Eine Einführung in Online-Algorithmen

Dennis Komm

Skript zur Vorlesung
Approximations- und Online-Algorithmen
Frühlingssemester 2024, ETH Zürich

Stand: 6. September 2024
Kontakt: dennis.komm@inf.ethz.ch

Nehmen wir an, Sie wollten zum ersten Mal in Ihrem Leben Ski fahren gehen, besitzen aber keine eigene Ausrüstung. Der Urlaub ist gebucht und prinzipiell so lang wie Sie wollen, ferner sind Sie in hervorragender körperlicher Verfassung. Das einzige, was Ihren Urlaub begrenzt, ist das Wetter und Sie erhalten leider erst zu Beginn eines Tages eine zuverlässige Vorhersage für diesen Tag. Sie haben nun die Möglichkeit, billig eine Ski-Ausrüstung für 1 CHF am Tag zu mieten oder sie für k CHF zu kaufen, wobei k eine positive ganze Zahl ist, die wesentlich grösser ist als 1. Ist es nun besser für Sie, die Ausrüstung wiederholt zu leihen oder sie zu kaufen?

Offensichtlich hängt die beste Strategie, die Sie verfolgen können, davon ab, wie viele Tage Sie tatsächlich Ski zu fahren imstande sind; gibt es wenige Tage mit gutem Wetter, würden Sie die Skier für diese Tage leihen, gibt es jedoch viele gute Tage, ergibt es mehr Sinn, sie zu kaufen. Sie möchten Ihre Kosten nun nicht unbedingt in einem allgemeinen Sinne gering halten, sondern derart, dass Sie sich möglichst wenig ärgern, wenn Sie das, was Sie tatsächlich bezahlt haben, mit dem vergleichen, was eine optimale Strategie Sie hätte zahlen lassen.

Es ist nicht sinnvoll, die Ski-Ausrüstung für eine beliebig lange Zeit zu mieten. Ab einer gewissen Anzahl von Tagen mit gutem Wetter sollte sie sicherlich gekauft werden, damit die Kosten überschaubar bleiben. Nehmen wir einmal an, Sie kaufen die Ausrüstung am i-ten Tag mit gutem Wetter, wobei $1 \leq i < k$ gilt. Somit zahlen Sie insgesamt $i - 1 + k$ CHF und, wie das Schicksal so will, ist das Wetter anschliessend für alle weiteren Tage so schlecht, dass Sie danach nicht mehr fahren können. Eine optimale Strategie wäre somit gewesen, die Ausrüstung alle i Tage jeweils zu leihen und insgesamt Kosten von i CHF zu haben. Das Verhältnis dieser beiden Grössen ist also $(i - 1 + k)/i \geq 2$. Entscheiden Sie sich auf der anderen Seite, die Ausrüstung am j-ten Tag mit schönem Wetter zu kaufen, wobei $j > k$ ist, wäre die optimale Strategie, dies gleich am ersten Tag zu tun und das Verhältnis ist $(j - 1 + k)/k \geq 2$. Die einzige bislang noch nicht betrachtete Option ist, die Ausrüstung am k-ten Tag mit schönem Wetter zu kaufen und somit $2k - 1$ zu zahlen. Das Verhältnis ist damit genau $2 - 1/k$ und folglich das Beste, worauf Sie im schlimmsten Fall hoffen dürfen.

Inhaltsverzeichnis

1 Online-Algorithmen und das Paging-Problem

Bisher haben wir effiziente Approximationsalgorithmen für $\mathcal{NP}$-schwere Probleme untersucht. Nun vernachlässigen wir die Laufzeit der betrachteten Algorithmen und widmen uns einem weiteren Problem, mit dem ein Informatiker beim Entwickeln von Programmen konfrontiert ist und das wir in unserem Einführungsbeispiel skizziert haben. Oftmals muss ein Algorithmus gute Entscheidungen treffen, ohne dass alle relevanten Informationen zur Verfügung stehen, die für die optimale Bearbeitung nötig wären. Dieses Problem kann offensichtlich nicht einmal mit beliebig starken Ressourcen gelöst werden.

Betrachten wir ein konkretes Beispiel. Von einem praktischen Standpunkt aus betrachtet ist das grundlegende Prinzip heutiger Rechner die Von-Neumann-Architektur. Als **Von-Neumann-Flaschenhals** wird wiederum die Tatsache bezeichnet, dass die Geschwindigkeit der CPU in der Regel die des Hauptspeichers um ein Vielfaches übersteigt. Deswegen wird ein weiterer weitaus schnellerer (und deswegen teurerer und somit wiederum kleinerer) Cache-Speicher verwendet, in dem Speicherinhalte, die häufig benutzt werden, zwischengespeichert werden sollen. Welche dies sind, hängt natürlich zu einem Grossteil vom jeweiligen Benutzer ab und ist deswegen allgemein schwer vorherzusehen.

Wir betrachten jetzt die Aufgabe eines Betriebssystems, den Speicher so zu verwalten, dass während der Laufzeit möglichst selten auf den Hauptspeicher zugegriffen werden muss. Wir wollen dieses Problem so weit es geht auf seine Essenz herunterbrechen und betrachten deswegen eine stark vereinfachte Version (siehe Abbildung 1 für eine schematische Darstellung). Die Situation, mit der wir in der Praxis konfrontiert sind, wenn wir den Speicher eines Rechners verwalten wollen, ist viel facettenreicher und komplizierter.

- Betrachten wir jetzt also eine Rechner-Architektur mit vereinfachter 2-Level-Speicherhierarchie bestehend aus dem Hauptspeicher und einem Cache.
- Beide Speicher werden in Speicherseiten fixer Grösse eingeteilt; eine Speicherseite hat die Grösse 1.
- Der Hauptspeicher hat Platz für m Speicherseiten und der Cache für k.
- Der Cache ist hochwertiger, deswegen teurer und somit viel kleiner, also gilt $m \gg k$.
- Wir unterteilen die Eingabe in diskrete Zeitschritte und während jedes Schrittes wird eine Seite «angefragt». Die CPU kann angefragte Seiten nur direkt aus dem Cache lesen. Das **Paging-Problem** (wir sagen kurz «Paging») besteht nun darin, möglichst viele Seiten, die während der Laufzeit benötigt werden, im Cache zu halten.
- Ist eine angefragte Seite nicht im Cache, entstehen Kosten durch das Laden der Seite aus dem Hauptspeicher in den Cache.
- Wenn eine neue Seite in den Cache geladen wird, muss unter Umständen eine im Cache vorhandene aus diesem gelöscht werden, um Platz zu schaffen. Ein Algorithmus für Paging muss entscheiden, welche Seite er in diesem Fall löscht, ohne zu wissen, welche Seite als nächstes angefragt wird.

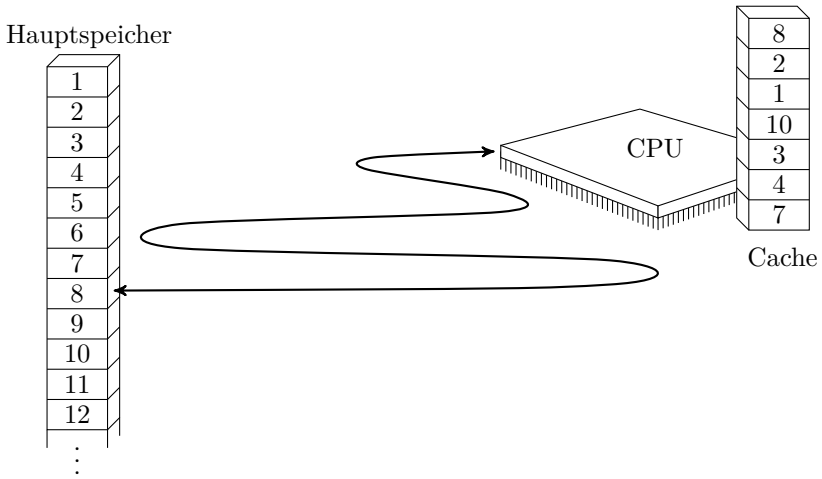


Abbildung 1.

1.1 Online-Algorithmen

Paging ist ein prominenter Verteter einer grossen Familie von Berechnungsproblemen, bei denen die konkrete Eingabe stückweise während der Laufzeit bekannt gegeben wird. Diese Probleme werden als **Online-Probleme** bezeichnet und es ist leicht einzusehen, dass wir sie in vielen Alltagssituationen vorfinden. Was wir zunächst brauchen, ist ein Formalismus, der es uns ermöglicht, Strategien für dieses Problem zu untersuchen; formalisieren wir diese Probleme jetzt allgemein.

Definition 1.1 (Online-Minimierungsproblem). *Ein **Online-Minimierungsproblem** besteht aus einer Klasse von Eingaben $\mathcal{I}$ und einer **Kostenfunktion** cost. Jede Eingabe $I \in \mathcal{I}$ ist eine Folge von **Anfragen** $I = (x_1, \ldots, x_n)$. Ferner gibt es eine Menge von zulässigen Lösungen für jede Eingabe. Jede Lösung O besteht wiederum aus einer Folge von **Antworten** $O = (y_1, \ldots, y_n)$. Die Funktion* cost *ordnet jedem Paar (I, O) einen reellen Wert $\text{cost}(I, O)$ zu. Das Optimierungsziel ist, diesen Wert zu minimieren.*

Die Definition eines Online-Maximierungsproblems ist analog. Anstatt «Kosten» reden wir bei diesen von «Gewinn» und schreiben $\text{gain}(I, O)$ anstatt $\text{cost}(I, O)$. Bei den weiteren Untersuchungen in diesem Kapitel werden wir der Einfachheit halber meistens von Online-Minimierungsproblemen ausgehen. Analoge Resultate lassen sich aber auch für Online-Maximierungsprobleme beweisen. Um die Notation so einfach wie möglich zu halten, schreiben wir ausserdem schlicht $\text{cost}(O)$ anstatt $\text{cost}(I, O)$ beziehungsweise $\text{gain}(O)$ anstatt $\text{gain}(I, O)$, wenn I aus dem Kontext klar ist. Definition 1.1 bringt noch nicht auf den Punkt, was wir ausdrücken wollen. Wir fordern insbesondere, dass ein Algorithmus, der ein derartiges Problem bearbeitet

- zu einem gegebenen Zeitpunkt nur einen Teil der Eingabe kennt,
- Entscheidungen trifft, die nur auf diesem Wissen basieren, und
- bereits getroffene Entscheidungen nicht mehr rückgängig machen kann.

Wir formalisieren dieses Modell in folgender Definition. Online-Algorithmen bezeichnen wir immer mit Buchstaben in Kapitälchen, also beispielsweise ALG, ALG′ oder OPT.

Definition 1.2 (Online-Algorithmus). *Sei $I = (x_1, \ldots, x_n)$ eine Eingabe für ein Online-Minimierungs- oder Online-Maximierungsproblem Π. Ein* ***Online-Algorithmus*** *ALG berechnet die Ausgabe $\text{ALG}(I) = (y_1, \ldots, y_n)$, wobei y_i nur von $x_1, \ldots, x_i$ und $y_1, \ldots, y_{i-1}$ abhängt. $\text{ALG}(I)$ ist hierbei eine zulässige Lösung für I.*

Wie beurteilen wir aber nun die Qualität einer von einem Online-Algorithmus berechneten Lösung? Wir wollen dies so machen, wie wir es im Einführungsbeispiel mit dem Problem Skier zu leihen getan haben. Wir möchten also, dass wir möglichst wenig verlieren, wenn wir unser Ergebnis mit einer optimalen Strategie vergleichen; das Dilemma ist natürlich, dass wir diese Strategie mit etwas Pech erst kennen, wenn die ganze Eingabe bekannt ist. Zu diesem Zweck vergleichen wir die Lösung eines Online-Algorithmus mit der optimalen Lösung eines optimalen **Offline-Algorithmus**, der die Ausgabe erst dann erzeugen muss, wenn die vollständige Instanz bekannt ist. Das Verhältnis der Kosten der berechneten Lösungen nennen wir den **kompetitiven Faktor** (auf englisch «competitive ratio», manchmal übersetzt mit «Wettbewerbsgüte,», «kompetitive Güte» oder «Konkurrenzgüte»).

Definition 1.3 (Kompetitiver Faktor). *Ein Online-Algorithmus ALG ist* ***c-kompetitiv****, wenn eine nicht negative Konstante α existiert, sodass für jede Eingabe I des betrachteten Online-Problems Π gilt, dass*

$$\text{cost}(\text{ALG}(I)) \leq c \cdot \text{cost}(\text{OPT}(I)) + \alpha \ ,$$

falls Π ein Online-Minimierungsproblem ist, oder

$$\text{gain}(\text{OPT}(I)) \leq c \cdot \text{gain}(\text{ALG}(I)) + \alpha \ ,$$

falls Π ein Online-Maximierungsproblem ist, und wobei OPT ein optimaler Offline-Algorithmus und $\text{OPT}(I)$ eine optimale Lösung für I ist. Das kleinste c, für das dies gilt, nennen wir hierbei den ***kompetitiven Faktor*** *von ALG. Gilt obige Ungleichung für $\alpha = 0$, so heisst ALG* ***strikt c-kompetitiv****; ist ALG strikt 1-kompetitiv, so nennen wir ALG* ***optimal****.*

Wir reden, wenn wir Online-Algorithmen aufgrund ihrer kompetitiven Faktoren klassifizieren, von **kompetitiver Analyse** und nennen einen Online-Algorithmus schlicht **kompetitiv** (auf englisch «competitive»), wenn er einen kompetitiven Faktor erreicht, der nicht von der Länge der Eingabe abhängt. Auf der anderen Seite ist ein Online-Algorithmus nicht kompetitiv, wenn er keinen konstanten kompetitiven Faktor besitzt. In vielen Fällen ist es allerdings so, dass der kompetitive Faktor von gewissen Parametern des Problems abhängt. Diese Parameter sind den Online-Algorithmen bekannt, beispielsweise die Cachegrösse bei Paging; es kann sich aber auch um ganze Graphen handeln.

Die Analogie zur bereits vorgestellten Approximationsgüte von Approximationsalgorithmen für Offline-Probleme ist offensichtlich, allerdings haben wir in der entsprechenden

Definition keine Konstante α. Diese ist hier wiederum wichtig, da sie dem Online-Algorithmus erlaubt, schlecht auf kurzen Eingaben zu sein, aber dennoch einen guten kompetitiven Faktor zu besitzen, wenn er mit zunehmender Eingabelänge besser wird. Für Approximationsalgorithmen ist so etwas offensichtlich nicht nötig, da hier optimale Lösungen für Eingaben konstanter Länge in diese hineinkodiert werden können. Dies macht es uns auf der anderen Seite etwas schwerer, untere Schranken zu zeigen.

- Betrachten wir ein Online-Minimierungsproblem, für das wir zeigen können, dass jeder Online-Algorithmus ALG (unabhängig von der Eingabelänge) Kosten von mindestens 10 hat, während ein optimaler Offline-Algorithmus für jede Instanz Kosten 1 besitzt.
- Es liegt nahe zu sagen, ALG sei nun niemals besser als 10-kompetitiv.
- Aber ist dies tatsächlich so? Betrachten wir Definition 1.3 und eine Instanz I des Problems, so sehen wir, dass

  $$\text{cost}(\text{ALG}(I)) \leq 1 \cdot \text{cost}(\text{OPT}(I)) + 9$$

  gelten kann.
- Also existiert eine Konstante α, für die ALG 1-kompetitiv ist (allerdings nicht optimal). Wir können mit unserem Beweis also keine nichttriviale untere Schranke zeigen.
- Können wir jedoch eine Klasse von Instanzen finden, sodass für jede betrachtete Instanz der Länge n gilt, dass ALG mindestens $10n$ zahlt, während OPT Kosten n hat, so können wir in der Tat sagen, dass ALG nicht besser ist als 10-kompetitiv, denn es existiert keine Konstante α sodass, für ein beliebiges $\varepsilon > 0$ und jede Instanz I, gilt

  $$\text{cost}(\text{ALG}(I)) \leq (10 - \varepsilon) \cdot \text{cost}(\text{OPT}(I)) + \alpha \ ,$$

  denn

  $$10n \leq (10 - \varepsilon)n + \alpha \iff \alpha \geq \varepsilon n \ ,$$

  für alle $n \in \mathbb{N}$, führt zu einem Widerspruch.

Dies bedeutet, dass es sinnvoll ist, wenn wir für die optimalen Kosten für alle Instanzen eines betrachteten Online-Problems immer probieren werden,

$$\lim_{|I| \to \infty} \text{cost}(\text{OPT}(I)) = \infty$$

zu fordern. Insbesondere für Paging gehen wir oftmals so vor, dass wir zeigen, dass der Algorithmus für eine bestimmte Anzahl von Zeitschritten um einen gewissen Faktor schlechter ist als ein optimaler Algorithmus. Wir fassen diese Zeitschritte dann zu **Phasen** zusammen, von denen wir beliebig viele wiederholen können.

Jetzt kommen wir zurück zu Paging und definieren es formal als Online-Problem.

Definition 1.4 (Paging). *Eine Eingabe für **Paging** besteht aus einer Folge von natürlichen Zahlen $I = (x_1, \ldots, x_n)$, die Indizes von Speicherseiten entsprechen. Insgesamt gibt es m Speicherseiten $s_1, \ldots, s_m$. In Zeitschritt i wird die Seite mit dem Index x_i angefragt. Ein Online-Algorithmus ALG für Paging verwaltet einen **Cache**-Speicher der Grösse k, gegeben durch die Folge $B = (s_{j_1}, \ldots, s_{j_k})$. Vor dem ersten Zeitschritt wird der Cache initialisiert mit $(s_1, \ldots, s_k)$, also den ersten k Seiten. Wenn ALG nun in einem Zeitschritt i eine Anfrage x_i erhält wobei $s_{x_i} \in B$, so gibt er $y_i = 0$ aus. Gilt allerdings $s_{x_i} \notin B$, so muss ALG eine Seite $s_j \in B$ bestimmen, die aus dem Cache gelöscht wird, um s_{x_i} aufzunehmen; die Ausgabe $y_i = j$ wird nun erzeugt und $B = B \setminus \{s_j\} \cup \{s_{x_i}\}$. Die Kosten sind $\text{cost}(\text{ALG}(x)) = |\{i \mid y_i > 0\}|$ und das Ziel ist, diese zu minimieren.*

Dass jeder Algorithmus mit demselben Cache-Inhalt beginnt, scheint eine sinnvolle Annahme zu sein; schliesslich wollen wir, wenn wir die Resultate eines gegebenen Online-Algorithmus ALG mit denen eines optimalen Algorithmus vergleichen, eine möglichst faire Ausgangssituation haben und sehen, was im besten Fall an ALGs Stelle zu erreichen wäre. Tatsächlich ist diese Annahme weniger restriktiv, als es zunächst erscheint, denn der Vorsprung, den eine optimale Lösung aufgrund des anfänglichen Cache-Inhaltes hat, kann in der Konstanten α aus Definition 1.3 versteckt werden.

