösen der Gleichung nach r ergibt

$$r_\infty^\star(m, M) = 1 + W\left(\frac{\varphi - 1}{e}\right) = \Theta(\log \varphi),$$

wobei W die lambertsche W-Funktion, also die Umkehrfunktion von $f(x) = xe^x$, bezeichnet. Damit folgt das folgende Korollar, dessen Beweis analog zu dem von Korollar 5.7 ist.

Korollar 5.8. *In dem Szenario, dass nur das Intervall $[m, M]$ bekannt ist, ist der Algorithmus* Threat$(r_\infty^\star(m, M))$ *ein optimaler Online-Algorithmus. Er erreicht einen strikten kompetitiven Faktor von $\Theta(\log \varphi)$.*

In der folgenden Abbildung betrachten wir für konkrete Werte von φ, wie sich die kompetitiven Faktoren von EXPO und Threat verhalten. Da wir EXPO nur für den Fall analysiert haben, dass φ eine Zweierpotenz ist, haben wir in der Abbildung die Schranke aus Theorem 5.3 verwendet und φ jeweils auf die nächste Zweierpotenz aufgerundet. Die obere Kurve zeigt den kompetitiven Faktor von EXPO und die untere Kurve zeigt den kompetitiven Faktor von Threat$(r_\infty^\star(m, M))$. Man sieht, dass Threat echt besser als EXPO ist.



Auch der Algorithmus Threat kann auf das Szenario verallgemeinert werden, dass nur der Parameter φ nicht aber das Intervall $[m, M]$ bekannt ist. Dadurch verschlechtert sich sein kompetitiver Faktor, er ist aber dennoch besser als der von EXPO.

5.2 Economical Caching

Wir betrachten nun ein mit One-Way-Trading verwandtes Online-Problem, das von Englert et al. [8] eingeführt wurde. Bei diesem Problem geht es darum, ein vorhandenes Lager möglichst effektiv einzusetzen, um Kosten beim Einkauf zu sparen. Betrachten wir als Beispiel einen Autofahrer, der jeden Morgen an einer Tankstelle vorbeifährt, den aktuellen Preis beobachtet und dann entscheidet, ob und wie viel er tankt. Auf der einen Seite möchte der Autofahrer bei einem schlechten Preis nicht tanken, auf der anderen Seite entsteht jeden Tag ein gewisser Verbrauch. Wartet der Autofahrer zu lange, so ist er irgendwann bei einem leeren Tank gezwungen, einen beliebig hohen Preis zu akzeptieren.

Formal ist bei dem *Economical-Caching-Problem* ein Parameter $\varphi \geq 1$ gegeben und jedes Ereignis σ_i einer Eingabe $\sigma = (\sigma_1, \ldots, \sigma_n)$ besteht aus einem Preis $p_i \in [1, \varphi]$ und einem Verbrauch $v_i \geq 0$. Die Aufgabe besteht darin, in jedem Schritt i die Anzahl $x_i \geq 0$ an Einheiten festzulegen, die in diesem Schritt gekauft werden. Wir gehen davon aus, dass ein Lager zur Verfügung steht, das zu Beginn leer ist und dessen Kapazität 1 beträgt. Wir bezeichnen mit L_i^A den Lagerbestand, den Algorithmus A nach Schritt i erreicht. Dieser hängt natürlich von der Eingabe σ ab, diese betrachten wir aber als gegeben und nehmen sie nicht als weiteren Parameter in die Notation auf. Der Lagerbestand zum Zeitpunkt i ergibt sich aus dem zum Zeitpunkt $i-1$ als $L_i^A = L_{i-1}^A - v_i + x_i$. Da der Lagerbestand stets zwischen 0 und 1 gehalten werden muss, ergibt sich daraus die Einschränkung

$$x_i \in \Big[\max\{0, v_i - L_{i-1}^A\}, 1 - L_{i-1}^A + v_i\Big]. \tag{5.9}$$

Die Kosten von Algorithmus A auf der gegebenen Eingabe σ entsprechen dann der

Summe $C_A(\sigma) = \sum_{i=1}^{n} p_i x_i$. Ziel ist es, die Werte x_i so zu wählen, dass diese Kosten so klein wie möglich werden.

Wir gehen im Folgenden stets davon aus, dass einem Online-Algorithmus der Parameter φ bekannt ist. Dies entspricht dem Szenario bei One-Way-Trading, in dem das Intervall $[m, M]$ bekannt ist. Man hätte auch das Economical-Caching-Problem ohne Weiteres so formulieren können, dass alle Preise aus einem bekannten Intervall $[m, M]$ stammen. Um einen Parameter zu sparen, gehen wir hier aber davon aus, dass das Intervall so skaliert ist, dass die untere Intervallgrenze 1 ist. Auch die Einschränkung, dass das Lager maximal 1 fasst, ist ohne Beschränkung der Allgemeinheit und nur eine Frage der Skalierung.

Wir werden zunächst einen Algorithmus kennenlernen, der einen kompetitiven Faktor von $\sqrt{\varphi}$ für das Economical-Caching-Problem erreicht. Anschließend werden wir untere Schranken für den kompetitiven Faktor zeigen, den Online-Algorithmen erreichen können. Danach werden wir einen optimalen Online-Algorithmus beschreiben, der einen kompetitiven Faktor erreicht, der echt zwischen $\frac{\sqrt{\varphi}+1}{2}$ und $\sqrt{\varphi}$ liegt.

5.2.1 Ein Reservationspreisalgorithmus

Als ersten naheliegenden Ansatz werden wir den Reservationspreisalgorithmus für One-Way-Trading auf das Economical-Caching-Problem übertragen. Der Algorithmus RPA, den wir betrachten, kauft bei einem Preis von höchstens $\sqrt{\varphi}$ so viel wie möglich und bei einem Preis über $\sqrt{\varphi}$ so wenig wie möglich. Aus (5.9) ergibt sich

$$x_i = \begin{cases} v_i + (1 - L_{i-1}^{\mathrm{RPA}}) & \text{falls } p_i \leq \sqrt{\varphi}, \\ \max\{0, v_i - L_{i-1}^{\mathrm{RPA}}\} & \text{falls } p_i > \sqrt{\varphi}. \end{cases}$$

Theorem 5.9. *Der Algorithmus* RPA *ist $\sqrt{\varphi}$-kompetitiv für das Economical-Caching-Problem.*

Beweis. Die Beweisidee ist ähnlich wie bei Theorem 5.2, die Argumentation ist aber etwas komplizierter. Es sei $\sigma = (\sigma_1, \ldots, \sigma_n)$ eine beliebige Eingabe für das Economical-Caching-Problem und es sei OPT ein beliebiger optimaler Offline-Algorithmus. Weiterhin bezeichnen wir mit p_i und v_i den Preis bzw. Verbrauch im i-ten Schritt. Wir nutzen die Potentialfunktion $\Phi(L^{\mathrm{OPT}}, L^{\mathrm{RPA}}) = \sqrt{\varphi} \cdot (L^{\mathrm{OPT}} - L^{\mathrm{RPA}})$, die jeder Konfiguration ein Potential zuweist. Dabei seien L^{RPA} und L^{OPT} die aktuellen Lagerbestände der Algorithmen RPA und OPT.

Anstatt diese Potentialfunktion direkt auf der Eingabe σ anzuwenden, was nicht zum gewünschten Ergebnis führen würde, teilen wir die Eingabe σ zunächst in Phasen ein. Dazu fassen wir in σ jede Sequenz von aufeinanderfolgenden Preisen größer als $\sqrt{\varphi}$ zu einer Phase zusammen. Ebenso fassen wir in σ jede Sequenz von aufeinanderfolgenden Preisen von höchstens $\sqrt{\varphi}$ zu einer Phase zusammen. Phasen mit Preisen größer als $\sqrt{\varphi}$ nennen wir *schlechte Phasen* und Phasen mit Preisen von höchstens $\sqrt{\varphi}$ nennen wir *gute Phasen*. Auf diese Weise können wir die Eingabe σ als eine Sequenz $(\Sigma_1, \ldots, \Sigma_k)$ von $k \leq n$ Phasen auffassen, wobei gute und schlechte Phasen abwechselnd auftreten.

Anstatt das Potential nach jedem Schritt σ_i zu betrachten, interessiert uns in diesem Beweis nur das Potential am Ende der Phasen. Sei Φ_i das Potential nach Phase Σ_i und sei $\Phi_0 = 0$ das Potential vor dem ersten Schritt σ_1. Außerdem bezeichnen wir mit L_i^{RPA} und L_i^{OPT} die Lagerbestände der Algorithmen RPA und OPT nach der i-ten Phase und wir bezeichnen mit V_i den Gesamtverbrauch in der i-ten Phase. Ferner sei $L_0^{\text{RPA}} = L_0^{\text{OPT}} = 0$.

Wir bezeichnen mit C_i^{RPA} und C_i^{OPT} die Kosten, die die Algorithmen RPA und OPT insgesamt in Phase i erzeugen. Analog zu Abschnitt 2.2.1 definieren wir die amortisierten Kosten a_i von Phase i als

$$a_i = C_i^{\text{RPA}} + \Phi_i - \Phi_{i-1} = C_i^{\text{RPA}} + \sqrt{\varphi} \cdot (L_{i-1}^{\text{RPA}} - L_i^{\text{RPA}}) + \sqrt{\varphi} \cdot (L_i^{\text{OPT}} - L_{i-1}^{\text{OPT}}).$$

Können wir zeigen, dass a_i für alle i durch $\sqrt{\varphi} \cdot C_i^{\text{OPT}}$ nach oben beschränkt ist, so folgt mit der Methode der Potentialfunktionen, dass der Algorithmus RPA $\sqrt{\varphi}$-kompetitiv ist, da das Potential nur Werte in einem konstanten Intervall $[-\sqrt{\varphi}, \sqrt{\varphi}]$ annehmen kann. Zur Abschätzung der amortisierten Kosten betrachten wir gute und schlechte Phasen separat.

- Es sei Σ_i eine gute Phase. Aus der Definition von RPA folgt, dass $L_i^{\text{RPA}} = 1$ gilt. Insgesamt kauft der Algorithmus RPA in der Phase Σ_i also $V_i + (1 - L_{i-1}^{\text{RPA}})$. Da alle Preise in dieser Phase maximal $\sqrt{\varphi}$ betragen, gilt

  $$C_i^{\text{RPA}} \leq \sqrt{\varphi} \cdot (V_i + 1 - L_{i-1}^{\text{RPA}}).$$

  Außerdem gilt

  $$C_i^{\text{OPT}} \geq V_i + (L_i^{\text{OPT}} - L_{i-1}^{\text{OPT}}),$$

  da OPT in dieser Phase insgesamt genau $V_i + (L_i^{\text{OPT}} - L_{i-1}^{\text{OPT}})$ kauft. Damit folgt

  $$\begin{aligned} a_i &\leq \sqrt{\varphi} \cdot (V_i + 1 - L_{i-1}^{\text{RPA}}) + \sqrt{\varphi} \cdot (L_{i-1}^{\text{RPA}} - 1) + \sqrt{\varphi} \cdot (L_i^{\text{OPT}} - L_{i-1}^{\text{OPT}}) \\ &= \sqrt{\varphi} \cdot (V_i + L_i^{\text{OPT}} - L_{i-1}^{\text{OPT}}) \\ &\leq \sqrt{\varphi} \cdot C_i^{\text{OPT}}. \end{aligned}$$

- Es sei Σ_i eine schlechte Phase. Da in dieser Phase alle Preise größer als $\sqrt{\varphi}$ sind, gilt

  $$C_i^{\text{OPT}} \geq \sqrt{\varphi} \cdot (V_i + L_i^{\text{OPT}} - L_{i-1}^{\text{OPT}}).$$

  Aus der Definition von RPA folgt, dass $L_i^{\text{RPA}} = \max\{0, L_{i-1}^{\text{RPA}} - V_i\}$ gilt und dass dementsprechend der Algorithmus RPA in dieser Phase $\max\{0, V_i - L_{i-1}^{\text{RPA}}\}$ kauft. Wir betrachten die beiden zugehörigen Unterfälle separat.

  - Gilt $V_i \leq L_{i-1}^{\text{RPA}}$, so ist $L_i^{\text{RPA}} = L_{i-1}^{\text{RPA}} - V_i$ und RPA kauft in dieser Phase nichts. Dementsprechend entstehen RPA keine Kosten und es gilt

    $$\begin{aligned} a_i &= \sqrt{\varphi} \cdot (L_{i-1}^{\text{RPA}} - L_i^{\text{RPA}}) + \sqrt{\varphi} \cdot (L_i^{\text{OPT}} - L_{i-1}^{\text{OPT}}) \\ &= \sqrt{\varphi} \cdot V_i + \sqrt{\varphi} \cdot (L_i^{\text{OPT}} - L_{i-1}^{\text{OPT}}) \\ &\leq C_i^{\text{OPT}} \leq \sqrt{\varphi} \cdot C_i^{\text{OPT}}. \end{aligned}$$

– Gilt $V_i > L_{i-1}^{\mathrm{RPA}}$, so ist $L_i^{\mathrm{RPA}} = 0$ und RPA kauft in dieser Phase $V_i - L_{i-1}^{\mathrm{RPA}}$. Dementsprechend gilt

$$C_i^{\mathrm{RPA}} \leq \varphi \cdot (V_i - L_{i-1}^{\mathrm{RPA}}).$$

Entweder gilt $i = 1$, was bedeutet, dass $L_{i-1}^{\mathrm{RPA}} = L_{i-1}^{\mathrm{OPT}} = 0$ gilt, oder vor der schlechten Phase i gab es eine gute Phase $i-1$, was $L_{i-1}^{\mathrm{RPA}} = 1$ bedeutet. In jedem Falle gilt $L_{i-1}^{\mathrm{RPA}} \geq L_{i-1}^{\mathrm{OPT}}$. Dies impliziert

$$\begin{aligned} a_i &\leq \varphi \cdot (V_i - L_{i-1}^{\mathrm{RPA}}) + \sqrt{\varphi} \cdot (L_i^{\mathrm{OPT}} - L_i^{\mathrm{RPA}} - L_{i-1}^{\mathrm{OPT}} + L_{i-1}^{\mathrm{RPA}}) \\ &= \varphi \cdot (V_i - L_{i-1}^{\mathrm{RPA}}) + \sqrt{\varphi} \cdot (L_i^{\mathrm{OPT}} + (L_{i-1}^{\mathrm{RPA}} - L_{i-1}^{\mathrm{OPT}})) \\ &\leq \varphi \cdot (V_i - L_{i-1}^{\mathrm{RPA}}) + \varphi \cdot (L_i^{\mathrm{OPT}} + (L_{i-1}^{\mathrm{RPA}} - L_{i-1}^{\mathrm{OPT}})) \\ &= \varphi \cdot (V_i + L_i^{\mathrm{OPT}} - L_{i-1}^{\mathrm{OPT}}) \\ &\leq \sqrt{\varphi} \cdot C_i^{\mathrm{OPT}}. \end{aligned}$$

In der zweiten Ungleichung haben wir ausgenutzt, dass $L_i^{\mathrm{OPT}} + (L_{i-1}^{\mathrm{RPA}} - L_{i-1}^{\mathrm{OPT}}) \geq 0$ wegen $L_{i-1}^{\mathrm{RPA}} \geq L_{i-1}^{\mathrm{OPT}}$ gilt.

Der Beweis ist abgeschlossen, da wir eine Potentialfunktion mit den gewünschten Eigenschaften gefunden haben. □

Wir stellen im Folgenden einen Schritt σ_i mit Preis p_i und Verbrauch v_i als $\sigma_i = \binom{p_i}{v_i}$ dar. Es ist leicht einzusehen, dass der Algorithmus RPA nicht besser als $\sqrt{\varphi}$-kompetitiv ist. Dazu betrachten wir die Sequenz

$$\sigma' = \binom{\sqrt{\varphi}}{0}, \binom{1}{0}, \binom{\varphi}{1}.$$

Der Algorithmus RPA füllt im ersten Schritt das Lager komplett, im zweiten Schritt kann er dementsprechend nichts mehr kaufen und im letzten Schritt kauft er ebenfalls nichts und deckt den Verbrauch komplett aus seinem Lager, das anschließend leer ist. Dadurch entstehen RPA Kosten in Höhe von $\sqrt{\varphi}$, wohingegen die Kosten eines optimalen Offline-Algorithmus nur 1 betragen. Dies zeigt zunächst nur, dass RPA nicht besser als strikt $\sqrt{\varphi}$-kompetitiv ist. Wir können jedoch eine Eingabe σ konstruieren, die aus beliebig vielen Wiederholungen von σ' besteht. Diese beeinflussen sich nicht, da RPA nach jeder Wiederholung von σ' ein leeres Lager hat. Ebenso können wir davon ausgehen, dass auch der optimale Offline-Algorithmus nach jeder Wiederholung ein leeres Lager hat. Mit ähnlichen Argumenten wie in Abschnitt 2.1.2 folgt daraus, dass RPA für kein $r < \sqrt{\varphi}$ einen kompetitiven Faktor von r erreicht, denn man kann für jedes $r < \sqrt{\varphi}$ und jede Konstante τ eine Anzahl an Wiederholungen finden, sodass die Kosten von RPA auf σ größer als $r \cdot \mathrm{OPT}(\sigma) + \tau$ sind.

5.2.2 Untere Schranken

In diesem Abschnitt beschäftigen wir uns mit unteren Schranken für den kompetitiven Faktor, der bei dem Economical-Caching-Problem erreicht werden kann. Wir fangen zum Aufwärmen mit einer einfachen Schranke an.

Theorem 5.10. *Es gibt keinen deterministischen oder randomisierten Online-Algorithmus für das Economical-Caching-Problem, der einen kompetitiven Faktor kleiner als $\frac{\sqrt{\varphi}+1}{2}$ erreicht.*

Beweis. Es sei A ein beliebiger deterministischer Online-Algorithmus für das Economical-Caching-Problem. Wir konstruieren eine Eingabe σ', die mit einem Schritt $\binom{\sqrt{\varphi}}{0}$ beginnt. Es sei x eine Variable, die angibt, wie viel Algorithmus A in diesem ersten Schritt kauft. Basierend auf x entscheiden wir, wie die Sequenz σ' weitergeht. Es gelte

$$\sigma' = \begin{cases} \binom{\sqrt{\varphi}}{0}, \binom{\varphi}{1} & \text{falls } x \leq \frac{1}{2}, \\ \binom{\sqrt{\varphi}}{0}, \binom{1}{0}, \binom{\varphi}{1} & \text{falls } x > \frac{1}{2}. \end{cases}$$

Daraus ergibt sich

$$C_A(\sigma') \geq \begin{cases} x \cdot \sqrt{\varphi} + (1-x) \cdot \varphi & \text{falls } x \leq \frac{1}{2} \\ x \cdot \sqrt{\varphi} + (1-x) & \text{falls } x > \frac{1}{2} \end{cases} \geq \begin{cases} \frac{1}{2} \cdot (\sqrt{\varphi} + \varphi) & \text{falls } x \leq \frac{1}{2} \\ \frac{1}{2} \cdot (\sqrt{\varphi} + 1) & \text{falls } x > \frac{1}{2} \end{cases}$$

und

$$\mathrm{OPT}(\sigma') = \begin{cases} \sqrt{\varphi} & \text{falls } x \leq \frac{1}{2}, \\ 1 & \text{falls } x > \frac{1}{2}. \end{cases}$$

Unabhängig von x gilt also

$$C_A(\sigma') \geq \frac{\sqrt{\varphi}+1}{2} \cdot \mathrm{OPT}(\sigma').$$

Dies zeigt zunächst nur, dass A nicht besser als strikt $\frac{\sqrt{\varphi}+1}{2}$-kompetitiv ist. Aber auch hier können wir eine Sequenz σ^ℓ konstruieren, die aus einer beliebigen Anzahl ℓ an Wiederholungen von σ' besteht. Allerdings ist hier ein wenig Vorsicht geboten, weil nicht klar ist, ob sich die Wiederholungen beeinflussen können. Um das auszuschließen, bemerken wir zunächst, dass wir ohne Beschränkung der Allgemeinheit davon ausgehen können, dass das Lager von Algorithmus A nach der Sequenz σ' leer ist. Der Leser sollte sich überlegen, dass wir Algorithmus A so modifizieren können, dass er diese Eigenschaft erfüllt, ohne dabei seine Kosten zu vergrößern. Zweitens weisen wir explizit darauf hin, dass der Algorithmus in jeder Wiederholung von σ' ein anderes x wählen kann. Dies ist aber unproblematisch, da wir bei der Konstruktion der Sequenz jedes Mal entsprechend darauf reagieren können. Insgesamt gilt für die Sequenz σ^ℓ

$$C_A(\sigma^\ell) \geq \frac{\sqrt{\varphi}+1}{2} \cdot \mathrm{OPT}(\sigma^\ell)$$

und außerdem gilt $\mathrm{OPT}(\sigma^\ell) \geq \ell$. Hätte Algorithmus A einen kompetitiven Faktor von $r < \frac{\sqrt{\varphi}+1}{2}$, so gäbe es eine Konstante τ, sodass

$$C_A(\sigma^\ell) \leq r \cdot \mathrm{OPT}(\sigma^\ell) + \tau$$

für alle ℓ gilt. Zusammen mit der vorangegangenen Formel folgt daraus

$$\frac{\sqrt{\varphi}+1}{2} \cdot \mathrm{OPT}(\sigma^\ell) \leq r \cdot \mathrm{OPT}(\sigma^\ell) + \tau,$$

was äquivalent zu

$$\mathrm{OPT}(\sigma^\ell) \leq \frac{\tau}{\frac{\sqrt{\varphi}+1}{2} - r}$$

ist. Diese Ungleichung ist aber für hinreichend große ℓ nicht erfüllt und damit ergibt sich ein Widerspruch zu der Annahme, dass Algorithmus A einen kompetitiven Faktor von r erreicht.

Wir haben bislang nur über deterministische Algorithmen gesprochen. Der Leser überlege sich als Übungsaufgabe, dass sich die Konstruktion komplett analog überträgt, wenn x angibt, wie viel Algorithmus A im Erwartungswert im ersten Schritt von σ' kauft. □

Wir werden nun eine bessere untere Schranke für das Economical-Caching-Problem herleiten. In dieser kommt wieder die lambertsche W-Funktion vor. Oben haben wir diese bereits als die Umkehrfunktion der Funktion $f(x) = x \cdot e^x$ definiert. Für positive Werte ist die lambertsche W-Funktion dadurch eindeutig bestimmt. Im Folgenden müssen wir die lambertsche W-Funktion aber für negative Werte auswerten, was problematisch ist, da die Funktion f nicht injektiv ist. Da das Minimum der Funktion f genau $-1/e$ beträgt, ist die lambertsche W-Funktion ohnehin nur für Werte $z \geq -1/e$ definiert. In dem Intervall $(-1/e, 0)$ gibt es zwei Möglichkeiten, die Funktion f umzukehren. Uns interessiert nur der sogenannte Hauptast der lambertschen W-Funktion. Diesen erhält man, wenn man zusätzlich $W(z) \geq -1$ fordert. Für $z \in [-1/e, 0)$ bezeichnen wir mit $W(z)$ also die eindeutige Zahl $x \geq -1$ mit $x \cdot e^x = z$.

Lemma 5.11. *Für alle $x \in [-\frac{1}{e}, 0)$ gilt $\ln(-W(x)) = \ln(-x) - W(x)$.*

Beweis. Für $x \in [-\frac{1}{e}, 0)$ gilt per Definition $W(x) \cdot e^{W(x)} = x$. Daraus folgt das Lemma:

$$\begin{aligned} & W(x) \cdot e^{W(x)} = x \\ \Rightarrow & -W(x) \cdot e^{W(x)} = -x \\ \Rightarrow & \ln\Big(-W(x) \cdot e^{W(x)}\Big) = \ln(-x) \\ \Rightarrow & \ln(-W(x)) + W(x) = \ln(-x). \end{aligned}$$

□

Theorem 5.12. *Es gibt keinen deterministischen oder randomisierten Online-Algorithmus für das Economical-Caching-Problem, der einen kompetitiven Faktor kleiner als*

$$r^\star(\varphi) = \frac{1}{W\left(\frac{1-\varphi}{e\varphi}\right) + 1}$$

erreicht.

Beweis. Wir schreiben im Folgenden der Einfachheit halber $r^\star$ statt $r^\star(\varphi)$. Für jedes $p \in [1, \varphi/r^\star]$ konstruieren wir eine Sequenz Σ_p. Diese beginnt mit einer Folge Σ'_p von Schritten, in denen kein Verbrauch auftritt und in denen der Preis monoton von $\varphi/r^\star$ auf p fällt. Um genau zu sein, sei

$$\frac{\varphi}{r^\star}, \frac{\varphi}{r^\star} - \varepsilon, \frac{\varphi}{r^\star} - 2\varepsilon, \ldots, p + \varepsilon, p$$

die Folge von Preisen in Σ'_p. Dabei sei $\varepsilon > 0$ ein kleiner Wert mit $(\varphi/r^\star - p)/\varepsilon \in \mathbb{N}$. Da wir ε beliebig klein wählen können, gehen wir im Folgenden davon aus, dass der Preis in der Sequenz Σ'_p kontinuierlich von $\varphi/r^\star$ auf p fällt. Die Sequenz Σ_p besteht aus der Sequenz Σ'_p und einem zusätzlichen Schritt mit Preis φ und Verbrauch 1, der an Σ'_p angehängt wird.

Bevor wir die untere Schranke für beliebige Online-Algorithmen zeigen, betrachten wir einen speziellen Algorithmus $A^\star$, der auf einer *Lagerfunktion* $g : [1, \varphi/r^\star] \to [0, 1]$ beruht. Diese Funktion gibt für jeden Preis $p \in [1, \varphi/r^\star]$ einen gewünschten Lagerbestand $g(p)$ an. Der Algorithmus $A^\star$ kauft bei einem Preis p_i soviel, dass der Lagerbestand nach Schritt i möglichst nah an $g(p_i)$ ist. Formal können wir den Betrag x_i, den Algorithmus $A^\star$ in Schritt i kauft, als

$$x_i = \begin{cases} v_i + g(p_i) - L_{i-1}^{A^\star} & \text{falls } g(p_i) \geq L_{i-1}^{A^\star} - v_i \\ 0 & \text{falls } g(p_i) < L_{i-1}^{A^\star} - v_i \end{cases}$$

beschreiben. Die Lagerfunktion $g : [1, \varphi/r^\star] \to [0, 1]$, die das Verhalten von $A^\star$ bestimmt, ist definiert als

$$g(x) = r^\star \cdot \Big(\ln\Big(1 - \frac{x}{\varphi}\Big) - \ln\Big(1 - \frac{1}{r^\star}\Big)\Big).$$

Diese Funktion ist streng monoton fallend und sie verhält sich nahezu linear (siehe Abbildung 5.1). Außerdem gilt $g(\varphi/r^\star) = 0$ und mit Lemma 5.11 folgt

$$\begin{aligned} g(1) &= r^\star \cdot \Big(\ln\Big(1 - \frac{1}{\varphi}\Big) - \ln\Big(- W\Big(\frac{1-\varphi}{e\varphi}\Big)\Big)\Big) \\ &= r^\star \cdot \Big(\ln\Big(1 - \frac{1}{\varphi}\Big) - \ln\Big(- \frac{1-\varphi}{e\varphi}\Big) + W\Big(\frac{1-\varphi}{e\varphi}\Big)\Big) \\ &= r^\star \cdot \Big(\ln\Big(\frac{\varphi-1}{\varphi}\Big) - \ln\Big(\frac{\varphi-1}{e\varphi}\Big) + W\Big(\frac{1-\varphi}{e\varphi}\Big)\Big) \\ &= r^\star \cdot \Big(\ln\Big(\frac{\varphi-1}{\varphi}\Big) - \ln\Big(\frac{\varphi-1}{\varphi}\Big) - \ln\Big(\frac{1}{e}\Big) + W\Big(\frac{1-\varphi}{e\varphi}\Big)\Big) \\ &= r^\star \cdot \Big(1 + W\Big(\frac{1-\varphi}{e\varphi}\Big)\Big) = 1. \end{aligned}$$

Die Kosten $C_{A^\star}(\Sigma_p)$ von Algorithmus $A^\star$ auf der Eingabe Σ_p können wir für jedes $p \in [1, \varphi/r^\star]$ durch den folgenden Term ausdrücken:

$$C_{A^\star}(\Sigma_p) = (1 - g(p)) \cdot \varphi + p \cdot g(p) + \int_p^{\varphi/r^\star} g(x)\, dx.$$

Der erste Summand rührt daher, dass der Algorithmus $A^\star$ auf der Eingabe Σ_p im letzten Schritt $1 - g(p)$ viele Einheiten zum Preis φ kaufen muss, um den Verbrauch von 1 zu decken. Der zweite Summand ist in Abbildung 5.1 näher erläutert.