Definition 1.4 sieht ferner vor, dass ein Online-Algorithmus für Paging nur dann eine Seite aus dem Cache löscht, wenn die angefragte Seite sich nicht bereits in diesem befindet. Prinzipiell wäre es auch denkbar, zu erlauben, dass in einem beliebigen Zeitschritt Seiten aus dem Cache gelöscht werden können. Es kann aber leicht gezeigt werden, dass dies nie einen Vorteil bringt.

Wir sehen, dass ein Paging-Algorithmus im Wesentlichen dadurch bestimmt wird, wie er die Seiten auswählt, die aus dem Cache gelöscht werden. Wenn eine Seite in den Cache geladen werden muss, sagen wir, dass ein **Seitenfehler** (auf englisch «page fault») auftritt und eine Seite gegebenenfalls aus dem Cache «verdrängt» werden muss; hierfür gibt es diverse Strategien.

- **First-In-First-Out (FIFO).** Bei dieser Strategie wird der Cache wie eine Warteschlange beziehungsweise Queue verwaltet. Muss eine Seite verdrängt werden, so wird diejenige ausgewählt, die bereits am längsten im Cache ist. Die k ersten Seiten werden hierbei beliebig verdrängt.

- **Last-In-First-Out (LIFO).** Diese Strategie ist das Gegenstück zu FIFO, denn sie behandelt den Cache wie einen Stack. Die verdrängte Seite ist jeweils die, die zuletzt in ihn aufgenommen wurde.

- **Least-Recently-Used (LRU).** Hier wird bei einem Seitenfehler die Seite verdrängt, deren letzte Anforderung am längsten her ist. Die ersten k Seiten werden auch hier beliebig verdrängt.

- **Longest-Forward-Distance (LFD).** Diese Strategie verdrängt die Seite, die erst möglichst spät wieder angefragt wird.

Es leuchtet ein, dass LFD recht geringe Kosten verursachen wird, allerdings haben wir es hier mit einer **Offline-Strategie** zu tun, da sie Informationen benötigt, die einem Online-Algorithmus nicht zur Verfügung stehen. Die anderen Strategien sind hingegen Online-Strategien. Wir zeigen nun, dass die FIFO-Strategie es ermöglicht, k-kompetitiv zu sein. Der erreichte kompetitive Faktor hängt also nicht von der Eingabelänge ab.

Satz 1.5. *Ein Online-Algorithmus für Paging, der die FIFO-Strategie umsetzt, ist strikt k-kompetitiv.*

Beweis. Bezeichne Fifo den entsprechenden Online-Algorithmus und sei $(x_1, \ldots, x_n)$ eine beliebige Eingabe für Paging. Beachten Sie, dass laut Definition 1.4 sowohl Fifo als auch ein optimaler Offline-Algorithmus Opt mit demselben Cache-Inhalt starten. Wir unterteilen diese Eingabe nun in aufeinanderfolgende disjunkte Phasen. Die erste endet nach dem ersten Seitenfehler, den Fifo verursacht. Alle weiteren Phasen sind jeweils so lang, dass Fifo genau k Seitenfehler auf ihnen macht und zwar so, dass jeweils direkt nach der k-ten Seite, die einen Fehler verursacht, die entsprechende Phase beendet wird; formal endet Phase P, für $P \geq 2$, nach Seitenfehler $(P-1)k+1$. Die letzte Phase kann offensichtlich kürzer sein.

Wir betrachten Phase P und zeigen, dass Opt in dieser Phase mindestens einen Seitenfehler verursacht. Zunächst stellen wir fest, dass sowohl Opt als auch Fifo in Phase 1 genau einen Seitenfehler machen. Sei für $P \geq 2$ die Seite s diejenige, die als letztes in Phase $P-1$ einen Seitenfehler von Fifo verursachte.

- Alle Seiten, die in P einen Seitenfehler verursachen, werden hiernach vor dem Ende von P nicht mehr gelöscht. Das bedeutet, dass die von Fifo verursachten Kosten sich durch mehrmaliges Anfragen derselben Seite nicht erhöhen. Also verursacht keine Seite in P zwei oder mehr Seitenfehler.

- Die Seite s wird erst mit dem k-ten beziehungsweise letzten Seitenfehler der Phase P verdrängt; sie während P anzufragen, verursacht also keine Kosten für Fifo.

- Da die Seite s am Ende von Phase $P-1$ als letztes einen Seitenfehler für Fifo verursachte und sofort hiernach diese Phase beendet wurde, ist s zu Beginn der Phase P auch im Cache von Opt.

- Dies bedeutet, dass es höchstens $k-1$ Seiten gibt (nämlich höchstens alle weiteren im Cache von Opt zum Beginn der Phase P), deren Anfrage dazu führt, dass Fifo einen Seitenfehler begeht, Opt aber nicht.

- Somit muss Opt während P mindestens einen Seitenfehler machen.

Somit macht Opt in jeder Phase, ausser in der letzten, mindestens einen Seitenfehler. Betrachten wir jetzt noch die letzte Phase. Ist diese vollständig, so macht Opt in ihr ebenfalls einen Seitenfehler. Ist sie unvollständig, so sind die Seitenfehler von Fifo während Phase 1 und der letzten Phase in der Summe höchstens k, wohingegen Opt in diesen beiden Phasen insgesamt mindestens einen Seitenfehler macht. Somit gilt die Aussage sogar, wenn die additive Konstante α aus Definition 1.3 auf 0 gesetzt wird und es folgt, dass Fifo strikt k-kompetitiv ist. □

Mit einer ähnlichen Vorgehensweise wie in obigem Beweis kann gezeigt werden, dass ein Online-Algorithmus, der eine LRU-Strategie umsetzt, ebenfalls einen (strikten) kompetitiven Faktor von höchstens k hat. Die LIFO-Strategie ist hingegen nicht kompetitiv. In Experimenten wurde ferner gezeigt, dass LRU in der Regel bessere Ergebnisse liefert als FIFO. Wir nehmen allerdings wieder den Standpunkt ein, dass wir Garantien für die Güte der betrachteten Algorithmen wünschen und von diesem Standpunkt aus sind die beiden Strategien gleich gut.

1.2 Eine untere Schranke für Paging

Überraschenderweise sind alle Online-Strategien recht schlecht, was Worst-Case-Eingaben betrifft. Um Worst-Case-Eingaben zu modellieren, denken wir uns einen **Gegenspieler** (auf englisch «adversary»), der den Quellcode des Online-Algorithmus genau kennt; somit weiss er, wie dieser auf entsprechende Eingaben reagiert und sein Ziel ist es, dem Online-Algorithmus eine möglichst schwere Eingabe zu geben. Wir können uns einen solchen Gegenspieler bei der Analyse jedweder Algorithmen denken, bei Online-Algorithmen tun wir dies allerdings sehr explizit, da man sich eine Interaktion der beiden Parteien einfacher vorstellen kann; der Gegenspieler macht eine Anfrage, der Online-Algorithmus gibt eine Antwort, der Gegenspieler stellt eine weiter Anfrage usw. Dieser Gegenspieler versucht nun, die Kosten des Online-Algorithmus, verglichen mit denen eines optimalen Offline-Algorithmus, so weit wie möglich nach oben zu treiben; er möchte also den kompetitiven Faktor maximieren.

Wir haben gerade gesehen, dass FIFO einen kompetitiven Faktor von k garantieren kann. Für jede Online-Strategie kann ein Gegenspieler nun eine Eingabe erzeugen, für die keine besserer Lösungsqualität erzielt werden kann. Zu diesem Zweck beschreiben wir unendlich viele Eingaben mit unendlich vielen Längen (beachten Sie unsere Diskussion über die Konstante α).

Satz 1.6. *Kein Online-Algorithmus für Paging kann einen besseren kompetitiven Faktor als k erreichen.*

Beweis. Sei Alg ein beliebiger Online-Algorithmus für Paging. Wir nehmen an, dass es insgesamt $k + 1$ Speicherseiten gäbe, d. h., Alg hat immer genau eine Seite nicht in seinem Cache. Offensichtlich macht diese Einschränkung die bewiesene untere Schranke nur stärker. Wir betrachten den in Algorithmus 1.1 dargestellten Gegenspieler, der also alle Entscheidungen von Alg kennt; wie auch im Folgenden bedienen wir uns einer intuitiven Pseudo-Programmiersprache, die auf Pascal basiert. Die vom Gegenspieler erzeugte Eingabe besteht aus genau k Anfragen.

Da der Gegenspieler genau weiss, welche Seite von Alg im Falle eines Seitenfehlers ausgewählt wird, führt diese Strategie also dazu, dass immer genau die Seite angefragt wird, die soeben aus dem Cache gelöscht wurde. Somit macht Alg garantiert einen Fehler in jedem Zeitschritt.

Dies allein reicht noch nicht, wir müssen ausserdem noch zeigen, dass es tatsächlich einen optimalen Offline-Algorithmus Opt gibt, der wesentlich weniger Fehler macht und zwar genau einen. Dies ist allerdings einfach einzusehen; Opt löscht einfach bei der ersten

Algorithmus 1.1: Gegenspieler für beliebigen Paging-Algorithmus ALG

output «Seite mit Index $k+1$»;
$i := 1$;
while $i \leq k-1$
  $j :=$ Index der soeben von ALG aus dem Cache gelöschten Seite;
  output «Seite mit Index j»;
  $i := i+1$;
end

Anfrage diejenige Seite, die nicht mehr angefragt wird. Das ist immer möglich, denn ausser der Seite mit dem Index $k+1$ werden nur noch $k-1$ verschiedene Seiten mit Indizes aus $\{1, \ldots, k\}$ angefragt.

Nun kann diese Strategie einfach iteriert werden, um eine Eingabe zu erzeugen, die ein beliebiges Vielfaches von k lang ist. Diese Eingabe ist in «Phasen» unterteilt, die genau die Länge k besitzen. OPT macht maximal einen Seitenfehler pro Phase mit demselben Argument wie oben. Dies passiert natürlich nicht notwendigerweise am Anfang. Es kann sogar sein, dass OPT gar keinen Seitenfehler mehr nach dem obligatorischen im ersten Zeitschritt verursacht (in diesem Fall ist ALG nicht kompetitiv). In jedem Fall verursacht OPT aber maximal n/k Seitenfehler, wobei n die Eingabelänge ist, wohingegen ALG genau n Seitenfehler verursacht. □

Wie wir also gesehen haben, kann ein Gegenspieler garantieren, dass jeder Online-Algorithmus in jedem Zeitschritt einen Seitenfehler begeht. Auf der anderen Seite erlaubt die FIFO-Strategie aber einen viel besseren kompetitiven Faktor als beispielsweise die LIFO-Strategie. In einfachen Worten liegt dies daran, dass zwar beide Strategien dieselbe Anzahl von Fehlern machen, FIFO aber dafür sorgt, dass nach einer gewissen Zeit auch ein optimaler Algorithmus eine gewisse Anzahl von Fehlern machen muss, wenn diese Strategie Fehler macht. Dies ist bei LIFO nicht der Fall.

Es ist natürlich zu beachten, dass wir hier einen sehr theoretischen Blick auf Online-Probleme werfen. In der Praxis beobachten wir beispielsweise das Prinzip der «Lokalität», das besagt, dass Speicherseiten, die nahe beeinander liegen, eine höhere Wahrscheinlichkeit haben, nacheinander angefragt zu werden als solche, die dies nicht tun.

1.3 Randomisierte Online-Algorithmen

Was können wir tun, um bessere Karten gegen den Gegenspieler zu haben? Die Idee ist, dem Online-Algorithmus zu erlauben, einen Teil seiner Entscheidungen zufällig zu treffen. Intuitiv verbietet dies dem Gegenspieler, die vom Algorithmus verfolgte Strategie genau zu kennen und somit eine schwere Eingabe zu konstruieren. Definieren wir randomisierte Online-Algorithmen zunächst formal.

Definition 1.7 (Randomisierter Online-Algorithmus). *Sei $I = (x_1, \ldots, x_n)$ eine Eingabe für ein Online-Minimierungs- oder ein Online-Maximierungsproblem Π. Ein* ***randomisierter Online-Algorithmus*** *RAND berechnet die Ausgabe $\text{RAND}^\phi(I) = (y_1, \ldots, y_n)$, wobei y_i nur von $\phi, x_1, \ldots, x_i$ und $y_1, \ldots, y_{i-1}$ abhängt; ϕ bezeichnet einen* ***binären Zufallsstring****, bei dem jedes Bit mit einer Wahrscheinlichkeit von jeweils $1/2$ und unabhängig von allen anderen Bits auf 0 oder 1 gesetzt wird.*

Wenn wir in Zukunft über randomisierte Online-Algorithmen sprechen, lassen wir, um die Notation einfach zu halten, ϕ weg. Wie bei randomisierten Offline-Algorithmen sieht unser Modell ein Zufallsband vor, das unendlich lang ist, und von dem die Zufallsbits sequentiell gelesen werden. Wir messen nun die Güte eines randomisierten Online-Algorithmus, indem wir seine **erwarteten Kosten** mit denen eines optimalen Offline-Algorithmus vergleichen. Zu diesem Zweck modellieren wir die Kosten eines randomisierten Online-Algorithmus RAND auf einer Eingabe I als Zufallsvariable, deren Erwartungswert wir bestimmen.

Definition 1.8 (Erwarteter kompetitiver Faktor). *Ein randomisierter Online-Algorithmus RAND ist* ***c-kompetitiv im Erwartungswert****, wenn eine nicht negative Konstante α existiert, sodass für jede Eingabe I des betrachteten Online-Problems Π gilt, dass*

$$\mathbb{E}[\text{cost}(\text{RAND}(I))] \leq c \cdot \text{cost}(\text{OPT}(I)) + \alpha \ ,$$

falls Π ein Online-Minimierungsproblem ist, oder

$$\text{gain}(\text{OPT}(I)) \leq c \cdot \mathbb{E}[\text{gain}(\text{RAND}(I))] + \alpha \ ,$$

falls Π ein Online-Maximierungsproblem ist, wobei OPT wieder ein optimaler Offline-Algorithmus ist. Das kleinste c, für das dies gilt, nennen wir hierbei den ***erwarteten kompetitiven Faktor*** *von RAND. Gilt obige Ungleichung für $\alpha = 0$, so heisst RAND* ***strikt c-kompetitiv im Erwartungswert****.*

Es wären hier auch andere Grössen als der Erwartungswert denkbar, beispielsweise könnten wir die Kosten betrachten, die RAND mit einer «guten» Wahrscheinlichkeit garantieren kann. Den Erwartungswert zu betrachten scheint allerdings intuitiv sinnvoll, wie folgende Überlegungen aufzeigen.

- Bei schlechten deterministischen Online-Algorithmen existiert eine Eingabe, für die der Online-Algorithmus immer eine schlechte Ausgabe erstellt.
- Für einen guten randomisierten Online-Algorithmus RAND fordern wir, dass er für jede Eingabe im Durchschnitt gut ist. Es kann also passieren, dass RAND für gewisse Zufallsentscheidungen, die er trifft, ebenfalls eine schlechte Ausgabe erzeugt. Ist sein erwarteter kompetitiver Faktor allerdings niedrig, so ist nun die Hoffnung, dass dies entweder selten passiert, dass also, für eine fixe Eingabe I, die Mehrheit der zufälligen Entscheidungen zu einer Ausgabe mit guter Qualität führt, oder er in

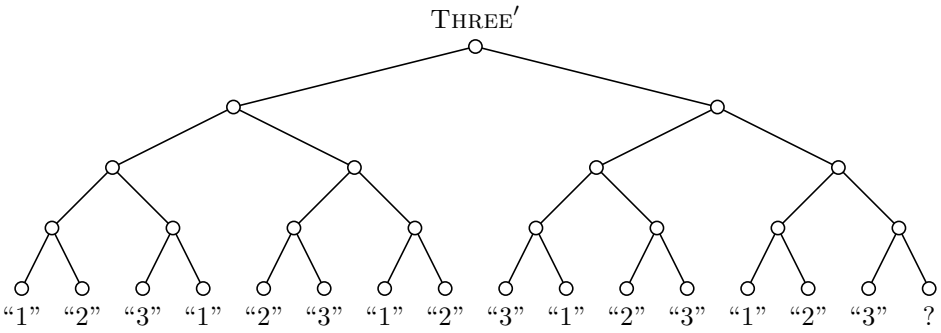


Abbildung 2.

wenigen Fällen eine extrem starke Ausgabe produziert. Es gibt also keine Eingabe, für die der Algorithmus immer schlecht ist.

Ein wichtiger Punkt, den wir bei randomisierten Online-Algorithmen berücksichtigen müssen, ist die Anzahl an Zufallsentscheidungen, die sie machen. Ein Problem für die folgenden Analysen ergibt sich zunächst einmal daraus, dass wir die Anzahl der Zufallsbits, die vom Zufallsband gelesen werden, eigentlich nicht mit absoluter Sicherheit nach oben beschränken können.

Betrachten wir hierzu einen einfachen randomisierten Online-Algorithmus THREE, der nichts anderes tun soll, als uniform eine Zahl 1, 2 oder 3 auszugeben. Wie erreichen wir dies? Die naheliegende Idee ist, THREE zunächst zwei Bits vom Zufallsband lesen zu lassen. Sind beide 0, so wird «1» ausgegeben, ist das erste 0 und das zweite 1, so wird «2» ausgeben und für den Fall, dass das erste Bit 1 und das zweite 0 ist, geben wir «3» aus. Was passiert aber, wenn beide Zufallsbits 1 sind? Nun, in diesem Fall liest THREE zwei weitere Zufallsbits und wiederholt obige Strategie. Allerdings können diese wieder beide 1 sein. Die Wahrscheinlichkeit hierfür ist 1/16 und in diesem Fall muss THREE zwei weitere Zufallsbits lesen. Allgemein ist die Wahrscheinlichkeit, dass wir nach dem n-ten Versuch noch immer kein Ergebnis haben, $1/4^n$, geht also exponentiell schnell gegen 0; aber das reicht uns nicht, um mit Sicherheit sagen zu können, dass THREE tatsächlich terminiert.