Die wesentliche Eigenschaft der Funktion g ist, dass die Kosten $C_{A^\star}(\Sigma_p)$ für alle $p \in [1, \varphi/r^\star]$ genau gleich $r^\star \cdot \mathrm{OPT}(\Sigma_p)$ sind. Dies zeigt die folgende Rechnung:

$$C_{A^\star}(\Sigma_p) = (1 - g(p)) \cdot \varphi + p \cdot g(p) + \int_p^{\varphi/r^\star} g(x)\, dx$$

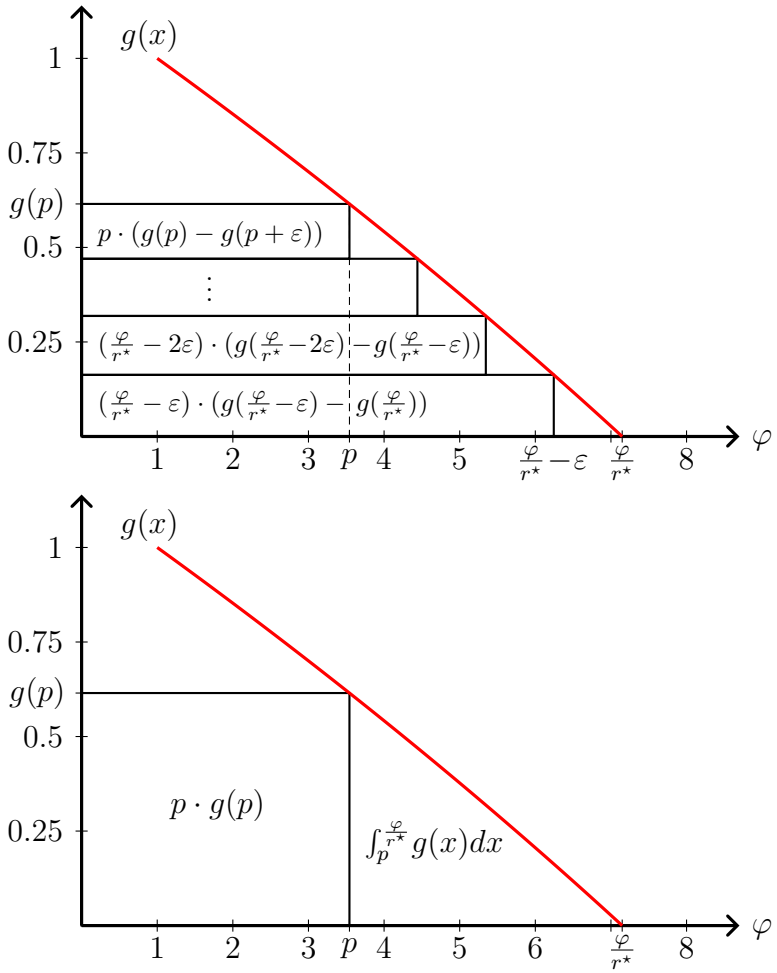


Abbildung 5.1: Die obere Abbildung illustriert die Kosten von Algorithmus $A^\star$ auf der diskreten Sequenz Σ_p. Die untere Abbildung illustriert die Kosten im kontinuierlichen Fall. Da die Funktion g monoton ist, ist sie auf dem Intervall $[p, \varphi/r^\star]$ integrierbar, d. h. im Grenzwert für $\varepsilon \to 0$ stimmen die Kosten für die diskrete Sequenz mit den Kosten für die kontinuierliche Sequenz überein. In dem Beispiel gilt $\varphi = 30$ und $r^\star \approx 4.2$.

$$
\begin{aligned}
&= (1 - g(p)) \cdot \varphi + p \cdot g(p) + \int_p^{\varphi/r^\star} r^\star \cdot \left(\ln\left(1 - \frac{x}{\varphi}\right) - \ln\left(1 - \frac{1}{r^\star}\right)\right) dx \\
&= (1 - g(p)) \cdot \varphi + p \cdot g(p) + \left[r^\star \cdot (-\varphi + x)\left(\ln\left(1 - \frac{x}{\varphi}\right) - 1\right)\right]_p^{\varphi/r^\star} \\
&\quad - r^\star \cdot \left(\frac{\varphi}{r^\star} - p\right) \ln\left(1 - \frac{1}{r^\star}\right) \\
&= (1 - g(p)) \cdot \varphi + p \cdot g(p) + r^\star \cdot \left((-\varphi + p) \ln\left(1 - \frac{1}{r^\star}\right) \right. \\
&\quad \left. - (-\varphi + p) \ln\left(1 - \frac{p}{\varphi}\right) + \left(p - \frac{\varphi}{r^\star}\right)\right) \\
&= (1 - g(p)) \cdot \varphi + p \cdot g(p) + (\varphi - p) \cdot g(p) + r^\star \cdot \left(p - \frac{\varphi}{r^\star}\right) \\
&= r^\star \cdot p = r^\star \cdot \mathrm{OPT}(\Sigma_p).
\end{aligned}
$$

Wir haben damit nachgewiesen, dass der Algorithmus $A^\star$ für jedes $p \in [1, \varphi/r^\star]$ auf der Sequenz Σ_p genau die $r^\star$-fachen Kosten eines optimalen Offline-Algorithmus verursacht. Bezogen auf diese Sequenzen handelt es sich bei $A^\star$ also um einen Drohungs-Algorithmus: er kauft in jedem Schritt den kleinstmöglichen Wert der garantiert, dass er strikt $r^\star$-kompetitiv ist.

Ähnlich wie in Korollar 5.7 können wir nun argumentieren, dass es keinen Online-Algorithmus für das Economical-Caching-Problem geben kann, der auf allen Sequenzen Σ_p einen besseren kompetitiven Faktor als $r^\star$ erreicht. Sei dazu A ein beliebiger deterministischer Online-Algorithmus. Die Erweiterung des folgenden Argumentes auf randomisierte Algorithmen überlassen wir wieder dem Leser als Übung. Wir betrachten das Verhalten von A auf den Sequenzen Σ_p, wobei wir implizit von einer Diskretisierung mit einem hinreichend kleinen ε ausgehen, bei der Beschreibung aber wieder auf kontinuierliche Sequenzen zurückgreifen.

Wir können das Verhalten von A auf den Sequenzen Σ_p wieder durch eine Lagerfunktion $f : [1, \varphi/r^\star] \to [0, 1]$ beschreiben. Dabei gibt $f(p)$ für $p \in [1, \varphi/r^\star]$ an, wie hoch der Lagerbestand von Algorithmus A nach der Sequenz Σ'_p ist. Da A ein Online-Algorithmus ist, ist die Funktion f monoton fallend. Außerdem sei ohne Beschränkung der Allgemeinheit $f(1) = 1$, da man jeden Algorithmus so transformieren kann, dass er bei Preis 1 soviel wie möglich kauft, ohne seine Kosten auf Sequenzen der Form Σ_p zu erhöhen. Analog zu den Kosten von $A^\star$ können wir die Kosten von Algorithmus A auf der Sequenz Σ_p als

$$C_A(\Sigma_p) = (1 - f(p)) \cdot \varphi + p \cdot f(p) + \int_p^{\varphi/r^\star} f(x)\,dx. \tag{5.10}$$

schreiben. Wir zeigen nun, dass es für jede Funktion f, d. h. für jeden Algorithmus A, ein p gibt, sodass die Kosten von A auf Σ_p mindestens $r^\star \cdot \mathrm{OPT}(\Sigma_p)$ betragen. Dazu unterscheiden wir zwei Fälle.

Zunächst betrachten wir den Fall, dass $f(x) \geq g(x)$ für alle $x \in [1, \varphi/r^\star]$ gilt. In diesem Fall gilt

$$C_A(\Sigma_1) = 1 + \int_1^{\varphi/r^\star} f(x)\,dx \geq 1 + \int_1^{\varphi/r^\star} g(x)\,dx = C_{A^\star}(\Sigma_1) = r^\star \cdot \mathrm{OPT}(\Sigma_1).$$

Gibt es ein $x \in [1, \varphi/r^\star]$ mit $f(x) < g(x)$, so setzen wir

$$p = \sup\{x \in [1, \varphi/r^\star] \mid f(x) < g(x)\}.$$

Für alle $x > p$ gilt dann $f(x) \geq g(x)$. Außerdem gilt $f(p) = g(p)$. Dies folgt, da g stetig ist und da f und g monoton fallen. Wir berechnen nun mithilfe von (5.10) die Kosten von Algorithmus A auf der Eingabe Σ_p. Es folgt

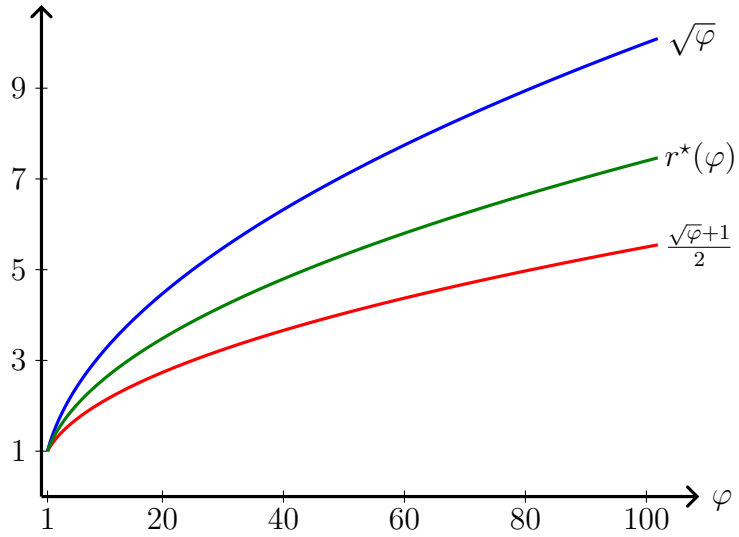
$$\begin{aligned} C_A(\Sigma_p) &= (1 - f(p)) \cdot \varphi + p \cdot f(p) + \int_p^{\varphi/r^\star} f(x)\,dx \\ &\geq (1 - g(p)) \cdot \varphi + p \cdot g(p) + \int_p^{\varphi/r^\star} g(x)\,dx \end{aligned}$$

$$= C_{A^\star}(\Sigma_p) = r^\star \cdot \mathrm{OPT}(\Sigma_p).$$

Mit dieser Argumentation folgt, dass es keinen Online-Algorithmus gibt, der auf allen Sequenzen der Form Σ_p besser als strikt $r^\star$-kompetitiv ist. Ähnlich wie im Beweis von Theorem 5.10 können wir durch eine hinreichend große Anzahl an Wiederholungen von Sequenzen der Form Σ_p zeigen, dass es keinen Online-Algorithmus gibt, der einen besseren kompetitiven Faktor als $r^\star$ erreicht. Dazu ist es wieder wichtig, dass sich die einzelnen Wiederholungen nicht beeinflussen. Dies ist hier der Fall, da jede Sequenz Σ_p mit einem Schritt mit Preis φ und Verbrauch 1 abschließt. Es ist nicht schwer, zu argumentieren, dass wir ohne Beschränkung der Allgemeinheit davon ausgehen können, dass der Algorithmus A nach einem solchen Schritt ein leeres Lager hat. Das bedeutet, dass jede Wiederholung mit einem leeren Lager startet und wir deshalb die obige Analyse auf jede Wiederholung anwenden können.

Natürlich muss Algorithmus A nicht in jeder Wiederholung dieselbe Lagerfunktion f zugrunde liegen. Dies ist aber kein Problem, da wir gezeigt haben, dass wir für jede Lagerfunktion f eine schlechte Sequenz Σ_p finden können. Der Rest des Argumentes ist komplett analog zu dem Beweis von Theorem 5.10. □

Die folgende Abbildung zeigt von unten nach oben die Funktionen $\frac{\sqrt{\varphi}+1}{2}$, $r^\star(\varphi)$ und $\sqrt{\varphi}$.



Man kann zeigen, dass

$$r^\star(\varphi) \in \left[\frac{\sqrt{\varphi}}{\sqrt{2}}, \frac{\sqrt{\varphi}+1}{\sqrt{2}}\right]$$

für alle $\varphi \geq 1$ gilt.

5.2.3 Algorithmen mit fester Lagerfunktion

Wir haben im Beweis von Theorem 5.12 einen Drohungs-Algorithmus für Sequenzen der Form Σ_p benutzt, um eine untere Schranke für den kompetitiven Faktor jedes Online-Algorithmus für das Economical-Caching-Problem zu zeigen. Dies legt die Vermutung nahe, dass genauso wie beim One-Way-Trading der Drohungs-Algorithmus

optimal ist und genau den kompetitiven Faktor $r^\star$ aus dem Theorem erreicht. Tatsächlich ist dies auch für alle Eingaben, die aus beliebig vielen Sequenzen der Form Σ_p zusammengesetzt sind, der Fall. Dies folgt direkt aus dem Beweis von Theorem 5.12. Wie sieht es aber mit beliebigen Eingaben aus, die anders aufgebaut sind als die Sequenzen Σ_p?

Wir betrachten allgemein die Klasse derjenigen Algorithmen, die auf einer statischen Lagerfunktion basieren. Jeder Algorithmus dieser Klasse ist durch eine Lagerfunktion $f : [1, \varphi] \to [0, 1]$ beschrieben, die zu jedem Preis $p \in [1, \varphi]$ den gewünschten Lagerbestand $f(p)$ angibt. Zu jeder solchen Funktion f gibt es einen Algorithmus A_f, der in jedem Schritt mit Preis p soviel kauft, dass sein Lagerbestand nach dem Schritt dem Wert $f(p)$ möglichst nahe ist. Formal können wir den Betrag x_i, den Algorithmus A_f in Schritt i kauft, als

$$x_i = \begin{cases} v_i + f(p_i) - L_{i-1}^{A_f} & \text{falls } f(p_i) \geq L_{i-1}^{A_f} - v_i, \\ 0 & \text{falls } f(p_i) < L_{i-1}^{A_f} - v_i, \end{cases}$$

beschreiben. Mit dieser Notation gilt $A^\star = A_g$ für die Lagerfunktion, die wir im Beweis von Theorem 5.12 definiert haben. Auch der Algorithmus RPA, den wir in Abschnitt 5.2.1 untersucht haben, fällt in diese Kategorie. Er entspricht dem Algorithmus A_f für die Lagerfunktion

$$f(p) = \begin{cases} 1 & \text{falls } p \leq \sqrt{\varphi}, \\ 0 & \text{falls } p > \sqrt{\varphi}. \end{cases}$$

Wir zeigen nun, dass kein Algorithmus mit einer festen Lagerfunktion besser als $\sqrt{\varphi}$-kompetitiv ist. Das bedeutet insbesondere, dass der Algorithmus $A^\star$ im Allgemeinen nicht besser als der Algorithmus RPA ist.

Theorem 5.13. *Jeder Algorithmus A_f mit einer festen Lagerfunktion $f : [1, \varphi] \to [0, 1]$ ist nicht besser als $\sqrt{\varphi}$-kompetitiv.*

Beweis. Zunächst beschränken wir unsere Betrachtung auf monoton fallende Funktionen f. Der Leser überlege sich als Übung, dass wir ohne Beschränkung der Allgemeinheit davon ausgehen können, dass $f(1) = 1$ und $f(\varphi) = 0$ gilt. Sei f eine beliebige Funktion mit diesen Eigenschaften und sei A_f der zugehörige Algorithmus. Wir konstruieren nun eine Eingabe σ, auf der A_f nicht besser als $\sqrt{\varphi}$-kompetitiv ist. Die Eingabe beginnt zunächst mit einer Sequenz Σ', die Σ_1' aus dem Beweis von Theorem 5.12 ähnelt. In der Sequenz Σ' sinkt der Preis monoton von φ auf 1 und in keinem der Schritte tritt Verbrauch auf. Dies können wir uns wieder kontinuierlich oder mit einem beliebig kleinen $\varepsilon > 0$ diskretisiert vorstellen. Wir haben in dem Beweis von Theorem 5.12 argumentiert, dass die Kosten von Algorithmus A_f auf dieser Sequenz genau

$$q := 1 + \int_1^\varphi f(x)\, dx$$

betragen. Gilt $q > \sqrt{\varphi}$, so ist der Algorithmus A_f auf der Sequenz, die aus Σ' und einem weiteren Schritt mit Preis φ und Verbrauch 1 besteht, nicht besser als strikt $\sqrt{\varphi}$-kompetitiv.

Interessant ist also nur noch der Fall $q \leq \sqrt{\varphi}$. In diesem Fall hängen wir an die Sequenz Σ' eine Sequenz Σ'' an. In dieser Sequenz steigt der Preis monoton von 1 auf φ und in jedem Schritt wird gerade soviel verbraucht, dass der Algorithmus A_f während der Sequenz Σ'' nichts kauft und sukzessive sein Lager leert. Formal entspricht Σ'' der Sequenz

$$\binom{1+\varepsilon}{f(1)-f(1+\varepsilon)}\binom{1+2\varepsilon}{f(1+\varepsilon)-f(1+2\varepsilon)}\cdots\binom{\varphi-\varepsilon}{f(\varphi-2\varepsilon)-f(\varphi-\varepsilon)}\binom{\varphi}{f(\varphi-\varepsilon)}.$$

Man überlegt sich leicht, dass diese Sequenz wirklich die gewünschten Eigenschaften besitzt, dass Algorithmus A_f während Σ'' nichts kauft und am Ende ein leeres Lager besitzt. Wir schließen die Sequenz σ durch einen letzten Schritt mit Preis φ und Verbrauch 1 ab. Es gilt also

$$\sigma = \Sigma', \Sigma'', \binom{\varphi}{1}.$$

Da Algorithmus A_f im letzten Schritt den kompletten Verbrauch zum Preis von φ decken muss, folgt

$$C_{A_f}(\sigma) = q + \varphi.$$

Um nun die Kosten eines optimalen Offline-Algorithmus nach oben abzuschätzen, betrachten wir einen Algorithmus A', der in allen Schritten von Σ' bis auf den letzten nichts kauft und der im letzten Schritt von Σ' zu einem Preis von 1 das Lager komplett füllt. Während der Sequenz Σ'' kauft dieser Algorithmus in jedem Schritt genau den aktuellen Verbrauch. Er hat also am Ende der Sequenz Σ'' ein volles Lager und kann im letzten Schritt den Verbrauch komplett aus dem Lager decken, ohne etwas kaufen zu müssen. Mit der Anschauung, die auch Abbildung 5.1 zugrunde liegt, erhält man für die Kosten dieses Algorithmus

$$\mathrm{OPT}(\sigma) \leq C_{A'}(\sigma) = 1 + \left(1 + \int_1^{\varphi} f(x)\,dx\right) = 1 + q.$$

Wegen $q \leq \sqrt{\varphi}$ gilt somit insgesamt

$$\frac{C_{A_f}(\sigma)}{\mathrm{OPT}(\sigma)} \geq \frac{q+\varphi}{q+1} \geq \frac{\sqrt{\varphi}+\varphi}{\sqrt{\varphi}+1} = \sqrt{\varphi}.$$

Damit ist gezeigt, dass der Algorithmus A_f für keine Wahl von f besser als strikt $\sqrt{\varphi}$-kompetitiv ist. Wieder können wir die Sequenz beliebig oft wiederholen, um zu zeigen, dass es keinen Algorithmus A_f gibt, der besser als $\sqrt{\varphi}$-kompetitiv ist. Auch dies gilt nur, weil wir ohne Einschränkung der Allgemeinheit davon ausgehen können, dass der Algorithmus A_f nach der Sequenz σ ein leeres Lager besitzt.

Bislang sind wir davon ausgegangen, dass f monoton fallend ist. Liegt dem Algorithmus A_f eine Lagerfunktion zugrunde, die nicht monoton ist, so können wir sie durch die monoton fallende Funktion

$$f^\star(x) = \sup\{f(y) \mid y \geq x\}$$

ersetzen. Auf der Sequenz Σ' verhalten sich die Algorithmen A_f und $A_{f^\star}$ identisch. Konstruieren wir dann die Sequenz Σ'' gemäß der Funktion $f^\star$, so überträgt sich die obige Analyse. Es ist eine Übung für den Leser, die Details auszuarbeiten. □

5.2.4 Ein optimaler Online-Algorithmus

In diesem Abschnitt lernen wir den optimalen Online-Algorithmus Threat kennen, der auf allen Eingaben einen kompetitiven Faktor von $r^\star$ erreicht. Es handelt sich dabei um einen Drohungs-Algorithmus, der stets davon ausgeht, dass nach dem aktuellen Schritt nur noch ein Schritt mit Preis φ und Verbrauch 1 kommt. Der Algorithmus kauft in jedem Schritt so wenig wie nötig, um zu garantieren, dass er $r^\star$-kompetitiv ist, falls diese Drohung eintritt. Auf den ersten Blick erinnert das an den Algorithmus $A^\star$ aus dem Beweis von Theorem 5.12 und tatsächlich verhält sich der Algorithmus Threat auf Sequenzen der Form Σ_p genauso wie der Algorithmus $A^\star$.

Optimaler Offline-Algorithmus

Um das Verhalten auf anderen Sequenzen definieren zu können, ist es zunächst wichtig, eine Beschreibung für die Kosten eines optimalen Offline-Algorithmus auf dem bisher gesehenen Teil der Eingabe zu finden. Sei $\sigma = (\sigma_1, \ldots, \sigma_n)$ eine beliebige Eingabe. Für jedes $i \in \{1, \ldots, n\}$ definieren wir eine Funktion $h_i : [0, \varphi] \to [0, 1]$. Für die Funktion h_i betrachten wir eine kostengünstigste Lösung, die Anfragen $\sigma_1, \ldots, \sigma_i$ zu bedienen und danach ein komplett volles Lager zu besitzen. Wir beschreiben mit der Funktion h_i, zu welchen Preisen der Lagerbestand, der am Ende vorhanden ist, gekauft wurde. Dabei gibt $1 - h_i(x)$ für jedes $x \in [1, \varphi]$ den Anteil des Lagerbestandes an, der zu einem Preis von x oder besser gekauft wurde.

Um dies zu formalisieren, definieren wir eine Funktion $h_0 : [0, \varphi] \to [0, 1]$ mit $h_0(x) = 1$ für $x < \varphi$ und $h_0(\varphi) = 0$. Wir geben nun eine Konstruktionsvorschrift von h_i an, wenn h_{i-1} bekannt ist. Diese folgt zwei Regeln.

1. Der aktuelle Verbrauch wird stets zu den bestmöglichen Konditionen bedient, d. h. wenn wir auf das Lager zurückgreifen, um den Verbrauch oder einen gewissen Anteil zu bedienen, so nehmen wir den Bestand, der am günstigsten eingekauft wurde.

2. Wenn Bestand im Lager vorhanden ist, der zu einem höheren Preis als dem aktuellen Preis p_i gekauft wurde, so wird er entfernt und in diesem Schritt zum Preis p_i ersetzt. Das bedeutet, dass nachträglich die Entscheidung rückgängig gemacht wird, den Bestand in einem der vorangegangenen Schritte zu kaufen. Dies ist nur möglich, da wir einen Offline-Algorithmus beschreiben.

Eine Konstruktionsvorschrift, die beide Regeln berücksichtigt, erhält man für $x \in [0, \varphi]$ durch

$$h_i(x) = \begin{cases} \min\{h_{i-1}(x) + v_i, 1\} & \text{falls } x \leq p_i, \\ 0 & \text{falls } x > p_i. \end{cases}$$

Diese Definition ist in den Abbildungen 5.2 und 5.3 veranschaulicht.

Unter der Annahme, dass der aktuelle Verbrauch immer zu den bestmöglichen Konditionen bedient wird, entsprechen die Kosten, die dem optimalen Offline-Algorithmus

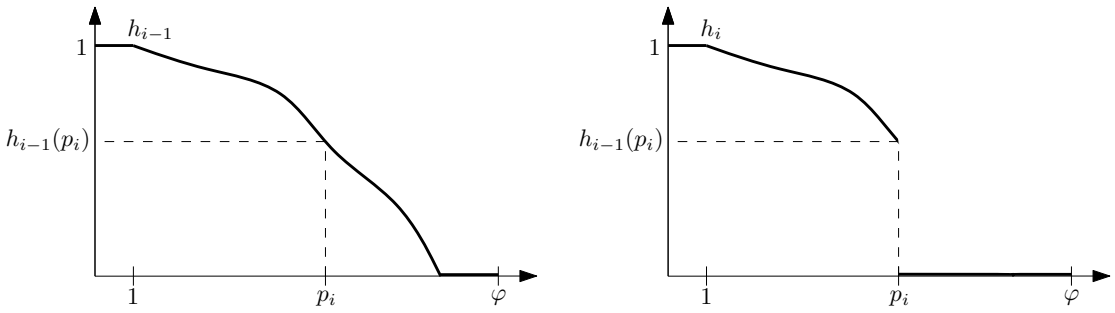


Abbildung 5.2: Konstruktion von h_i im Fall $v_i = 0$.

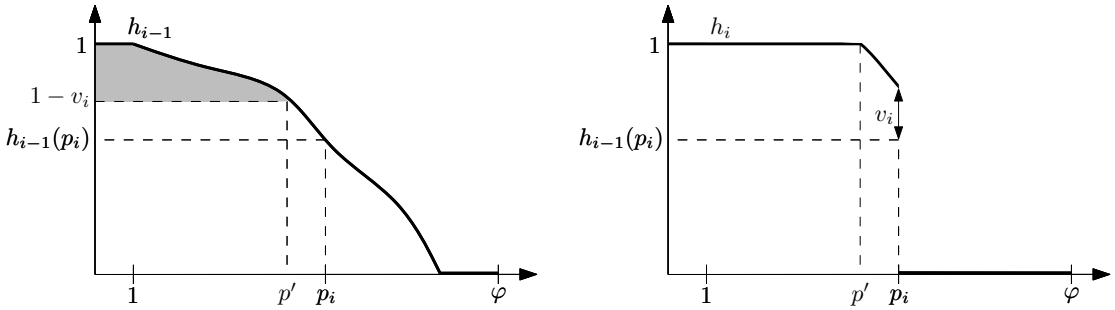


Abbildung 5.3: Konstruktion von h_i im Fall $v_i > 0$.

in Schritt i anfallen, um den Verbrauch v_i zu decken, intuitiv dem Flächeninhalt des grauen Bereichs in Abbildung 5.3. Deshalb definieren wir

$$C_i = \int_0^{p_i} \max\{h_{i-1}(x) + v_i - 1, 0\}\, dx.$$

Lemma 5.14. *Die Kosten eines optimalen Offline-Algorithmus auf der Eingabe σ entsprechen der Summe $\sum_{i=1}^{n} C_i$.*

Wir werden dieses Lemma nicht beweisen, da der Beweis relativ technisch ist. Die wesentliche Idee ist es, induktiv nachzuweisen, dass

$$A_y^i = \Big(\sum_{j=1}^{i} C_j\Big) + \int_0^{\varphi} \max\{h_i(x) + y - 1, 0\}\, dx$$

für jedes i und jedes $y \in [0, 1]$ die Kosten eines optimalen Offline-Algorithmus angibt, der die Anfragen $\sigma_1, \ldots, \sigma_i$ abarbeitet und danach einen Lagerbestand von genau y besitzt. Ist diese Eigenschaft gezeigt, so folgt das Lemma direkt, denn

$$\mathrm{OPT}(\sigma) = A_0^n = \sum_{j=1}^{n} C_j + \int_0^{\varphi} \max\{h_n(x) - 1, 0\}\, dx = \sum_{j=1}^{n} C_j.$$

Drohungs-Algorithmus

Jede Funktion h_i basiert nur auf den Schritten $\sigma_1, \ldots, \sigma_i$. Das heißt, der Drohungs-Algorithmus Threat darf in Schritt σ_i auf die Funktionen h_j mit $j \leq i$ zurückgreifen.

Der Algorithmus Threat kauft in Schritt σ_i mit Preis p_i und Verbrauch v_i genau

$$x_i = v_i + r^\star \cdot \int_1^{\varphi/r^\star} \frac{h_{i-1}(x) - h_i(x)}{\varphi - x}\, dx.$$

Mit einigen Rechnungen kann man nachweisen, dass x_i stets eine gültige Entscheidung ist, d. h. dass in keinem Schritt der Lagerbestand unter 0 sinkt oder über 1 steigt.

Um die Wahl von x_i zu begründen, vergleichen wir die angedrohten Eingaben $\Sigma_{i-1} = \left(\sigma_1, \ldots, \sigma_{i-1}, \binom{\varphi}{1}\right)$ und $\Sigma_i = \left(\sigma_1, \ldots, \sigma_i, \binom{\varphi}{1}\right)$. Der Einfachheit halber beschränken wir uns auf den Fall, dass $p_i \leq \varphi/r^\star$ gilt. In diesem Fall kann man nachweisen, dass für das gewählte x_i

$$\begin{aligned} C_{\text{Threat}}(\Sigma_i) - C_{\text{Threat}}(\Sigma_{i-1}) &= p_i \cdot x_i + \varphi \cdot (L_{i-1}^{\text{Threat}} - L_i^{\text{Threat}}) \\ &\leq r^\star \cdot \left(C_i + \int_0^{\varphi/r^\star} h_i(x) - h_{i-1}(x)\, dx\right) \end{aligned}$$

gilt. Daraus folgt mit Lemma 5.14, dass der Algorithmus Threat einen kompetitiven Faktor von $r^\star$ erreicht, denn

$$\begin{aligned} C_{\text{Threat}}(\Sigma_n) &= \sum_{i=1}^{n} \left(C_{\text{Threat}}(\Sigma_i) - C_{\text{Threat}}(\Sigma_{i-1})\right) + C_{\text{Threat}}(\Sigma_0) \\ &\leq \sum_{i=1}^{n} \left(r^\star \cdot \left(C_i + \int_0^{\varphi/r^\star} h_i(x) - h_{i-1}(x)\, dx\right)\right) + \varphi \\ &= r^\star \cdot \sum_{i=1}^{n} C_i + r^\star \cdot \int_0^{\varphi/r^\star} h_n(x) - h_0(x)\, dx + \varphi \\ &= r^\star \cdot \sum_{i=1}^{n} C_i + r^\star \cdot \int_0^{\varphi/r^\star} h_n(x) - 1\, dx + \varphi \\ &\leq r^\star \cdot \sum_{i=1}^{n} C_i + \varphi \\ &= r^\star \cdot \text{OPT}(\sigma) + \varphi. \end{aligned}$$

Da die Kosten von Threat auf der Eingabe σ nicht größer sind als die auf der Eingabe Σ_n, erhalten wir das folgende Theorem.

Theorem 5.15. *Der Algorithmus* Threat *erreicht einen kompetitiven Faktor von $r^\star$ für das Economical-Caching-Problem.*

Den vollständigen Beweis dieses Theorems werden wir in der Vorlesung nicht besprechen, da er relativ technisch ist.

Kapitel 6

Scheduling

Scheduling ist ein wichtiges Problem in der Informatik, bei dem es darum geht, Prozessen Ressourcen zuzuteilen. In dem Modell, das wir betrachten, ist eine Menge $J = \{1, \ldots, n\}$ von *Jobs* oder *Prozessen* gegeben, die auf eine Menge $M = \{1, \ldots, m\}$ von *Maschinen* oder *Prozessoren* verteilt werden muss. Jeder Job $j \in J$ besitzt dabei eine *Größe* $p_j \in \mathbb{R}_{>0}$ und jede Maschine $i \in M$ hat eine *Geschwindigkeit* $s_i \in \mathbb{R}_{>0}$. Wird ein Job $j \in J$ auf einer Maschine $i \in M$ ausgeführt, so werden dafür p_j/s_i Zeiteinheiten benötigt. Ein *Schedule* $\pi : J \to M$ ist eine Abbildung, die jedem Job eine Maschine zuweist, auf der er ausgeführt wird. Wir bezeichnen mit $L_i(\pi)$ die *Ausführungszeit* von Maschine $i \in M$ in Schedule π, d. h.

$$L_i(\pi) = \frac{\sum_{j \in J : \pi(j) = i} p_j}{s_i}.$$

Der *Makespan* $C(\pi)$ eines Schedules entspricht der Ausführungszeit derjenigen Maschine, die am längsten benötigt, die ihr zugewiesenen Jobs abzuarbeiten, d. h.

$$C(\pi) = \max_{i \in M} L_i(\pi).$$

Wir betrachten den Makespan als Zielfunktion, die es zu minimieren gilt.

In der Online-Variante dieses Problems werden die Jobs nacheinander präsentiert, und sobald ein neuer Job präsentiert wird, muss unwiderruflich entschieden werden, auf welcher Maschine er ausgeführt wird. Ein Online-Algorithmus kennt also von Anfang an die Menge M der Maschinen und die zugehörigen Geschwindigkeiten. Die Größen $p_1, p_2, \ldots$ werden nach und nach präsentiert und er muss Job j auf einer Maschine platzieren, sobald er die Größe p_j erfährt, ohne zu wissen wie die Größen $p_{j+1}, p_{j+2}, \ldots$ aussehen und wie viele Jobs noch kommen. Ein einmal platzierter Job kann nachträglich nicht mehr einer anderen Maschine zugewiesen werden.

Wir betrachten im Folgenden zunächst den Spezialfall *Online-Scheduling mit identischen Maschinen*. In diesem Spezialfall haben alle Maschinen $i \in M$ die Geschwindigkeit $s_i = 1$. Danach betrachten wir den Fall mit allgemeinen Geschwindigkeiten, den wir einfach als *Online-Scheduling* bezeichnen werden.

6.1 Identische Maschinen

Haben alle Maschinen die gleiche Geschwindigkeit, so erscheint es natürlich, jeden neuen Job der Maschine zuzuweisen, die momentan die kleinste Ausführungszeit besitzt. Diesen Greedy-Algorithmus nennen wir LEAST-LOADED-Algorithmus.

Theorem 6.1. *Der* LEAST-LOADED*-Algorithmus ist für Online-Scheduling mit identischen Maschinen strikt* $(2 - 1/m)$*-kompetitiv.*

Beweis. Sei eine beliebige Eingabe für Online-Scheduling mit identischen Maschinen gegeben. Wie immer bei Approximations- und Online-Algorithmen beruht der Beweis auf einer unteren Schranke für den Makespan des optimalen Schedules π^*. In diesem Fall haben wir sogar zwei untere Schranken:

$$C(\pi^*) \geq \frac{1}{m} \sum_{j \in J} p_j \qquad \text{und} \qquad C(\pi^*) \geq \max_{j \in J} p_j.$$

Die erste Schranke basiert auf der Beobachtung, dass die durchschnittliche Ausführungszeit der Maschinen in jedem Schedule gleich $\frac{1}{m} \sum_{j \in J} p_j$ ist. Deshalb muss es in jedem Schedule (also auch in π^*) eine Maschine geben, deren Ausführungszeit mindestens diesem Durchschnitt entspricht. Die zweite Schranke basiert auf der Beobachtung, dass die Maschine, der der größte Job zugewiesen ist, eine Ausführungszeit von mindestens $\max_{j \in J} p_j$ besitzt.

Wir betrachten nun den Schedule π, den der LEAST-LOADED-Algorithmus berechnet. In diesem Schedule sei $i \in M$ eine Maschine mit größter Ausführungszeit. Es gilt also $C(\pi) = L_i(\pi)$. Es sei $j \in J$ der Job, der als letztes Maschine i hinzugefügt wurde. Zu dem Zeitpunkt, zu dem diese Zuweisung erfolgt ist, war Maschine i die Maschine mit der kleinsten Ausführungszeit. Die Jobs $1, \ldots, j-1$ waren bereits verteilt und dementsprechend muss es eine Maschine gegeben haben, deren Ausführungszeit höchstens dem Durchschnitt $\frac{1}{m} \sum_{k=1}^{j-1} p_k$ entsprach. Wir können den Makespan von π nun wie folgt abschätzen:

$$\begin{aligned}
C(\pi) = L_i(\pi) &\leq \frac{1}{m} \left(\sum_{k=1}^{j-1} p_k \right) + p_j \leq \frac{1}{m} \left(\sum_{k \in J \setminus \{j\}} p_k \right) + p_j \\
&= \frac{1}{m} \left(\sum_{k \in J} p_k \right) + \left(1 - \frac{1}{m} \right) p_j \\
&\leq \frac{1}{m} \left(\sum_{k \in J} p_k \right) + \left(1 - \frac{1}{m} \right) \cdot \max_{k \in J} p_k \\
&\leq C(\pi^*) + \left(1 - \frac{1}{m} \right) C(\pi^*) = \left(2 - \frac{1}{m} \right) \cdot C(\pi^*),
\end{aligned}$$

wobei wir die beiden oben angegebenen unteren Schranken für $C(\pi^*)$ benutzt haben. □

Das folgende Beispiel zeigt, dass der Least-Loaded-Algorithmus im Worst-Case nicht besser als $(2 - 1/m)$-kompetitiv ist.

Untere Schranke für den Least-Loaded-Algorithmus

Sei eine Anzahl m an Maschinen vorgegeben. Wir betrachten eine Instanz mit $n = m(m-1)+1$ vielen Jobs. Die ersten $m(m-1)$ Jobs haben jeweils eine Größe von 1 und der letzte Job hat eine Größe von m. Der optimale Offline-Algorithmus verteilt die ersten $m(m-1)$ Jobs gleichmäßig auf den Maschinen $1, \ldots, m-1$ und platziert den letzten Job auf Maschine m. Dann haben alle Maschinen eine Ausführungszeit von genau m. Der Least-Loaded-Algorithmus hingegen verteilt die ersten $m(m-1)$ Jobs gleichmäßig auf den Maschinen $1, \ldots, m$ und platziert den letzten Job auf einer beliebigen Maschine i. Diese Maschine hat dann eine Ausführungszeit von $(m-1)+m = 2m-1$. Es gilt

$$\frac{2m-1}{m} = 2 - \frac{1}{m}.$$

Damit ist gezeigt, dass der Least-Loaded-Algorithmus im Allgemeinen keinen besseren strikten kompetitiven Faktor als $2 - 1/m$ erreicht.

Es ist bekannt, dass der Least-Loaded-Algorithmus für $m = 2$ und $m = 3$ der optimale Online-Algorithmus ist. Für mehr als drei Maschinen ist das aber nicht mehr der Fall. So gibt es beispielsweise für den Fall $m = 4$ einen Algorithmus, der einen kompetitiven Faktor von $1{,}7333 < 2 - 1/4$ erreicht. Außerdem gibt es einen Algorithmus, der für jedes m einen kompetitiven Faktor von 1,9201 erreicht, und man weiß, dass kein deterministischer Online-Algorithmus existiert, der für alle m einen Faktor besser als 1,88 erreicht. Auch dieses einfache Scheduling-Problem ist also noch nicht vollständig gelöst, und es ist nicht klar, ob es für allgemeine m einen besseren Online-Algorithmus als den o. g. 1,9201-kompetitiven gibt. Der interessierte Leser sei für weitere Details auf die Übersichtsartikel von Jiri Sgall [21] und Yossi Azar [2] verwiesen.

Da der Least-Loaded-Algorithmus und der o. g. 1,9201-kompetitive Algorithmus effizient sind, handelt es sich insbesondere um Approximationsalgorithmen für die Offline-Variante des Scheduling-Problems mit identischen Maschinen. Diese Offline-Variante ist NP-schwer, was durch eine einfache Reduktion von Partition gezeigt werden kann. Somit liegt die Frage nahe, ob es effiziente Offline-Algorithmen gibt, die einen besseren Approximationsfaktor als 1,9201 erreichen. Wir beantworten diese Frage positiv und betrachten einen einfachen Offline-Algorithmus, der eine $\frac{4}{3}$-Approximation erreicht.

Longest-Processing-Time **(LPT)**

1. Sortiere die Jobs so, dass $p_1 \geq p_2 \geq \ldots \geq p_n$ gilt.
2. Führe den Least-Loaded-Algorithmus auf den so sortierten Jobs aus.

Es ist klar, dass der LPT-Algorithmus kein Online-Algorithmus ist, da er zunächst alle Jobs kennen muss, um sie bezüglich ihrer Größe zu sortieren. Diese Sortierung verbessert den Approximationsfaktor allerdings deutlich.

Theorem 6.2. *Der* Longest-Processing-Time*-Algorithmus ist ein $\frac{4}{3}$-Approximationsalgorithmus für Scheduling mit identischen Maschinen.*

Beweis. Wir führen einen Widerspruchsbeweis und nehmen an, es gäbe eine Eingabe mit m Maschinen und n Jobs mit Größen $p_1 \geq \ldots \geq p_n$, auf der der LPT-Algorithmus einen Schedule π berechnet, dessen Makespan mehr als $\frac{4}{3}$-mal so groß ist wie der Makespan des optimalen Schedules π^*. Außerdem gehen wir davon aus, dass wir eine Instanz mit der kleinstmöglichen Anzahl an Jobs gewählt haben, auf der der LPT-Algorithmus keine $\frac{4}{3}$-Approximation liefert.

Es sei nun $i \in M$ eine Maschine, die in Schedule π die größte Ausführungszeit besitzt, und es sei $j \in J$ der letzte Job, der vom LPT-Algorithmus Maschine i zugewiesen wird. Es gilt $j = n$, da ansonsten $p_1, \ldots, p_j$ eine Eingabe mit weniger Jobs ist, auf der der LPT-Algorithmus keine $\frac{4}{3}$-Approximation liefert. Genauso wie im Beweis von Theorem 6.1 können wir argumentieren, dass es zum Zeitpunkt der Zuweisung von Job n eine Maschine mit Ausführungszeit höchstens $\frac{1}{m}\left(\sum_{k=1}^{n-1} p_k\right) \leq \mathrm{OPT}(\sigma)$ gibt. Da wir Job n der Maschine i mit der bisher kleinsten Ausführungszeit zuweisen, gilt

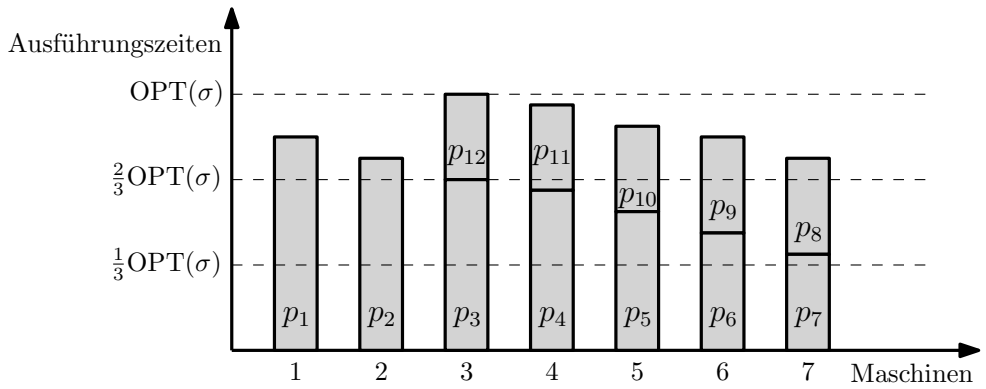
$$C(\pi) = L_i(\pi) \leq \frac{1}{m}\left(\sum_{k=1}^{n-1} p_k\right) + p_n \leq \mathrm{OPT}(\sigma) + p_n.$$

Der Makespan von π kann also nur dann größer als $4\mathrm{OPT}(\sigma)/3$ sein, wenn $p_n > \mathrm{OPT}(\sigma)/3$ gilt. Wegen der Sortierung der Jobs gilt $p_1 \geq \ldots \geq p_n$ und somit gilt $p_j > \mathrm{OPT}(\sigma)/3$ dann für alle Jobs $j \in J$.

Das bedeutet, dass in jedem optimalen Schedule jeder Maschine höchstens zwei Jobs zugewiesen sein können. Für Eingaben, die diese Bedingung erfüllen, gilt insbesondere $n \leq 2m$ und man kann einen optimalen Schedule explizit angeben:

- Jeder Job $j \in \{1, \ldots, \min\{n, m\}\}$ wird Maschine j zugewiesen.
- Jeder Job $j \in \{m+1, \ldots, n\}$ wird Maschine $2m - j + 1$ zugewiesen.

Dieser Schedule ist in folgender Abbildung dargestellt. Den einfachen Beweis, dass



dies wirklich ein optimaler Schedule ist, falls alle Jobs echt größer als $\frac{1}{3}\mathrm{OPT}(\sigma)$ sind, überlassen wir dem Leser.

Wir zeigen nun, dass der optimale Schedule, den wir gerade beschrieben haben, dem Schedule entspricht, den der LPT-Algorithmus berechnet: Zunächst ist klar, dass die ersten $\min\{n, m\}$ Jobs jeweils einer eigenen Maschine zugewiesen werden, und wir deshalb ohne Beschränkung der Allgemeinheit davon ausgehen können, dass jeder Job $j \in \{1, \ldots, \min\{n, m\}\}$ Maschine j zugewiesen wird. Wir betrachten nun die Zuweisung eines Jobs $j \in \{m+1, \ldots, n\}$ und gehen induktiv davon aus, dass die Jobs aus $\{1, \ldots, j-1\}$ bereits so wie im oben beschriebenen optimalen Schedule zugewiesen wurden. Für eine Maschine $i \in M$ bezeichnen wir mit L_i die Ausführungszeit, die durch die bisher zugewiesenen Jobs verursacht wird. Im optimalen Schedule wird Job j Maschine $2m - j + 1$ zugewiesen. Deshalb gilt $L_{2m-j+1} + p_j \leq \mathrm{OPT}(\sigma)$. Ferner gilt $L_1 \geq L_2 \geq \ldots \geq L_{2m-j+1}$. Alle Maschinen $i \in \{2m - j + 2, \ldots, m\}$ haben bereits zwei Jobs zugewiesen. Da alle Jobs größer als $\frac{1}{3}\mathrm{OPT}(\sigma)$ sind, folgt für diese Maschinen $L_i + p_j > \mathrm{OPT}(\sigma)$ und damit $L_i > L_{2m-j+1}$. Damit ist gezeigt, dass Maschine $2m-j+1$ bei der Zuweisung von Job j tatsächlich eine Maschine mit kleinster Ausführungszeit ist.

Damit haben wir einen Widerspruch zu der Annahme, dass der LPT-Algorithmus auf der gegebenen Eingabe keine $\frac{4}{3}$-Approximation berechnet. Wir haben gezeigt, dass der LPT-Algorithmus auf Eingaben, in denen alle Jobs echt größer als $\frac{1}{3}\mathrm{OPT}(\sigma)$ sind, den optimalen Schedule berechnet. Für alle anderen Eingaben haben wir bewiesen, dass der LPT-Algorithmus eine $\frac{4}{3}$-Approximation berechnet. □

Ist der LPT-Algorithmus der bestmögliche Approximationsalgorithmus für das Scheduling-Problem mit identischen Maschinen? Die Antwort auf diese Frage lautet nein, denn es gibt ein polynomielles Approximationsschema für dieses Problem, d. h. man kann den optimalen Schedule in polynomieller Zeit beliebig gut approximieren.

6.2 Maschinen mit Geschwindigkeiten

Wir betrachten nun den Fall, dass die Maschinen verschiedene Geschwindigkeiten haben können. Als erstes betrachten wir wieder einen Greedy-Algorithmus à la Least-Loaded. Es gibt zwei Varianten dieses Algorithmus. Wird ein neuer Job präsentiert, so weisen wir ihn entweder der Maschine zu, die *derzeit* die kleinste Ausführungszeit besitzt, oder der Maschine, die *nach der Zuweisung des neuen Jobs* die kleinste Ausführungszeit besitzt.

Bei identischen Maschinen sind diese beiden Alternativen äquivalent, bei Maschinen mit verschiedenen Geschwindigkeiten sind sie es nicht mehr, wie das folgende einfache Beispiel zeigt. Sei $s_1 = 3$, $s_2 = 1$, $p_1 = p_2 = 3$ und sei der erste Jobs bereits Maschine 1 zugewiesen. Dann hat Maschine 2 derzeit die kleinste Ausführungszeit (0 statt 1), Maschine 1 hat jedoch nach der Zuweisung des neuen Jobs die kleinste Ausführungszeit (2 statt 3). Dieses Beispiel zeigt auch direkt, dass die erste Alternative, nur die derzeitige Ausführungszeit zu berücksichtigen, wenig sinnvoll ist. Wir hätten die erste Maschine in diesem Beispiel beliebig viel schneller machen können als die zweite und somit hätte der Algorithmus eine beliebig schlechte Lösung geliefert.

Es besteht also nur die Hoffnung, dass der Greedy-Algorithmus, der die Ausführungszeit nach der Zuweisung des neuen Jobs betrachtet, einen endlichen kompetitiven Faktor erreicht. Man kann jedoch zeigen, dass der kompetitive Faktor dieses Algorithmus nur $\Theta(\log m)$ beträgt.

Wir lernen jetzt einen Algorithmus mit einem konstanten kompetitiven Faktor kennen. Um diesen zu beschreiben, betrachten wir zunächst einen mit $\alpha \in \mathbb{R}_{>0}$ parametrisierten Online-Algorithmus SlowFit(α), der auf allen Eingaben σ mit $\mathrm{OPT}(\sigma) \leq \alpha$ einen Schedule π mit $C(\pi) \leq 2\alpha$ berechnet. Wäre uns bei einer Eingabe σ also bereits von Anfang an der Makespan $\mathrm{OPT}(\sigma)$ des optimalen Schedules bekannt, so könnten wir entsprechend den Algorithmus SlowFit($\mathrm{OPT}(\sigma)$) ausführen und würden so einen 2-kompetitiven Online-Algorithmus erhalten. Das Problem ist aber natürlich, dass wir zu Beginn nicht wissen, wie groß $\mathrm{OPT}(\sigma)$ tatsächlich ist. Bevor wir dieses Problem lösen, stellen wir zunächst den Algorithmus SlowFit(α) vor.

SlowFit(α)

Wird ein neuer Job $j \in J$ der Größe p_j präsentiert, so weise ihn der langsamsten Maschine $i \in M$ zu, die nach der Zuweisung von Job j eine Ausführungszeit von höchstens 2α besitzt. Falls keine solche Maschine existiert, gib eine Fehlermeldung aus.

Um diesen Algorithmus formaler zu beschreiben, gehen wir davon aus, dass die Maschinen aufsteigend gemäß ihrer Geschwindigkeiten sortiert sind, d. h. $s_1 \leq s_2 \leq \ldots \leq s_m$. Für $j \in J$ bezeichnen wir den (partiellen) Schedule, den SlowFit(α) für die Jobs $1, \ldots, j$ berechnet, mit π_j. Für $i \in M$ und $j \in J$ bezeichnet $L_i(\pi_j)$ dann entsprechend die Ausführungszeit von Maschine i zu dem Zeitpunkt, zu dem die Jobs $1, \ldots, j$ zugewiesen sind. Der Algorithmus SlowFit(α) weist einen Job $j \in J$ der Maschine

$$\min\{i \in M \mid L_i(\pi_{j-1}) + p_j/s_i \leq 2\alpha\}$$

zu.

Lemma 6.3. *Es sei $\alpha \in \mathbb{R}_{>0}$ beliebig und σ sei eine beliebige Eingabe für Online-Scheduling mit $\mathrm{OPT}(\sigma) \leq \alpha$. Dann gibt der Algorithmus* SlowFit(α) *auf der Eingabe σ keine Fehlermeldung aus und berechnet einen Schedule π mit $C(\pi) \leq 2\alpha$.*

Beweis. Falls SlowFit(α) keine Fehlermeldung ausgibt, so berechnet er per Definition einen Schedule π mit $C(\pi) \leq 2\alpha$. Somit müssen wir nur zeigen, dass er auf der Eingabe σ keine Fehlermeldung ausgibt. Wir führen einen Widerspruchsbeweis und gehen davon aus, dass die Eingabe σ aus n Jobs mit Größen $p_1, \ldots, p_n$ besteht und dass die Fehlermeldung erst beim letzten Job p_n ausgegeben wird. Dies können wir ohne Beschränkung der Allgemeinheit annehmen, denn wird die Fehlermeldung bereits früher ausgegeben, so können wir die nachfolgenden Jobs einfach aus der Eingabe σ entfernen.

Zunächst halten wir fest, dass $L_i(\pi_{n-1}) > \mathrm{OPT}(\sigma)$ nicht für alle Maschinen $i \in M$ gelten kann, denn ansonsten wäre

$$\sum_{j=1}^{n-1} p_j = \sum_{i \in M} (s_i \cdot L_i(\pi_{n-1})) > \sum_{i \in M} (s_i \cdot \mathrm{OPT}(\sigma)) \geq \sum_{i \in M} (s_i \cdot L_i(\pi^*)) = \sum_{j=1}^{n} p_j,$$

wobei π^* einen optimalen Schedule bezeichnet. Bei dieser Rechnung haben wir ausgenutzt, dass sich die Gesamtgröße der Jobs, die Maschine i in Schedule π_{n-1} zugewiesen sind, als $s_i \cdot L_i(\pi_{n-1})$ schreiben lässt. Außerdem haben wir ausgenutzt, dass die Gesamtgröße der Jobs, die Maschine i in einem optimalen Schedule π^* zugewiesen sind, nicht größer als $s_i \cdot \mathrm{OPT}(\sigma)$ sein kann.