In der Tat ist es einfach einzusehen, dass es keine absolute Garantie geben kann. Nehmen wir an, es gäbe eine natürliche Zahl $\overline{n}$, sodass ein randomisierter Online-Algorithmus THREE′ mit absoluter Sicherheit unter Verwendung von $\overline{n}$ Zufallsbits uniform eine Zahl 1, 2 oder 3 ausgibt. THREE′ liest ebenfalls keine Eingabe und somit ist seine Ausgabe offensichtlich durch den Inhalt der ersten $\overline{n}$ Bits auf dem Zufallsband vollständig determiniert. Mit anderen Worten kann sich THREE′ auf $2^{\overline{n}}$ verschiedene Arten verhalten. Wenn uniform eine natürliche Zahl zwischen 1 und 3 ausgegeben werden soll, müssen wir diese Ausgänge gleichmässig auf die Ausgaben «1», «2» und «3» verteilen. Dies ist aber nicht möglich, da 3 keine Zweierpotenz ohne Rest teilt, insbesondere nicht $2^{\overline{n}}$ (siehe Abbildung 2). Beachten Sie, dass es uns ebenfalls im Allgemeinen nicht hilft, das Alphabet des Zufallsbandes zu vergrössern, denn dieses darf nicht von der Eingabe abhängen. Genau genommen müssten wir also selbst den sehr simplen Algorithmus THREE′ analog zu Abbildung 2 als Baum darstellen, der unendlich oft verzweigt.

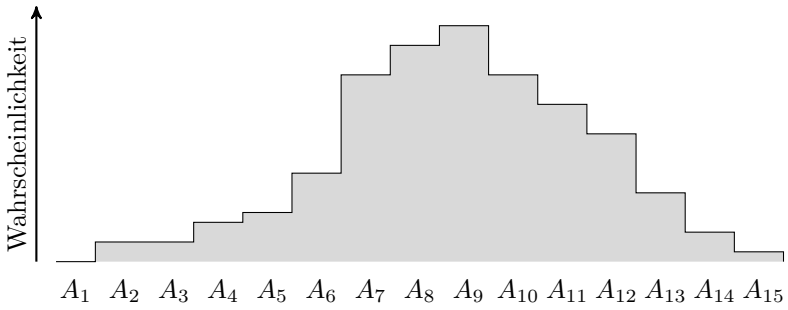


Abbildung 3.

Ein Ausweg aus diesem Dilemma ist, zu analysieren, wie viele Versuche wir im Erwartungswert brauchen, um erfolgreich zu sein. Mit der **Markov-Ungleichung** können wir dann folgern, dass die Wahrscheinlichkeit, t-mal länger zu brauchen, höchstens $1/t$ ist. Wir können deswegen die Anzahl der gelesenen Zufallsbits derart nach oben beschränken, dass die Wahrscheinlichkeit, dass der betrachtete randomisierte Online-Algorithmus mehr Bits braucht, sehr klein ist. Diese letzte Unsicherheit werden wir im Folgenden ignorieren (ähnlich wie wir auch beispielsweise die Wahrscheinlichkeit für das Auftreten von Hardware-Fehlern ausser Acht lassen) und davon ausgehen, dass jeder randomisierte Online-Algorithmus, wenn er eine endlich lange Eingabe liest, eine endliche Anzahl an Zufallsbits benutzt.

Eine weitere Beobachtung, die wir machen, ist, dass es für jede Eingabelänge n für jedes betrachtete Online-Problem eine endliche Anzahl von möglichen Eingaben gibt.

Um untere und obere Schranken für den erwarteten kompetitiven Faktor zu zeigen, benutzen wir meist folgende Betrachtungsweise, wenn wir über randomisierte Online-Algorithmen sprechen und diese analysieren wollen. Sei im Folgenden $b\colon \mathbb{N}^+ \to \mathbb{N}^+$ eine Funktion, die die maximale Anzahl der zufälligen binären Entscheidungen (gewissermassen also der Münzwürfe) eines randomisierten Online-Algorithmus RAND für alle Eingaben der Länge n angibt. Wie gerade besprochen, gehen wir immer davon aus, dass diese Funktion b wohldefiniert ist, dass $b(n)$ also wirklich eine natürliche Zahl ist. Wir können also wieder sagen, dass RAND sich auf maximal $2^{b(n)}$ unterschiedliche Weisen verhält, wenn er eine Eingabe der Länge n liest, wobei wir diesen Wert berechnen können, wenn wir RAND kennen und, wie oben besprochen, berücksichtigen, dass es für jedes n endlich viele mögliche Eingaben gibt.

Die folgende Beobachtung fasst unsere Überlegungen unter obigen Annahmen nun zusammen.

Beobachtung 1.9. *Jeder randomisierte Online-Algorithmus* RAND*, der $b(n)$ Zufallsbits für Eingaben der Länge n liest, kann für Eingaben der Länge n als eine Menge* $\mathrm{strat}(\textsc{Rand}) = \{A_1, \ldots, A_{\ell(n)}\}$ *von $\ell(n) \leq 2^{b(n)}$ deterministischen Online-Algorithmen aufgefasst werden, aus denen* RAND *einen zufällig wählt.*

Schematisch ist diese Sichtweise in Abbildung 3 dargestellt. Um den Kontext hervorzuheben, bezeichnen wir die deterministischen Strategien aus der Menge strat(RAND) immer mit kursiven Buchstaben, also A, B oder C. Die Wahrscheinlichkeitsverteilung, die RAND zu Grunde liegt, ist natürlich beliebig, muss aber mit $b(n)$ Zufallsbits zu realisieren sein.

Um die Analyse später einfacher zu gestalten, machen wir nun eine weitere Beobachtung. Dazu stellen wir fest, dass jede der obigen deterministischen Strategien mit einer Wahrscheinlichkeit ausgewählt wird, die ein Vielfaches von $1/2^{b(n)}$ ist. Ferner können wir, anstatt einen Algorithmus mit einer Wahrscheinlichkeit $a/2^{b(n)}$ auszuwählen, einfach a identische Algorithmen mit einer Wahrscheinlichkeit von jeweils $1/2^{b(n)}$ nehmen. Dies führt zu folgender Beobachtung.

Beobachtung 1.10. *Jeder randomisierte Online-Algorithmus* RAND*, der $b(n)$ Zufallsbits für Eingaben der Länge n liest, kann für Eingaben der Länge n als eine Menge* $\text{strat}(\text{RAND}) = \{A_1, \ldots, A_{2^{b(n)}}\}$ *von $2^{b(n)}$ deterministischen Online-Algorithmen aufgefasst werden, von denen einer mit jeweils einer Wahrscheinlichkeit von $1/2^{b(n)}$ ausgewählt wird.*

Wir werden im Folgenden an manchen Stellen davon ausgehen, dass RAND gleichverteilt einen deterministischen Algorithmus aus strat(RAND) zieht.

Beachten Sie, dass diese Sichtweise für Offline-Algorithmen leichter nachzuvollziehen ist. Wenn wir über das Modell der Turingmaschine argumentieren, so können wir für einen gegebenen Algorithmus RAND einen Algorithmus RAND$'$ konstruieren, der wie folgt funktioniert. Bevor er die erste Anfrage liest, kopiert RAND$'$ einfach $2^{b(n)}$ Bits vom Zufallsband auf sein Arbeitsband. Danach benutzt RAND$'$ letzteres genau so, wie RAND das Zufallsband gebraucht, und greift nicht mehr auf sein Zufallsband zu. Für randomisierte Online-Algorithmen können wir eine solche Betrachtungsweise nicht wählen. Ein randomisierter Online-Algorithmus kann seine Zufallsentscheidungen im Allgemeinen nicht zu Beginn machen; dies würde nämlich voraussetzen, dass dieser Algorithmus selber berechnen kann, wie viele Zufallsbits er brauchen wird. Hierfür muss er wiederum die Eingabelänge kennen und dies verbieten wir natürlich ausdrücklich. Wichtig ist aber, dass wir $2^{b(n)}$ berechnen können, wenn wir diesen randomisierten Online-Algorithmus analysieren.

Wir haben in Definition 1.8 nicht über optimale randomisierte Online-Algorithmen gesprochen. Prinzipiell wäre dies schon sinnvoll; dies wäre also ein randomisierter Online-Algorithmus, der für jede Wahl von Zufallsbits eine optimale Lösung für jede Eingabe berechnet. Die Existenz eines solchen Algorithmus lässt dann jedoch direkt eine viel stärkere Aussage zu.

Satz 1.11. *Wenn für ein Online-Problem Π ein optimaler randomisierter Online-Algorithmus existiert, so existiert für Π auch ein optimaler deterministischer Online-Algorithmus.*

Beweis. Nehmen wir also an, RAND wäre ein optimaler randomisierter Algorithmus für Π. Dann betrachten wir, Beobachtung 1.9 folgend, RAND für jede Eingabelänge als eine Menge strat(RAND) von deterministischen Strategien, aus denen RAND eine auswählt; wichtig ist, dass RAND für eine feste Eingabelänge eine endliche Zahl von Zufallsbits benutzt und die Kardinalität von strat(RAND) somit ebenfalls endlich ist. Dass RAND optimal ist, bedeutet nichts anderes, als dass er für jede Zufallsentscheidung auf jeder Eingabe minimale Kosten hat. Da eine Zufallsentscheidung einfach der Wahl einer Strategie A_j aus strat(RAND) entspricht, ist also jeder deterministische Algorithmus in strat(RAND) optimal für Π. □

Ein ähnlicher Beweis für 1-kompetitive randomisierte Online-Algorithmen ist indes nicht möglich. Hierzu betrachten wir kurz ein künstliches Online-Problem, für das ein «fast» optimaler randomisierter Online-Algorithmus existiert, für das aber jeder deterministische Online-Algorithmus beliebig schlecht ist.

- Sei $\varepsilon > 0$ beliebig, sodass $1/\varepsilon \in \mathbb{N}^+$.
- Die Eingabe $I = (x_1, \ldots, x_n)$ beginnt mit einer Anfrage $x_1 = n$, wobei n ein Vielfaches von $1/\varepsilon$ ist und $n \gg 1/\varepsilon$ gilt.
- Jeder Online-Algorithmus muss auf x_1 eine Antwort y_1 geben, wobei $1 \leq y_1 \leq n$ ist.
- Die zweite Anfrage ist x_2 mit $1 \leq x_2 \leq n$.
- Gilt nun $y_1 = x_2$, so muss der Algorithmus unabhängig von seinen weiteren Antworten in jedem $(1/\varepsilon)$-ten Zeitschritt Kosten von 1 zahlen.
- Gilt hingegen $y_1 \neq x_2$, so zahlt er einmalig Kosten von 1 und sonst nichts mehr.
- Opt hat also immer Kosten 1.
- Offensichtlich sind sowohl die Kosten als auch der kompetitive Faktor jedes deterministischen Online-Algorithmus im schlechtesten Fall genau εn.
- Wir können aber einfach einen randomisierten Online-Algorithmus Rand entwerfen, der nach dem ersten Zeitschritt eine Antwort y_1, $1 \leq y_1 \leq n$, uniform, also jeweils mit einer Wahrscheinlichkeit von $1/n$, wählt und dessen erwartete Kosten

  $$\frac{1}{n} \cdot \varepsilon \cdot n + \left(1 - \frac{1}{n}\right) \cdot 1 \leq 1 + \varepsilon$$

  sind.
- Somit ist Rand also 1-kompetitiv, wobei wir Definition 1.8 entsprechend $\alpha = \varepsilon$ setzen.

Unser Argument funktioniert natürlich nur, weil n nach dem ersten Zeitschritt bekannt ist, wir es aber nicht wie einen Parameter behandeln. Sicherlich ist diese Argumentation hier nicht sehr schön, aber formal richtig.

Wenn wir uns mit randomisierten Offline-Algorithmen beschäftigen, so nutzen wir häufig die sogenannte **Wahrscheinlichkeitsverstärkung** (auf englisch «amplification»). Die Idee ist hierbei, einen Algorithmus mit einseitigem Fehler (einen sogenannten **Monte-Carlo-Algorithmus**), der mit einer Wahrscheinlichkeit von weniger als $1/2$ beispielsweise eine falsche Antwort zu einer Instanz I eines Entscheidungsproblems gibt, mehrmals auf I laufen zu lassen. Bei k Läufen ist die Wahrscheinlichkeit, eine falsche Ausgabe zu erzielen, dann nur noch $1/2^k$. Bei randomisierten Online-Algorithmen können wir eine solche Technik nicht anwenden, da wir hier Antworten auf Anfragen geben müssen, die nicht mehr rückgängig gemacht werden dürfen.

1.4 Ein randomisierter Online-Algorithmus für Paging

Wir betrachten nun den randomisierten Online-Algorithmus RMark (Algorithmus 1.2), der in Phasen arbeitet und Seiten, die bereits angefragt wurden, markiert. Verdrängt werden lediglich unmarkierte Seiten, solange es welche im Cache gibt. Sind alle Seiten markiert und ein Seitenfehler tritt auf, so endet die aktuelle Phase und eine neue beginnt, nachdem zunächst die Markierung aller Cache-Seiten aufgehoben wurde. Vor dem Lesen des ersten Teils der Eingabe markiert RMark alle Seiten im Cache, damit der erste Seitenfehler eine neue Phase einleitet.

Algorithmus 1.2: Algorithmus RMark

```
mark alle Seiten im Cache;
while Eingabe ist noch nicht beendet
  s := Seite mit Index x_i;
  if s ist im Cache
    if s ist unmarkiert
      mark s;
    output «0»;
  else
    if es existiert keine unmarkierte Seite mehr im Cache
      unmark alle Seiten im Cache;
    s' := zufällig gewählte unmarkierte Seite;
    verdränge s' und füge s an der alten Stelle von s' ein;
    mark s;
    output «Index von s'»;
  i := i + 1;
end
```

Unsere Hoffnung ist nun also, dass ein Gegenspieler den Algorithmus RMark nicht mehr so einfach ausspielen kann, da die Seiten, die zu einem gegebenen Zeitpunkt im Cache sind, von Zufallsentscheidungen abhängen. Für die Analyse brauchen wir die folgenden Zahlen. Für jedes $l \in \mathbb{N}^+$ bezeichne H_l die ***l*-te harmonische Zahl**, definiert als

$$H_l := 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{l} = \sum_{i=1}^{l} \frac{1}{i} .$$

Satz 1.12. *Der randomisierte Online-Algorithmus RMark für Paging hat einen erwarteten kompetitiven Faktor von $2H_k$.*

Beweis. Bevor die erste Anfrage bearbeitet wird, werden, wie bereits erwähnt, alle Seiten im Cache markiert. Damit leitet der erste Seitenfehler also eine neue Phase ein. Analysieren wir jetzt zunächst wieder eine einzelne Phase P.

- P endet genau dann, wenn k verschiedene Seiten angefragt wurden. Für die folgende Argumentation gehen wir davon aus, dass keine Seite in P zweimal angefragt wird.

- Bezeichnen wir alle Seiten, die zum Beginn von P im Cache sind, als «alt» und nicht alte Seiten, die während P angefragt werden, als «neu».

- Für jede neue Seite wird eine alte aus dem Cache verdrängt. Es ist somit für den Algorithmus am schlechtesten, wenn zunächst neue Seiten angefragt werden und dann alte, die eventuell bereits entfernt wurden. Erst alte Seiten anzufragen, ist keine Strategie, die dem Gegenspieler einen Vorteil gegenüber ersterer verschafft.

- Bezeichne nun l die Anzahl an neuen Seiten, die in P angefragt werden; diese führen offensichtlich zu l Seitenfehlern bei RMark.

- Weiterhin werden in P auch noch $k-l$ alte Seiten angefragt, die RMark mit einer gewissen Wahrscheinlichkeit nach den ersten l Anfragen noch im Cache hat.

- Wird nun die erste alte Seite angefragt, so gibt es insgesamt k unmarkierte alte Seiten, von denen ein Teil im Cache ist und ein Teil bereits verdrängt wurde; genauer wurden zu diesem Zeitpunkt $k-l$ alte Seiten noch nicht verdrängt. Die Wahrscheinlichkeit, dass die gerade angefragte alte Seite also noch im Cache ist, beträgt somit $(k-l)/k$.

- Anschliessend gibt es insgesamt $k-1$ unmarkierte alte Seiten und $k-l-1$ nicht verdrängte unmarkierte alte Seiten. Die Wahrscheinlichkeit, dass die nächste angefragte Seite noch im Cache ist, beträgt somit $(k-l-1)/(k-1)$.

- Allgemein erhalten wir für die i-te angefragte alte Seite eine Wahrscheinlichkeit von

  $$\frac{k-l-(i-1)}{k-(i-1)}\ ,$$

  dass sie sich noch in RMarks Cache befindet und nicht nachgeladen werden muss.

- Die entsprechende Seite ist also nicht im Cache mit einer Wahrscheinlichkeit von

  $$1-\frac{k-l-(i-1)}{k-(i-1)}=\frac{l}{k-(i-1)}\ ,$$

  mit der für RMark in diesem Zeitschritt Kosten von 1 entstehen.

- Zusammen mit den Kosten l für die neuen Seiten ergibt sich insgesamt für die ewarteten Kosten von RMark in P

  $$\begin{aligned}
  l+\sum_{i=1}^{k-l}\frac{l}{k-(i-1)} &= l+l\Big(\frac{1}{k}+\frac{1}{k-1}+\cdots+\frac{1}{l+1}\Big)\\
  &= l+l\Bigg(\underbrace{\frac{1}{k}+\frac{1}{k-1}+\cdots+1}_{H_k}-\Big(\underbrace{\frac{1}{l}+\frac{1}{l-1}+\cdots+1}_{H_l}\Big)\Bigg)\\
  &= l(H_k-H_l+1)\leq lH_k\ ,
  \end{aligned}$$

wobei wir verwendet haben, dass $l \geq 1$ sein muss (denn jede Phase beginnt ja mit der Anfrage einer neuen Seite). Nun müssen wir die Kosten eines optimalen Algorithmus Opt nach unten abschätzen.