Wir betrachten nun die schnellste Maschine $f \in M$, für die $L_f(\pi_{n-1}) \leq \mathrm{OPT}(\sigma)$ gilt, d. h.

$$f = \max\{i \in M \mid L_i(\pi_{n-1}) \leq \mathrm{OPT}(\sigma)\}.$$

Da m die schnellste Maschine ist, muss $f < m$ gelten. Denn wäre $L_m(\pi_{n-1}) \leq \mathrm{OPT}(\sigma)$, dann wäre

$$L_m(\pi_{n-1}) + p_n/s_m \leq 2 \cdot \mathrm{OPT}(\sigma) \leq 2\alpha$$

und somit hätte SlowFit(α) keine Fehlermeldung ausgegeben. Wir haben bei dieser Ungleichung ausgenutzt, dass p_n/s_m eine untere Schranke für $\mathrm{OPT}(\sigma)$ ist, da m die schnellste Maschine ist.

Wir setzen $\Gamma = \{i \in M \mid i > f\}$. Dann haben alle Maschinen in Γ im Schedule π_{n-1} eine Ausführungszeit größer als $\mathrm{OPT}(\sigma)$. Wegen $f < m$ gilt außerdem $\Gamma \neq \emptyset$. Diese Notationen und Definitionen sind in Abbildung 6.1 noch einmal dargestellt.

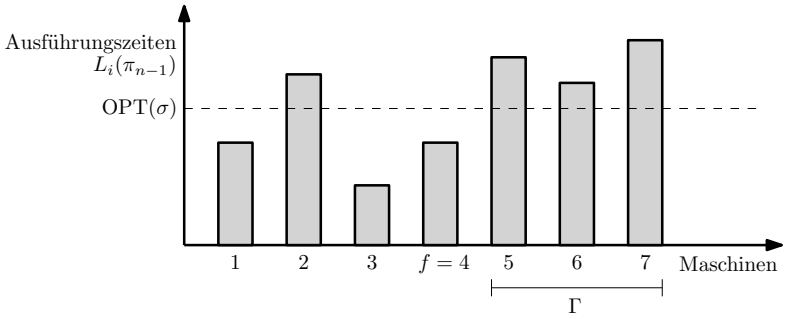


Abbildung 6.1: Illustration der Definitionen im Beweis von Lemma 6.3.

Die Gesamtgröße aller Jobs, die in Schedule π_{n-1} Maschinen aus Γ zugewiesen sind, beträgt

$$\sum_{i \in \Gamma}(s_i \cdot L_i(\pi_{n-1})) > \sum_{i \in \Gamma}(s_i \cdot \mathrm{OPT}(\sigma)).$$

Die rechte Seite dieser Ungleichung ist eine obere Schranke für die Gesamtgröße der Jobs, die in einem optimalen Schedule Maschinen aus Γ zugewiesen sind. Somit muss es einen Job $j \in J \setminus \{n\}$ mit $\pi_{n-1}(j) \in \Gamma$ geben, der in einem optimalen Schedule einer Maschine $i \notin \Gamma$, also $i \leq f$, zugewiesen wird. Für diesen Job j und diese Maschine i muss demnach $p_j/s_i \leq \mathrm{OPT}(\sigma)$ gelten. Wegen $i \leq f$ und der Sortierung der Maschinen gemäß ihrer Geschwindigkeiten muss somit auch $p_j/s_f \leq \mathrm{OPT}(\sigma)$ gelten. Betrachten wir nun den Zeitpunkt, zu dem Job j eingefügt wurde. Zu diesem Zeitpunkt waren die Jobs $1, \ldots, j-1$ gemäß des partiellen Schedules π_{j-1} verteilt. Es gilt

$$L_f(\pi_{j-1}) + p_j/s_f \leq L_f(\pi_{n-1}) + p_j/s_f \leq \mathrm{OPT}(\sigma) + \mathrm{OPT}(\sigma) \leq 2\alpha.$$

Somit hätte zu diesem Zeitpunkt Job j Maschine f zugewiesen werden dürfen. Stattdessen wurde er aber einer Maschine $i' \in \Gamma$ zugewiesen. Wegen $f < i'$ ist dies ein

Widerspruch zur Definition des Algorithmus. Damit ist gezeigt, dass der Algorithmus auf der Eingabe σ keine Fehlermeldung ausgibt. □

Wir müssen nun die Frage klären, wie wir den Algorithmus SlowFit(α) nutzen können, ohne das richtige α zu kennen. Wir werden dazu einfach den Makespan der optimalen Lösung schätzen und unsere Schätzung im Laufe des Algorithmus gegebenenfalls anpassen. Wir fangen mit einer Schätzung α an und jedes Mal, wenn SlowFit(α) eine Fehlermeldung ausgibt, verdoppeln wir unsere Schätzung. Formal ist der Algorithmus SlowFit wie folgt definiert.

SlowFit

1. Setze $\alpha_0 = p_1/s_m$.
2. Beginne mit Phase $k = 0$.
3. Für jeden Job $j = 1, \ldots, n$ führe folgende Anweisungen durch:
4. Versuche, Job j mit SlowFit(α_k) einer Maschine zuzuweisen.
   Ignoriere dabei alle Jobs, die in früheren Phasen zugewiesen wurden.
5. Gibt SlowFit(α_k) keine Fehlermeldung aus, so übernimm die Zuweisung.
6. Ansonsten erhöhe k um eins, setze $\alpha_k = 2^k\alpha_0$ und gehe zurück zu Schritt 4.

Theorem 6.4. *Der Algorithmus* SlowFit *ist ein strikt* 8*-kompetitiver Algorithmus für Online-Scheduling.*

Beweis. Es sei σ eine beliebige Eingabe, auf der der Algorithmus SlowFit die Phasen $0, 1, \ldots, h$ durchläuft. Wir bezeichnen mit σ_k die Teilsequenz der Jobs, die in Phase k zugewiesen werden. Wir können mit Lemma 6.3 einen Rückschluss auf den Makespan des optimalen Schedules ziehen. Gilt $h = 0$, so beträgt der Makespan des Schedules π, den der Algorithmus berechnet, höchstens $2\alpha_0$. Da α_0 eine untere Schranke für $\mathrm{OPT}(\sigma)$ ist, liefert der Algorithmus in diesem Fall sogar eine 2-Approximation. Wir betrachten nun den Fall $h > 0$. Dazu betrachten wir Phase $h - 1$ und den ersten Job $j \in J$, der zu Phase h gehört. Da in Phase $h - 1$ alle Jobs ignoriert werden, die bereits in früheren Phasen zugewiesen wurden, erzeugt der Algorithmus SlowFit(α_{h-1}) gemäß Lemma 6.3 nur dann bei der Einfügung von Job j eine Fehlermeldung, wenn für die Teilsequenz $\sigma_{h-1}j$ (alle Jobs aus Phase $h - 1$ gefolgt von Job j) gilt

$$\mathrm{OPT}(\sigma_{h-1}j) > \alpha_{h-1} = 2^{h-1}\alpha_0.$$

Nun schätzen wir den Makespan des Schedules π ab, den der Algorithmus SlowFit berechnet. Dazu betrachten wir alle Phasen $k = 0, \ldots, h$ und addieren jeweils den Makespan auf, der durch die Jobs verursacht wird, die in Phase k zugewiesen werden. Aus Lemma 6.3 folgt, dass der Makespan, der durch die Jobs in Phase k verursacht wird, höchstens $2\alpha_k$ beträgt. Damit ergibt sich insgesamt

$$C(\pi) \leq \sum_{k=0}^{h} 2\alpha_k = 2\sum_{k=0}^{h} 2^k\alpha_0 = 2(2^{h+1} - 1)\alpha_0 \leq 2^{h+2}\alpha_0.$$

Damit ist gezeigt, dass

$$C(\pi) \leq 2^{h+2}\alpha_0 = 8 \cdot 2^{h-1}\alpha_0 < 8 \cdot \mathrm{OPT}(\sigma_{h-1}j) \leq 8 \cdot \mathrm{OPT}(\sigma)$$

gilt. □

Der Algorithmus SlowFit ist nicht der beste bekannte Online-Algorithmus für das betrachtete Scheduling-Problem. Es ist ein Algorithmus bekannt, der einen kompetitiven Faktor von 5,828 erreicht, und es ist bekannt, dass kein deterministischer Online-Algorithmus existiert, der einen Faktor kleiner als 2,438 erreicht. Auch dieses grundlegende Scheduling-Problem ist also noch nicht vollständig gelöst. Der interessierte Leser sei für weitere Details wieder auf die Übersichtsartikel von Jiri Sgall [21] und Yossi Azar [2] verwiesen.

Kapitel 7

Das Online-Matching-Problem

Matching-Probleme sind ein klassisches Themengebiet der kombinatorischen Optimierung. Ein Matching in einem Graphen $G = (V, E)$ ist eine Teilmenge $M \subseteq E$ der Kanten, sodass keine zwei Kanten aus M einen gemeinsamen Endknoten besitzen. Die Aufgabe besteht darin, ein *maximales Matching*, also ein Matching mit so vielen Kanten wie möglich, zu finden. Für die Lösung dieses Problems und vieler Problemvarianten, in denen die Kanten beispielsweise verschiedene Gewichte haben dürfen, sind effiziente Algorithmen bekannt.

Wir betrachten in dieser Vorlesung das *Online-Matching-Problem*. Dabei konzentrieren wir uns auf den Fall, dass die Eingabe aus einem bipartiten Graphen $G = (V \cup U, E)$ besteht. In einem solchen Graphen besitzt jede Kante jeweils einen Endknoten aus V und U. Während die Knoten aus V einem Online-Algorithmus von vornherein bekannt sind, werden die Knoten aus U Schritt für Schritt aufgedeckt. Sobald ein Knoten $u \in U$ aufgedeckt wird, werden auch alle zu ihm inzidenten Kanten präsentiert. Ein Online-Algorithmus startet mit einem leeren Matching $M = \emptyset$ und muss direkt nach dem Aufdecken eines Knotens u entscheiden, ob er dem aktuellen Matching M eine zu u inzidente Kante hinzufügt und wenn ja, welche. Dabei können nur solche Kanten hinzugefügt werden, die mit keiner bislang im Matching enthaltenen Kante einen Endknoten teilen. Dies kann insbesondere dazu führen, dass dem aktuellen Matching keine Kante hinzugefügt werden darf. Fügt der Algorithmus dem Matching die Kante (v, u) hinzu, so sagen wir, dass die Knoten v und u einander zugewiesen werden.

Die Untersuchung des Online-Matching-Problems geht auf Karp, Vazirani und Vazirani zurück [11]. Die Präsentation in diesem Kapitel beruht in Teilen auf einem Artikel von Birnbaum und Mathieu [5].

7.1 Deterministische Algorithmen

Zunächst gehen wir der Frage nach, wie gut das Online-Matching-Problem durch deterministische Algorithmen gelöst werden kann. Dazu betrachten wir einen einfachen Greedy-Algorithmus, der jeden ankommenden Knoten aus U einem beliebigen freien Nachbarn aus V zuweist, sofern ein solcher existiert.

Theorem 7.1. *Jeder Greedy-Algorithmus, der nach dem oben beschriebenen Prinzip arbeitet, ist strikt 2-kompetitiv für das Online-Matching-Problem.*

Beweis. Jeder Greedy-Algorithmus berechnet ein *nicht erweiterbares Matching* $M \subseteq E$. Dabei handelt es sich um ein Matching, dem keine Kante hinzugefügt werden kann. Formal heißt ein Matching M nicht erweiterbar, wenn $M \cup \{e\}$ für keine Kante $e \in E \setminus M$ ein gültiges Matching ist.[1] Dies kann durch einen einfachen Widerspruchsbeweis gezeigt werden: Angenommen, es gäbe eine Kante $e = (v, u) \in E \setminus M$ mit $v \in V$ und $u \in U$, für die $M \cup \{e\}$ ein Matching ist. Dann besitzen die Knoten v und u in M keine inzidenten Kanten. Insbesondere wurde der Knoten u bei seiner Ankunft vom Greedy-Algorithmus keinem Knoten aus V zugewiesen, obwohl der adjazente Knoten v frei war. Dies ist ein Widerspruch zum Verhalten des Greedy-Algorithmus.

Zum Beweis des Theorems fehlt nun nur noch eine einfache Aussage aus der Graphentheorie: Es sei M ein beliebiges nicht erweiterbares Matching und es sei $M^\star$ ein maximales Matching. Dann gilt $|M^\star| \leq 2|M|$. Wir bezeichnen mit $V(M) \subseteq V \cup U$ und $V(M^\star) \subseteq V \cup U$ die Knoten, die in M bzw. $M^\star$ eine inzidente Kante besitzen. Es gilt dann $|V(M)| = 2|M|$ und $|V(M^\star)| = 2|M^\star|$. Außerdem besitzt für jede Kante $e \in M^\star$ mindestens einer der beiden Endknoten eine inzidente Kante in M, da ansonsten $M \cup \{e\}$ ein Matching wäre. Demzufolge gilt $|V(M)| \geq |M^\star|$. Insgesamt folgt damit

$$|M| = \frac{|V(M)|}{2} \geq \frac{|M^\star|}{2},$$

womit das Theorem bewiesen ist. □

Als nächstes zeigen wir, dass Greedy-Algorithmen optimale deterministische Online-Algorithmen für das Online-Matching-Problem sind.

Theorem 7.2. *Es gibt keinen deterministischen Online-Algorithmus für das Online-Matching-Problem, der einen besseren kompetitiven Faktor als 2 erreicht.*

Beweis. Es sei A ein beliebiger deterministischer Online-Algorithmus für das Online-Matching-Problem. Wir betrachten zunächst einen bipartiten Graphen $G = (V \cup U, E)$ mit jeweils zwei Knoten auf beiden Seiten. Sei also $V = \{v_1, v_2\}$ und $U = \{u_1, u_2\}$. Zunächst wird Knoten u_1 aufgedeckt, der sowohl zu v_1 als auch zu v_2 benachbart ist. Da der Algorithmus A deterministisch ist, wird er beim Aufdecken von u_1 stets dieselbe Entscheidung treffen. Entweder weist er den Knoten u_1 keinem Knoten zu oder er weist ihn Knoten v_i für ein $i \in \{1, 2\}$ zu.

Abhängig von dem Verhalten des Algorithmus A können wir die Nachbarschaft des Knoten u_2 wählen. Entscheidet A sich dazu, den Knoten u_1 keinem Knoten aus V zuzuweisen, so besitzt u_2 keine inzidenten Kanten. Entscheidet A sich dazu, den Knoten u_1 dem Knoten v_i zuzuweisen, so besitzt u_2 genau einen adjazenten Knoten, nämlich ebenfalls v_i. Bezeichnen wir mit M das Matching, das Algorithmus A berechnet, und

[1] In englischen Texten werden maximale Matchings als *maximum matchings* und nicht erweiterbare Matchings als *maximal matchings* bezeichnet.

mit $M^\star$ ein maximales Matching, so gilt im ersten Fall $|M| = 0$ sowie $|M^\star| = 1$ und im zweiten Fall gilt $|M| = 1$ sowie $|M^\star| = 2$. In jedem Falle gilt $\frac{|M|}{|M^\star|} \leq \frac{1}{2}$.

Dies zeigt bereits, dass es keinen Algorithmus gibt, der einen strikt kompetitiven Faktor kleiner als 2 erreicht. Um auch einen nicht strikt kompetitiven Faktor kleiner als 2 ausschließen zu können, konstruieren wir einen Graphen, der aus mehreren Wiederholungen der obigen Konstruktion besteht. Sei dazu $\ell \in \mathbb{N}$ beliebig und sei $V = \{v_1, \ldots, v_{2\ell}\}$ und $U = \{u_1, \ldots, u_{2\ell}\}$. Der durch die Knoten $\{v_{2i-1}, v_{2i}, u_{2i-1}, u_{2i}\}$ induzierte Teilgraph ist für jedes $i \in \{1, \ldots, \ell\}$ so konstruiert wie die oben beschriebe Eingabe. Dabei kann sich der Algorithmus zwar auf jedem dieser Teilgraphen unterschiedlich verhalten, wir haben aber bereits oben argumentiert, dass stets $\frac{|M|}{|M^\star|} \leq \frac{1}{2}$ gilt, wobei M und $M^\star$ wieder das von Algorithmus A berechnete bzw. ein maximales Matching bezeichnen. Außerdem gilt $|M^\star| \geq \ell$. Damit folgt das Theorem analog zu den unteren Schranken aus Abschnitt 2.1.2, da wir ℓ beliebig groß wählen können. □

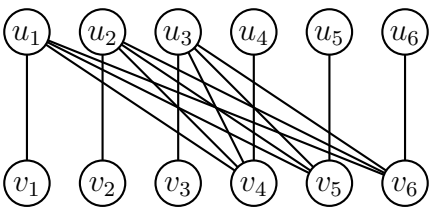
7.2 Randomisierte Algorithmen

Nachdem wir den optimalen kompetitiven Faktor für deterministische Algorithmen beim Online-Matching-Problem im letzten Abschnitt exakt bestimmt haben, liegt nun die Frage auf der Hand, ob randomisierte Algorithmen einen besseren kompetitiven Faktor erreichen können. Diese Frage werden wir positiv beantworten.

Wir bezeichnen mit RANDOM den randomisierten Greedy-Algorithmus, der bei dem Aufdecken eines Knotens $u \in U$ uniform zufällig einen Knoten $v \in V$ aus der Menge aller noch freien Nachbarn von u wählt und dann u und v einander zuweist. Da es sich bei diesem Algorithmus um einen speziellen Greedy-Algorithmus handelt, folgt aus Theorem 7.1, dass RANDOM 2-kompetitiv ist.

Theorem 7.3. *Es gibt keine Konstante $r < 2$, für die der Algorithmus RANDOM r-kompetitiv ist.*

Beweis. Es sei $n \in \mathbb{N}$ beliebig. Wir betrachten das Verhalten von RANDOM auf dem Graphen $G = (V \cup U, E)$ mit $V = \{v_1, \ldots, v_{2n}\}$ und $U = \{u_1, \ldots, u_{2n}\}$. Dabei enthalte E für jedes $i \in \{1, \ldots, 2n\}$ die Kante (v_i, u_i) und für jedes $i \in \{1, \ldots, n\}$ und jedes $j \in \{n+1, \ldots, 2n\}$ die Kante (v_j, u_i). Dieser Graph ist in der folgenden Abbildung für $n = 3$ dargestellt.



Die Knoten aus U treffen nun in der Reihenfolge $u_1, u_2, \ldots$ ein. Aus der Struktur des Graphen und der Definition des Algorithmus folgt, dass RANDOM jedem Knoten u_i mit $i \in \{1, \ldots, n\}$ einen Knoten aus V zuweist, denn bei der Ankunft von u_i ist insbesondere v_i noch frei. Wir bezeichnen mit p_i die Wahrscheinlichkeit, dass die Knoten u_i

und v_i einander zugewiesen werden. Außerdem bezeichnen wir mit X die Zufallsvariable, die angibt, für wie viele $i \in \{1, \ldots, n\}$ die Knoten u_i und v_i einander zugewiesen werden. Für das Matching M, das RANDOM berechnet, gilt $|M| = n + X$, denn nach der Ankunft der Knoten $u_1, \ldots, u_n$ enthält die Menge $\{v_{n+1}, \ldots, v_{2n}\}$ genau X freie Knoten. Genau diese Knoten werden bei der Ankunft der Knoten $u_{n+1}, \ldots, u_{2n}$ noch zugewiesen.

Aufgrund der Linearität des Erwartungswertes gilt

$$\mathbf{E}[|M|] = n + \mathbf{E}[X] = n + \sum_{i=1}^{n} p_i.$$

Wenn der Knoten u_i für $i \in \{1, \ldots, n\}$ eintrifft, so sind mindestens $n - (i-1)$ der Knoten $v_{n+1}, \ldots, v_{2n}$ noch frei. Die Wahrscheinlichkeit p_i, dass v_i und u_i einander zugewiesen werden, beträgt also höchstens $1/(n-(i-1)+1)$. Demzufolge gilt

$$\mathbf{E}[X] = \sum_{i=1}^{n} p_i \leq \sum_{i=1}^{n} \frac{1}{n-i+2} = \sum_{i=2}^{n+1} \frac{1}{i} \leq \sum_{i=1}^{n} \frac{1}{i} \leq \ln(n) + 1,$$

wobei wir im letzten Schritt eine bekannte Abschätzung für die n-te harmonische Zahl benutzt haben.

Seien nun $r < 2$ und $\tau \in \mathbb{R}$ beliebige Konstanten. Der Graph G besitzt ein maximales Matching mit $2n$ Kanten. Man rechnet leicht nach, dass für hinreichend große $n \in \mathbb{N}$ die Ungleichung $r \cdot \mathbf{E}[|M|] + \tau < 2n$ gilt. Damit ist gezeigt, dass RANDOM für kein $r < 2$ einen kompetitiven Faktor von r erreicht. □

Wir betrachten nun den randomisierten Algorithmus RANKING, der als eine leichte Abwandlung des Algorithmus RANDOM betrachtet werden kann und einen besseren kompetitiven Faktor erreicht. Der Algorithmus RANKING wählt vor dem Eintreffen des ersten Knotens aus U eine uniform zufällige Permutation σ der Knoten aus V. Kommt ein Knoten $v_1 \in V$ in dieser Permutation vor einem Knoten $v_2 \in V$, so schreiben wir $\sigma(v_1) < \sigma(v_2)$ und sagen, dass v_1 einen kleineren Rang als v_2 hat. Anschließend arbeitet RANKING deterministisch und weist jedem Knoten $u \in U$ bei dessen Ankunft den Knoten aus V zu, der unter allen noch freien Nachbarn von u den kleinsten Rang hat, sofern der Knoten u überhaupt noch freie Nachbarn besitzt.

Würde man die Permutation σ bei jedem Eintreffen eines Knotens neu unabhängig zufällig wählen, so würden sich die Algorithmen RANDOM und RANKING identisch verhalten, da in einer zufälligen Permutation jeder Nachbar des ankommenden Knotens die gleiche Wahrscheinlichkeit besitzt, den kleinsten Rang zu haben. Der einzige Unterschied zwischen RANDOM und RANKING besteht darin, dass durch die einmalige Wahl der Permutation die einzelnen Schritte von RANKING korreliert sind. Wir werden nun beweisen, dass diese Korrelation dazu führt, dass RANKING einen besseren kompetitiven Faktor erreicht.

Theorem 7.4. *Der Algorithmus RANKING ist auf Graphen, in denen das maximale Matching die Größe n besitzt, strikt $\frac{1}{1-\left(1-\frac{1}{n+1}\right)^n}$-kompetitiv.*

Dieser kompetitive Faktor konvergiert für wachsendes n gegen $e/(e-1) \approx 1{,}582$. Der Beweis erfolgt in mehreren Schritten. Zunächst argumentieren wir, dass wir uns ohne Beschränkung der Allgemeinheit auf Graphen beschränken können, die ein *perfektes Matching* besitzen, also ein Matching, in dem jeder Knoten eine inzidente Kante besitzt. In solchen Graphen $G = (V \cup U, E)$ gilt $|V| = |U| = |M^\star|$, wobei $M^\star$ ein maximales Matching bezeichnet.

Dazu führen wir zunächst einige Begriffe und Notationen aus der Graphentheorie ein. Für ein gegebenes Matching M bezeichnen wir einen Pfad P im Graphen G als *M-alternierenden Pfad*, wenn er abwechselnd Kanten aus M und $E \setminus M$ enthält. Wir sagen, dass zwei Matchings M und M' sich *um den M-alternierenden Pfad P unterscheiden*, wenn

$$M' = (M \setminus P) \cup (P \setminus M)$$

gilt. Das bedeutet, M' entsteht aus M dadurch, dass alle Kanten des Pfades P, die zu M gehören, gelöscht werden und alle Kanten des Pfades P, die nicht zu M gehören, hinzugefügt werden. Wir schreiben dann auch $M' = M \oplus P$. Ferner bezeichnen wir im Folgenden für einen Graphen $G = (V \cup U, E)$ und eine Permutation σ der Knoten aus V mit RANKING(G, σ) das Matching, das der Algorithmus RANKING auf dem Graphen G für die Permutation σ ausgibt.

Lemma 7.5. *Es sei σ eine beliebige Permutation der Menge V, $x \in V \cup U$ ein beliebiger Knoten und H der Graph, der aus G entsteht, indem der Knoten x und alle zu ihm inzidenten Kanten entfernt werden. Dann sind die Matchings $M = RANKING(G, \sigma)$ und $M' = RANKING(H, \sigma_x)$ entweder identisch oder sie unterscheiden sich um einen M-alternierenden Pfad P, der mindestens so viele Kanten aus M wie aus $E \setminus M$ enthält. Dabei bezeichne σ_x die Permutation, die σ auf $V \setminus \{x\}$ durch Löschen von x induziert (für $x \in U$ gilt $\sigma = \sigma_x$), und wir gehen davon aus, dass die Knoten aus $U \setminus \{x\}$ in dem Graphen H in derselben Reihenfolge ankommen wie in G.*

Aus diesem Lemma folgt, dass wir uns bei dem Beweis von Theorem 7.4 auf Graphen beschränken können, die ein perfektes Matching besitzen. Besitzt der Graph G kein solches Matching, so sei $M^\star$ ein beliebiges maximales Matching und x sei ein Knoten, der in $M^\star$ nicht zugewiesen ist. Entfernen wir den Knoten x und alle zu ihm inzidenten Kanten aus dem Graphen G, so erhalten wir einen Graphen H, für den $M^\star$ immer noch ein maximales Matching ist. Das bedeutet, der Wert der optimalen Lösung ist in G und H derselbe.

Lemma 7.5 besagt, dass RANKING für jede Permutation σ auf dem Graphen G ein Matching berechnet, das mindestens so groß ist wie das, das er auf dem Graphen H berechnet. Daraus folgt, dass RANKING auch im Erwartungswert auf dem Graphen G ein mindestens so großes Matching wie auf dem Graphen H berechnet. Eine obere Schranke für den kompetitiven Faktor auf der Eingabe H impliziert also direkt eine obere Schranke für den kompetitiven Faktor auf der Eingabe G. Wir können mit diesem Argument solange Knoten aus dem Graphen löschen, bis ein Graph vorliegt, der ein perfektes Matching besitzt.

Beweis von Lemma 7.5. Wir betrachten in diesem Beweis nur den Fall $x \in U$, der in

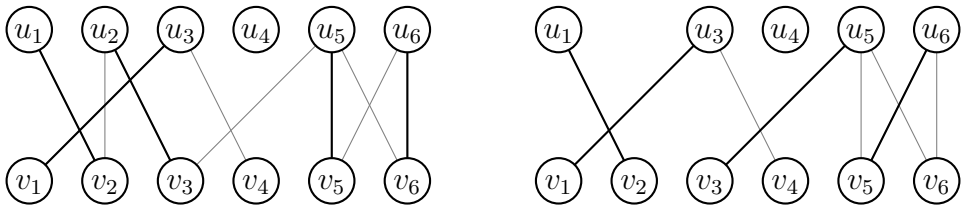


Abbildung 7.1: Diese Abbildung zeigt links das Matching M und rechts das Matching M', das nach dem Löschen des Knotens u_2 berechnet wird. Wir gehen davon aus, dass die Knoten aus V in der Reihenfolge $v_1, \ldots, v_6$ in der Permutation σ vorkommen und dass die Knoten aus U in der Reihenfolge $u_1, \ldots, u_6$ eintreffen. In diesem Beispiel ist $P = (u_2, v_3, u_5, v_5, u_6, v_6)$ ein M-alternierender Pfad mit $M' = M \oplus P$.

Abbildung 7.1 illustriert ist. Die Korrektheit des Lemmas im Fall $x \in V$ folgt mit ähnlichen Argumenten und ist dem Leser als Übung überlassen.

Es sei $n = |U|$ und $u_1, u_2, \ldots, u_n$ sei die Reihenfolge, in der die Knoten aus U eintreffen. Wir bezeichnen mit $M_i \subseteq E$ und $M'_i \subseteq E$ die Matchings, die RANKING auf den Graphen G bzw. H bei Permutation σ berechnet hat, nachdem die Knoten aus $\{u_1, \ldots, u_i\}$ bzw. $\{u_1, \ldots, u_i\} \setminus \{x\}$ eingetroffen sind. Wir zeigen die folgende Invariante: Für jedes i gilt entweder $M_i = M'_i$ oder es gibt einen M_i-alternierenden Pfad P_i, der mit einer Kante aus M beginnt und für den $M'_i = M_i \oplus P_i$ gilt. Aus dieser Invariante folgt das Lemma, denn es gilt $M = M_n$ und $M' = M'_n$.

Sei $x = u_j$. Beim Eintreffen der Knoten $u_1, \ldots, u_{j-1}$ verhält sich RANKING auf den Graphen G und H identisch. Für $i \in \{1, \ldots, j-1\}$ gilt also $M_i = M'_i$. Fügt RANKING beim Eintreffen des Knotens u_j dem Matching im Graphen G keine zu x inzidente hinzu, so gilt $M_j = M'_j$. Ansonsten sei e die zu u_j inzidente Kante, die RANKING dem Matching im Graphen G hinzufügt. Es gilt dann $M'_j = M_j \oplus P_j$ für den Pfad P_j, der nur aus der Kante $e \in M_j \subseteq M$ besteht.

Sei nun $i > j$ und gelte die Invariante für $i - 1$. Gilt $M_{i-1} = M'_{i-1}$, so sind beim Eintreffen des Knotens u_i in den Graphen G und H genau dieselben Knoten aus V frei. RANKING verhält sich also identisch und es gilt $M_i = M'_i$. Gilt $M'_{i-1} = M_{i-1} \oplus P_{i-1}$ für einen M_{i-1}-alternierenden Pfad, der mit einer Kante aus M beginnt, so unterscheiden sich die Mengen der in den Matchings M_{i-1} und M'_{i-1} freien Knoten aus V um maximal einen Knoten, nämlich den letzten Knoten v auf dem Pfad P_{i-1}, sofern dieser zu V gehört. Dieser Knoten ist in Matching M'_{i-1} frei, nicht jedoch in Matching M_{i-1}.

Sei $e' = (u_i, v')$ die Kante, die beim Eintreffen des Knotens u_j dem Matching im Graphen G hinzugefügt wird, sofern es eine solche Kante gibt. Hat v einen größeren Rang als v', so wird die Kante e' auch im Graphen H zum aktuellen Matching hinzugefügt. In diesem Fall gilt $M'_i = M_i \oplus P_{i-1}$. Hat v einen kleineren Rang als v' oder existiert gar keine Kante e', so wird im Graphen H die Kante $e = (u_i, v)$ eingefügt, sofern diese existiert. In diesem Fall gilt $M'_i = M_i \oplus P_i$ für den Pfad P_i, der aus P_{i-1} durch das Anhängen der Kanten e und e' entsteht, soweit sie existieren. □

Wir gehen im Folgenden davon aus, dass der Graph G ein perfektes Matching besitzt, das durch eine bijektive Funktion $m^\star : U \to V$ beschrieben ist. Ferner sei $n = |V| = |U|$ und M sei das zufällige Matching, das RANKING auf dem Graphen G berechnet.

Lemma 7.6. *Es sei $u \in U$ beliebig und es sei $v = m^\star(u)$. Falls der Knoten v in dem Matching M nicht zugewiesen ist, so ist u in M einem Knoten zugewiesen, dessen Rang kleiner ist als der von v.*

Beweis. Wir betrachten den Zeitpunkt, zu dem der Knoten u aufgedeckt wird. Da der Knoten v zu diesem Zeitpunkt frei ist, u aber dennoch einem anderen Knoten v' zugewiesen wird, folgt direkt aus der Definition von RANKING, dass v' einen kleineren Rang als v haben muss. □

Für $t \in \{1, \ldots, n\}$ bezeichne p_t die Wahrscheinlichkeit, dass der Knoten aus V mit Rang t in dem Matching M eine inzidente Kante besitzt.

Lemma 7.7. *Für $t \in \{1, \ldots, n\}$ gilt*

$$1 - p_t \leq \frac{1}{n} \sum_{s=1}^{t} p_s.$$

Beweis. Der Artikel von Karp, Vazirani und Vazirani [11] enthält einen einfachen, aber fehlerhaften Beweis dieses Lemmas, den auch wir zunächst angeben. Wir werden dann diskutieren, wo der Fehler liegt und wie dieser repariert werden kann. Sei $t \in \{1, \ldots, n\}$ beliebig und sei v der Knoten aus V, der in der Permutation σ den Rang t besitzt. Per Definition ist $1 - p_t$ die Wahrscheinlichkeit, dass der Knoten v vom Algorithmus RANKING nicht zugewiesen wird. Es sei $u \in U$ der Knoten, der in dem perfekten Matching $M^\star$ dem Knoten v zugewiesen ist, also der Knoten mit $v = m^\star(u)$. Ferner sei $R_{t-1} \subseteq U$ die Menge der Knoten, die im Matching M Knoten aus V mit Rang höchstens $t - 1$ zugewiesen sind.

Lemma 7.6 besagt, dass der Knoten v in dem Matching M nur dann nicht zugewiesen sein kann, wenn $u \in R_{t-1}$ gilt. Die Kardinalität der Menge R_{t-1} entspricht der Anzahl der Knoten in V mit Rang maximal $t-1$, die in Matching M zugewiesen sind. Es gilt demnach

$$\mathbf{E}\left[|R_{t-1}|\right] = \sum_{s=1}^{t-1} p_s.$$

Würde der Knoten u uniform zufällig und unabhängig von der Menge R_{t-1} gewählt, so könnten wir die Wahrscheinlichkeit, dass der Knoten v im Matching M nicht zugewiesen ist, wie folgt abschätzen:

$$1 - p_t \leq \mathbf{Pr}\left[u \in R_{t-1}\right] = \frac{\mathbf{E}\left[|R_{t-1}|\right]}{|U|} = \frac{1}{n} \sum_{s=1}^{t-1} p_s.$$

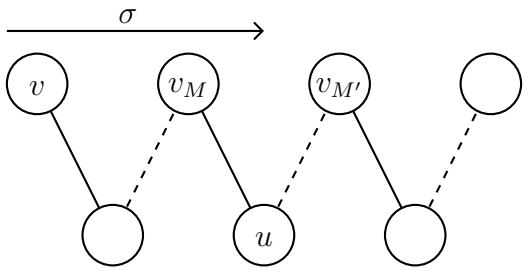
Dies entspricht dem Vorgehen in dem Artikel [11], ist jedoch nicht korrekt, da der Knoten u und die Menge R_{t-1} stochastisch nicht unabhängig sind, weil sie beide von der zufälligen Permutation σ abhängen.

Um diese Abhängigkeiten zu beseitigen, konstruieren wir eine zweite zufällige Permutation σ'. Dazu starten wir mit der Permutation σ, wählen einen Knoten $v \in V$ uniform zufällig aus und schieben diesen in der Permutation an Position t. Die so

erzeugte Permutation σ' ist genauso wie σ eine uniform zufällige Permutation der Knoten aus V. Natürlich sind σ und σ' voneinander abhängig, aber für sich betrachtet ist σ' uniform zufällig, da nach dem Verschieben eines uniform zufälligen Knotens an Position t nach wie vor jede Permutation die gleiche Wahrscheinlichkeit besitzt, als σ' realisiert zu werden. Insbesondere beträgt die Wahrscheinlichkeit, dass der Knoten v in dem Matching $M' = \text{RANKING}(G, \sigma')$ zugewiesen ist, genau p_t.

Sei nun $u \in U$ wieder der Knoten mit $v = m^\star(u)$ und sei R_{t-1} weiterhin bezüglich des Matchings $M = \text{RANKING}(G, \sigma)$ definiert. Da der Knoten v und damit auch der Knoten u uniform zufällig und unabhängig von dem Matching M gewählt wird, gilt nun tatsächlich $\mathbf{Pr}[u \in R_{t-1}] = \mathbf{E}[|R_{t-1}|]/|U|$. Das Ereignis $u \in R_{t-1}$ ist nun aber bedeutungslos, da die Menge R_{t-1} sich auf das Matching M bezieht und der Knoten $v = m^\star(u)$ in der Permutation σ', aber im Allgemeinen nicht in der Permutation σ den Rang t besitzt.

Um die Unabhängigkeit des Knotens u von der Menge R_{t-1} auszunutzen zu können, müssen wir zunächst einen Zusammenhang zwischen den Matchings M und M' herstellen. Wir möchten die Wahrscheinlichkeit $1 - p_t$ für das Ereignis beschränken, dass der Knoten v im Matching M' nicht zugewiesen ist. Tritt dieses Ereignis ein, so kann man mit ähnlichen Argumenten wie im Beweis von Lemma 7.5 zeigen, dass M und M' entweder identisch sind oder sich um einen M-alternierenden Pfad P unterscheiden, der an Knoten v mit einer Kante aus M beginnt. Ferner lässt sich zeigen, dass die Knoten aus V, die auf dem Pfad P vorkommen, in aufsteigender Reihenfolge bezüglich der Permutation σ durchlaufen werden. Wir verzichten auf einen formalen Beweis dieser Aussage und überlassen ihn dem Leser als Übung. Die folgende Abbildung illustriert den Pfad P.



Seien v_M und $v_{M'}$ die Knoten, denen der Knoten u im Matching M bzw. M' zugewiesen ist. Dann gelten die folgenden drei Aussagen.

- $\sigma'(v_{M'}) < \sigma'(v) = t$ folgt aus Lemma 7.6, da der Knoten v gemäß Annahme im Matching M' nicht zugewiesen ist.
- $|\sigma(v_{M'}) - \sigma'(v_{M'})| \leq 1$, da in σ' nur ein Knoten v verschoben wird und sich dadurch der Rang der anderen Knoten um maximal 1 ändert.
- $\sigma(v_M) \leq \sigma(v_{M'})$, da die Knoten aus V von P in aufsteigender Reihenfolge bezüglich σ durchlaufen werden.

Insgesamt folgt

$$\sigma(v_M) \leq \sigma(v_{M'}) \leq \sigma'(v_{M'}) + 1 < t + 1,$$

was äquivalent zu $\sigma(v_M) \leq t$ ist. Damit ist gezeigt, dass $u \in R_t$ gilt, wenn v in M' nicht zugewiesen ist. Also

$$1 - p_t = \frac{\mathbf{E}\left[|R_t|\right]}{|U|} = \frac{1}{n} \sum_{s=1}^{t} p_s. \qquad \square$$

Mithilfe des vorangegangenen Lemmas können wir nun den kompetitiven Faktor von RANKING analysieren.

Beweis von Theorem 7.4. Es sei M das zufällige Matching, das RANKING auf dem Graphen G berechnet. Aufgrund der Linearität des Erwartungswertes gilt

$$\mathbf{E}\left[|M|\right] = \sum_{s=1}^{n} p_s. \tag{7.1}$$

Unser Ziel ist es, diese Summe unter den in Lemma 7.7 gegebenen Nebenbedingungen zu minimieren. Dazu lösen wir die Ungleichung aus diesem Lemma zunächst nach p_t auf und erhalten

$$p_t \geq 1 - \frac{1}{n} \sum_{s=1}^{t} p_s.$$

Als nächstes substituieren wir $S_t = \sum_{s=1}^{t} p_s$ und erhalten mit kurzer Rechnung für jedes $t \in \{2, \ldots, n\}$ die zu Lemma 7.7 äquivalente Ungleichung

$$S_t \geq \frac{n}{n+1}(1 + S_{t-1}).$$

Aus Lemma 7.7 folgt direkt

$$S_1 = p_1 \geq \frac{n}{n+1} = 1 - \frac{1}{n+1}. \tag{7.2}$$

Mithilfe vollständiger Induktion kann man beweisen, dass

$$S_t \geq \sum_{s=1}^{t} \left(1 - \frac{1}{n+1}\right)^s$$

für jedes $t \in \{1, \ldots, n\}$ gilt. Der Induktionsanfang folgt direkt aus (7.2). Gelte die Behauptung für $t - 1 \geq 1$, so gilt

$$\begin{aligned}
S_t &\geq \frac{n}{n+1} \cdot (1 + S_{t-1}) \\
&\geq \left(1 - \frac{1}{n+1}\right) \cdot \left(1 + \sum_{s=1}^{t-1} \left(1 - \frac{1}{n+1}\right)^s\right) \\
&= \left(1 - \frac{1}{n+1}\right) + \sum_{s=2}^{t} \left(1 - \frac{1}{n+1}\right)^s \\
&= \sum_{s=1}^{t} \left(1 - \frac{1}{n+1}\right)^s.
\end{aligned}$$

Insgesamt haben wir damit Folgendes bewiesen:

$$\begin{aligned}\mathbf{E}\left[|M|\right] &= \sum_{s=1}^{n} p_s = S_n \geq \sum_{s=1}^{n} \left(1 - \frac{1}{n+1}\right)^s \\ &= \frac{n}{n+1} \cdot \sum_{s=0}^{n-1} \left(\frac{n}{n+1}\right)^s \\ &= \frac{n}{n+1} \cdot \left(\frac{1 - (n/(n+1))^n}{1/(n+1)}\right) \\ &= n - n \cdot \left(1 - \frac{1}{n+1}\right)^n \\ &= \left(1 - \left(1 - \frac{1}{n+1}\right)^n\right) \cdot n.\end{aligned}$$

Damit folgt das Theorem, da ein maximales Matching die Größe n besitzt. □

Karp, Vazirani und Vazirani [11] haben bewiesen, dass es keinen randomisierten Algorithmus für das Online-Matching-Problem gibt, der einen besseren kompetitiven Faktor als $e/(e-1)$ erreicht.

7.3 Random-Order-Modell

Die Graphen, die wir bislang in diesem Kapitel betrachtet haben, waren ungewichtet. Es liegt nun die Frage nahe, ob sich die Algorithmen, die wir kennengelernt haben, auf den Fall übertragen lassen, dass die Kanten in dem Graphen gewichtet sind. In diesem Fall ist es das Ziel, ein Matching mit einem möglichst großen Gesamtgewicht zu finden. Offline gibt es auch für diese Problemvariante effiziente Algorithmen. In dem Modell, das wir untersuchen werden, ist ein bipartiter Graph $G = (V \cup U, E)$ mit einer Kantengewichtung $w : E \to \mathbb{R}_{\geq 0}$ gegeben. Während die Knoten aus V wieder von Anfang an bekannt sind, werden die Knoten aus U nach und nach mitsamt ihrer Kanten und den entsprechenden Kantengewichten aufgedeckt.

Man überlegt sich leicht, dass jeder deterministische Online-Algorithmus für dieses Problem beliebig schlecht ist, denn sobald der Algorithmus in einem Schritt eine Kante $e = (v, u)$ auswählt und dem Matching hinzufügt, könnte im nächsten Schritt eine Kante $e' = (v, u')$ mit einem Gewicht aufgedeckt werden, das um einen beliebigen Faktor größer ist als das von e. Die Kante e' kann dann aber nicht mehr zum Matching hinzugefügt werden, da der Knoten v bereits eine inzidente Kante im Matching besitzt. Der Leser sollte sich als Übung überlegen, dass auch mit einem randomisierten Online-Algorithmus kein endlicher kompetitiver Faktor erreicht werden kann.

Das gewichtete Online-Matching-Problem ist somit zunächst nicht besonders interessant, da es für jeden Online-Algorithmus Eingaben gibt, auf denen er beliebig schlecht abschneidet. Insbesondere schneidet jeder Algorithmus in einer kompetitiven Analyse genauso gut (oder schlecht) ab wie der Algorithmus, der niemals Kanten zum Matching hinzufügt und auf jeder Eingabe ein leeres Matching als Ausgabe liefert. Intuitiv

erscheint dies wenig sinnvoll und man hat das Gefühl, dass manche Algorithmen durchaus besser sind als andere. Um dies formal zu untermauern, werden wir das Modell der kompetitiven Analyse in diesem Abschnitt abändern.

Für den kompetitiven Faktor eines Online-Algorithmus ist gemäß Definition 1.1 nur sein Verhalten im Worst Case von Bedeutung. Anschaulich muss man es als Algorithmenentwickler mit einem Gegner aufnehmen, der eine möglichst schlechte Eingabe für den verwendeten Algorithmus sucht. Eingaben, die in praktischen Anwendungen auftreten, sind jedoch normalerweise nicht gegnerisch mit dem Ziel erzeugt, den verwendeten Algorithmus möglichst schlecht dastehen zu lassen. Deshalb kann es irreführend sein, sich bei der Beurteilung eines Algorithmus nur auf sein Worst-Case-Verhalten zu konzentrieren. Tatsächlich sind viele Algorithmen in praktischen Anwendungen deutlich besser als im Worst Case.

Ein abgeschwächtes Worst-Case-Modell, das in den letzten Jahren auf viel Interesse gestoßen ist, ist das *Random-Order-Modell*. In diesem Modell darf der Gegner zwar immer noch den bipartiten Graphen und die Kantengewichtung beliebig wählen, die Knoten aus U werden jedoch in einer uniform zufälligen Reihenfolge aufgedeckt. Dies steht im Gegensatz zur klassischen Worst-Case-Analyse, in der der Gegner bestimmen kann, in welcher Reihenfolge diese Knoten erscheinen. Wir werden sehen, dass es Online-Algorithmen für das gewichtete Online-Matching-Problem gibt, die im Random-Order-Modell einen konstanten kompetitiven Faktor erreichen, wenn ihnen zu Beginn die Kardinalität $|U|$ bekannt ist. In diesem Modell kann man also deutlich besser zwischen guten und schlechten Algorithmen differenzieren als in einer klassischen Worst-Case-Analyse.

Bevor wir in Abschnitt 7.3.2 den allgemeinen Fall untersuchen, betrachten wir in Abschnitt 7.3.1 zunächst den Spezialfall, in dem die von vornherein bekannte Seite V des Graphen aus nur einem einzigen Knoten besteht. Diese seit den 60er-Jahren untersuchte Variante wird aus Gründen, die wir gleich erläutern werden, auch als *Sekretärinnenproblem* bezeichnet. In Abschnitt 7.3.3 weisen wir nach, dass die Algorithmen, die wir in den Abschnitten 7.3.1 und 7.3.2 kennenlernen werden, optimal sind.

7.3.1 Das klassische Sekretärinnenproblem

Das klassische Sekretärinnenproblem kann wie folgt beschrieben werden. Ein Unternehmen möchte eine freie Stelle (zum Beispiel im Sekretariat) besetzen. Es gibt n Bewerbungen und jeder Bewerber $i \in \{1, \dots, n\}$ besitzt eine gewisse Eignung $w_i \in \mathbb{R}_{\geq 0}$ für die Stelle. Das Ziel des Unternehmens ist es, einen Bewerber mit einer möglichst großen Eignung einzustellen. Dazu kann es die Bewerber nacheinander zum Bewerbungsgespräch einladen. Erst im Bewerbungsgespräch mit Kandidat i erfährt das Unternehmen seine Eignung w_i. Es muss sich dann direkt entscheiden, ob es Kandidat i einstellt oder nicht. Stellt es Kandidat i ein, so ist das Bewerbungsverfahren abgeschlossen und die verbleibenden Bewerber werden nicht mehr eingeladen. Lehnt das Unternehmen Kandidat i nach dem Bewerbungsgespräch ab, so gibt es später keine Möglichkeit mehr, diese Entscheidung zu revidieren.

Zugegebenermaßen sind die Annahmen in diesem Modell nicht unbedingt realistisch, dennoch ist es interessant, sich mit dem Sekretärinnenproblem zu beschäftigen, da es ein gut untersuchter Spezialfall des gewichteten Online-Matching-Problems ist. Wie oben bereits angedeutet, ist es der Spezialfall, bei dem die bekannte Seite V aus nur einem einzigen Knoten v besteht. Dieser Knoten repräsentiert die zu besetzende Stelle und jeder Knoten aus U entspricht einem Bewerber. Für jedes $u \in U$ gibt es eine Kante (v, u), wobei das Gewicht $w(v, u)$ der Eignung des Kandidaten u entspricht. Ein nichtleeres Matching in diesem Graphen besteht aus genau einer Kante und entspricht somit der Auswahl eines der Kandidaten.

Wie sollte sich das Unternehmen verhalten, um einen möglichst guten Kandidaten einzustellen? Mit den gleichen Argumenten wie für das gewichtete Online-Matching-Problem kann man auch hier zeigen, dass es für jeden deterministischen oder randomisierten Online-Algorithmus eine Eingabe gibt, auf der er beliebig schlecht abschneidet. Wir betrachten deshalb das Random-Order-Modell, in dem ein Gegner zwar die Eignungen $w_1, \ldots, w_n \in \mathbb{R}_{\geq 0}$ beliebig wählen kann, die Kandidaten aber in einer uniform zufälligen Reihenfolge zum Bewerbungsgespräch eingeladen werden. Außerdem ist dem Algorithmus in diesem Modell die Anzahl n an Bewerbern von vornherein bekannt. Wir werden sehen, dass in diesem Modell ein konstanter kompetitiver Faktor erreicht werden kann.

Wir bezeichnen im Folgenden mit $\sigma = \{w_1, \ldots, w_n\}$ die Menge der vom Gegner gewählten Eignungen und gehen davon aus, dass diese paarweise verschieden sind. Sei $w_1 > w_2 > \ldots > w_n$. Für einen Online-Algorithmus A bezeichnen wir im Random-Order-Modell mit $w_A(\sigma)$ die *erwartete* Eignung des von A ausgewählten Kandidaten. Dabei bezieht sich der Erwartungswert auf die zufällige Reihenfolge, in der die Kandidaten betrachtet werden, und auf die zufälligen Entscheidungen von A, sofern es sich bei A um einen randomisierten Algorithmus handelt. Ferner sei $\mathrm{OPT}(\sigma) = \max_i w_i = w_1$. Wir sagen, dass A im Random-Order-Modell strikt r-kompetitiv ist, wenn $w_A(\sigma) \geq \frac{1}{r} \cdot \mathrm{OPT}(\sigma)$ für alle Eingaben σ gilt.

Sprechen wir im Folgenden vom ersten Kandidaten, den ersten k Kandidaten oder Ähnlichem so bezieht sich dies stets auf die zufällige Reihenfolge, in der die Kandidaten betrachtet werden. Im Allgemeinen ist also die Eignung des ersten Kandidaten ungleich w_1. Der Algorithmus A_k, den wir nun untersuchen, lehnt die ersten k Kandidaten ab und wählt den ersten Kandidaten $i > k$ aus, der besser ist als alle bisher gesehenen Kandidaten $1, \ldots, i-1$. Bei A_k handelt es sich für jedes k um einen deterministischen Online-Algorithmus. Als erstes nehmen wir der Einfachheit halber an, dass n gerade ist, und betrachten den Algorithmus $A_{n/2}$. Wir zeigen, dass dieser Algorithmus mit einer Wahrscheinlichkeit von mindestens $1/4$ den besten Kandidaten auswählt, woraus direkt folgt, dass er strikt 4-kompetitiv ist.

Wir bezeichnen mit $H_1 \subseteq \{w_1, \ldots, w_n\}$ und $H_2 = \{w_1, \ldots, w_n\} \setminus H_1$ die Menge der Eignungen der Kandidaten, die sich in der zufälligen Reihenfolge in der ersten bzw. zweiten Hälfte befinden. In der folgenden Rechnung nutzen wir die Beobachtung aus, dass der beste Kandidat auf jeden Fall gewählt wird, wenn er sich selbst in der zweiten Hälfte befindet und sich der zweitbeste Kandidat in der ersten Hälfte befindet:

$$w_{A_{n/2}}(\sigma) \geq \mathrm{OPT}(\sigma) \cdot \mathbf{Pr}\left[\text{Kandidat mit Eignung } w_1 \text{ wird ausgewählt}\right]$$

$$\begin{aligned}&\geq \mathrm{OPT}(\sigma) \cdot \mathbf{Pr}\left[w_2 \in H_1 \text{ und } w_1 \in H_2\right] \\ &= \mathrm{OPT}(\sigma) \cdot \mathbf{Pr}\left[w_2 \in H_1\right] \cdot \mathbf{Pr}\left[w_1 \in H_2 \mid w_2 \in H_1\right].\end{aligned}$$

Da die Kandidaten in einer uniform zufälligen Reihenfolge eintreffen, beträgt die Wahrscheinlichkeit für das Ereignis $w_2 \in H_1$ genau $1/2$. Gilt $w_2 \in H_1$, so gibt es in H_1 und H_2 noch $n/2 - 1$ bzw. $n/2$ verfügbare Plätze für w_1. Da die Bewerber in einer uniform zufälligen Reihenfolge eintreffen, wird für w_1 aus diesen verfügbaren Plätzen uniform zufällig einer ausgewählt. Die bedingte Wahrscheinlichkeit für $w_1 \in H_2$ beträgt demnach $\frac{n/2}{n-1} \geq \frac{1}{2}$. Insgesamt ergibt sich

$$w_{A_{n/2}}(\sigma) \geq \frac{1}{4} \cdot \mathrm{OPT}(\sigma).$$

Das folgende Theorem zeigt, dass man für eine geschickte Wahl von k einen besseren kompetitiven Faktor erreichen kann.

Theorem 7.8. *Der Algorithmus $A_{\lfloor n/e \rfloor}$ ist im Random-Order-Modell strikt $c(n)$-kompetitiv, wobei $c(n) = \frac{en}{n-e}$ für wachsendes n gegen e konvergiert.*

Beweis. Wieder genügt es zur Analyse des Algorithmus die Wahrscheinlichkeit abzuschätzen, mit der der beste Kandidat ausgewählt wird. Es gilt

$$\begin{aligned}&\mathbf{Pr}\left[w_1 \text{ wird ausgewählt}\right] \\ &= \sum_{i=\lceil n/e \rceil}^{n} \mathbf{Pr}\left[w_1 \text{ an Position } i \text{ und wird ausgewählt}\right] \\ &= \sum_{i=\lceil n/e \rceil}^{n} \mathbf{Pr}\left[w_1 \text{ an Position } i\right] \cdot \mathbf{Pr}\left[w_1 \text{ wird ausgewählt} \mid w_1 \text{ an Position } i\right] \\ &= \sum_{i=\lceil n/e \rceil}^{n} \frac{1}{n} \cdot \mathbf{Pr}\left[w_1 \text{ wird ausgewählt} \mid w_1 \text{ an Position } i\right].\end{aligned}$$

Die Summen in der obigen Rechnung beginnen bei $i = \lceil n/e \rceil = \lfloor n/e \rfloor + 1$, da die ersten $\lfloor n/e \rfloor$ Kandidaten stets abgelehnt werden. Die letzte Gleichung gilt, da der Kandidat mit Eignung w_1 an einer uniform zufälligen Position steht und somit jede Position i eine Wahrscheinlichkeit von $1/n$ hat.