- Wenn wir zwei aufeinanderfolgende Phasen P_{j-1} und P_j betrachten, so wurden in diesen Phasen mindestens $k + l_j$ verschiedene Seiten angefragt, wobei l_j nun die neuen Seiten, die während P_j angefragt wurden, bezeichnet.
- Somit macht Opt mindestens l_j Seitenfehler während dieser zwei Phasen. Wir können die Phasen nun auf zwei Weisen partitionieren, wobei wir entweder mit P_1 oder P_2 anfangen können, also

$$\underbrace{P_1,\ P_2}_{l_2 \text{ Fehler}},\ \underbrace{P_3,\ P_4}_{l_4 \text{ Fehler}},\ P_5,\ \ldots \quad \text{oder} \quad P_1,\ \underbrace{P_2,\ P_3}_{l_3 \text{ Fehler}},\ \underbrace{P_4,\ P_5}_{l_5 \text{ Fehler}},\ \ldots\ .$$

- Ausserdem macht Opt in der ersten Phase l_1 Seitenfehler, da beide Algorithmen mit demselben Cache-Inhalt starten und deswegen gilt

$$\text{cost}(\text{Opt}(I)) \geq \sum_{i=1}^{\lfloor N/2 \rfloor} l_{2i} \quad \text{und} \quad \text{cost}(\text{Opt}(I)) \geq \sum_{i=1}^{\lceil N/2 \rceil} l_{2i-1}\ .$$

- Es folgt

$$\begin{aligned}
\text{cost}(\text{Opt}(I)) &\geq \max\left\{ \sum_{i=1}^{\lfloor N/2 \rfloor} l_{2i},\ \sum_{i=1}^{\lceil N/2 \rceil} l_{2i-1} \right\} \\
&\geq \frac{1}{2}\left(\sum_{i=1}^{\lfloor N/2 \rfloor} l_{2i} + \sum_{i=1}^{\lceil N/2 \rceil} l_{2i-1} \right) \\
&= \sum_{i=1}^{N} \frac{1}{2}\, l_i\ .
\end{aligned}$$

Insgesamt erhalten wir so eine obere Schranke für den kompetitiven Faktor von RMark von

$$\sum_{j=1}^{N} H_k l_j \Big/ \sum_{j=1}^{N} \frac{1}{2}\, l_j = 2H_k\ ,$$

was zu zeigen war. □

Um besser einschätzen zu können, wie weit uns Randomisierung bei Paging hilft, stellen wir fest, dass

$$\sum_{i=1}^{n} \frac{1}{i} \leq 1 + \int_1^n \frac{1}{i}\, \mathrm{d}i = \ln n + 1 \in \mathcal{O}(\log n)\ .$$

Zusammenfassend haben wir also gezeigt (siehe Abbildung 4), dass

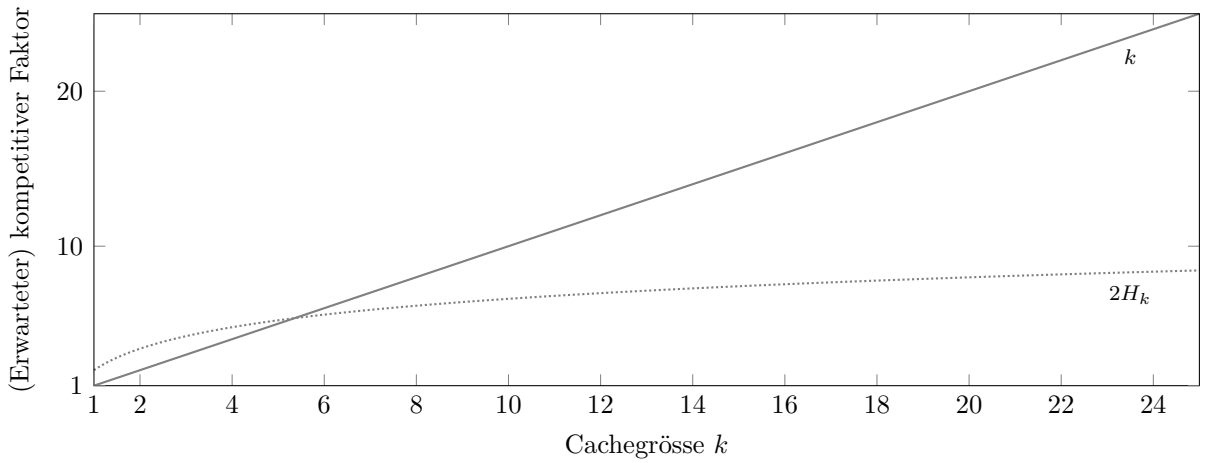


Abbildung 4.

- kein deterministischer Online-Algorithmus besser als k-kompetitiv ist, aber
- ein randomisierter Online-Algorithmus existiert, der im Erwartungswert $\mathcal{O}(\log k)$-kompetitiv ist.
- Somit erlaubt uns Randomisierung eine asymptotische exponentielle Verbesserung der Ausgabequalität im Erwartungswert.

Asymptotisch ist dies wiederum das Beste, worauf wir bei randomisierten Online-Algorithmen hoffen können, wie wir im Folgenden sehen werden. Bevor wir hier aber eine untere Schranke für den kompetitiven Faktor von randomisierten Online-Algorithmen für Paging beweisen, lernen wir eine allgemeine Technik kennen, mit der wir oftmals relativ einfach solche unteren Schranken zeigen können.

1.5 Yaos Prinzip

Jetzt stellen wir einen Bezug zwischen randomisierten und deterministischen Online-Algorithmen her. Der folgende Satz erlaubt uns, eine untere Schranke für den erwarteten kompetitiven Faktor von randomisierten Online-Algorithmen auf untere Schranken für den kompetitiven Faktor von deterministischen Online-Algorithmen zurückzuführen.

Die Erkenntnis, dass alle deterministischen Strategien für ein gegebenes Problem schlecht sind, reicht selbstverständlich nicht. Dies zeigen wir für verschiedene Algorithmen anhand verschiedener Worst-Case-Eingaben. Die jeweiligen deterministischen Strategien könnten aber gut für viele andere sein, sodass dennoch ein guter randomisierter Online-Algorithmus existiert, der zwischen ihnen wählt (das ist ja gerade der Punkt bei randomisierten Verfahren). Wir müssen hier etwas Stärkeres zeigen und zwar, dass für eine feste Wahrscheinlichkeitsverteilung über irgendwelche Instanzen alle deterministischen Strategien schlechte Resultate liefern. Dies ist im folgenden Satz formal ausgedrückt. Intuitiv zeigen wir jetzt, dass wir eine Aussage über eine Wahrscheinlichkeitsverteilung über die Eingaben übertragen können auf eine Aussage über eine Wahrscheinlichkeitsverteilung über die deterministischen Strategien eines randomisierten Algorithmus.

Wir gehen im Folgenden davon aus, dass sowohl die Anzahl der Instanzen des gegebenen Online-Problems, als auch die Anzahl der deterministischen Online-Algorithmen, aus denen ein gegebener randomisierter Online-Algorithmus wählt, endlich sind. Dies bedeutet, dass strat(RAND) nun die Menge aller deterministischen Strategien von RAND bezeichnet und nicht von der Eingabelänge abhängt. Beachten Sie, dass es für diesen Fall sinnvoll ist, die additive Konstante α aus Definition 1.3 auf Null zu setzen. Da jede der endlich vielen Instanzen nämlich eine endliche Länge besitzt und pro Zeitschritt keine unendlich grossen Kosten verursacht werden können, ist jeder deterministische Online-Algorithmus 1-kompetitiv.

Lemma 1.13. *Sei* Π *ein Online-Minimierungsproblem und* $\mathcal{I} = \{I_1, \ldots, I_m\}$ *eine Klasse von Instanzen für* Π. *Ferner sei* Prob_{ADV} *eine Wahrscheinlichkeitsverteilung auf* $\mathcal{I}$, *sodass für jeden deterministischen Online-Algorithmus* ALG *gilt, dass seine erwarteten Kosten (bezüglich des durch* Prob_{ADV} *definierten Wahrscheinlichkeitsraums der Eingaben) mindestens c-mal so schlecht sind wie die erwarteten Kosten eines optimalen Algorithmus* OPT *für ein* $c \in \mathbb{Q}^+$. *Dann existiert für jeden randomisierten Online-Algorithmus* RAND *eine Instanz* $I \in \mathcal{I}$, *sodass die erwarteten Kosten (bezüglich der Wahrscheinlichkeitsverteilung von* RAND *über seine Strategien) auf* I *nicht besser als c-mal die Kosten von* OPT *sind.*

Beweis. Sei RAND ein randomisierter Online-Algorithmus für Π, der eine Wahrscheinlichkeitsverteilung $\text{Prob}_{\text{RAND}}$ realisiert, mit der zufällig einer der deterministischen Algorithmen $A_1, \ldots, A_\ell$ aus strat(RAND) gewählt wird.

- Nun betrachten wir eine beliebige Wahrscheinlichkeitsverteilung Prob_{ADV} auf $\mathcal{I}$, die jeder Instanz $I \in \mathcal{I}$ die Wahrscheinlichkeit $\text{Prob}_{\text{ADV}}[I]$ zuordnet.
- Wir können dann die erwarteten Kosten des deterministischen Algorithmus A_j bezüglich Prob_{ADV} berechnen als

$$\mathbb{E}_{\text{ADV}}[\text{cost}(A_j(\mathcal{I}))] = \sum_{i=1}^{m} \text{Prob}_{\text{ADV}}[I_i] \cdot \text{cost}(A_j(I_i)) \ .$$

- Die erwarteten Kosten von RAND bezüglich Prob_{ADV} sind

$$\begin{aligned}
\mathbb{E}_{\text{ADV}}[\mathbb{E}_{\text{RAND}}[\text{cost}(\text{RAND}(\mathcal{I}))]] &= \sum_{i=1}^{m} \text{Prob}_{\text{ADV}}[I_i] \cdot \mathbb{E}_{\text{RAND}}[\text{cost}(\text{RAND}(I_i))] \\
&= \sum_{i=1}^{m} \text{Prob}_{\text{ADV}}[I_i] \cdot \left(\sum_{j=1}^{\ell} \text{Prob}_{\text{RAND}}[A_j] \cdot \text{cost}(A_j(I_i)) \right) \\
&= \sum_{i=1}^{m} \left(\sum_{j=1}^{\ell} \text{Prob}_{\text{ADV}}[I_i] \cdot \text{Prob}_{\text{RAND}}[A_j] \cdot \text{cost}(A_j(I_i)) \right) \\
&= \sum_{j=1}^{\ell} \text{Prob}_{\text{RAND}}[A_j] \cdot \left(\sum_{i=1}^{m} \text{Prob}_{\text{ADV}}[I_i] \cdot \text{cost}(A_j(I_i)) \right) \\
&= \sum_{j=1}^{\ell} \text{Prob}_{\text{RAND}}[A_j] \cdot \mathbb{E}_{\text{ADV}}[\text{cost}(A_j(\mathcal{I}))] \ .
\end{aligned}$$

Genau genommen betrachten wir hier einen neuen Wahrscheinlichkeitsraum, der sich aus dem Produkt der gegebenen Wahrscheinlichkeitsräume über den Strategien von RAND und den Eingaben ergibt.

- Nehmen wir nun an, jeder deterministische Online-Algorithmus hat im Erwartungswert Kosten, die c-mal grösser als die erwarteten Kosten einer optimalen Lösung sind. Dies gilt dann natürlich insbesondere auch für jeden Online-Algorithmus $A_j \in \text{strat}(\text{RAND})$.
- Somit erhalten wir schliesslich

$$\begin{aligned}\mathbb{E}_{\text{ADV}}[\mathbb{E}_{\text{RAND}}[\text{cost}(\text{RAND}(\mathcal{I}))]] &= \sum_{j=1}^{\ell} \text{Prob}_{\text{RAND}}[A_j] \cdot \mathbb{E}_{\text{ADV}}[\text{cost}(A_j)] \\ &\geq \sum_{j=1}^{\ell} \text{Prob}_{\text{RAND}}[A_j] \cdot c \cdot \mathbb{E}_{\text{ADV}}[\text{cost}(\text{OPT})] \\ &= c \cdot \mathbb{E}_{\text{ADV}}[\text{cost}(\text{OPT}(\mathcal{I}))] \; .\end{aligned}$$

- Wir sind an dieser Stelle jedoch noch nicht ganz zufrieden, denn wir haben es hier mit einem randomisierten Gegenspieler zu tun, der seine Strategie zufällig wählt und das entspricht eigentlich nicht unserem Modell.
- Wir können unseren Gegenspieler aber sehr einfach «derandomisieren», indem wir uns klarmachen, dass aus obiger Ungleichung unmittelbar folgt, dass es eine Instanz $I \in \mathcal{I}$ geben muss, sodass

  $$\mathbb{E}_{\text{RAND}}[\text{cost}(\text{RAND}(I))] \geq c \cdot \text{cost}(\text{OPT}(I))$$

  gilt, da wir sonst einen Widerspruch zur bewiesenen Ungleichung der Erwartungswerte erhalten würden. Würde konkret für jede Instanz $I_i \in \mathcal{I}$, $1 \leq i \leq m$, gelten, dass $\mathbb{E}_{\text{RAND}}[\text{cost}(\text{RAND}(I_i))] < c \cdot \text{cost}(\text{OPT}(I_i))$ ist, so ist auch $\text{Prob}_{\text{ADV}}[I_i] \cdot \mathbb{E}_{\text{RAND}}[\text{cost}(\text{RAND}(I_i))] < c \cdot \text{Prob}_{\text{ADV}}[I_i] \cdot \text{cost}(\text{OPT}(I_i))$ für jedes i. Damit würde aber auch

  $$\sum_{i=1}^{m} \text{Prob}_{\text{ADV}}[I_i] \cdot \mathbb{E}_{\text{RAND}}[\text{cost}(\text{RAND}(I_i))] < c \sum_{i=1}^{m} \text{Prob}_{\text{ADV}}[I_i] \cdot \text{cost}(\text{OPT}(I_i))$$

  und somit

  $$\mathbb{E}_{\text{ADV}}[\mathbb{E}_{\text{RAND}}[\text{cost}(\text{RAND}(\mathcal{I}))]] < c \cdot \mathbb{E}_{\text{ADV}}[\text{cost}(\text{OPT}(\mathcal{I}))]$$

  gelten.

Es folgt sofort, dass eine Instanz I existiert, sodass der erwartete kompetitive Faktor von RAND durch

$$c \leq \frac{\mathbb{E}_{\text{RAND}}[\text{cost}(\text{RAND}(I))]}{\text{cost}(\text{OPT}(I))}$$

beschränkt werden kann. □

Wir haben in Lemma 1.13 über das Verhältnis der erwarteten Kosten von Online-Algorithmen und der erwarteten Kosten einer optimalen Lösung argumentiert. Dies erlaubt uns aber noch nicht, auch über den erwarteten strikten kompetitiven Faktor zu argumentieren. Sei ALG ein beliebiger deterministischer Online-Algorithmus und eine Wahrscheinlichkeitsverteilung über Eingaben gegeben; dann gilt im Allgemeinen

$$\frac{\mathbb{E}_{\text{ADV}}[\text{cost}(\text{ALG}(\mathcal{I}))]}{\mathbb{E}_{\text{ADV}}[\text{cost}(\text{OPT}(\mathcal{I}))]} \neq \mathbb{E}_{\text{ADV}}\left[\frac{\text{cost}(\text{ALG}(\mathcal{I}))}{\text{cost}(\text{OPT}(\mathcal{I}))}\right] .$$

Das folgende Lemma kann allerdings analog zu Lemma 1.13 bewiesen werden, und somit können wir auch über den Erwartungswert des strikten kompetitiven Faktors argumentieren.

Lemma 1.14. *Seien* Π, $\mathcal{I}$ *und* Prob_{ADV} *wie oben. Ferner sei der erwartete kompetitive Faktor jedes deterministischen Online-Algorithmus* ALG *mindestens* c *für ein* $c \in \mathbb{Q}^+$. *Dann ist der kompetitive Faktor jedes randomisierten Online-Algorithmus ebenfalls nicht besser als* c.

Beweis. Sei also RAND wieder ein randomisierter Online-Algorithmus, der eine Wahrscheinlichkeitsverteilung $\text{Prob}_{\text{RAND}}$ über $A_1, \ldots, A_\ell$ realisiert.

- Nach Voraussetzung gilt

  $$\mathbb{E}_{\text{ADV}}\left[\frac{\text{cost}(A_j(\mathcal{I}))}{\text{cost}(\text{OPT}(\mathcal{I}))}\right] \geq c .$$

  für jeden deterministischen Online-Algorithmus A_j.

- Für $\mathbb{E}_{\text{ADV}}$ folgt ferner

  $$\mathbb{E}_{\text{ADV}}\left[\frac{\text{cost}(A_j(\mathcal{I}))}{\text{cost}(\text{OPT}(\mathcal{I}))}\right] = \sum_{i=1}^{m} \text{Prob}_{\text{ADV}}[I_i] \cdot \frac{\text{cost}(A_j(I_i))}{\text{cost}(\text{OPT}(I_i))} .$$

- Für RAND erhalten wir somit

  $$\begin{aligned}
  &\mathbb{E}_{\text{ADV}}\left[\frac{\mathbb{E}_{\text{RAND}}[\text{cost}(\text{RAND}(\mathcal{I}))]}{\text{cost}(\text{OPT}(\mathcal{I}))}\right] \\
  &= \mathbb{E}_{\text{ADV}}\left[\frac{\sum_{j=1}^{\ell} \text{Prob}_{\text{RAND}}[A_j] \cdot \text{cost}(A_j(\mathcal{I}))}{\text{cost}(\text{OPT}(\mathcal{I}))}\right] \\
  &= \sum_{i=1}^{m}\left(\text{Prob}_{\text{ADV}}[I_i] \cdot \frac{\sum_{j=1}^{\ell} \text{Prob}_{\text{RAND}}[A_j] \cdot \text{cost}(A_j(I_i))}{\text{cost}(\text{OPT}(I_i))}\right) \\
  &= \sum_{i=1}^{m}\left(\text{Prob}_{\text{ADV}}[I_i] \sum_{j=1}^{\ell} \text{Prob}_{\text{RAND}}[A_j] \cdot \frac{\text{cost}(A_j(I_i))}{\text{cost}(\text{OPT}(I_i))}\right) \\
  &= \sum_{i=1}^{m}\left(\sum_{j=1}^{\ell} \text{Prob}_{\text{ADV}}[I_i] \cdot \text{Prob}_{\text{RAND}}[A_j] \cdot \frac{\text{cost}(A_j(I_i))}{\text{cost}(\text{OPT}(I_i))}\right) \\
  &= \sum_{j=1}^{\ell}\left(\text{Prob}_{\text{RAND}}[A_j] \sum_{i=1}^{m} \text{Prob}_{\text{ADV}}[I_i] \cdot \frac{\text{cost}(A_j(I_i))}{\text{cost}(\text{OPT}(I_i))}\right) .
  \end{aligned}$$

- In der letzten Zeile können wir nun die Definition des Erwartungswertes $\mathbb{E}_{\text{ADV}}$ einsetzen und erhalten

$$\mathbb{E}_{\text{ADV}}\left[\frac{\mathbb{E}_{\text{RAND}}[\text{cost}(\text{RAND}(\mathcal{I}))]}{\text{cost}(\text{OPT}(\mathcal{I}))}\right] = \sum_{j=1}^{\ell} \text{Prob}_{\text{RAND}}[A_j] \cdot \mathbb{E}_{\text{ADV}}\left[\frac{\text{cost}(A_j(\mathcal{I}))}{\text{cost}(\text{OPT}(\mathcal{I}))}\right] .$$

- Mit der unteren Schranke von c folgt jetzt

$$\mathbb{E}_{\text{ADV}}\left[\frac{\mathbb{E}_{\text{RAND}}[\text{cost}(\text{RAND}(\mathcal{I}))]}{\text{cost}(\text{OPT}(\mathcal{I}))}\right] \geq \sum_{j=1}^{\ell} \text{Prob}_{\text{RAND}}[A_j] \cdot c = c .$$

Wie auch im Beweis von Lemma 1.13) können wir wegen der Definition von $\mathbb{E}_{\text{ADV}}$ argumentieren, dass es eine Instanz $I \in \mathcal{I}$ geben muss, sodass

$$\frac{\mathbb{E}_{\text{RAND}}[\text{cost}(\text{RAND}(I))]}{\text{cost}(\text{OPT}(I))} \geq c$$

gilt. □

Aus den Lemmata 1.13 und 1.14 folgt nun Yaos Prinzip für Online-Minimierungsprobleme für eine konstante Anzahl von Instanzen und Algorithmen.

Satz 1.15 (Yaos Prinzip). *Seien wieder* Π, $\mathcal{I}$ *und* Prob_{ADV} *wie oben. Für jeden randomisierten Online-Algorithmus existiert dann eine Eingabe* I, *sodass*

$$\frac{\mathbb{E}_{RAND}[\text{cost}(RAND(I))]}{\text{cost}(OPT(I))} \geq \max\left\{\min_j\left\{\frac{\mathbb{E}_{ADV}[\text{cost}(A_j(\mathcal{I}))]}{\mathbb{E}_{ADV}[\text{cost}(OPT(\mathcal{I}))]}\right\}, \min_j\left\{\mathbb{E}_{ADV}\left[\frac{\text{cost}(A_j(\mathcal{I}))}{\text{cost}(OPT(\mathcal{I}))}\right]\right\}\right\} .$$

Mit ähnlichen Überlegungen und Argumenten können wir eine analoge Aussage für Online-Maximierungsprobleme machen. Ferner können ähnliche Aussagen bewiesen werden, wenn sowohl $\mathcal{I}$ als auch strat(RAND) unbeschränkt gross sind; hierbei darf die Konstante α aus Definition 1.3 beliebig sein.

Im übernächsten Kapitel werden wir die soeben vorgestellte Technik benutzen, um eine untere Schranke für randomisierte Online-Algorithmen für Paging aufzustellen, die über untere Schranken für deterministische Algorithmen argumentiert. Zunächst betrachten wir, um unsere Intuition zu festigen, eine andere Sichtweise auf Online-Probleme und das Spiel zwischen einem Algorithmus und einem Gegenspieler.

1.6 Eine alternative Sichtweise: Spieltheorie

Wir betrachten einen randomisierten Online-Algorithmus RAND und einen Gegenspieler, der RAND so sehr wie möglich schaden möchte. Nehmen wir wieder an, RAND trifft $b \in \mathbb{N}^+$ binäre Zufallsentscheidungen und zieht somit nach Beobachtung 1.9 zufällig einen von ℓ deterministischen Algorithmen $A_1, \ldots, A_\ell$ aus strat(RAND), wobei $\ell \leq 2^b$ ist. Auf der

anderen Seite kann ADV eine beliebige Eingabe $I_1, \ldots, I_m$ aus einer Klasse $\mathcal{I}$ wählen, die RAND dann bearbeiten muss; dabei nehmen wir immer an, dass ℓ und m beliebig grosse, aber endliche Zahlen sind und setzen aus den eben erwähnten Gründen $\alpha = 0$.

Wir bezeichnen $A_1, \ldots, A_\ell$ als **Strategien** von RAND und $I_1, \ldots, I_m$ als die Strategien von ADV. Weiterhin definieren wir $c_{i,j}$, mit $1 \leq i \leq m, 1 \leq j \leq \ell$, als den kompetitiven Faktor des deterministischen Algorithmus A_j auf der Eingabe I_i, also

$$c_{i,j} := \frac{\text{cost}(A_j(I_i))}{\text{cost}(\text{OPT}(I_i))} .$$

Wir können mit diesen Werten nun die folgende Matrix T konstruieren.

	A_1	A_2	A_3	$\ldots$
I_1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$\ldots$
I_2	$c_{2,1}$	$c_{2,2}$	$c_{2,3}$	
I_3	$c_{3,1}$	$c_{3,2}$	$c_{3,3}$	
$\vdots$	$\vdots$			$\ddots$

Bleiben wir zunächst einmal im deterministischen Fall. Jetzt können wir uns also vorstellen, dass RAND eine Spalte j auswählt, sodass, wenn eine Zeilenauswahl von ADV gegeben ist, der Wert $c_{i,j}$ möglichst klein ist; analog könnten wir sagen, dass RAND garantieren möchte, dass $-c_{i,j}$ möglichst gross sein soll. Auf der anderen Seite möchte ADV den Wert $c_{i,j}$ maximieren. Da für eine konkrete Auswahl einer Zeile und einer Spalte RAND einen Gewinn hat, der genau dem negierten Gewinn von ADV entspricht, reden wir in diesem Zusammenhang von einem **Nullsummenspiel** für zwei Personen. Für unsere weiteren Untersuchungen ist es von Vorteil, davon zu sprechen, dass RAND den Gewinn von ADV minimieren möchte (was gleichbedeutend damit ist, den eigenen Gewinn zu maximieren, aber eine intuitivere Betrachtungsweise darstellt).

Wir bezeichnen ADV als den Zeilenspieler und RAND als den Spaltenspieler. Wir dürfen hier annehmen, dass beide **Spieler** die möglichen Strategien ihres Gegners, und somit T, kennen. Ein einfaches Beispiel ist durch die folgende Matrix T_1 gegeben.

	A_1	A_2
I_1	1	2
I_2	2	1

Beide Spieler gehen nun mögliche Verläufe eines Spiels mit der gegebenen Matrix im Kopf durch, um danach zu bestimmen, welche Strategie für sie die beste wäre, denn sie sollen nachher gleichzeitig ziehen. Allerdings sehen sie sehr schnell ein, dass einer von ihnen die Strategie immer wechseln wollen würde. Hätten sich beide Spieler entschieden, ihre ersten Strategien, also A_1 und I_1 zu wählen, würde ADV einen Gewinn von $c_{1,1} = 1$ erhalten und RAND einen von -1. Da ADV die Matrix T_1 jedoch kennt, wird er in diesem Fall seine Strategie wechseln wollen und I_2 spielen. Dies führt jedoch zu einem Gewinn von RAND von $-c_{1,2} = -2$, weswegen nun dieser seine Strategie auf A_2 wechselt. Anschliessend hat ADV wiederum einen **Anreiz**, seine Strategie zu ändern und wir sehen schnell, dass wir

uns im Kreis drehen; egal, was gespielt wird, bleibt die Strategie des anderen unverändert, so hat einer der Spieler immer einen Vorteil, wenn er seine eigene ändern würde.

	A_1	A_2
I_1		
I_2		

Betrachten wir nun die Matrix T_2, so sehen wir ein anderes Phänomen.

	A_1	A_2	A_3	A_4
I_1	7	2	3	1
I_2	1	4	5	4
I_3	10	9	6	9
I_4	6	3	2	11
I_5	12	1	2	5

Hier gibt es ein Paar von Strategien, das dazu führt, dass keiner der beiden Spieler einen Anreiz hat, seine eigene Strategie zu ändern, wenn der andere seine Strategie beibehält. Konkret ist dies der Fall, wenn RAND die Strategie A_3 und ADV die Strategie I_3 spielt.

	A_1	A_2	A_3	A_4
I_1				
I_2				
I_3				
I_4				
I_5				

In diesem Fall sagen wir, dass das Spiel in einem **Gleichgewicht** ist und wir nennen $c_{3,3}$ den **Wert des Spiels**.

- Wählt RAND die Strategie A_3, so kann er sicher sein, dass der Gewinn für ADV nicht grösser als 6 ist. Würde RAND beispielsweise A_1 wählen, so könnte der Gewinn von ADV wesentlich grösser sein, nämlich 12.
- Wählt ADV auf der anderen Seite die Strategie I_3, so kann er davon ausgehen, dass sein Gewinn mindestens 6 ist. Würde er beispielsweise I_5 wählen, könnte sein Gewinn mit 1 viel niedriger ausfallen.

Wir können also in einem solchen Fall die Strategien der beiden Spieler wie folgt beschreiben.

- RAND wählt seine Strategie A_{j^*} derart, dass der maximale Wert über alle Einträge in dieser Spalte minimal ist, also

  $$j^* = \arg\min_j \{\max_i \{c_{i,j}\}\}$$

  und wir setzen

  $$v_{\text{RAND}} := \max_i \{c_{i,j^*}\} \ .$$

- ADV wählt seine Strategie I_{i^*} hingegen so, dass der minimale Wert über alle Spalten in dieser Zeile maximal ist, also

  $$i^* = \arg\max_i \{\min_j \{c_{i,j}\}\}$$

  und wir setzen analog

  $$v_{\text{ADV}} := \min_j \{c_{i^*,j}\} \ .$$

- Wie wir gerade gesehen haben, ist

  $$v_{\text{ADV}} = v_{\text{RAND}} \ ,$$

  wenn das Spiel ein Gleichgewicht hat. Wenn i^* und j^* gespielt werden, so kann keiner der beiden Spieler durch einen alleinigen Wechsel seiner Strategie dafür sorgen, dass sein Gewinn grösser wird.

Wir wissen bereits, dass der dritte Punkt im Allgemeinen nicht erfüllt ist, auch wenn die Werte v_{RAND} und v_{ADV} immer existieren; wir können lediglich davon ausgehen, dass $v_{\text{ADV}} \leq v_{\text{RAND}}$ ist. Als letztes Beispiel betrachten wir die Matrix T_3.

	A_1	A_2	A_3	A_4
I_1	6	5	8	4
I_2	7	6	2	7
I_3	1	3	3	2

Nehmen wir also an, beide Spieler verfahren nach obigem Prinzip.

- RAND kann immer garantieren, dass der Gewinn von ADV nicht grösser als 6 ist, indem er A_2 spielt. Für ADV wäre es in diesem Fall offensichtlich am besten, I_2 zu spielen. Allerdings würde RAND seine Strategie dann ändern und A_3 wählen.

- Auf der anderen Seite kann ADV durch die Wahl der Strategie I_1 garantieren, dass sein Gewinn mindestens 4 ist. RAND würde A_4 wählen und ADV hätte wiederum einen Anreiz, seine Strategie zu ändern.

- Insbesondere ist die Wahl I_1 und A_2 kein Gleichgewicht in diesem Spiel.

Das präsentierte Vorgehen scheint nicht sehr aussichtsreich, um deterministische Algorithmen zu analysieren, aber wir wollen ja auch eigentlich über randomisierte Online-Algorithmen sprechen und wir nehmen im Folgenden zunächst einmal an, dass ADV ebenfalls randomisiert agiert. Wenn wir diese Betrachtungsweise auf unser Spiel übertragen, so haben wir es mit folgender Situation zu tun.

- RAND arbeitet mit einer Wahrscheinlichkeitsverteilung $\text{Prob}_{\text{RAND}}\colon \text{strat}(\text{RAND}) \to [0,1]$ über den Spalten von T. Wir bezeichnen die Wahrscheinlichkeit $\text{Prob}_{\text{RAND}}[A_j]$, dass RAND die Strategie A_j wählt, mit q_j; $q = (q_1, \ldots, q_\ell)$ heisst die **gemischte Strategie** von RAND.

- Analog arbeitet ADV mit einer Wahrscheinlichkeitsverteilung $\text{Prob}_{\text{ADV}} \colon \mathcal{I} \to [0,1]$ über den Zeilen von T. Die Wahrscheinlichkeit $\text{Prob}_{\text{ADV}}[I_i]$, dass ADV die Strategie I_i wählt, bezeichnen wir mit p_i; $p = (p_1, \ldots, p_m)$ heisst die gemischte Strategie von ADV.

Die Auszahlung modellieren wir nun als eine Zufallsvariable $G \colon \mathcal{I} \times \text{strat}(\text{RAND}) \to \mathbb{R}^+$, deren Erwartungswert sich zu

$$\mathbb{E}[G] := \sum_{i=1}^{m} \sum_{j=1}^{\ell} p_i \cdot c_{i,j} \cdot q_j = p^\intercal T q$$

berechnet. Offensichtlich möchte RAND diesen Erwartungswert durch die Wahl seiner Wahrscheinlichkeitsverteilung minimieren und ADV möchte ihn maximieren. Wie sehen aber nun konkret geschickte Wahlen für Wahrscheinlichkeiten der beiden Spieler aus? Wir argumentieren wieder so wie wir es oben für deterministische Strategien getan haben.

- RAND kennt T und weiss, dass ADV ein Interesse daran hat, seine gemischte Strategie p so zu wählen, dass $\mathbb{E}[G]$ für die gegebene Wahl q maximiert wird. Somit kann RAND eine Strategie q^* wählen, sodass

  $$q^* = \arg\min_q \{\max_p \{p^\intercal T q\}\}$$

  ist. Also hat ADV höchstens einen Gewinn von

  $$v_{\text{RAND}} := \max_p \{p^\intercal T q^*\} \; .$$

- Auf der anderen Seite kann ADV eine gemischte Strategie p^* wählen, sodass

  $$p^* = \arg\max_p \{\min_q \{p^\intercal T q\}\}$$

  ist, kann also einen Gewinn von mindestens

  $$v_{\text{ADV}} := \min_q \{p^{*\intercal} T q\}$$

  garantieren.

Analog zu unseren Überlegungen zu deterministischen Strategien ist klar, dass $v_{\text{ADV}} \leq v_{\text{RAND}}$ gilt, tatsächlich gilt diese Beziehung aber mit Gleichheit. Der folgende Satz ist eines der wichtigsten Resultate aus der Spieltheorie, denn er besagt, dass $v_{\text{RAND}} = v_{\text{ADV}}$ gilt. Wir werden ihn hier nicht beweisen.

Satz 1.16 (Minimax-Theorem). *Für jedes Nullsummenspiel für zwei Personen mit endlichen Strategien gilt*

$$\min_q \{\max_p \{p^\intercal T q\}\} = \max_p \{\min_q \{p^\intercal T q\}\} \; .$$

□

Das Minimax-Theorem hat für uns eine wichtige Konsequenz. Es sagt aus, dass es für Nullsummenspiele für zwei Personen ein Gleichgewicht in gemischten Strategien p^* und q^* gibt, die auf die obige Weise berechnet werden können. Was bedeutet dies nun für randomisierte Online-Algorithmen?

Wir stellen fest, dass, wenn p oder q fest sind, der jeweils andere Spieler seinen Gewinn maximieren kann, indem er lediglich eine reine Strategie wählt. Nehmen wir beispielsweise an, die Strategie q sei gegeben; dies entspricht natürlich genau der Situation, die wir bei randomisierten Online-Algorithmen vorfinden, denn ADV kennt die Wahrscheinlichkeitsverteilung, mit der RAND seine deterministischen Strategien wählt. Dann gilt

$$\begin{pmatrix} c_{1,1} & \dots & c_{1,\ell} \\ \vdots & \ddots & \vdots \\ c_{m,1} & \dots & c_{m,\ell} \end{pmatrix} \cdot \begin{pmatrix} q_1 \\ \vdots \\ q_\ell \end{pmatrix} = \begin{pmatrix} c'_1 \\ \vdots \\ c'_m \end{pmatrix},$$

sodass ADV seine Strategie nun als Einheitsvektor wählen kann, bei dem genau der Eintrag nicht 0 ist, für welchen der entsprechende Eintrag in $(c'_1 \ \dots \ c'_m)^\intercal$ am grössten ist, um seinen Gewinn

$$\begin{pmatrix} p_1 & \dots & p_m \end{pmatrix} \cdot \begin{pmatrix} c'_1 \\ \vdots \\ c'_m \end{pmatrix}$$

zu maximieren. Analog können wir umgekehrt argumentieren. Wenn wir diese Erkenntnis mit Satz 1.16 kombinieren, erhalten wir das folgende Lemma, wobei e_i, für $1 \leq i \leq m$ beziehungsweise $1 \leq i \leq \ell$, den entsprechenden Einheitsvektor bezeichnet, bei dem der i-te Eintrag 1 ist und alle anderen 0.