Wir müssen nur noch die bedingte Wahrscheinlichkeit bestimmen, mit der der Kandidat mit Eignung w_1 ausgewählt wird, wenn er in der zufälligen Permutation an Position i steht. Dies passiert genau dann, wenn sich der beste der ersten $i-1$ Kandidaten unter den ersten $\lfloor n/e \rfloor$ Kandidaten befindet. Da die Permutation uniform zufällig ist, beträgt die Wahrscheinlichkeit dafür $\frac{\lfloor n/e \rfloor}{i-1}$. Somit gilt

$$\begin{aligned}\mathbf{Pr}\left[w_1 \text{ wird ausgewählt}\right] &= \sum_{i=\lceil n/e \rceil}^{n} \frac{1}{n} \cdot \frac{\lfloor n/e \rfloor}{i-1} = \frac{\lfloor n/e \rfloor}{n} \cdot \sum_{i=\lfloor n/e \rfloor}^{n-1} \frac{1}{i} \\ &\geq \frac{n/e - 1}{n} \cdot \ln\left(\frac{n}{n/e}\right) = \frac{1}{e} - \frac{1}{n}. \qquad (7.3)\end{aligned}$$

Die Ungleichung folgt aus der Abschätzung

$$\sum_{i=\lfloor n/e \rfloor}^{n-1} \frac{1}{i} \geq \int_{\lfloor n/e \rfloor}^{n} \frac{1}{x}\, dx = \ln\left(\frac{n}{\lfloor n/e \rfloor}\right) \geq \ln\left(\frac{n}{n/e}\right) = 1.$$

Damit ist der Beweis abgeschlossen, denn es gilt

$$\begin{aligned} w_{A_{\lfloor n/e \rfloor}}(\sigma) &\geq \mathrm{OPT}(\sigma) \cdot \mathbf{Pr}\left[w_1 \text{ wird ausgewählt}\right] \\ &\geq \left(\frac{1}{e} - \frac{1}{n}\right) \cdot \mathrm{OPT}(\sigma) = \frac{1}{c(n)} \cdot \mathrm{OPT}(\sigma). \end{aligned}$$

□

7.3.2 Das Online-Matching-Problem in gewichteten Graphen

Wir betrachten nun das allgemeine Online-Matching-Problem in gewichteten bipartiten Graphen und lernen einen strikt e-kompetitiven Online-Algorithmus von Kesselheim et al. [12] kennen. Der Algorithmus ist im Folgenden in Pseudocode dargestellt. Für eine Teilmenge $U' \subseteq U$ bezeichnen wir mit $G[U']$ den Teilgraphen von G, der durch die Knotenmenge $U' \cup V$ induziert wird. Der Graph $G[U']$ enthält also die Knoten $U' \cup V$ und alle Kanten, die zwischen diesen Knoten in G verlaufen. Die Kantengewichte in $G[U']$ sind identisch mit denen in G. Ist U' die Menge der bereits aufgedeckten Knoten, so ist $G[U']$ genau der Teil des Graphen, der dem Algorithmus bereits bekannt ist.

RANDOM-ORDER-MATCHING

1. Es sei $U' \subseteq U$ die Menge der ersten $\lfloor n/e \rfloor$ ankommenden Knoten aus U. Füge für keinen Knoten aus U' eine Kante in das Matching ein.
2. $M = \emptyset$;
3. **for each** (ankommenden Knoten $u \in U$ nach den ersten $\lfloor n/e \rfloor$)
4. $\quad U' := U' \cup \{u\}; \quad i := |U'|$;
5. $\quad$ Sei $M^{(i)}$ ein Matching mit maximalem Gewicht in $G[U']$.
6. $\quad$ **if** (u besitzt inzidente Kante in $M^{(i)}$)
7. $\quad\quad$ Sei $e^{(i)} = (v, u)$ die zu u inzidente Kante in $M^{(i)}$.
8. $\quad\quad$ **if** ($M \cup \{e^{(i)}\}$ ist Matching) $M := M \cup \{e^{(i)}\}$;
9. **return** M;

Wir fassen den Algorithmus RANDOM-ORDER-MATCHING noch einmal in Worten zusammen. Die ersten $\lfloor n/e \rfloor$ Knoten erhalten keine inzidente Kante im Matching. Anschließend wird für jeden ankommenden Knoten u auf dem bisher bekannten Teil des Graphen ein Matching mit maximalem Gewicht berechnet und zwar vollkommen unabhängig von den Kanten, die wir dem Matching bereits hinzugefügt haben. Besitzt u in diesem Matching eine inzidente Kante, so testen wir, ob diese zusammen mit den bereits ausgewählten Kanten ein Matching bildet. Ist dies der Fall, so fügen wir sie dem bisher ausgewählten Matching hinzu. Dies garantiert, dass die Menge M zu jedem Zeitpunkt ein Matching in G bildet.

Theorem 7.9. *Der Algorithmus* Random-Order-Matching *ist im Random-Order-Modell strikt $c(n)$-kompetitiv, wobei $c(n) = \frac{en}{n-e}$ für wachsendes n gegen e konvergiert.*

Um das Theorem zu beweisen, führen wir zunächst Zufallsvariablen $X_{\lceil n/e \rceil}, \ldots, X_n$ ein. Dabei gibt X_i das Gewicht der Kante an, die Random-Order-Matching in Schritt i dem Matching hinzufügt, sofern es eine solche gibt. Formal sei

$$X_i = \begin{cases} w(e^{(i)}) & \text{falls } e^{(i)} \text{ existiert und } M \text{ hinzugefügt wird,} \\ 0 & \text{sonst.} \end{cases}$$

Das erwartete Gewicht des von Random-Order-Matching ausgewählten Matchings auf dem Graphen G kann dann als

$$w_{\mathrm{ROM}}(G) = \mathbf{E}\left[\sum_{i=\lceil n/e \rceil}^{n} X_i\right] = \sum_{i=\lceil n/e \rceil}^{n} \mathbf{E}[X_i] \tag{7.4}$$

geschrieben werden, wobei wir die Linearität des Erwartungswertes ausgenutzt haben. Wir bezeichnen mit $\mathrm{OPT}(G)$ den Wert eines Matchings in G mit maximalem Gewicht. Der wesentliche Baustein im Beweis von Theorem 7.9 ist das folgende Lemma.

Lemma 7.10. *Für jedes $i \in \{\lceil n/e \rceil, \ldots, n\}$ gilt*

$$\mathbf{E}[X_i] \geq \frac{\lfloor n/e \rfloor}{i-1} \cdot \frac{\mathrm{OPT}(G)}{n}.$$

Beweis. Da Random-Order-Matching ein Online-Algorithmus ist, sind für den Wert von X_i nur die ersten i Knoten in der zufälligen Reihenfolge relevant. Wir unterteilen das Zufallsexperiment, das diese i Knoten und ihre Reihenfolge erzeugt, in drei Phasen.

1. In der ersten Phase wird aus der Menge U eine uniform zufällige Teilmenge U' mit Kardinalität i ausgewählt.
2. In der zweiten Phase wird aus der Menge U' uniform zufällig ein Knoten ausgewählt, der an Position i gesetzt wird.
3. In der dritten Phase wird die Reihenfolge der übrigen Knoten aus U' in der gleichen Art und Weise bestimmt. Das heißt, zunächst wird aus den $i-1$ verbleibenden Knoten aus U' uniform zufällig einer ausgewählt und an Position $i-1$ gesetzt. Dann wird aus den verbleibenden $i-2$ Knoten aus U' uniform zufällig einer ausgewählt und an Position $i-2$ gesetzt, und so weiter.

In diesen drei Phasen werden die ersten i Knoten und ihre Reihenfolge bestimmt. Die Verteilung ist dieselbe wie in dem eigentlichen Zufallsexperiment, in dem eine uniform zufällige Reihenfolge aller Knoten gewählt wird, die Einteilung in die drei Phasen erleichtert uns jedoch die Analyse.

Nach der ersten Phase steht das Matching $M^{(i)}$ bereits fest, da dieses Matching nur von dem bekannten Teilgraphen $G[U']$ abhängt, nicht jedoch von der Reihenfolge, in

der die Knoten aus U' aufgedeckt wurden. (Wir gehen davon aus, dass in Schritt 5 von Random-Order-Matching ein fester deterministischer Algorithmus zur Bestimmung eines optimalen Matchings eingesetzt wird. Sollte es in dem Graphen $G[U']$ mehrere optimale Matchings geben, so sei $M^{(i)}$ das eindeutige optimale Matching, das dieser Algorithmus auswählt.) Es bezeichne $M^\star$ ein Matching in G mit maximalem Gewicht. Dann gilt $\mathrm{OPT}(G) = w(M^\star)$. Außerdem bezeichne $M^\star[U']$ die Einschränkung von $M^\star$ auf die Knotenmenge $U' \cup V$. Das heißt, das Matching $M^\star[U']$ enthält genau die Kanten aus $M^\star$, die zu einem Knoten aus U' inzident sind. Bei $M^\star[U']$ handelt es sich um ein Matching in $G[U']$. Da es sich bei $M^{(i)}$ um ein Matching in $G[U']$ mit maximalem Gewicht handelt, gilt $w(M^{(i)}) \geq w(M^\star[U'])$.

Für jeden Knoten aus U beträgt die Wahrscheinlichkeit, dass er zu U' gehört, genau i/n, da es sich bei U' um eine uniform zufällige Teilmenge der Größe i von U handelt. Für $u \in U$ sei w_u das Gewicht der zu u inzidenten Kante in $M^\star$, sofern eine solche Kante existiert, und 0 sonst. Es sei

$$Y_u = \begin{cases} w_u & \text{falls } u \in U', \\ 0 & \text{sonst.} \end{cases}$$

Dann gilt

$$\begin{aligned} \mathbf{E}\left[w(M^{(i)})\right] &\geq \mathbf{E}\left[w(M^\star[U'])\right] = \mathbf{E}\left[\sum_{u \in U} Y_u\right] = \sum_{u \in U} \mathbf{E}\left[Y_u\right] = \sum_{u \in U} \frac{i}{n} \cdot w_u = \frac{i}{n} \cdot \sum_{u \in U} w_u \\ &= \frac{i}{n} \cdot w(M^\star) = \frac{i}{n} \cdot \mathrm{OPT}(G). \end{aligned} \tag{7.5}$$

Wir haben bislang nur den Zufall aus Phase 1 genutzt. Nun nutzen wir den Zufall aus Phase 2, um den Erwartungswert $\mathbf{E}\left[w(e^{(i)})\right]$ zu bestimmen. Wir gehen davon aus, dass die Menge U' bereits feststeht und bezeichnen ihre Realisierung mit Z. Für eine Menge $Z \subseteq U$ mit $|Z| = i$ bezeichnen wir mit $M^{(i)}(Z)$ das optimale Matching in dem Graphen $G[Z]$. In Phase 2 wird uniform zufällig ein Knoten u aus U' ausgewählt und an Position i gesetzt. Es gilt für jedes $Z \subseteq U$ mit $|Z| = i$

$$\mathbf{E}\left[w(e^{(i)}) \mid U' = Z\right] = \frac{1}{i} \cdot w(M^{(i)}(Z)),$$

da die in $M^{(i)}(Z)$ inzidente Kante eines uniform zufällig gewählten Knotens aus Z im Erwartungswert wegen $|Z| = i$ genau einen $(1/i)$-Bruchteil zum Gesamtgewicht des Matchings beisteuert. Gemeinsam mit (7.5) folgt daraus

$$\begin{aligned} \mathbf{E}\left[w(e^{(i)})\right] &= \sum_{Z \subseteq U, |Z| = i} \mathbf{E}\left[w(e^{(i)}) \mid U' = Z\right] \cdot \mathbf{Pr}\left[U' = Z\right] \\ &= \sum_{Z \subseteq U, |Z| = i} \frac{1}{i} \cdot w(M^{(i)}(Z)) \cdot \mathbf{Pr}\left[U' = Z\right] \\ &= \frac{1}{i} \cdot \sum_{Z \subseteq U, |Z| = i} w(M^{(i)}(Z)) \cdot \mathbf{Pr}\left[U' = Z\right] \\ &= \frac{1}{i} \cdot \mathbf{E}\left[w(M^{(i)})\right] \geq \frac{1}{i} \cdot \frac{i}{n} \cdot \mathrm{OPT}(G) = \frac{1}{n} \cdot \mathrm{OPT}(G). \end{aligned}$$

Erst in Phase 3 entscheidet sich, ob die Kante $e^{(i)}$ tatsächlich zum Matching hinzugefügt werden kann. Wir zeigen, dass die Wahrscheinlichkeit hierfür mindestens $\frac{\lfloor n/e \rfloor}{i-1}$ beträgt. Seien die Phasen 1 und 2 bereits abgeschlossen und sei $e^{(i)} = (v, u)$. Uns interessiert die Wahrscheinlichkeit, dass der Knoten v noch frei ist. Dies ist genau dann der Fall, wenn der Knoten v in keiner der Kanten $e^{(\lceil n/e \rceil)}, \ldots, e^{(i-1)}$ enthalten ist.

Betrachten wir zunächst die Wahrscheinlichkeit für das Ereignis $v \in e^{(i-1)}$. Das Matching $M^{(i-1)}$ steht bereits fest, da wir nach den ersten beiden Phasen die Menge $U' \setminus \{u\}$ der ersten $i-1$ aufgedeckten Knoten kennen. Die Reihenfolge, in der diese Knoten aufgedeckt wurden, haben wir in den Phasen 1 und 2 noch nicht festgelegt, sie spielt aber für das Matching $M^{(i-1)}$ auch keine Rolle. Wir ziehen nun uniform zufällig einen Knoten aus $U' \setminus \{u\}$ und setzen ihn an Position $i-1$. Die Wahrscheinlichkeit einen in $M^{(i-1)}$ zu v adjazenten Knoten zu ziehen, beträgt maximal $1/(i-1)$, da nur maximal einer der Knoten aus $U' \setminus \{u\}$ adjazent zu v ist. Somit gilt $\mathbf{Pr}\left[v \in e^{(i-1)}\right] \leq 1/(i-1)$.

Analog kann man auch für jede Runde k von $i-2$ bis $\lceil n/e \rceil$ argumentieren. Zu Beginn von Runde k ist die Menge U' bereits bekannt, ebenso wie die Knoten aus U' an den Positionen $k+1, \ldots, i$. Unabhängig von der Wahl dieser Knoten und der Menge U' folgt mit denselben Argumenten wie oben $\mathbf{Pr}\left[v \in e^{(k)}\right] \leq 1/k$. Wir nutzen in dieser Analyse essentiell die Reihenfolge aus, in der wir in Phase 3 Informationen über die Reihenfolge der Knoten erhalten.

Insgesamt ergibt sich für jedes $Z \subseteq U$ mit $|Z| = i$ und für jeden Knoten $z \in Z$

$$\begin{aligned}
\mathbf{Pr}\left[v \text{ in Runde } i \text{ frei} \mid U'=Z,\, u=z\right] &= \mathbf{Pr}\left[\bigwedge_{k=\lceil n/e \rceil}^{i-1} v \notin e^{(k)}\right] \\
&\geq \prod_{k=\lceil n/e \rceil}^{i-1} \left(1 - \frac{1}{k}\right) = \prod_{k=\lceil n/e \rceil}^{i-1} \left(\frac{k-1}{k}\right) \\
&= \frac{\lceil n/e \rceil - 1}{i-1} = \frac{\lfloor n/e \rfloor}{i-1}.
\end{aligned}$$

Damit können wir den Beweis des Lemmas abschließen, denn es gilt

$$\begin{aligned}
\mathbf{E}\left[X_i\right] &= \sum_{Z \subseteq U, |Z|=i} \sum_{z \in Z} \mathbf{Pr}\left[U'=Z,\, u=z\right] \cdot \mathbf{Pr}\left[v \text{ in Runde } i \text{ frei} \mid U'=Z,\, u=z\right] \cdot w(e^{(i)}) \\
&\geq \sum_{Z \subseteq U, |Z|=i} \sum_{z \in Z} \mathbf{Pr}\left[U'=Z,\, u=z\right] \cdot \frac{\lfloor n/e \rfloor}{i-1} \cdot w(e^{(i)}) \\
&= \frac{\lfloor n/e \rfloor}{i-1} \cdot \sum_{Z \subseteq U, |Z|=i} \sum_{z \in Z} \mathbf{Pr}\left[U'=Z,\, u=z\right] \cdot w(e^{(i)}) \\
&= \frac{\lfloor n/e \rfloor}{i-1} \cdot \mathbf{E}\left[w(e^{(i)})\right] \\
&\geq \frac{\lfloor n/e \rfloor}{i-1} \cdot \frac{\mathrm{OPT}(G)}{n}.
\end{aligned}$$

Formal korrekt müssten wir in der obigen Rechnung in den ersten drei Zeilen $e^{(i)}(Z, z)$ statt nur $e^{(i)}$ schreiben, da die Kante $e^{(i)}$ von der Wahl von U' und u abhängt. □

Beweis von Theorem 7.9. Das Theorem folgt einfach aus Lemma 7.10 und (7.4):

$$\begin{aligned} w_{\mathrm{ROM}}(G) &= \sum_{i=\lceil n/e \rceil}^{n} \mathbf{E}\left[X_i\right] \geq \sum_{i=\lceil n/e \rceil}^{n} \frac{\lfloor n/e \rfloor}{i-1} \cdot \frac{\mathrm{OPT}(G)}{n} \\ &= \mathrm{OPT}(G) \cdot \frac{\lfloor n/e \rfloor}{n} \cdot \sum_{i=\lceil n/e \rceil}^{n} \frac{1}{i-1} \\ &\geq \mathrm{OPT}(G) \cdot \left(\frac{1}{e} - \frac{1}{n}\right). \end{aligned}$$

Die letzte Ungleichung folgt analog zu (7.3). □

7.3.3 Eine untere Schranke

Der Algorithmus in Abschnitt 7.3.1 wählt den besten Kandidaten mit einer Wahrscheinlichkeit von ungefähr $1/e$ aus. Wir zeigen nun noch, dass dies optimal ist. Der Beweis, den wir präsentieren werden, geht auf eine recht aktuelle Arbeit von Buchbinder et al. [7] zurück. Die Aussage wurde aber bereits in den 60er-Jahren (mehrfach) bewiesen.

Für die untere Schranke betrachten wir zunächst nur *ordnungsbasierte* Algorithmen. Darunter verstehen wir Algorithmen, die bei ihren Entscheidungen nur die Rangfolge der Eignungen berücksichtigen, nicht jedoch die konkreten Werte. Ein ordnungsbasierter Algorithmus erfährt bei einem Bewerbungsgespräch also nicht die Eignung des Kandidaten, sondern nur welche der bisher gesehenen Kandidaten besser oder schlechter sind. Der Algorithmus A_k aus Abschnitt 7.3.1 gehört zu der Klasse der ordnungsbasierten Algorithmen.

Theorem 7.11. *Es gibt keinen deterministischen oder randomisierten ordnungsbasierten Online-Algorithmus für das Sekretärinnenproblem im Random-Order-Modell, der den besten Kandidaten mit einer größeren Wahrscheinlichkeit als* $1/e$ *auswählt.*

Beweis. Sei A ein beliebiger randomisierter Online-Algorithmus für das Sekretärinnenproblem. Mit p_i bezeichnen wir die Wahrscheinlichkeit, dass er den Kandidaten an Position i auswählt. Die Wahrscheinlichkeit bezieht sich dabei sowohl auf die zufällige Reihenfolge der Kandidaten als auch auf die zufälligen Entscheidungen des Algorithmus.

Da wir die Wahrscheinlichkeit, mit der der Algorithmus A den besten Kandidaten auswählt, nach oben abschätzen wollen, können wir ohne Beschränkung der Allgemeinheit davon ausgehen, dass der Algorithmus A einen Kandidaten nur dann auswählt, wenn er der beste bislang gesehene Kandidat ist. Mit dieser Annahme können wir die Wahrscheinlichkeiten p_i auch wie folgt schreiben:

$$\begin{aligned} p_i &= \mathbf{Pr}\left[\text{Kandidat an Position } i \text{ wird ausgewählt}\right] \\ &= \mathbf{Pr}\left[\text{Kandidat an Position } i \text{ wird ausgewählt und ist besser als alle Vorgänger}\right]. \end{aligned}$$

Wir bezeichnen im Folgenden mit $\mathcal{A}_i$ das Ereignis, dass der Algorithmus den Kandidaten an Position i auswählt. Ferner bezeichnen wir mit $\mathcal{B}_i$ das Ereignis, dass der Kandidat an Position i besser ist als alle Kandidaten davor. Es gilt dann

$$\begin{aligned} p_i &= \mathbf{Pr}[\mathcal{A}_i] = \mathbf{Pr}[\mathcal{A}_i \cap \mathcal{B}_i] = \mathbf{Pr}[\mathcal{A}_i \mid \mathcal{B}_i] \cdot \mathbf{Pr}[\mathcal{B}_i] \\ &\leq \mathbf{Pr}[\neg(\mathcal{A}_1 \cup \ldots \cup \mathcal{A}_{i-1}) \mid \mathcal{B}_i] \cdot \frac{1}{i} \\ &= \mathbf{Pr}[\neg(\mathcal{A}_1 \cup \ldots \cup \mathcal{A}_{i-1})] \cdot \frac{1}{i}. \end{aligned}$$

In der Abschätzung haben wir ausgenutzt, dass $\mathcal{A}_i \subseteq \neg(\mathcal{A}_1 \cup \ldots \cup \mathcal{A}_{i-1})$ gilt, da Kandidat i nur dann ausgewählt werden kann, wenn alle vorangegangenen Kandidaten abgelehnt wurden. Ebenso haben wir ausgenutzt, dass Kandidat i mit einer Wahrscheinlichkeit von genau $1/i$ der beste der ersten i Kandidaten ist. Die letzte Gleichung gilt, da A ein Online-Algorithmus ist. Das heißt, die Qualität von Kandidat i beeinflusst nicht die Entscheidungen in den ersten $i-1$ Runden. Da die Ereignisse $\mathcal{A}_1, \ldots, \mathcal{A}_{i-1}$ paarweise disjunkt sind (es wird maximal ein Kandidat ausgewählt), gilt

$$\begin{aligned} p_i &\leq \mathbf{Pr}[\neg(\mathcal{A}_1 \cup \ldots \cup \mathcal{A}_{i-1})] \cdot \frac{1}{i} \\ &= \left(1 - \sum_{j=1}^{i-1} \mathbf{Pr}[\mathcal{A}_j]\right) \cdot \frac{1}{i} \\ &= \left(1 - \sum_{j=1}^{i-1} p_j\right) \cdot \frac{1}{i}, \end{aligned}$$

wobei die letzte Gleichung aus der Definition von p_j folgt. Insgesamt können wir festhalten, dass die Werte $p_1, \ldots, p_n$ den folgenden Bedingungen genügen müssen:

$$\begin{aligned} &\forall i \in \{1, \ldots, n\} : \quad \sum_{j=1}^{i-1} p_j + i p_i \leq 1, \\ &\forall i \in \{1, \ldots, n\} : \quad p_i \geq 0. \end{aligned} \tag{7.6}$$

Das nächste Ziel ist es, die Wahrscheinlichkeit, den besten Kandidaten auszuwählen, mithilfe der Werte p_i auszudrücken. Diese Wahrscheinlichkeit entspricht dem folgenden Ausdruck:

$$\begin{aligned} &\sum_{i=1}^{n} \mathbf{Pr}[\text{bester Kandidat an Position } i] \cdot \mathbf{Pr}[\mathcal{A}_i \mid \text{bester Kandidat an Position } i] \\ &= \sum_{i=1}^{n} \frac{1}{n} \cdot \mathbf{Pr}[\mathcal{A}_i \mid \mathcal{B}_i] = \sum_{i=1}^{n} \frac{1}{n} \cdot \frac{\mathbf{Pr}[\mathcal{A}_i \cap \mathcal{B}_i]}{\mathbf{Pr}[\mathcal{B}_i]} = \sum_{i=1}^{n} \frac{i}{n} \cdot p_i. \end{aligned}$$

Die erste Gleichung folgt, da ein Online-Algorithmus nicht unterscheiden kann, ob der Kandidat an Position i insgesamt der beste Kandidat ist oder nur der beste bisher gesehene Kandidat. Die zweite Gleichung folgt aus der Definition der bedingten Wahrscheinlichkeit und die dritte Gleichung folgt aus der Definition von p_i und der

Tatsache, dass der Kandidat an Position i mit Wahrscheinlichkeit $1/i$ der beste der ersten i Kandidaten ist.

Wir sind nun in der Lage, einen optimalen ordnungsbasierten Online-Algorithmus für das Sekretärinnenproblem zu charakterisieren. Er muss dergestalt sein, dass die Werte $p_1, \ldots, p_n$ die Nebenbedingungen (7.6) einhalten und unter diesen Nebenbedingungen die Zielfunktion

$$\sum_{i=1}^{n} \frac{i}{n} \cdot p_i$$

maximieren. Da sowohl die Zielfunktion als auch die Nebenbedingungen linear in den p_i sind, handelt es sich hierbei um ein lineares Programm mit den Variablen $p_1, \ldots, p_n$. Durch Lösen dieses linearen Programms erhält man einen optimalen Online-Algorithmus und eine obere Schranke, wie groß die Wahrscheinlichkeit, den besten Kandidaten zu wählen, maximal sein kann.