Lemma 1.17 (Das Lemma von Loomis). *Für jedes Nullsummenspiel für zwei Personen mit endlichen Strategien gilt*

$$\min_q\{\max_i\{e_i^\intercal T q\}\} = \max_p\{\min_j\{p^\intercal T e_j\}\}.$$

□

Da für jede feste gemischte Strategie p' für ADV offensichtlich

$$\max_p\{\min_j\{p^\intercal T e_j\}\} \geq \min_j\{p'^\intercal T e_j\}$$

gilt, können wir zusammenfassend somit unter Verwendung des Minimax-Theorems beziehungsweise des Lemmas von Loomis schlussfolgern, dass für jedes p'

$$\min_q\{\max_i\{e_i^\intercal T q\}\} \geq \min_j\{p'^\intercal T e_j\}$$

ist. Sei wieder q^* eine optimale gemischte Strategie für RAND. Dann ist

$$\max_i\{e_i^\intercal T q^*\} \geq \min_j\{p'^\intercal T e_j\}.$$

Bei genauerer Betrachtung stellen wir fest, dass diese Ungleichung wiederum genau Yaos Prinzip entspricht, das wir im letzten Kapitel vorgestellt haben. Die Aussage ist, dass eine Eingabe (nämlich $e_i^\intercal$) existiert, sodass für einen besten randomisierten Online-Algorithmus (nämlich q^*) die Kosten mindestens so gross sind wie die eines besten deterministischen Online-Algorithmus (nämlich e_j) auf einer beliebigen festen Verteilung auf den Eingaben (nämlich p').

1.7 Eine untere Schranke für randomisierte Paging-Algorithmen

Wir haben in Satz 1.12 gezeigt, dass RMark einen erwarteten kompetitiven Faktor von $2H_k$ erreicht. Jetzt zeigen wir, dass dies nur maximal eine multiplikative Konstante von 2 von dem entfernt ist, was ein randomisierter Online-Algorithmus für Paging überhaupt schaffen kann.

Satz 1.18. *Kein randomisierter Online-Algorithmus für Paging kann einen besseren erwarteten kompetitiven Faktor als H_k erreichen.*

Beweis. Es reicht hier (genau wie beim Beweis von Satz 1.6), wenn wir davon ausgehen, dass es insgesamt nur $k+1$ Seiten gibt. Wir betrachten folgende Wahrscheinlichkeitsverteilung über alle Eingaben und zeigen, dass jeder deterministische Online-Algorithmus hohe erwartete Kosten hat. Mit Yaos Prinzip können wir dann auf die Kosten eines besten randomisierten Online-Algorithmus schliessen.

- Im ersten Zeitschritt wird die Seite s_{k+1} angefragt, die nach Definition nicht im Speicher ist.
- In allen weiteren Zeitschritten wird mit jeweils einer Wahrscheinlichkeit von $1/k$ eine beliebige Seite ausser der unmittelbar zuvor angefragten Seite angefragt.

Wir unterteilen die Eingabe wieder in Phasen. Phase 1 beginnt mit dem ersten Zeitschritt und endet im Zeitschritt j, wobei in Zeitschritt $j+1$ das erste Mal alle $k+1$ verschiedenen Seiten angefragt wurden. Danach beginnt Phase 2, die analog definiert ist, etc. Es ist hier zu beachten, dass die Länge einer Phase nicht fix ist.

Wir betrachten wieder eine einzelne Phase und zeigen, dass die erwarteten Kosten jedes deterministischen Online-Algorithmus (ungefähr) H_k-mal höher sind als die einer optimalen Lösung.

- Jeder deterministische Online-Algorithmus hat zu jedem Zeitpunkt genau k Seiten im Cache. Das bedeutet, dass für jede angefragte Seite diese gerade mit einer Wahrscheinlichkeit von $1/k$ nicht im Cache ist (eine Ausnahme bildet natürlich der erste Zeitschritt). Er macht also in jedem Zeitschritt einen Seitenfehler mit dieser Wahrscheinlichkeit.
- Auf der anderen Seite existiert für jede Phase ein optimaler Algorithmus Opt, der nicht mehr als einen Fehler macht. In Phase 1 macht er einen mit der ersten Anfrage; er verdrängt dann die Seite, die als erstes zu Beginn von Phase 2 angefragt wird etc.

- Wenn wir mit $|P|$ die erwartete Länge von P bezeichnen, dann gilt für die erwarteten Kosten jedes deterministischen Algorithmus ALG während P

$$\frac{\mathbb{E}_{\text{ADV}}[\text{cost}(\text{ALG}(P))]}{\mathbb{E}_{\text{ADV}}[\text{cost}(\text{OPT}(P))]} \geq \frac{|P| \cdot \frac{1}{k}}{1} = \frac{|P|}{k} \ .$$

Die wesentliche Grösse, die es nun abzuschätzen gilt, ist die erwartete Länge einer Phase P. Diese lässt uns direkt über die erwarteten Kosten jedes deterministischen Algorithmus sprechen.

- Die Phase P ist genau dann beendet, wenn die nächste Anfrage das erste Auftauchen der $(k+1)$-ten Seite seit Beginn von P ist.

- Wir müssen also abschätzen, wie lange es im Erwartungswert dauert, bis alle Seiten $s_1, \ldots, s_{k+1}$ angefragt wurden (dieses Problem ist allgemein als Sammelbilderproblem, englisch «coupon collector's problem», bekannt). Die erwartete Anzahl von Zeitschritten ist diese Zahl minus 1.

- Sei zu diesem Zweck t_i, $1 \leq i \leq k+1$ eine Zufallsvariable, die die Anzahl der Schritte zählt, die vergehen, bis die i-te Seite angefragt wird, nachdem bereits $i-1$ Seiten angefragt wurden; diese Wahrscheinlichkeit ist dann in jedem Zeitschritt gleich.

- Betrachten wir für P zunächst die Wahrscheinlichkeit p_i, die i-te neue Seite anzufragen, wenn in früheren Zeitschritten schon $i-1$ Seiten angefragt wurden.

- Es ist klar, dass $p_1 = 1$ gilt und, weil wir im zweiten Zeitschritt von P die erste Seite nicht anfragen, ist ebenfalls $p_2 = 1$.

- Allgemein gilt für $2 \leq i \leq k+1$, dass

$$p_i = \frac{k-(i-2)}{k} \ .$$

- Nun können wir den Erwartungswert von t_i wie folgt berechnen. Nehmen wir an, wir hätten soeben die $(i-1)$-te Seite angefragt und zählen ab jetzt die Zeitschritte, die wir brauchen, bis wir die i-te Seite anfragen. Die Wahrscheinlichkeit, dass dies j Schritte braucht, ist $p_i \cdot (1-p_i)^{j-1}$, d. h., wir haben $j-1$ Schritte eine schon angefragte Seite gezogen und beim j-ten Mal eine noch nicht gefragte. Somit folgt

$$\begin{aligned}
\mathbb{E}_{\text{ADV}}[t_i] &= \sum_{j=1}^{\infty} j \cdot p_i \cdot (1-p_i)^{j-1} \\
&= \sum_{j=0}^{\infty} (j+1) \cdot p_i \cdot (1-p_i)^{j} \\
&= \sum_{j=0}^{\infty} j \cdot p_i \cdot (1-p_i)^{j} + \sum_{j=0}^{\infty} p_i \cdot (1-p_i)^{j} \\
&= (1-p_i) \cdot \sum_{j=1}^{\infty} j \cdot p_i \cdot (1-p_i)^{j-1} + p_i \cdot \sum_{j=0}^{\infty} (1-p_i)^{j}
\end{aligned}$$

$$= (1 - p_i) \cdot \mathbb{E}_{\text{ADV}}[t_i] + \frac{p_i}{1 - (1 - p_i)}$$

und schliesslich

$$\mathbb{E}_{\text{ADV}}[t_i] = \frac{1}{p_i} .$$

- Sei nun $t = t_1 + \cdots + t_{k+1}$ eine Zufallsvariable, die alle Schritte zählt, bis $k + 1$ verschiedene Seiten seit dem Beginn von P angefragt wurden. Jetzt erhalten wir

$$\begin{aligned}
\mathbb{E}_{\text{ADV}}[t] &= \mathbb{E}_{\text{ADV}}[t_1 + \cdots + t_{k+1}] \\
&= \mathbb{E}_{\text{ADV}}[t_1] + \cdots + \mathbb{E}_{\text{ADV}}[t_{k+1}] \\
&= 1 + \frac{1}{p_2} + \cdots + \frac{1}{p_{k+1}} \\
&= 1 + \sum_{i=2}^{k+1} \frac{1}{p_i} \\
&= 1 + \sum_{i=2}^{k+1} \frac{k}{k - (i - 2)} \\
&= 1 + k \cdot \sum_{i=2}^{k+1} \frac{1}{k - (i - 2)} \\
&= 1 + k \cdot H_k
\end{aligned}$$

  und damit, dass P nach erwarteten kH_k Schritten beendet ist.

Aus den Kosten für eine Phase können wir schliesslich auf die Kosten der gesamten Eingabe schliessen. Wir bleiben an dieser Stelle allerdings auf einer intuitiven Ebene und verzichten auf die mathematischen Details. Mit Yaos Prinzip beziehungsweise einer Verallgemeinerung von Satz 1.15 folgt dann sofort die Aussage des Satzes. □

1.8 Ein Barely-Random-Algorithmus für Paging

Wir haben nun eine gewisse Übersicht über Paging erhalten. Insbesondere sind randomisierte Verfahren exponentiell besser als deterministische und zwar sogar nachweislich, d. h., die oberen und unteren Schranken sind asymptotisch scharf. Wenn wir etwas genauer hinschauen, sehen wir ausserdem, dass die bislang betrachteten Algorithmen sicherlich effizient sind. Was könnten wir nun noch verbessern? Ein Punkt, der für uns auch in folgenden Kapiteln eine nicht unwesentliche Rolle spielt, ist die Frage, wieviel Randomisierung wir eigentlich benötigen. Der eben besprochene randomisierte Online-Algorithmus RMark benutzt beispielsweise eine Anzahl an Zufallsbits, die von der Eingabelänge abhängt, muss also aus einer sehr grossen Menge strat(RMark) auswählen. Wir können jedoch zeigen, dass dies nicht nötig ist, um asymptotisch ein genau so gutes Ergebnis zu erzielen. Konkret geben wir jetzt einen Online-Algorithmus an, der nur eine konstante Anzahl an Zufallsbits benötigt. Einen solchen Algorithmus nennen wir **Barely-Random-Algorithmus** (zu Deutsch also so etwas wie «gerade noch zufälliger Algorithmus»).

Der hier betrachtete Barely-Random-Algorithmus RMarkBarely benutzt zufällig einen deterministischen Online-Algorithmus, der analog zu RMark funktioniert, angefragte Seiten also markiert und nur unmarkierte Seiten verdrängt. Genauer benutzt RMarkBarely b Zufallsbits, um uniform einen deterministischen Algorithmus aus der Menge strat(RMarkBarely) $= \{Mark_1, \ldots, Mark_{2^b}\}$ zu wählen. Die Online-Algorithmen $Mark_i$, $1 \leq i \leq 2^b$, verdrängen unmarkierte Seiten bei einem Seitenfehler jeweils deterministisch und sind so konstruiert, dass sie dies so tun, dass, wenn eine Seite angefragt wird, diese bei möglichst wenigen von ihnen nicht im Cache ist.

Probieren wir zunächst, uns die Idee beispielhaft klar zu machen. Nehmen wir an, wir würden 2 Zufallsbits lesen, würden also einen von 4 Algorithmen $Mark_1$, $Mark_2$, $Mark_3$ und $Mark_4$ wählen. Weiterhin habe der Cache eine Grösse von 7 und sei mit irgendwelchen Seiten $s_1, \ldots, s_7$ initialisiert, sodass wir zu Beginn der Berechnung folgende Situation vorfinden.

s_1	s_2	s_3	s_4	s_5	s_6	s_7

Alle Algorithmen merken sich diese Seiten in dieser Reihenfolge. Nun wird eine Seite s_8 angefragt, die nicht im Cache ist. Dies führt dazu, dass $Mark_i$ die Seite s_i verdrängt, um Platz für s_8 zu schaffen; wir heben markierte Seiten hier hervor, indem wir sie grau hinterlegen.

$Mark_1$:

s_8	s_2	s_3	s_4	s_5	s_6	s_7

$Mark_3$:

s_1	s_2	**s_8**	s_4	s_5	s_6	s_7

$Mark_2$:

s_1	**s_8**	s_3	s_4	s_5	s_6	s_7

$Mark_4$:

s_1	s_2	s_3	**s_8**	s_5	s_6	s_7

Wird nun die Seite s_9 angefragt, so verdrängen die Algorithmen die nächsten Seiten entsprechend der oben festgelegten Reihenfolge der Seiten, die ursprünglich im Cache waren.

$Mark_1$:

s_8	**s_9**	s_3	s_4	s_5	s_6	s_7

$Mark_3$:

s_1	s_2	**s_8**	**s_9**	s_5	s_6	s_7

$Mark_2$:

s_1	**s_8**	**s_9**	s_4	s_5	s_6	s_7

$Mark_4$:

s_1	s_2	s_3	**s_8**	**s_9**	s_6	s_7

Die Idee ist also, dass, wenn nun beispielsweise die alte Seite s_2 angefragt wird, diese nur bei manchen Algorithmen, nämlich bei $Mark_1$ und $Mark_2$ nicht im Cache ist. Wird dann gleich verteilt einer der 4 Algorithmen gewählt, so ist s_2 zu diesem Zeitpunkt also mit einer Wahrscheinlichkeit von $1/2$ noch im Cache.

Allerdings entsteht ein Problem, wenn wir diese Idee weiterdenken; und zwar kann es passieren, dass mehrere Algorithmen ab einem gewissen Zeitpunkt gleich verfahren. Nehmen wir an, es wird nun die alte Seite s_1 angefragt. Dann führt dies zu folgender Situation.

$Mark_1$:

s_8	**s_9**	**s_1**	s_4	s_5	s_6	s_7

$Mark_3$:

s_1	s_2	**s_8**	**s_9**	s_5	s_6	s_7

$Mark_2$:

s_1	**s_8**	**s_9**	s_4	s_5	s_6	s_7

$Mark_4$:

s_1	s_2	s_3	**s_8**	**s_9**	s_6	s_7

Wir sehen, dass die Algorithmen $Mark_1$ und $Mark_2$ nun dieselben markierten (die Reihenfolge ist offensichtlich irrelevant) und unmarkierten Seiten im Cache haben. Ab dem aktuellen

Zeitschritt handelt es sich hier also tatsächlich um denselben deterministischen Online-Algorithmus. Offensichtlich wollen wir eine solche Situation möglichst vermeiden. Ein einfacher Ausweg wäre hier, dass $Mark_1$ nicht die Seite s_3 verdrängt, sondern eine andere Seite, von der wir sicher sein können, dass sie noch kein Algorithmus verdrängt hat. Als Beispiel nehmen wir die Seite s_7 und erhalten die folgende Situation.

$Mark_1$:	s_8	s_9	s_3	s_4	s_5	s_6	s_1	$Mark_3$:	s_1	s_2	s_8	s_9	s_5	s_6	s_7
$Mark_2$:	s_1	s_8	s_9	s_4	s_5	s_6	s_7	$Mark_4$:	s_1	s_2	s_3	s_8	s_9	s_6	s_7

Jetzt wird die Seite s_3 angefragt und wieder würden, wenn alle Algorithmen unserer ursprünglichen Idee folgen, zwei Algorithmen anschliessend gleich arbeiten. Dieses Mal handelt es sich hierbei um die Algorithmen $Mark_3$ und $Mark_4$, sodass die Cache-Inhalte der 4 Algorithmen wie folgt aussehen.

$Mark_1$:	s_8	s_9	s_3	s_4	s_5	s_6	s_1	$Mark_3$:	s_1	s_2	s_8	s_9	s_3	s_6	s_7
$Mark_2$:	s_1	s_8	s_9	s_3	s_5	s_6	s_7	$Mark_4$:	s_1	s_2	s_3	s_8	s_9	s_6	s_7

Auch hier können wir aber einfach Abhilfe verschaffen, indem $Mark_3$ nicht die Seite s_5, sondern die Seite s_6 verdrängt, sodass sich folgende Situation ergibt.

$Mark_1$:	s_8	s_9	s_3	s_4	s_5	s_6	s_1	$Mark_3$:	s_1	s_2	s_8	s_9	s_5	s_3	s_7
$Mark_2$:	s_1	s_8	s_9	s_3	s_5	s_6	s_7	$Mark_4$:	s_1	s_2	s_3	s_8	s_9	s_6	s_7

Nun stossen wir allerdings an eine Grenze, denn ein weiteres Mal können wir diese Strategie nicht verfolgen, da wir nun keine alten Seiten mehr übrig haben, die von noch keinem der Algorithmen verdrängt wurden.

Um RMarkBarely zu analysieren, unterteilen wir die Eingabe wieder in Phasen, die so aufgebaut sind, dass sie aus k verschiedenen Anfragen bestehen und die wir jeweils wieder einzeln analysieren. Nehmen wir an, es würden am Anfang der betrachteten Phase drei neue Seiten angefragt. Jeder der betrachteten Algorithmen hat somit drei Seiten nach dem obigen Prinzip verdrängt. Konkret funktioniert dies so, dass jeder der vier Online-Algorithmen, nachdem die drei neuen Seiten angefragt wurden, mit einer Folge von drei verdrängten alten unmarkierten Seiten assoziiert ist; $Mark_3$ beispielsweise mit (s_3, s_4, s_5) usw. Dies erreichen wir, indem wir die Seiten, die am Anfang im Cache sind, beliebig (z. B. so, wie sie im Cache auftauchen) ordnen und jeden der Algorithmen eine Teilfolge hiervon verdrängen lassen (siehe Abbildung 5).

Im weiteren Verlauf der Berechnung müssen wir immer darauf achten, dass diese Teilfolgen für jeden der Algorithmen eindeutig bleiben, sodass keine zwei Algorithmen dasselbe tun. Beachten Sie, dass die markierten Seiten innerhalb einer Phase für alle Algorithmen immer gleich sind; deswegen werden soeben markierte Seiten aus den Teilfolgen gelöscht.