Wie die optimale Lösung des linearen Programms aussieht, ist auf den ersten Blick nicht direkt ersichtlich. Für konkrete Werte von n kann man das lineare Programm mit einem entsprechenden Solver[2] lösen. Man erhält dann eine Folge von Werten, die für wachsendes n gegen $1/e$ zu konvergieren scheint. Dass dies tatsächlich der Fall ist, kann man mit dem Dualitätsprinzip der linearen Programmierung beweisen. Das duale lineare Programm hat die folgende Form:

$$\begin{aligned} &\text{minimiere } \sum_{i=1}^{n} x_i \\ &\forall i \in \{1, \ldots, n\}: \quad i \cdot x_i + \sum_{j=i+1}^{n} x_j \geq \frac{i}{n} \\ &\forall i \in \{1, \ldots, n\}: \quad x_i \geq 0. \end{aligned}$$

Jede gültige Lösung für dieses duale lineare Programm liefert eine obere Schranke für den Wert, den das primale lineare Programm annehmen kann. Sei $\tau \in \mathbb{N}$ so gewählt, dass $\sum_{i=\tau}^{n-1} \frac{1}{i} < 1 \leq \sum_{i=\tau-1}^{n-1} \frac{1}{i}$ gilt. Dann ist

$$x_i = \begin{cases} 0 & \text{falls } 1 \leq i < \tau, \\ \frac{1}{n}\left(1 - \sum_{j=i}^{n-1} \frac{1}{j}\right) & \text{falls } \tau \leq i \leq n, \end{cases}$$

eine gültige Lösung des dualen Programms mit Wert $\frac{\tau-1}{n} \sum_{j=\tau-1}^{n-1} \frac{1}{j}$. Aus der Definition von τ folgt leicht, dass dieser Wert für wachsendes n gegen $1/e$ konvergiert. Wir überlassen es dem Leser als Übung, die fehlenden Details und Rechnungen in diesem Beweis zu ergänzen. □

Alexander Gnedin ist es 1994 gelungen, Theorem 7.11 auf Algorithmen zu übertragen, die ihre Entscheidungen von den konkreten Werten der Kandidaten und nicht nur der Rangfolge abhängig machen dürfen [10]. Seinen Beweis werden wir in dieser Vorlesung nicht mehr besprechen. Erwähnt sei nur, dass auch daraus noch nicht

[2]Ein empfehlenswerter LP Solver ist zum Beispiel SoPlex `http://soplex.zib.de/`.

direkt folgt, dass die präsentierten Algorithmen für das klassische Sekretärinnenproblem und für das Online-Matching-Problem optimal sind, denn eigentlich interessiert uns nicht die Wahrscheinlichkeit, den besten Kandidaten auszuwählen, sondern die erwartete Eignung des ausgewählten Kandidaten. Ist der zweitbeste Kandidat immer noch gut, so kann der kompetitive Faktor eines Algorithmus besser als e sein, auch wenn er den besten Kandidaten nur mit einer Wahrscheinlichkeit von $1/e$ auswählt. Ausgehend von dem Ergebnis von Gnedin kann man zeigen, dass es für das klassische Sekretärinnenproblem (und somit auch für das Online-Matching-Problem) keinen Online-Algorithmen gibt, der einen besseren kompetitiven Faktor als e erreicht.

Kapitel 8

Das Minimax-Prinzip

Für viele Probleme in dieser Vorlesung haben wir randomisierte Online-Algorithmen entworfen. Wir haben uns stets die Frage gestellt, ob diese Algorithmen optimal sind oder ob es andere Online-Algorithmen mit besseren kompetitiven Faktoren gibt. Für die meisten Probleme und Algorithmen aus dieser Vorlesung konnten wir diese Frage zwar beantworten, für viele Probleme ist es aber schwierig eine untere Schranke für den besten kompetitiven Faktor zu zeigen, der mit randomisierten Online-Algorithmen erreicht werden kann.

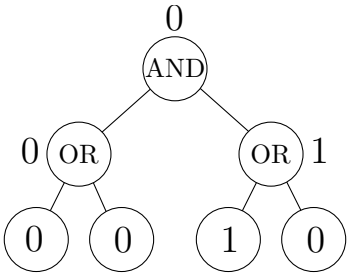
Wir lernen in diesem Abschnitt eine recht allgemeine Methode kennen, mit deren Hilfe man untere Schranken für randomisierte Algorithmen zeigen kann. Diese Methode kann nicht nur im Kontext von Online-Problemen angewendet werden, sondern auch bei bestimmten Arten von Offline-Problemen. Bevor wir diese Methode beschreiben und auf Paging sowie das Online-Matching-Problem anwenden, führen wir zunächst das Problem der *Spielbaumauswertung* ein, anhand dessen wir die Methode illustrieren werden. Die Inhalte dieses Kapitels stammen zu großen Teilen aus den Büchern [6] und [19].

8.1 Spielbaumauswertung

Ein Spielbaum ist ein Baum, in dem jedes Blatt einen reellen Wert besitzt. Jeder innere Knoten besitzt ebenfalls einen Wert, der sich durch eine bestimmte Operation aus den Werten seiner Kinder berechnet. Gängige Operationen sind zum Beispiel MIN und MAX, die den kleinsten bzw. größten Wert eines der Kinder ausgeben. Unter *Spielbaumauswertung* verstehen wir das Problem, den Wert der Wurzel zu bestimmen. Der Name dieses Problems kommt daher, dass Spielbäume bei Schach und anderen Spielen dazu genutzt werden, um möglichst gute Strategien zu berechnen.

Wir betrachten Spielbaumauswertung für den Spezialfall, dass ein vollständiger binärer Baum ausgewertet werden soll, dessen Blätter nur die Werte 0 oder 1 haben. Als Operationen erlauben wir nur MIN- und MAX-Operationen, die wir aufgrund der

booleschen Werte auch als AND- und OR-Operationen bezeichnen werden. Wir bezeichnen einen solchen Baum als T_k-Baum, wenn er die Tiefe $2k$ besitzt, alle inneren Knoten mit einem geraden Abstand von der Wurzel (insbesondere die Wurzel selbst) AND-Operationen berechnen und alle anderen inneren Knoten OR-Operationen berechnen. Die folgende Abbildung zeigt einen T_1-Baum.



Anstatt die Laufzeit von Algorithmen zur Spielbaumauswertung zu analysieren, betrachten wir im Folgenden nur, wie viele Blätter zur Bestimmung des Wertes der Wurzel ausgelesen werden. Alle anderen Operationen, die die Algorithmen durchführen, vernachlässigen wir. Zum einen erleichtert das die Analyse und zum anderen ist die erwartete Anzahl von ausgelesenen Blättern bei dem Algorithmus, den wir betrachten werden, in der gleichen Größenordnung wie seine Laufzeit.

Einen deterministischen Algorithmus können wir als eine adaptive Reihenfolge betrachten, in der die Werte der Blätter ausgelesen werden. Ein solcher Algorithmus liest deterministisch zunächst immer den Wert desselben Blattes. Abhängig davon, ob dort eine 0 oder eine 1 steht, wählt er dann deterministisch das zweite zu lesende Blatt aus und so weiter. Man kann leicht zeigen, dass es für jeden deterministischen Algorithmus eine Eingabe gibt, die ihn dazu zwingt, die Werte aller Blätter auszulesen. Die wesentliche Idee zur Konstruktion dieser Eingabe können wir wie folgt beschreiben. Wir nennen einen Knoten determiniert, wenn sich sein Wert aus den Werten der bisher ausgelesenen Blätter ergibt. Wir nennen einen Knoten vollständig aufgedeckt, wenn die Werte aller Blätter in dem Teilbaum, dessen Wurzel er ist, gelesen wurden. Wir wählen die Werte der gelesenen Blätter so, dass nur vollständig aufgedeckte Knoten determiniert sind. Das heißt insbesondere, dass wir alle Blätter auslesen müssen, bevor der Wert der Wurzel feststeht. Man kann induktiv zeigen, dass stets eine solche Wahl der Werte der Blätter existiert.

Der naheliegende randomisierte Algorithmus RandomGameTree, den wir nun betrachten werden, wertet die Teilbäume eines jeden Knotens in zufälliger Reihenfolge aus. Wertet er beispielsweise einen OR-Knoten aus, der den Wert 1 hat, so hat mindestens einer seiner beiden Söhne ebenfalls den Wert 1. Wählt er diesen zuerst aus, was mit Wahrscheinlichkeit 1/2 passiert, so muss er den anderen nicht mehr auswerten. Genauso verhält es sich mit AND-Knoten mit Wert 0. Wie sieht es jedoch mit OR-Knoten mit Wert 0 und AND-Knoten mit Wert 1 aus? Bei solchen Knoten, können wir durch die zufällige Reihenfolge der Auswertung nichts sparen. Es kann jedoch nicht sein, dass alle Knoten dieser Gestalt sind, denn der Vater eines OR-Knotens mit Wert 0 ist ein AND-Knoten mit Wert 0, also ein für uns guter Knoten. Ebenso ist der Vater eines AND-Knotens mit Wert 1 ein OR-Knoten mit Wert 1, also ebenfalls ein für uns guter Knoten.

Der rekursive Algorithmus RandomGameTree ist im folgenden Pseudocode dargestellt.

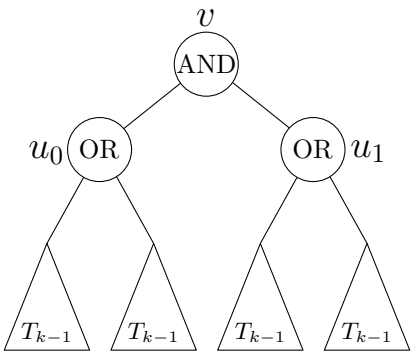
Der initiale Aufruf erfolgt mit den Parametern T und v, wobei T den auszuwertenden Baum und v die Wurzel von T bezeichnet.

RandomGameTree(T, v)

1. **if** (v ist Blatt) **return** Wert von v.
2. **else**
3. Seien u_0 und u_1 die Kinder von v.
4. Wähle $i \in \{0, 1\}$ uniform zufällig.
5. **if** (v ist OR-Knoten) **then**
6. **if** (RandomGameTree$(T, u_i) = 1$) **then return** 1.
7. **else return** RandomGameTree(T, u_{1-i}).
8. **if** (v ist AND-Knoten) **then**
9. **if** (RandomGameTree$(T, u_i) = 0$) **then return** 0.
10. **else return** RandomGameTree(T, u_{1-i}).

Theorem 8.1. *Die erwartete Anzahl an Blättern, die der Algorithmus RandomGameTree ausliest, beträgt für jeden T_k-Baum höchstens 3^k.*

Beweis. Wir zeigen die Aussage mittels vollständiger Induktion über k. Der Induktionsanfang für $k = 0$ ist trivial. Wir betrachten nun den Induktionsschritt für $k \geq 1$. In diesem Fall hat der T_k-Baum einen AND-Knoten v als Wurzel mit zwei OR-Knoten u_0 und u_1 als Söhnen, die jeweils zwei T_{k-1}-Bäume als Nachkommen haben. Diese Situation ist in der folgenden Abbildung dargestellt.



Wir analysieren zunächst die OR-Knoten u_0 und u_1. Wir können für beide Knoten dieselbe Analyse durchführen, weswegen wir nicht zwischen den beiden Knoten unterscheiden und einfach allgemein über den Knoten u_i für $i \in \{0, 1\}$ sprechen. Wir unterscheiden zwei Fälle. Hat der OR-Knoten u_i den Wert 1, so liefert mindestens einer seiner Söhne ebenfalls den Wert 1 zurück. Werten wir diesen zuerst aus, was mit Wahrscheinlichkeit 1/2 passiert, so genügen zur Auswertung von u_i im Erwartungswert gemäß Induktionsvoraussetzung 3^{k-1} Blätter. Der andere Sohn von u_i ergibt im schlimmsten Fall eine 0. Werten wir diesen zuerst aus, so müssen wir auch noch den anderen Sohn auswerten und mit der Linearität des Erwartungswertes ergibt sich eine

erwartete Anzahl ausgelesener Blätter von $2 \cdot 3^{k-1}$. Da beide Fälle mit Wahrscheinlichkeit 1/2 eintreten, können wir die erwartete Anzahl ausgelesener Blätter durch

$$\frac{1}{2} \cdot 3^{k-1} + \frac{1}{2} \cdot 2 \cdot 3^{k-1} = \frac{3}{2} \cdot 3^{k-1}$$

nach oben beschränken. Hat der OR-Knoten u_i den Wert 0, so müssen beide Kinder ausgewertet werden, was eine erwartete Anzahl von höchstens $2 \cdot 3^{k-1}$ ausgelesenen Blättern bedeutet.

Nun betrachten wir die Wurzel v des T_k-Baumes. Dies ist ein AND-Knoten. Falls dieser den Wert 1 hat, so haben beide Söhne ebenfalls den Wert 1. Nutzen wir wieder die Linearität des Erwartungswertes, so ergibt sich mit der obigen Analyse für u_i, dass die erwartete Anzahl ausgelesener Blätter durch

$$2 \cdot \frac{3}{2} \cdot 3^{k-1} = 3^k$$

nach oben beschränkt ist. Ist der Wert des AND-Knotens v gleich 0, so hat mindestens einer seiner Söhne ebenfalls den Wert 0. Wird dieser zuerst ausgewertet, so beträgt die erwartete Anzahl ausgelesener Blätter $2 \cdot 3^{k-1}$. Ansonsten, falls der andere Sohn den Wert 1 hat und dieser zuerst ausgewertet wird, so beträgt die erwartete Anzahl ausgelesener Blätter höchstens $\frac{3}{2} \cdot 3^{k-1} + 2 \cdot 3^{k-1} = \frac{7}{2} \cdot 3^{k-1}$. Da beide Fälle mit Wahrscheinlichkeit 1/2 eintreten, beträgt die erwartete Anzahl ausgelesener Blätter insgesamt höchstens

$$\frac{1}{2} \cdot 2 \cdot 3^{k-1} + \frac{1}{2} \cdot \frac{7}{2} \cdot 3^{k-1} = \frac{11}{4} \cdot 3^{k-1} \leq 3^k. \qquad \square$$

Da die Anzahl an Blättern in einem T_k-Baum $n = 4^k$ beträgt, ist die erwartete Anzahl an Blättern, die der Algorithmus RandomGameTree ausliest, durch

$$3^k = 3^{\log_4 n} = n^{\log_4 3} \leq n^{0{,}793}$$

nach oben beschränkt.

8.2 Das Minimax-Prinzip

8.2.1 2-Personen-Nullsummenspiele

Die Methode zum Beweis von unteren Schranken für randomisierte Algorithmen ist durch Ideen aus der Spieltheorie inspiriert. Zunächst betrachten wir *2-Personen-Nullsummenspiele*. Dabei handelt es sich um Spiele, die durch eine $(n \times m)$-Matrix mit reellen Einträgen beschrieben sind. Es gibt einen Spalten- und einen Zeilenspieler, die jeweils eine Spalte j und eine Zeile i der Matrix auswählen. Der Spaltenspieler muss dann einen Betrag in Höhe von M_{ij} an den Zeilenspieler zahlen. Der Spaltenspieler möchte M_{ij} minimieren, während der Zeilenspieler es maximieren möchte.

Wir können beispielsweise das berühmte Spiel Schere-Stein-Papier als ein 2-Personen-Nullsummenspiel mit der folgenden (3×3)-Matrix M auffassen.

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

Verliert der Spaltenspieler, so zahlt er 1 € an den Zeilenspieler. Gewinnt er, so zahlt er -1 €, erhält also 1 € vom Zeilenspieler. Bei einem Unentschieden findet keine Zahlung statt.

Spieler, die auf Nummer sicher gehen und sich gegen alle möglichen Strategien des anderen Spielers wappnen möchten, verfolgen eine sogenannte Minimax-Strategie. Das heißt, der Zeilenspieler wählt die Zeile i, für die $\min_j M_{ij}$ am größten ist. Damit kann er erreichen, dass er mindestens

$$V_R = \max_i \min_j M_{ij}$$

vom Spaltenspieler erhält. Der Spaltenspieler wählt die Spalte j, für die $\max_i M_{ij}$ am kleinsten ist. Damit kann er erreichen, dass er höchstens einen Betrag von

$$V_C = \min_j \max_i M_{ij}$$

an den Zeilenspieler zahlen muss. Mit dieser Anschauung ist klar, dass $V_R \leq V_C$ für jede Matrix M gilt.

Gilt sogar $V_R = V_C$, so ist es für beide Spieler vernünftig, sich an die Minimax-Strategie zu halten. Dadurch wird dann ein sogenanntes Nash-Gleichgewicht erreicht, in dem es für keinen Spieler Sinn ergibt, seine Strategie zu ändern. Gilt aber $V_R < V_C$ wie im Beispiel Schere-Stein-Papier, dann ist die Minimax-Strategie kein Gleichgewicht, weil man das Wissen über die Strategie des anderen Spielers ausnutzen kann, um die eigene Strategie zu verbessern.

Um dieses Problem zu lösen, nutzt man Randomisierung. Statt eine Zeile zu wählen, wählt der Zeilenspieler eine Wahrscheinlichkeitsverteilung p auf den Zeilen. Wir können $p \in [0,1]^n$ als einen Vektor $p = (p_1, \ldots, p_n)^T$ mit $p_1 + \ldots + p_n = 1$ auffassen. Ebenso wählt der Spaltenspieler eine Wahrscheinlichkeitsverteilung $q \in [0,1]^m$ auf den Spalten, wobei $q = (q_1, \ldots, q_m)^T$ ein Vektor mit $q_1 + \ldots + q_m = 1$ ist. Wählen beide Spieler ihre Strategie unabhängig gemäß dieser Verteilungen, dann beträgt der erwartete Betrag, den der Spaltenspieler an den Zeilenspieler zahlen muss, aufgrund der Linearität des Erwartungswertes genau

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

Wir definieren nun V_R und V_C wie vorhin, nur dass wir diesmal randomisierte Strategien zulassen. Dann ist

$$V_R = \max_p \min_q \left(p^T M q \right)$$

die größte erwartete Auszahlung, die sich der Zeilenspieler gegen jede Gegenstrategie des Spaltenspielers sichern kann. Ebenso kann der Spaltenspieler erreichen, dass er im Erwartungswert niemals mehr als

$$V_C = \min_q \max_p \left(p^T M q\right)$$

an den Zeilenspieler zahlen muss. John von Neumann [18] hat das folgende erstaunliche Resultat bewiesen.

Theorem 8.2. *Für jede Matrix M gilt*

$$V_R = \max_p \min_q \left(p^T M q\right) = \min_q \max_p \left(p^T M q\right) = V_C.$$

Lassen wir also Randomisierung zu, so besitzt jedes 2-Personen-Nullsummenspiel ein Nash-Gleichgewicht, d. h. ein Paar von Strategien, von dem keiner der beiden Spieler einen Anreiz hat, abzuweichen. Eine einfache Folgerung aus diesem Theorem ist das folgende Korollar.

Korollar 8.3. *Es bezeichne e_i den i-ten Einheitsvektor. Für jede Matrix M gilt*

$$V_R = \max_p \min_j \left(p^T M e_j\right) = \min_q \max_i \left(e_i^T M q\right) = V_C.$$

Ist die randomisierte Strategie des Gegners bekannt, so existiert also stets eine beste Gegenstrategie, die deterministisch ist.

8.2.2 Anwendung auf randomisierte Algorithmen

Nun diskutieren wir, wie uns der obige Exkurs in die Spieltheorie dabei hilft, untere Schranken für randomisierte Algorithmen zu zeigen. Dazu betrachten wir ein Problem Π, für das es für jedes $n \in \mathbb{N}$ nur eine endliche Anzahl an Eingaben der Länge n gibt. Wir gehen davon aus, dass n fixiert ist, und bezeichnen mit $\mathcal{I}$ die Menge der Eingaben der Länge n. Außerdem darf es für jede Eingabelänge $n \in \mathbb{N}$ nur endlich viele essentiell verschiedene deterministische Algorithmen für das Problem Π geben. Wieder gehen wir davon aus, dass n fixiert ist, und bezeichnen mit $\mathcal{A}$ die Menge aller Algorithmen für Eingabelänge n. Natürlich kann man für jedes Problem beliebig viele deterministische Algorithmen dadurch erhalten, dass man einen beliebigen korrekten deterministischen Algorithmus nimmt und irgendwelche nicht notwendigen Operationen hinzufügt, die die Ausgabe nicht verändern. Wenn wir von essentiell verschiedenen Algorithmen sprechen, so müssen wir für jedes Problem genau definieren, was damit gemeint ist. Bei dem Problem der Spielbaumauswertung interessiert uns beispielsweise nur die adaptive Reihenfolge, in der die Werte der Blätter ausgelesen werden. Von diesen Reihenfolgen gibt es für jedes feste n nur eine endliche Anzahl.

Für $A \in \mathcal{A}$ und $I \in \mathcal{I}$ sei $C(A, I) \in \mathbb{R}_{\geq 0}$ ein Maß dafür, wie gut Algorithmus A auf der Eingabe I ist. Uns interessiert für das Problem der Spielbaumauswertung der Spezialfall, dass $C(A, I)$ die Anzahl der von A ausgelesenen Blätter bei Eingabe I

angibt, es können aber auch andere Maße eingesetzt werden wie zum Beispiel im Kontext von Online-Algorithmen der kompetitive Faktor von A auf I. Wir nennen $C(A, I)$ im Folgenden auch die Kosten von Algorithmus A auf Eingabe I. Wir definieren nun ein 2-Personen-Nullsummenspiel, in dem der Spaltenspieler dem Algorithmendesigner entspricht und der Zeilenspieler einem Gegner, der eine möglichst schlechte Eingabe wählen möchte. Für einen Algorithmus $A \in \mathcal{A}$ und eine Eingabe $I \in \mathcal{I}$ enthält die $(|\mathcal{I}| \times |\mathcal{A}|)$-Matrix M an der Stelle M_{IA} den Wert $C(A, I)$.

Eine reine Strategie des Spaltenspielers entspricht einem deterministischen Algorithmus aus der Menge $\mathcal{A}$. Eine gemischte Strategie des Spaltenspielers ist eine Wahrscheinlichkeitsverteilung $q \in [0,1]^{|\mathcal{A}|}$ auf den deterministischen Algorithmen. Diese können wir als einen Las-Vegas-Algorithmus betrachten (ein Las-Vegas-Algorithmus ist ein randomisierter Algorithmus, der stets ein korrektes Ergebnis berechnet, deren Laufzeit jedoch eine Zufallsvariable ist), der zufällig gemäß der Verteilung q einen deterministischen Algorithmus auswählt, welchen er dann auf die gegebene Eingabe anwendet. Andersherum können wir jeden Las-Vegas-Algorithmus als eine Wahrscheinlichkeitsverteilung auf der Menge $\mathcal{A}$ der deterministischen Algorithmen darstellen. Der Leser möge an dieser Stelle über ein formales Argument hierfür nachdenken. Das bedeutet insgesamt, dass die Menge der gemischten Strategien des Spaltenspielers genau mit der Menge der Las-Vegas-Algorithmen übereinstimmt.

Theorem 8.2 und Korollar 8.3 liefern angewendet auf das soeben definierte 2-Personen-Nullsummenspiel das folgende Korollar.

Korollar 8.4. *Es bezeichnen P und Q die Mengen der gemischten Strategien des Zeilen- bzw. Spaltenspielers. Für $p \in P$ und $q \in Q$ bezeichnen I_p und A_q eine zufällige Eingabe, die gemäß p gewählt wird, bzw. einen zufälligen Algorithmus, der gemäß q gewählt wird. Es gilt*

$$\max_{p \in P} \min_{q \in Q} \mathbf{E}\left[C(I_p, A_q)\right] = \min_{q \in Q} \max_{p \in P} \mathbf{E}\left[C(I_p, A_q)\right]$$

und

$$\max_{p \in P} \min_{A \in \mathcal{A}} \mathbf{E}\left[C(I_p, A)\right] = \min_{q \in Q} \max_{I \in \mathcal{I}} \mathbf{E}\left[C(I, A_q)\right].$$

Aus diesem Korollar ergibt sich direkt ein weiteres Korollar, das wir zum Beweis unterer Schranken einsetzen werden.

Korollar 8.5. *Für alle Verteilungen $p \in P$ über $\mathcal{I}$ und für alle Verteilungen $q \in Q$ über $\mathcal{A}$ gilt*

$$\min_{A \in \mathcal{A}} \mathbf{E}\left[C(I_p, A)\right] \leq \max_{I \in \mathcal{I}} \mathbf{E}\left[C(I, A_q)\right].$$

Zunächst machen wir uns klar, was die Terme in diesem Korollar bedeuten. Auf der rechten Seite steht $\max_{I \in \mathcal{I}} \mathbf{E}\left[C(I, A_q)\right]$, wobei q eine beliebige Wahrscheinlichkeitsverteilung auf der Menge $\mathcal{A}$ ist, also ein beliebiger Las-Vegas-Algorithmus. Da das Maximum über alle Eingaben $I \in \mathcal{I}$ gebildet wird, entspricht dieser Term genau den erwarteten Kosten des durch q beschriebenen Las-Vegas-Algorithmus auf der schlechtesten Eingabe. Der Term $\min_{A \in \mathcal{A}} \mathbf{E}\left[C(I_p, A)\right]$ auf der linken Seite bezeichnet die erwarteten Kosten des besten deterministischen Algorithmus auf Eingaben, die gemäß der Verteilung p gewählt werden.

Möchten wir mithilfe dieses Korollars also zeigen, dass es keinen Las-Vegas-Algorithmus gibt, dessen erwarteten Worst-Case-Kosten eine gewisse Schranke B unterschreitet, so genügt es, eine Wahrscheinlichkeitsverteilung p auf der Menge der Eingaben zu finden, für die kein deterministischer Algorithmus erwartete Kosten kleiner als B erreichen kann. Auf diese Weise können wir das Problem umgehen, über alle möglichen Las-Vegas-Algorithmen argumentieren zu müssen. Stattdessen genügt es, eine einzige Verteilung auf den Eingaben zu finden, für die alle deterministischen Algorithmen im Erwartungswert mindestens Kosten B erzeugen.

Zwar folgt Korollar 8.5 direkt aus dem Minimax-Theorem über 2-Personen-Nullsummenspiele, da wir dieses aber nicht bewiesen haben, präsentieren wir hier noch einen elementaren Beweis des Korollars, der ohne Spieltheorie auskommt.

Beweis von Korollar 8.5. Es seien $p \in P$ und $q \in Q$ beliebig. Dann gilt

$$\begin{aligned}
\max_{I \in \mathcal{I}} \mathbf{E}\left[C(I, A_q)\right] &= \max_{I \in \mathcal{I}} \sum_{A \in \mathcal{A}} \Big(q(A) \cdot C(I, A) \Big) \\
&\geq \sum_{I \in \mathcal{I}} \Big(p(I) \cdot \sum_{A \in \mathcal{A}} \Big(q(A) \cdot C(I, A) \Big) \Big) \\
&= \sum_{A \in \mathcal{A}} \Big(q(A) \cdot \sum_{I \in \mathcal{I}} \Big(p(I) \cdot C(I, A) \Big) \Big) \\
&\geq \min_{A \in \mathcal{A}} \Big(\sum_{I \in \mathcal{I}} \Big(p(I) \cdot C(I, A) \Big) \Big) \\
&= \min_{A \in \mathcal{A}} \mathbf{E}\left[C(I_p, A)\right],
\end{aligned}$$

wobei wir in der dritten Zeile bei der Umordnung der Summe ausgenutzt haben, dass $\mathcal{A}$ und $\mathcal{I}$ endlich sind. □

In der Literatur wird Korollar 8.5 oft als *Yaos Ungleichung* oder *Yaos Minimax-Prinzip* bezeichnet. Dies geht auf den Informatiker Andrew Yao zurück, der diese Methode zum Beweis unterer Schranken für randomisierte Algorithmen eingeführt hat.

8.3 Untere Schranke für Spielbaumauswertung

Wir möchten nun Korollar 8.5 anwenden, um zu zeigen, dass es keinen Las-Vegas-Algorithmus zur Spielbaumauswertung gibt, der im Erwartungswert wesentlich weniger Blätter betrachtet als der Algorithmus RandomGameTree, den wir oben beschrieben haben. Wir betrachten wieder T_k-Bäume. Anstatt diese über wechselnde Ebenen von AND- und OR-Knoten zu beschreiben, ersetzen wir alle Knoten durch NOR-Knoten. Mithilfe der De Morgan'schen Gesetze sieht man leicht, dass sich der Wert der Wurzel des Spielbaumes dadurch nicht verändert. Wir gehen also davon aus, dass wir einen vollständigen binären Baum der Höhe $2k$ vorliegen haben, in dem alle inneren Knoten die NOR-Funktion berechnen.

Theorem 8.6. *Für jeden Las-Vegas-Algorithmus A und jede hinreichend große Zweierpotenz n gibt es eine Eingabe mit n Blättern, auf der A im Erwartungswert die Werte von mindestens $n^{0,694}$ Blättern ausliest.*

Beweis. Wir zeigen eine Schranke für die erwartete Anzahl an Blättern, die sich jeder Las-Vegas-Algorithmus anschauen muss, bevor er sicher den korrekten Wert an der Wurzel kennt. Korollar 8.5 besagt, dass es dazu genügt, eine Wahrscheinlichkeitsverteilung auf der Menge der Eingaben anzugeben, auf der alle deterministischen Algorithmen schlecht abschneiden. Wir wählen eine Verteilung auf T_k-Bäumen, in der jedes Blatt unabhängig von den anderen mit einer Wahrscheinlichkeit von $r = \left(3 - \sqrt{5}\right)/2$ auf 1 und sonst auf 0 gesetzt wird. Diese Verteilung hat die Eigenschaft, dass die Wahrscheinlichkeit, dass ein NOR zweier Blätter den Wert 1 ergibt, wieder genau

$$(1-r)^2 = \left(1 - \frac{3-\sqrt{5}}{2}\right)^2 = \left(\frac{\sqrt{5}-1}{2}\right)^2 = \frac{6-2\sqrt{5}}{4} = r$$

beträgt. Somit verhalten sich alle Ebenen des T_k-Baumes in gewisser Weise identisch.

Sei v ein Knoten eines solchen T_k-Baumes. Intuitiv ergibt es Sinn, dass ein optimaler deterministischer Algorithmus zunächst den Wert eines Teilbaumes von v bestimmt, bevor er damit beginnt, Blätter in dem anderen Teilbaum auszulesen. Wir nennen eine Reihenfolge von Blättern DFS-Reihenfolge, wenn die Blätter in dieser Reihenfolge von einer Tiefensuche besucht werden können. Dann ergibt es intuitiv Sinn, wenn der Algorithmus sich die Blätter in einer DFS-Reihenfolge anschaut, wobei er Teilbäume überspringt, deren Werte nicht mehr relevant sind. Solche Algorithmen nennen wir *DFS-Pruning-Algorithmen*. Tatsächlich kann man zeigen, dass es für die Verteilung, die wir gewählt haben, einen optimalen Algorithmus gibt, der ein DFS-Pruning-Algorithmus ist.

Lemma 8.7. *Es sei T ein NOR-Baum, dessen Blätter unabhängig voneinander mit einer festen Wahrscheinlichkeit $r \in [0,1]$ auf 1 gesetzt werden. Es sei $W(T)$ die erwartete Anzahl an Blättern, die ein optimaler deterministischer Algorithmus auslesen muss, um T auszuwerten. Dann gibt es einen DFS-Pruning-Algorithmus, der im Erwartungswert $W(T)$ Blätter ausliest, um T auszuwerten.*

Der Beweis dieses Lemmas, den wir hier nicht präsentieren, kann per Induktion über die Höhe von T geführt werden.

Sei nun ein beliebiger deterministischer DFS-Pruning-Algorithmus fixiert. Die erwartete Anzahl an Blättern $W(h)$, die er für einen NOR-Baum der Höhe h ausliest, können wir induktiv wie folgt beschreiben. Es gilt $W(0) = 1$ und für $h \geq 1$ gilt

$$W(h) = W(h-1) + (1-r) \cdot W(h-1) = (2-r) \cdot W(h-1).$$

Dies liegt daran, dass der erste Teilbaum der Wurzel, den der Algorithmus auswertet, mit einer Wahrscheinlichkeit von $(1-r)$ den Wert 0 besitzt. Nur in diesem Fall muss auch der zweite Teilbaum ausgewertet werden. Somit gilt $W(h) \geq (2-r)^h$. Die Anzahl der Blätter eines NOR-Baumes der Höhe h beträgt $n = 2^h$ und damit gilt

$$W(h) \geq (2-r)^{\log_2 n} = n^{\log_2 (2-r)} \geq n^{0,694}.$$

Damit ist gezeigt, dass jeder DFS-Pruning-Algorithmus im Erwartungswert mindestens $n^{0,694}$ viele Blätter ausliest. Aufgrund von Lemma 8.7 bedeutet das, dass jeder deterministische Algorithmus auf Eingaben, die gemäß der oben beschriebenen Verteilung gewählt werden, im Erwartungswert mindestens $n^{0,694}$ viele Blätter ausliest. □

Die Schranke aus dem vorangegangenem Theorem entspricht nicht ganz der oberen Schranke, die der Algorithmus RandomGameTree erreicht. Mit deutlich mehr Aufwand kann man jedoch sogar zeigen, dass der Algorithmus RandomGameTree optimal ist.

8.4 Anwendungen auf Online-Probleme

Wir werden das Minimax-Prinzip nun auf zwei Online-Probleme anwenden, die wir bereits in dieser Vorlesung diskutiert haben. Zum einen werden wir einen weiteren Beweis für Theorem 2.14 angeben, der das Minimax-Prinzip benutzt und eleganter als der Beweis aus Kapitel 2 ist. Zum anderen werden wir die in Abschnitt 7.2 getätigte Aussage beweisen, dass es sich bei RANKING, um einen (nahezu) optimalen Online-Algorithmus für das Online-Matching-Problem handelt.

8.4.1 Paging

Wir geben nun einen alternativen Beweis für Theorem 2.14.

Theorem 8.8. *Es existiert kein randomisierter Online-Algorithmus für das Paging-Problem mit einem kleineren kompetitiven Faktor als H_k.*

Beweis. Zunächst beobachten wir, dass wir bei einer Paging-Eingabe σ der Länge n ohne Beschränkung der Allgemeinheit annehmen können, dass nur die Seitennummern $1, \dots, n$ verwendet werden. Dies bedeutet, dass es nur endlich viele essentiell verschiedene Eingaben der Länge n gibt. Ebenso gibt es nur endlich viele essentiell verschiedene deterministische Algorithmen für Eingaben der Länge n. Die Voraussetzungen zur Anwendung des Minimax-Prinzip sind also gegeben.

Seien $r < H_k$ und $\tau \in \mathbb{R}_{\geq 0}$ beliebige Konstanten. Für einen Online-Algorithmus A und eine Eingabe σ setzen wir

$$C(A, \sigma) = w_A(\sigma) - (r \cdot \mathrm{OPT}(\sigma) + \tau).$$

Für jedes n können wir nun eine endliche Matrix erstellen, die für jede Eingabe der Länge n eine Zeile und für jeden deterministischen Online-Algorithmus für Eingaben der Länge n eine Spalte enthält und deren Einträge den Werten $C(A, \sigma)$ entsprechen.

Als nächstes müssen wir eine Wahrscheinlichkeitsverteilung p auf der Menge der Eingaben der Länge n definieren. Dazu definieren wir einfach, dass jedes σ_i für $i \in \{1, \dots, n\}$ unabhängig und uniform zufällig aus der Menge $\{1, \dots, k+1\}$ gewählt wird. Das bedeutet, dass p eine Gleichverteilung auf der Menge aller Eingaben der Länge n ist, die nur aus Seitenzugriffen auf die Seiten $\{1, \dots, k+1\}$ bestehen.

Es bezeichne σ^p eine zufällige Eingabe, die gemäß dieser Verteilung p erzeugt wird. Sei A ein beliebiger deterministischer Online-Algorithmus. Dann gilt $\mathbf{E}[w_A(\sigma^p)] \geq n/(k+1)$, da der Algorithmus A in jedem Schritt mindestens eine Seite nicht im Cache hat und diese mit einer Wahrscheinlichkeit von $1/(k+1)$ angefragt wird.

Um die Anzahl der Seitenfehler eines optimalen Offline-Algorithmus abzuschätzen, teilen wir die Eingabe σ^p wie bei Markierungsalgorithmen in Phasen ein. Da es insgesamt nur $k+1$ viele verschiedene Seiten in der Eingabe σ^p gibt, können wir die Anzahl an Seitenfehlern eines optimalen Offline-Algorithmus durch $k+(\ell-1)$ nach oben abschätzen, wobei ℓ die Anzahl an Phasen in der Eingabe σ^p bezeichnet (k Seitenfehler in der ersten Phase sowie jeweils ein Seitenfehler in jeder weiteren Phase). Um die erwarteten Kosten $\mathbf{E}[\mathrm{OPT}(\sigma^p)]$ nach oben abzuschätzen, genügt es also die erwartete Anzahl an Phasen in σ^p nach oben abzuschätzen.

Die erwartete Länge einer Phase lässt sich leicht mit dem *Coupon Collector's Problem* bestimmen. Dieses gut untersuchte Zufallsexperiment lässt sich wie folgt beschreiben. Anlässlich einer großen Sportveranstaltung gibt es eine Menge von m unterschiedlichen Sammelbildern. Anstatt gezielt eines dieser Sammelbilder zu kaufen, kann man lediglich ein verschlossenes Paket kaufen, das ein uniform zufälliges Sammelbild enthält. Es ist ein bekanntes Ergebnis, dass man im Erwartungswert genau mH_m Pakete kaufen muss, um alle Sammelbilder zu erhalten. Hier steht H_m wie üblich für die m-te harmonische Zahl. Der Leser sollte sich als Übung einen Beweis dieser Aussage überlegen. Die zufälligen Seiten, auf die in der Sequenz σ^p zugegriffen wird, entsprechen den Sammelbildern. Da eine Phase genau einen Schritt vor dem Zugriff auf die letzten Seite endet, beträgt die erwartete Länge einer Phase genau $(k+1)H_{k+1} - 1 = (k+1)H_k$.

An dieser Stelle ist Vorsicht geboten. Eine naheliegende Schlussfolgerung wäre es, dass die erwartete Anzahl an Phasen $n/((k+1)H_k)$ beträgt. Dies ist jedoch nicht korrekt, da eine solche Rechenregel für Erwartungswerte nicht gilt. Das korrekte Ergebnis ist allerdings ähnlich. Sei $Z(n)$ die Anzahl an Phasen in der zufälligen Eingabe σ^p. Aus Ergebnissen der sogenannten *Erneuerungstheorie (renewal theory)*, auf die wir hier nicht weiter eingehen werden, folgt

$$\lim_{n\to\infty} \frac{\mathbf{E}[Z(n)]}{n} = \frac{1}{(k+1)H_k}.$$

Intuitiv gilt diese Formel, da sich die durchschnittliche Phasenlänge für große n der erwarteten Phasenlänge annähert. Somit gilt für große n also tatsächlich, dass die erwartete Anzahl an Phasen in σ^p in etwa $n/((k+1)H_k)$ beträgt. Damit gilt

$$\mathbf{E}[\mathrm{OPT}(\sigma^p)] = \frac{n}{(k+1)H_k} + O(1).$$

Zusammen mit $\mathbf{E}[w_A(\sigma^p)] = n/(k+1)$ impliziert dies, dass der Algorithmus A auf der zufälligen Eingabe σ^p keinen kompetitiven Faktor kleiner als H_k erreicht.

Für jedes $r < H_k$ und τ gibt es somit ein hinreichend großes n, sodass $\mathbf{E}[C(A, \sigma^p)] > 0$ für jeden deterministischen Algorithmus A gilt. Mit Korollar 8.5 folgt für alle randomisierten Algorithmen A_q

$$\max_{I\in\mathcal{I}} \mathbf{E}[C(I, A_q)] \geq \min_{A\in\mathcal{A}} \mathbf{E}[C(\sigma^p, A)] > 0.$$

Somit existiert für jeden randomisierten Online-Algorithmus A_q eine Eingabe I mit

$$\mathbf{E}[C(I, A_q)] > r \cdot \mathrm{OPT}(\sigma) + \tau.$$

Dies bedeutet, dass für kein $r < H_k$ ein r-kompetitiver randomisierter Online-Algorithmus für Paging existiert. □

8.4.2 Matching

Wir betrachten nun das Online-Matching-Problem aus Abschnitt 7.2. Karp, Vazirani und Vazirani [11] haben mithilfe des Minimax-Prinzips nachgewiesen, dass RANKING ein asymptotisch optimaler randomisierter Online-Algorithmus ist, dass es also keinen randomisierten Online-Algorithmus gibt, der einen besseren kompetitiven Faktor als $e/(e-1) \approx 1{,}58$ erreicht. Wir zeigen im Folgenden eine etwas schwächere untere Schranke, die auf der gleichen Konstruktion beruht.

Theorem 8.9. *Es gibt keinen randomisierten Online-Algorithmus für das Online-Matching-Problem, der einen kompetitiven Faktor echt kleiner als* 1,52 *erreicht.*

Beweis. Zunächst halten wir fest, dass es für eine feste Eingabelänge nur endlich viele Eingaben und endlich viele deterministische Algorithmus gibt. Genau wie im Beweis von Theorem 8.8 konstruieren wir eine Wahrscheinlichkeitsverteilung auf der Menge der Eingaben, für die kein deterministischer Online-Algorithmus existiert, der einen besseren kompetitiven Faktor als 1,52 erreicht. Daraus folgt dann mit denselben Argumenten wie im Beweis von Theorem 8.8, dass es keinen randomisierten Online-Algorithmus gibt, der einen besseren kompetitiven Faktor als 1,52 erreicht.

Sei n gegeben. Wir nehmen ohne Beschränkung der Allgemeinheit an, dass n gerade ist und $n \geq 1000$ gilt. Wir konstruieren zunächst einen Graphen $G = (V \cup U, E)$ mit $|V| = |U| = n$. Es sei $U = \{u_1, \ldots, u_n\}$ und $V = \{v_1, \ldots, v_n\}$. Für $i = \{1, \ldots, n\}$ sei $V_i \subseteq V$ die Menge der Knoten aus V, zu der der Knoten u_i benachbart ist. Der Knoten u_1 ist zu allen Knoten aus V benachbart. Es gilt also $V_1 = V$. Für $i \in \{1, \ldots, n-1\}$ sei a_i ein uniform zufälliger Knoten aus V_i. Der Knoten u_{i+1} ist zu genau denselben Knoten aus V benachbart wie der Knoten u_i bis auf a_i, das heißt $V_{i+1} = V_i \setminus \{a_i\}$. Es bezeichne p die Wahrscheinlichkeitsverteilung auf der Menge der Eingaben, die durch diesen Prozess induziert wird.

Sei A ein bester deterministischer Online-Algorithmus für die Wahrscheinlichkeitsverteilung p. Wir können ohne Beschränkung der Allgemeinheit annehmen, dass A ein Greedy-Algorithmus ist, also stets eine Zuweisung vornimmt, sofern dies möglich ist. Der Leser überlege sich, dass man einen beliebigen Algorithmus, der in einem Schritt keine Kante zum Matching hinzufügt, obwohl dies noch möglich wäre, so umbauen kann, dass er eine Kante hinzufügt, ohne ihn dadurch zu verschlechtern.

Sei nun Y_i für $i \in \{1, \ldots, n\}$ eine Zufallsvariable, die angibt, wie viele freie Nachbarn in V der Knoten u_i besitzt, wenn er eintrifft. Es gilt $Y_1 = n$ und die Größe des zufälligen Matchings M, das der Algorithmus A berechnet, lässt sich schreiben als

$$Z := \max\{i \in \{1, \ldots, n\} \mid Y_i > 0\}.$$

Dies gilt, da A ein Greedy-Algorithmus ist, der stets eine Kante zum Matching hinzufügt, sofern dies noch möglich ist.

Gilt $Y_i = 1$, so folgt $Y_{i+1} = 0$, da A ein Greedy-Algorithmus ist und den Knoten u_i dem letzten verbleibenden Nachbarn in V zuweist. Gilt $Y_i \geq 2$, so gilt $Y_{i+1} \in \{Y_i - 1, Y_i - 2\}$. Dies folgt zum einen, da der Algorithmus A den Knoten u_i einem freien Nachbarn zuweist. Dies reduziert die Menge, der für u_{i+1} zur Verfügung stehenden Knoten, um eins. Zum anderen fällt der zufällig aus V_i gewählte Knoten a_i als Nachbar für u_{i+1} weg. Handelt es sich dabei um einen bereits vergebenen Knoten, so hat dies keinen Effekt auf Y_{i+1} und es gilt $Y_{i+1} = Y_i - 1$. Ist der Knoten a_i nach den ersten i Schritten noch frei, so gilt $Y_{i+1} = Y_i - 2$. Aus dieser Überlegung und der Tatsache, dass a_i uniform zufällig aus V_i gewählt wird, folgt für $\ell \geq 2$ direkt

$$\mathbf{Pr}\left[Y_{i+1} = Y_i - 2 \mid Y_i = \ell\right] = \frac{\ell - 1}{|V_i|} = \frac{\ell - 1}{n - i + 1}.$$

Im Zähler steht $\ell - 1$ statt ℓ, da $Y_i = \ell$ die Zahl an freien Knoten in V_i vor dem Eintreffen von u_i bezeichnet. Nachdem u_i ebenfalls zugewiesen wurde, gibt es dementsprechend nur noch $\ell - 1$ freie Knoten in V_i. Wir haben in der obigen Rechnung implizit ausgenutzt, dass es sich bei A um einen Online-Algorithmus handelt, weshalb seine Entscheidungen in den ersten i Schritten unabhängig von der Wahl von a_i sind.

Aus der obigen Gleichung folgt für $\ell \geq 2$

$$\begin{aligned}
\mathbf{E}\left[Y_{i+1} \mid Y_i = \ell\right] &= (\ell - 2) \cdot \frac{\ell - 1}{n - i + 1} + (\ell - 1) \cdot \left(1 - \frac{\ell - 1}{n - i + 1}\right) \\
&= (\ell - 1) - \frac{\ell - 1}{n - i + 1} \\
&= (\ell - 1) \cdot \frac{n - i}{n - i + 1}.
\end{aligned}$$

Für $i \in \{1, \ldots, n\}$ gilt $Y_i \geq n - 2(i - 1)$. Insbesondere gilt für $i \leq n/2$ stets $Y_i \geq 2$ und somit $\mathbf{Pr}\left[Y_i = 0\right] = \mathbf{Pr}\left[Y_i = 1\right] = 0$ sowie $Z \geq n/2 + 1$. Für $i \in \{1, \ldots, n/2\}$ folgt demnach

$$\begin{aligned}
\mathbf{E}\left[Y_{i+1}\right] &= \sum_{\ell=2}^{n} \mathbf{Pr}\left[Y_i = \ell\right] \cdot \mathbf{E}\left[Y_{i+1} \mid Y_i = \ell\right] \\
&= \sum_{\ell=2}^{n} \mathbf{Pr}\left[Y_i = \ell\right] \cdot (\ell - 1) \cdot \frac{n - i}{n - i + 1} \\
&= \frac{n - i}{n - i + 1} \cdot \sum_{\ell=2}^{n} \mathbf{Pr}\left[Y_i = \ell\right] \cdot (\ell - 1) \\
&= \frac{n - i}{n - i + 1} \cdot \left(\sum_{\ell=0}^{n} \mathbf{Pr}\left[Y_i = \ell\right] \cdot (\ell - 1) + \mathbf{Pr}\left[Y_i = 0\right] \right) \\
&= \frac{n - i}{n - i + 1} \cdot \sum_{\ell=0}^{n} \mathbf{Pr}\left[Y_i = \ell\right] \cdot (\ell - 1) \\
&= \frac{n - i}{n - i + 1} \cdot (\mathbf{E}\left[Y_i\right] - 1).
\end{aligned}$$

Mit $\mathbf{E}[Y_1] = n$ folgt aus dieser Formel durch wiederholtes Einsetzen in sich selbst für $i \in \{2, ..., n/2+1\}$

$$\mathbf{E}[Y_i] = (n-i+1) \cdot \Big(1 - \sum_{k=n-i+2}^{n} \frac{1}{k}\Big).$$

Somit gilt für $i \in \{2, ..., n/2+1\}$

$$\begin{aligned}\mathbf{E}[Y_i] &\leq (n-i+1) \cdot \Big(1 - \int_{n-i+2}^{n} \frac{1}{x}\, dx\Big) \\ &= (n-i+1) \cdot \Big(1 - \ln(n) + \ln(n-i+2)\Big) \\ &= (n-i+1) \cdot \Big(1 - \ln\Big(\frac{n}{n-i+2}\Big)\Big).\end{aligned}$$

Es gilt

$$\mathbf{E}\Big[Y_{n/2+1}\Big] \leq \frac{n}{2} \cdot \Big(1 - \ln\Big(\frac{n}{n/2+1}\Big)\Big) \leq 0{,}154n,$$

wobei wir bei der letzten Ungleichung $n \geq 1000$ ausgenutzt haben.

Wir möchten den Erwartungswert von Z nach oben abschätzen. Dazu gehen wir pessimistisch davon aus, dass für $i \geq n/2+2$ stets $Y_{i+1} = \max\{0, Y_i - 1\}$ gilt. Das heißt, wir gehen davon aus, dass in der zweiten Hälfte nur solche Knoten als a_i gewählt werden, die bereits eine inzidente Kante im Matching besitzen. Diese Annahme kann den Erwartungswert von Z nur vergrößern. Unter dieser Annahme ist $Y_{n/2+1} = i$ äquivalent $Z = n/2+1+i$. Somit gilt

$$\begin{aligned}\mathbf{E}[Z] &= \sum_{i=1}^{n} \mathbf{Pr}[Z=i] \cdot i = \sum_{i=n/2+1}^{n} \mathbf{Pr}[Z=i] \cdot i \\ &= \sum_{i=n/2+1}^{n} \mathbf{Pr}\Big[Y_{n/2+1} = i - n/2 - 1\Big] \cdot i \\ &= \sum_{i=0}^{n/2-1} \mathbf{Pr}\Big[Y_{n/2+1} = i\Big] \cdot (i + n/2 + 1) \\ &\leq \sum_{i=0}^{n} \mathbf{Pr}\Big[Y_{n/2+1} = i\Big] \cdot (i + n/2 + 1) \\ &= \mathbf{E}\Big[Y_{n/2+1}\Big] + n/2 + 1 \\ &\leq 0{,}654n + 1 \leq 0{,}655n,\end{aligned}$$

wobei wir bei der letzten Ungleichung wieder $n \geq 1000$ ausgenutzt haben.

Da $\mathrm{OPT}(\sigma^p) = n$ gilt, ist damit gezeigt, dass es für die konstruierte Wahrscheinlichkeitsverteilung auf der Menge der Eingaben keinen deterministischen Online-Algorithmus gibt, der einen besseren kompetitiven Faktor als $\frac{1}{0{,}655} \geq 1{,}52$ erreicht. Das Theorem folgt nun aus Korollar 8.5. □

Literaturverzeichnis

[1] Dimitris Achlioptas, Marek Chrobak und John Noga: **Competitive Analysis of Randomized Paging Algorithms**. In *Proc. of the 4th Annual European Symposium on Algorithms (ESA)*, Seiten 419–430, 1996.

[2] Yossi Azar: **On-line Load Balancing**. In Amos Fiat und Gerhard J. Woeginger (Herausgeber): *Online Algorithms: The State of the Art*, Band 1442 der Reihe *Lecture Notes in Computer Science*, Seiten 178–195. Springer, 1996.

[3] Nikhil Bansal, Niv Buchbinder, Aleksander Madry und Joseph Naor: **A Polylogarithmic-Competitive Algorithm for the k-Server Problem**. In *Proc. of the 52nd Annual Symposium on Foundations of Computer Science (FOCS)*, Seiten 267–276, 2011.

[4] Yair Bartal, Béla Bollobás und Manor Mendel: **A Ramsy-type Theorem for Metric Spaces and its Applications for Metrical Task Systems and Related Problems**. In *Proc. of the 42nd Annual Symposium on Foundations of Computer Science (FOCS)*, Seiten 396–405, 2001.

[5] Benjamin E. Birnbaum und Claire Mathieu: **On-line bipartite matching made simple**. SIGACT News, 39(1):80–87, 2008.

[6] Allan Borodin und Ran El-Yaniv: **Online computation and competitive analysis**. Cambridge University Press, 1998.

[7] Niv Buchbinder, Kamal Jain und Mohit Singh: **Secretary Problems via Linear Programming**. In *Proc. of the 14th Intl. Conference on Integer Programming and Combinatorial Optimization (IPCO)*, Seiten 163–176, 2010.

[8] Matthias Englert, Heiko Röglin, Jacob Spönemann und Berthold Vöcking: **Economical Caching**. ACM Transactions on Computation Theory, 5(2), 2013.

[9] Jittat Fakcharoenphol, Satish Rao und Kunal Talwar: **A tight bound on approximating arbitrary metrics by tree metrics**. Journal of Computer and System Sciences, 69(3):485–497, 2004.

[10] Alexander V. Gnedin: **A Solution to the Game of Googol**. The Annals of Probability, 22(3):1588–1595, 1994.

[11] Richard M. Karp, Umesh V. Vazirani und Vijay V. Vazirani: **An Optimal Algorithm for On-line Bipartite Matching**. In *Proc. of the 22nd Annual ACM Symposium on Theory of Computing (STOC)*, Seiten 352–358, 1990.

[12] Thomas Kesselheim, Klaus Radke, Andreas Tönnis und Berthold Vöcking: **An Optimal Online Algorithm for Weighted Bipartite Matching and Extensions to Combinatorial Auctions.** In *Proc. of the 21st Annual European Symposium on Algorithms (ESA)*, Seiten 589–600, 2013.

[13] Bernhard Korte und Jens Vygen: **Combinatorial Optimization: Theory and Algorithms**. Springer, 4. Auflage, 2007.

[14] Ulrich Krengel: **Einführung in die Wahrscheinlichkeitstheorie und Statistik**. Vieweg+Teubner Verlag, 2005.

[15] Mark S. Manasse, Lyle A. McGeoch und Daniel Dominic Sleator: **Competitive Algorithms for Server Problems**. J. Algorithms, 11(2):208–230, 1990.

[16] Lyle A. McGeoch und Daniel Dominic Sleator: **A Strongly Competitive Randomized Paging Algorithm**. Algorithmica, 6(6):816–825, 1991.

[17] Michael Mitzenmacher und Eli Upfal: **Probability and Computing**. Cambridge University Press, 2005.

[18] John von Neumann: **Zur Theorie der Gesellschaftsspiele**. Mathematische Annalen, 100:295–320, 1928.

[19] Rajeev Motwani und Prabhakar Raghavan: **Randomized Algorithms**. Cambridge University Press, 1995.

[20] Heiko Röglin: **Randomisierte und approximative Algorithmen**, Vorlesungsskript, Universität Bonn, Wintersemester 2011/12. `http://www.roeglin.org/teaching/WS2011/RandomisierteAlgorithmen/RandomisierteAlgorithmen.pdf`.

[21] Jiri Sgall: **On-line Scheduling**. In Amos Fiat und Gerhard J. Woeginger (Herausgeber): *Online Algorithms: The State of the Art*, Band 1442 der Reihe *Lecture Notes in Computer Science*, Seiten 196–231. Springer, 1996.