Wird in einem Zeitschritt eine neue Seite angefragt, werden die jeweiligen Teilfolgen einfach in dem entsprechenden Zeitschritt um 1 erweitert, was bedeutet, dass eine alte

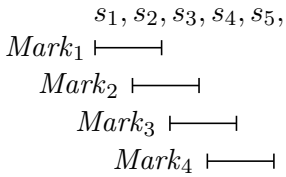


Abbildung 5.

unmarkierte Seite verdrängt wird, um diese Seite in den Cache zu laden. Wird allerdings eine alte unmarkierte Seite s angefragt, ist die Situation komplexer. Die Algorithmen, die s schon im Cache haben, müssen ihre Teilfolgen offensichtlich nicht verändern, schliesslich haben sie s per Definition noch nicht verdrängt. Die Algorithmen, die die Seite allerdings nicht im Cache haben, dürfen nun nicht die der ursprünglichen Ordnung nach nächste Seite s' verdrängen; wie wir oben gesehen haben, können wir in diesem Fall nämlich nicht garantieren, dass die Algorithmen nach einem solchen Schritt noch unterschiedlich agieren. Stattdessen benutzen sie eine Seite s'', die bislang bei allen Algorithmen im Cache ist, die also von noch keinem Algorithmus verdrängt wurde. Wann eine solche Seite noch existiert, werden wir gleich genauer analysieren, nehmen wir jetzt einfach an, dies wäre der Fall. In allen Teilsequenzen tauschen wir somit s gegen s'' aus.

- Wenn wir konkret auf obiges Beispiel zurückkommen, so waren nach den Anfragen der Seiten s_8 und s_9 mit $Mark_1$ und $Mark_2$ die Teilfolgen (s_1, s_2) und (s_2, s_3) assoziiert.
- Wird nun s_1 angefragt, so wird die Teilfolge von $Mark_2$ nicht verändert, denn er hat s_1 noch im Cache und muss sie lediglich markieren.
- Bei der Teilfolge von $Mark_1$ hingegen wird s_1 gelöscht, denn die Seite ist fortan markiert. Wird dann, um s_1 zu laden, die Seite s_3 verdrängt, so sind beide Teilfolgen offensichtlich (s_2, s_3).
- Was wir also tun, ist, $Mark_1$ die unmarkierte Seite s_7 verdrängen zu lassen, was zu einer Teilfolge (s_2, s_7) führt.
- Somit ist garantiert, dass beide Algorithmen nach dieser Anfrage weiterhin unterschiedliche unmarkierte Seiten im Cache haben.

Allgemein werden wir, nachdem wir dieses Vorgehen einige Male wiederholt haben, an einen Punkt kommen, an dem wir keine freien (d. h. unmarkierte alte) Seiten mehr zur Verfügung haben und der Gegenspieler uns zwingen kann, dass fortan mehrere Algorithmen gleich fortfahren. Der randomisierte Online-Algorithmus RMarkBarely arbeitet deswegen in Runden, sodass er während einer Runde garantieren kann, dass noch genug alte unmarkierte Seiten übrig sind und die Algorithmen $Mark_1, \ldots, Mark_{2^b}$ nie denselben Cache-Inhalt haben. Am Ende einer Runde werden wiederum Algorithmen zu Gruppen zusammengefasst, sodass Algorithmen in derselben Gruppe ab diesem Zeitpunkt Seiten nach derselben Strategie verdrängen. Mit anderen Worten, gehen wir hier so vor, dass wir das unumgängliche Zusammenfügen verschiedener Algorithmen so weit wie es geht hinauszögern und dies in einem gewissen Sinne «kontrolliert» tun, anstatt es den Gegenspieler

an einer anderen Stelle tun zu lassen. Wir verfolgen diesen Ansatz, um die Analyse von RMarkBarely einfacher zu gestalten.

Im obigen Beispiel können die zu verdrängenden Seiten so gewählt werden, dass die Algorithmen nach dem Bearbeiten der Anfragen dennoch unterschiedliche Cache-Inhalt haben. Es existiert allerdings eine Folge von vier Seitenanfragen, sodass, wenn die vorgestellte Strategie verfolgt wird, mit der fünften Anfrage unweigerlich folgt, dass zwei Algorithmen fortan gleich agieren.

Beachten Sie, dass RMarkBarely im Folgenden alle deterministischen Algorithmen gleichzeitig simulieren muss, um zu wissen, wie der konkret ausgewählte Algorithmus die Seiten verdrängt.

Wir beschreiben und analysieren RMarkBarely jetzt genauer, wenn wir den folgenden Satz beweisen, der im Wesentlichen aussagt, dass er asymptotisch nicht schlechter als der beste randomisierte Online-Algorithmus für Paging ist. Hierfür werden wir die soeben diskutierten Ideen formalisieren.

Satz 1.19. *Der randomisierte Online-Algorithmus RMarkBarely für Paging liest b Zufallsbits, wobei $2^b < k$ gilt, und erreicht einen erwarteten kompetitiven Faktor von*

$$3b + \frac{2(k+1)}{2^b} .$$

Beweis. Wir betrachten die deterministischen Algorithmen $Mark_1, \ldots, Mark_{2^b}$ und unterteilen die Eingabe wie im Beweis von Satz 1.12 in Phasen, in denen k verschiedene Seiten angefragt werden.

- Wie bereits besprochen, haben die Algorithmen jeweils andere Vorgehensweisen, wie unmarkierte Seiten verdrängt werden.
- Die Menge markierter Seiten ist allerdings nach Definition für alle Algorithmen zu jedem Zeitpunkt gleich. Dies bedeutet wiederum, dass jeder solche Algorithmus zum Beginn einer Phase dieselben Seiten im Cache hat.

Wir betrachten wieder eine beliebige Phase P und bezeichnen Seiten ebenfalls analog zum Beweis von Satz 1.12 mit alt und neu je nachdem, ob sie während P schon einmal im Cache waren. Wieder ignorieren wir ausserdem alle Anfragen an bereits markierte Seiten, weil diese bei keinem der betrachteten Algorithmen zu Seitenfehlern führen. Wir können also der Einfachheit halber sagen, dass P aus k Anfragen $x_1, \ldots, x_k$ an alte und neue Seiten besteht. Diese k Anfragen werden nun wiederum in $b+1$ Runden bearbeitet.

- Runde 0 beginnt mit der Anfrage x_1 und dauert bis Anfrage x_{k-2^b+1} bearbeitet wurde.
- Für $R \geq 1$ besteht Runde R aus den Anfragen $x_{k-2^b/2^{R-1}+2}$ bis $x_{k-2^b/2^R+1}$, also aus genau $2^b/2^R$ Anfragen.

In jeder Runde werden die 2^b Algorithmen nun in $2^b/2^R$ Gruppen $G_1, \ldots, G_{2^b/2^R}$ der Grösse 2^R eingeteilt. In Runde 0 besteht jede Gruppe, wie oben gezeigt, somit aus einem eindeutigen Algorithmus, also ist $G_i = \{Mark_i\}$. Zu Beginn jeder Runde R, für $R \geq 1$, werden die Gruppen G_i und $G_{i+2^b/2^R+1}$ zur Gruppe G_i zusammengefasst. Wenn wir also

acht Algorithmen haben, dann fassen wir unmittelbar vor Runde 1 die beiden Gruppen G_1 und G_5 zusammen zu G_1 und ausserdem G_2 und G_6 zu G_2 usw.

Betrachten wir jetzt eine Anfrage x_{t+1}.

- Mit l_t bezeichnen wir die Anzahl der neuen Seiten, die während des Teils $x_1, \ldots, x_t$ der Eingabe angefragt wurden.
- Somit gibt es $t - l_t$ alte Seiten, die angefragt wurden und bereits markiert sind.
- Also gibt es insgesamt $k - t + l_t$ alte unmarkierte Seiten, von denen manche im Cache sind und manche nicht.
- Wir bezeichnen diese mit $\overline{s}_1, \ldots, \overline{s}_{k-t+l_t}$ und setzen

  $$S_t := (\overline{s}_1, \ldots, \overline{s}_{k-t+l_t}) ,$$

  S_t ist also eine Folge der bis zum Zeitschritt t unmarkierten alten Seiten.

Jede Gruppe G_i verdrängt nun Seiten nach einer bestimmten Vorgabe. Konkret assoziieren wir mit G_i in jedem Zeitschritt t (der Ordnung, die durch S_t implizit gegeben ist, entsprechend) eine Menge von unmarkierten alten Seiten

$$E_{i,t} := \{\overline{s}_i, \ldots, \overline{s}_{i+l_t-1}\}$$

für $l_t \geq 1$ und $E_{i,t} = \emptyset$ für $l_t = 0$. Algorithmen in G_i verdrängen nur Seiten aus ihrem Cache, wenn sie in dieser Menge im entsprechenden Zeitschritt sind (wir zeigen gleich noch, dass dies immer möglich ist). Es ist offensichtlich, dass jede unmarkierte alte Seite höchstens in l_t Mengen $E_{i,t}$ ist. Da wir nur Anfragen an unmarkierte Seiten betrachten, können wir folgende Fallunterscheidung machen.

- Sei x_{t+1} eine neue Seite. In diesem Fall ist $S_{t+1} = S_t$. Weiterhin wird allen Gruppen ein weiteres Objekt zugewiesen, also $E_{i,t+1} = E_{i,t} \cup \{\overline{s}_{i+l_t}\}$. Dies bedeutet nichts anderes, als dass die Seite $\overline{s}_{i+l_t}$ verdrängt wurde, um x_{t+1} in den Cache zu laden.
- Sei x_{t+1} eine unmarkierte alte Seite. Dann wird x_{t+1} aus S_t gelöscht und durch eine beliebige alte Seite ersetzt, die bislang keiner Gruppe zugewiesen ist.

  Es bleibt zu zeigen, dass dies auch möglich ist, so eine Seite also tatsächlich existiert. Wie oben bereits erwähnt, gibt es $k - t + l_t$ alte unmarkierte Seiten, von denen eine x_{t+1} ist. Andererseits sind maximal $2^b/2^R - 1 + l_t - 1$ unmarkierte Seiten auf die Gruppen verteilt. Wegen

  $$(k - t + l_t - 1) - \left(\frac{2^b}{2^R} - 1 + l_t - 1\right) = k - t - \frac{2^b}{2^R} \geq 1$$

  gibt es dann mindestens eine Seite, die die Position von x_{t+1} einnehmen kann.

  Wir haben bereits gesehen, dass jede Seite in maximal l_t Mengen $E_{i,t}$ ist; deswegen ändern sich höchstens l_t Mengen $E_{i,t+1}$ gegenüber $E_{i,t}$.

Nun haben wir gezeigt, wie die Mengen G_i aussehen, in denen die Algorithmen sich die Seiten merken, die sie ersetzen dürfen, wenn eine angefragte Seite nicht im Cache ist.

Sei also $Mark_j$ ein Algorithmus der Gruppe G_i und sei x_t eine Anfrage, die einen Seitenfehler verursacht. Dann nimmt $Mark_j$ diese Seite also in den Cache auf und verdrängt dafür eine Seite, die sich in $E_{i,t+1}$ befindet. Eine solche Seite existiert, denn

- $Mark_j$ hat in seinem Cache bislang $k - t$ alte unmarkierte Seiten,
- in der Menge $E_{i,t+1}$ befinden sich nach Definition l_{t+1} alte unmarkierte Seiten und
- es gibt insgesamt $k + l_{t+1} - (t + 1)$ alte unmarkierte Seiten.
- Somit existieren lediglich $k - t + 1$ alte unmarkierte Seiten, die nicht in $E_{i,t+1}$ sind, also muss der Cache von $Mark_j$ mindestens eine Seite aus $E_{i,t+1}$ enthalten.

Die konstruierten Algorithmen sind also konsistent. Nun müssen wir noch die erwartete Anzahl der Seitenfehler von RMarkBarely abschätzen. Dies tun wir, indem wir die durchschnittliche Fehleranzahl der Algorithmen $Mark_j$ abschätzen.

Berechnen wir zunächst die gesamte Fehleranzahl aller dieser Algorithmen. Bezeichne l nun die Anzahl der neuen Seiten, die in Phase P angefragt werden.

- Für Anfragen an neue Seiten macht jeder Algorithmus offensichtlich einen Fehler, deswegen summieren sich diese Fehler zu $l2^b$.
- Für Anfragen an alte unmarkierte Seiten ist es etwas komplizierter. Wir haben weiter oben gesehen, dass eine solche Anfrage dazu führt, dass im entsprechenden Zeitschritt höchstens l Gruppen einen Seitenfehler machen. Jede Gruppe in Runde R besteht aus genau 2^R Algorithmen. Für $R > 0$ besteht Runde R aus $2^b/2^R$ Zeitschritten, weswegen in diesen Runden höchstens $l2^b$ Seitenfehler passieren. Runde 0 dauert wiederum $k - 2^b + 1$ Zeitschritte, weswegen wir für diese Runde $l(k - 2^b + 1)$ Seitenfehler zählen. Insgesamt ergibt sich eine maximale Anzahl von $l(k - 2^b + 1) + 2^b bl$ Fehlern.
- Zuletzt müssen wir noch die Fehler, die durch das Zusammenfügen von Gruppen entstehen, berücksichtigen. Wenn die Algorithmen der Gruppe $G_{i+2^b/2^R+1}$ denen der Gruppe G_i hinzugefügt werden, so gehen wir davon aus, dass sie anschliessend ebenfalls nur noch Seiten aus der Menge $E_{i,t}$ verdrängen. Die entsprechenden Seiten sind aber unter Umständen gar nicht mehr in ihrem Cache. Um hier auf der sicheren Seite zu sein, nehmen wir einfach an, dass sie die unmarkierten alten Seiten $E_{i,t}$ einfach in ihren Cache laden und unmarkiert lassen. Dies verursacht pro betroffenen Algorithmus maximal l weitere Fehler. Bei jedem Zusammenfügen ist genau die Hälfte der Algorithmen, also 2^{b-1} viele, davon betroffen und dies passiert genau b-mal, nämlich zu Beginn jeder Runde $R \geq 1$. In der Summe kommen wir damit auf $2^{b-1}bl_t \leq 2^{b-1}bl$ Seitenfehlern.

Insgesamt haben somit alle Algorithmen in strat(RMarkBarely) höchstens Kosten von

$$l2^b + l(k - 2^b + 1) + 2^b bl + l2^{b-1}b = l\left(k + 1 + \frac{3}{2}2^b\, b\right)$$

pro Phase, was zu durchschnittlichen Kosten pro Phase von

$$l\left(\frac{k+1}{2^b}+\frac{3}{2}b\right)$$

führt. Nun können wir uns wieder am Beweis von Satz 1.12 orientieren, uns daran erinnern, dass OPT durchschnittlich Kosten $l/2$ pro Phase hat, und dann über alle Phasen aufsummieren.

Wir können den kompetitiven Faktor von RMARKBARELY dann nach oben durch

$$3b+\frac{2(k+1)}{2^b}$$

beschränken. □

Nun können wir Satz 1.19 benutzen, und RMARKBARELY $\lfloor \log_2 k \rfloor - 1$ Zufallsbits lesen lassen, woraus sofort ein kompetitiver Faktor von

$$3\lfloor \log_2 k \rfloor - 3 + \frac{2(k+1)}{2^{\lfloor \log_2 k \rfloor - 1}} = 3\log_2 k + \mathcal{O}(1) \in \mathcal{O}(\log k)$$

folgt. Interpretieren wir die Ergebnisse kurz: Kein deterministischer Online-Algorithmus kann für Paging besser als k-kompetitiv sein. Allerdings kann dieses negative Resultat asymptotisch exponentiell verbessert werden, indem die verfolgte Strategie zufällig aus einer Menge von Strategien gewählt werden, die lediglich eine konstante Grösse hat. Derartige Phänomene motivieren die Analyse der Advice-Komplexität, die wir an späterer Stelle betrachten werden.

1.9 Verwendete und weiterführende Literatur

Dieses Skript richtet sich nach dem Buch «An Introduction to Online Computation: Determinism, Randomization, Advice» des Autors [25]. «Online Computation and Competitive Analysis» von Borodin und El-Yaniv [9] ist sicherlich als das Standardwerk zu Online-Algorithmen zu bezeichnen. Eine Einführung in die Spieltheorie, insbesondere die Theorie der Nullsummenspiele, wird von Straffin in «Game Theory and Strategy» gegeben [39]. Als Einführung in randomisierte Algorithmen empfehlen wir «Randomisierte Algorithmen» von Hromkovič [16] und «Randomized Algorithms» von Motwani und Raghavan [35].

Das Konzept der kompetitiven Analyse beziehungsweise des kompetitiven Faktors geht auf Sleator und Tarjan zurück [38]. In dieser einflussreichen Arbeit wurde unter anderem Paging untersucht und die untere Schranke von k gezeigt. Der randomisierte Online-Algorithmus RMARK stammt von Fiat et al. [14]. Das Minimax-Theorem geht auf von Neumann (nach dem die eingangs erwähnte Rechner-Architektur benannt ist) zurück [40]. Später zeigte Nash [36], dass alle Spiele mit endlich vielen Strategien ein Gleichgewicht in gemischten Strategien besitzen (es wird deswegen hierbei von Nash-Gleichgewichten gesprochen). Das Prinzip von Yao wurde von diesem erstmals angewendet [41] und seitdem für viele Online-Probleme benutzt. Das Lemma von Loomis wurde 1946 von Loomis bewiesen [29]. Borodin und El-Yaniv [9] und Komm [25] formulieren jeweils allgemeinere

Versionen als wir sie hier benutzen. Die kurz diskutierte Diskrepanz zwischen den Paging-Strategien LRU und FIFO in der praktischen Anwendung wurde von Young [42] untersucht. Die untere Schranke für randomisierte Paging-Algorithmen stammt ebenfalls von Fiat et al. [14]; der hier präsentierte Beweis richtet sich nach Motwani und Raghavan [35]. Tatsächlich ist die untere Schranke von H_k nicht nur asymptotisch scharf; McGeoch und Sleator [34] haben einen randomisierten Paging-Algorithmus vorgestellt, der tatsächlich H_k-kompetitiv ist. Der hier präsentierte Barely-Random-Algorithmus stammt von Böckenhauer et al. [8] beziehungsweise Komm und Královič [26].

2 Das Ski-Rental-Problem

Kommen wir einmal kurz zurück zum einführenden Beispiel des Ausleihens einer Ski-Ausrüstung (wir orientieren uns an der englischen Bezeichnung und sagen kurz «Ski-Rental») ohne zu wissen, wie das Wetter in den kommenden Tagen wird. Wir haben informell einen deterministischen Online-Algorithmus BreakEven besprochen, der die Skier am k-ten Tag mit gutem Wetter kauft, wobei k den Preis für das Kaufen bezeichnet. Wir können dies jetzt in der Terminologie, die wir gelernt haben, ausdrücken.

Satz 2.1. *Der deterministische Online-Algorithmus BreakEven für Ski-Rental ist strikt $(2 - 1/k)$-kompetitiv.*

Ausserdem haben wir gesehen, dass diese Schranke scharf ist, denn es existiert ein Gegenspieler, der für jeden deterministischen Online-Algorithmus Alg eine Eingabe konstruieren kann, sodass Alg nicht besser als strikt $(2 - 1/k)$-kompetitiv ist. Hier können wir tatsächlich nur über den strikten kompetitiven Faktor reden, weil die Kosten nicht unbedingt beliebig mit der Eingabelänge wachsen.

Satz 2.2. *Kein deterministischer Online-Algorithmus für Ski-Rental ist besser als strikt $(2 - 1/k)$-kompetitiv.*

2.1 Ein randomisierter Online-Algorithmus

Der strikte kompetitive Faktor jedes deterministischen Online-Algorithmus ist also von unten durch einen Wert beschränkt, der mit wachsenden Kaufkosten k gegen 2 konvergiert. Wir haben gesehen, dass uns Randomisierung bei Paging geholfen hat, die Ausgabequalität eines Online-Algorithmus exponentiell zu verbessern und fragen jetzt, wie weit wir hier mit diesem Ansatz kommen. Offensichtlich werden wir kein so signifikantes Ergebnis erzielen, weil BreakEven schon einen konstanten kompetitiven Faktor erreicht.

Wenn wir kurz darüber nachdenken, wie wir einen randomisierten Online-Algorithmus Rand entwerfen können, stellen wir zunächst fest, dass die Wahrscheinlichkeit, dass Rand die Skier nie kauft, 0 sein muss. Ansonsten ist der erwartete kompetitive Faktor abhängig von der Eingabelänge, wenn der Gegenspieler eine Eingabe konstruiert, in der jeder Tag gutes Wetter hat. Damit wäre Rand wiederum schlechter als BreakEven.

Für jedes i, $1 \leq i \leq k$, sei Buy_i der deterministische Online-Algorithmus, der die Skier bis zum Tag $(i-1)$-ten Tag mit schönem Wetter leiht, um sie dann am i-ten Tag mit schönem Wetter (wenn dieser existiert) zu kaufen. Wir betrachten nun den randomisierten Online-Algorithmus RandSki mit $\text{strat}(\text{RandSki}) = \{Buy_1, \ldots, Buy_k\}$. Sei im Folgenden

$$\delta := \frac{k}{k-1}$$

und

$$\gamma := \frac{\delta - 1}{\delta^k - 1} \, .$$

RandSki realisiert eine Wahrscheinlichkeitsverteilung Prob auf strat(RandSki), wobei die Wahrscheinlichkeit, dass der Online-Algorithmus Buy_i gewählt wird, gegeben ist durch

$$\text{Prob}[Buy_i] := \gamma\delta^{i-1} \ .$$

Wir sehen sofort, dass

$$\sum_{i=1}^{k} \text{Prob}[Buy_i] = \sum_{i=1}^{k} \gamma\delta^{i-1} = \gamma \sum_{i=0}^{k-1} \delta^i = \left(\frac{\delta - 1}{\delta^k - 1}\right)\left(\frac{\delta^k - 1}{\delta - 1}\right) = 1$$

gilt, RandSki ist also wohldefiniert. Wir zeigen zunächst den folgenden Satz, dessen Aussage wir danach genauer untersuchen.

Satz 2.3. *Der randomisierte Online-Algorithmus RandSki für Ski-Rental erreicht einen erwarteten strikten kompetitiven Faktor von*

$$\frac{\delta^k}{\delta^k - 1} \ .$$

Beweis. Als erstes können wir unsere Sichtweise auf die vom Gegenspieler erzeugten Instanzen etwas vereinfachen, um die Analyse nicht unnötig kompliziert zu machen.

- Da jeder deterministische Online-Algorithmus aus strat(RandSki) spätestens am Tag k mit gutem Wetter die Skier gekauft hat, müssen wir nur Eingaben betrachten, die höchstens k solche Tage beinhalten.
- Ferner können wir Tage, an denen das Wetter schlecht ist und die zwischen denen mit gutem Wetter liegen, vernachlässigen, da diese am erreichten kompetitiven Faktor der betrachteten Online-Algorithmen offensichtlich nichts ändern.
- Dies bedeutet, dass wir davon ausgehen können, dass alle betrachteten Instanzen aus j Tagen mit schönem Wetter bestehen, wonach nur noch schlechtes Wetter folgt, $1 \leq j \leq k$.
- Für diese Klasse von Instanzen hat eine optimale Lösung immer Kosten j.

Wenn es j Tage mit schönem Wetter gibt, so hat der Online-Algorithmus Buy_i Kosten von $i - 1 + k$, falls $i \leq j$ ist. Ist hingegen $i > j$, so hat Buy_i Kosten von j. Die erwarteten Kosten von RandSki auf einer solchen Eingabe I, die genau j Tage mit schönem Wetter beinhaltet, sind gegeben durch

$$\begin{aligned}\mathbb{E}[\text{cost}(\text{RandSki}(I))] &= \sum_{i=1}^{j}(i - 1 + k)\text{Prob}[Buy_i] + \sum_{i=j+1}^{k} j\text{Prob}[Buy_i] \\ &= \sum_{i=1}^{j}(i-1)\text{Prob}[Buy_i] + \sum_{i=1}^{j} k\text{Prob}[Buy_i] + \sum_{i=j+1}^{k} j\text{Prob}[Buy_i] \ .\end{aligned}$$

Nun setzen wir die gewählte Wahrscheinlichkeitsverteilung ein und erhalten

$$\mathbb{E}[\text{cost}(\text{RandSki}(I))] = \gamma \sum_{i=1}^{j}(i-1)\delta^{i-1} + \gamma k \sum_{i=1}^{j} \delta^{i-1} + \gamma j \sum_{i=j+1}^{k} \delta^{i-1}$$

$$
\begin{aligned}
&= \gamma \sum_{i=0}^{j-1} i\delta^i + \gamma k \sum_{i=0}^{j-1} \delta^i + \gamma j \sum_{i=j}^{k-1} \delta^i \\
&= \gamma \sum_{i=0}^{j-1} i\delta^i + \gamma k \sum_{i=0}^{j-1} \delta^i + \gamma j \sum_{i=0}^{k-1} \delta^i - \gamma j \sum_{i=0}^{j-1} \delta^i \ .
\end{aligned}
$$

Jetzt können wir die geschlossenen Formen dieser geometrischen Reihen einsetzen,

$$
\delta = \frac{k}{k-1} \iff k = \frac{\delta}{\delta - 1}
$$

benutzen, und erhalten

$$
\begin{aligned}
\mathbb{E}[\text{cost}(\textsc{RandSki}(I))] &= \gamma \frac{(j-1)\delta^{j+1} - j\delta^j + \delta}{(\delta-1)^2} + \gamma k \frac{\delta^j - 1}{\delta - 1} + \gamma j \frac{\delta^k - 1}{\delta - 1} - \gamma j \frac{\delta^j - 1}{\delta - 1} \\
&= \frac{\gamma}{\delta - 1} \left(\frac{(j-1)\delta^{j+1} - j\delta^j + \delta}{\delta - 1} + k(\delta^j - 1) + j(\delta^k - \delta^j) \right) \\
&= \frac{\gamma}{\delta - 1} \Big(k(j-1)\delta^j - kj\delta^{j-1} + k + k(\delta^j - 1) + j(\delta^k - \delta^j) \Big) \\
&= \frac{\gamma}{\delta - 1} \Big(jk\delta^j - kj\delta^{j-1} + j\delta^k - j\delta^j \Big) \\
&= \frac{\gamma}{\delta - 1} \left(\delta^j \underbrace{\left(k - \frac{k}{\delta} - 1 \right)}_{0} + \delta^k \right) j \\
&= \frac{\delta^k \gamma}{\delta - 1} \, j \ .
\end{aligned}
$$

Nun setzen wir $\text{cost}(\textsc{Opt}(I)) = j$ und $\gamma = (\delta - 1)/(\delta^k - 1)$ ein, woraus schliesslich

$$
\begin{aligned}
\mathbb{E}[\text{cost}(\textsc{RandSki}(I))] &= \frac{\delta^k(\delta - 1)}{(\delta - 1)(\delta^k - 1)} \, \text{cost}(\textsc{Opt}(I)) \\
&= \frac{\delta^k}{\delta^k - 1} \, \text{cost}(\textsc{Opt}(I))
\end{aligned}
$$

folgt. □

Uns interessiert das asymptotische Verhalten des erreichten erwarteten kompetitiven Faktors von RandSki, wenn k gegen Unendlich geht. Wir betrachten also

$$
\lim_{k \to \infty} \delta^k = \lim_{k \to \infty} \left(\frac{k}{k-1} \right)^k = \mathrm{e} \ ,
$$

wobei $\mathrm{e} = 2.718\ldots$ die Eulersche Zahl ist, und erhalten

$$
\lim_{k \to \infty} \frac{\delta^k}{\delta^k - 1} = \frac{\mathrm{e}}{\mathrm{e} - 1} \approx 1.582 \ .
$$

Aber ab wann ist RandSki besser als BreakEven? Um diese Frage zu beantworten

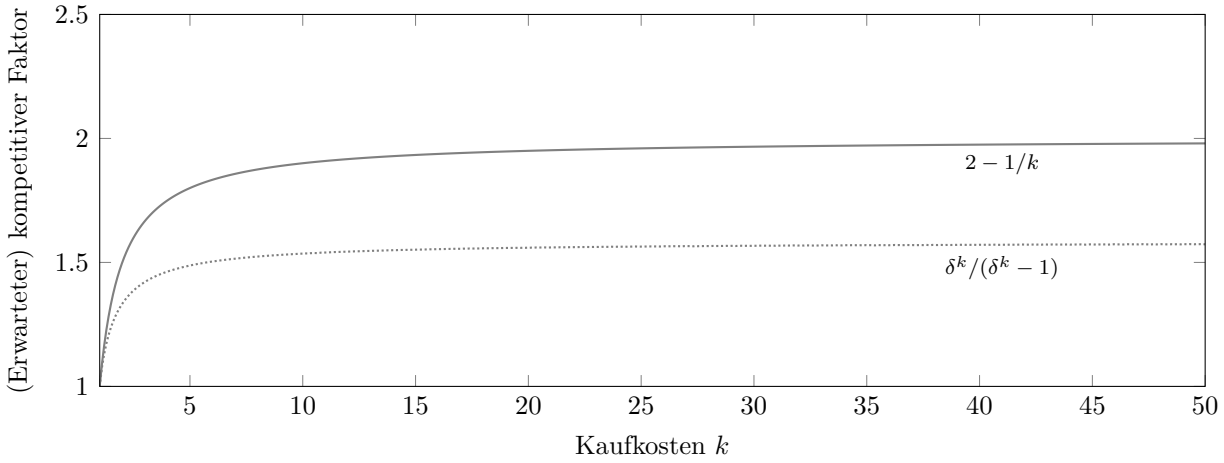


Abbildung 6.

setzen wir voraus, dass $k > 2$ ist und stellen fest, dass

$$\begin{aligned}
& \frac{\left(\frac{k}{k-1}\right)^k}{\left(\frac{k}{k-1}\right)^k - 1} \leq 2 - \frac{1}{k} \\
\iff \quad & \frac{1}{\left(\frac{k}{k-1}\right)^k - 1} \leq \frac{k-1}{k} \\
\iff \quad & \log_{k/(k-1)}\left(1 + \frac{k}{k-1}\right) \leq k \\
\impliedby \quad & \log_{k/(k-1)}\left(2\frac{k}{k-1}\right) \leq k \\
\iff \quad & \log_{k/(k-1)}(2) \leq k - 1 \\
\iff \quad & \log_2\left(\frac{k}{k-1}\right) \geq \frac{1}{k-1}
\end{aligned}$$

für jedes solche k gilt, da die linke Seite der Ungleichung grösser 1 ist und monoton steigt, während die rechte Seite immer kleiner 1 ist. Somit folgern wir, dass RandSki bereits für $k > 2$ besser als jeder deterministische Online-Algorithmus ist (siehe Abbildung 6).

2.2 Verwendete und weiterführende Literatur

Ski-Rental ist ein sehr generisches Problem, auf das wir häufig in praktischen Situationen treffen. Die vorgestellte obere Schranke wurde, zusammen mit einer scharfen unteren Schranke, zuerst von Karlin et al. [22] gezeigt.

3 Das k-Server-Problem

In diesem Kapitel lernen wir ein drittes, sehr allgemeines Online-Problem kennen, das wahrscheinlich als das unter Theoretikern berühmteste bezeichnet werden kann. Bei dem k-Server-Problem (wir in reden in Zukunft einfach kurz von k-Server) bewegen wir k Objekte durch den Raum hin zu Punkten, die in den einzelnen Zeitschritten angefragt werden (siehe Abbildung 7).

Die Motivation kommt aus praktischen Situationen, in denen wir beispielsweise in einer Polizeizentrale per Funk die verschiedenen Polizisten, die an irgendwelchen Orten stationiert sind, zu Einsatzorten schicken wollen. Hierbei wissen wir allerdings natürlich nicht, ob ihr derzeitiger Standpunkt zum Ort eines Verbrechens wird, sobald sie am Einsatzort angekommen sind.

Im Folgenden vereinfachen wir das Problem wieder ein wenig. Um k-Server formalisieren zu können, brauchen wir ausserdem zunächst den Begriff eines metrischen Raums.

Definition 3.1 (Metrischer Raum). *Sei S eine Menge von **Punkten** und $\mathrm{dist}\colon S \times S \to \mathbb{R}$ eine Distanzfunktion. $\mathcal{M} = (S, \mathrm{dist})$ ist ein **metrischer Raum**, wenn folgende Bedingungen gelten.*

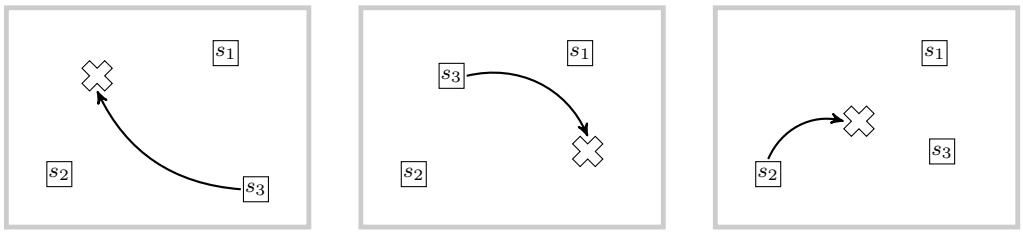
1. ***Definitheit.*** $\mathrm{dist}(v_i, v_i) = 0$ *für alle* $v_i \in S$ *und*
   $\mathrm{dist}(v_i, v_j) > 0$ *für alle* $v_i \neq v_j, v_i, v_j \in S$.
2. ***Symmetrie.*** $\mathrm{dist}(v_i, v_j) = \mathrm{dist}(v_j, v_i)$ *für alle* $v_i, v_j \in S$.
3. ***Dreiecksungleichung.*** $\mathrm{dist}(v_i, v_j) \leq \mathrm{dist}(v_i, v_k) + \mathrm{dist}(v_k, v_j)$ *für alle* $v_i, v_j, v_k \in S$.

Als Teilklasse endlicher metrischer Räume können wir auch vollständige gewichtete ungerichtete Graphen betrachten, deren Kantengewichte die Dreiecksungleichung erfüllen. Ausserdem können wir noch die folgende einfache Beobachtung machen.

Beobachtung 3.2. *Alle Graphen, deren Kantenkosten jeweils nur 1 oder 2 sind, erfüllen die Dreiecksungleichung.*

Definieren wir k-Server nun formal auf allgemeinen metrischen Räumen; diese können endlich viele oder unendlich viele Punkte besitzen.

Definition 3.3 (k-Server). *Sei $\mathcal{M} = (S, \mathrm{dist})$ ein metrischer Raum. Weiterhin seien k **Server** $s_1, \ldots, s_k$ gegeben, die auf einigen der Punkte aus S positioniert sind. Eine **Konfiguration** $C_i \subseteq S$ mit $|C_i| = k$ ist eine Multimenge von Punkten und beschreibt die Positionen der Server im Zeitschritt i. Eine Eingabe $I = (x_1, \ldots, x_n)$ besteht aus n Anfragen, die jeweils ein Punkt aus S sind. Wird im Zeitschritt i ein Punkt x_i angefragt, auf dem kein Server positioniert ist, so muss mindestens ein Server zum Punkt x_i bewegt werden; dies führt zu einer neuen Konfiguration C_{i+1}. Die **Distanz** zwischen zwei Konfigurationen $C_{j'}$ und $C_{j''}$ ist gegeben durch die Kosten eines **Matchings mit minimalen Kosten** zwischen ihnen. Das Ziel ist, die Summe aller Distanzen von direkt aufeinanderfolgenden Konfigurationen zu minimieren.*



(a) (b) (c)

Abbildung 7.

Als Beispiel können wir uns die Ebene mit der Euklidischen Kostenfunktion denken; der Abstand zwischen zwei Punkten $x = (x_1, x_2)$ und $y = (y_1, y_2)$ ist also gegeben durch

$$\text{dist}(x, y) := \sqrt{(y_2 - x_2)^2 + (y_1 - x_1)^2} \, .$$

In diesem Fall können wir die zurückgelegten Strecken der Server mit dem Lineal nachmessen, sofern es sich um einen endlichen Raum handelt.

Prinzipiell erlauben wir einem Online-Algorithmus für k-Server, in jedem Zeitschritt beliebig viele Server zu bewegen. Um an späterer Stelle aber die Analysen zu vereinfachen, betrachten wir manchmal nur sogenannte «träge» Algorithmen.

Definition 3.4 (Träge Online-Algorithmen). *Ein Online-Algorithmus für k-Server heisst **träge**, falls er nur dann in einem Zeitschritt einen Server bewegt, wenn der auf dem angefragten Punkt nicht bereits ein Server positioniert ist. Ferner bewegt ein solcher Algorithmus nie mehr als einen Server in einem Zeitschritt.*

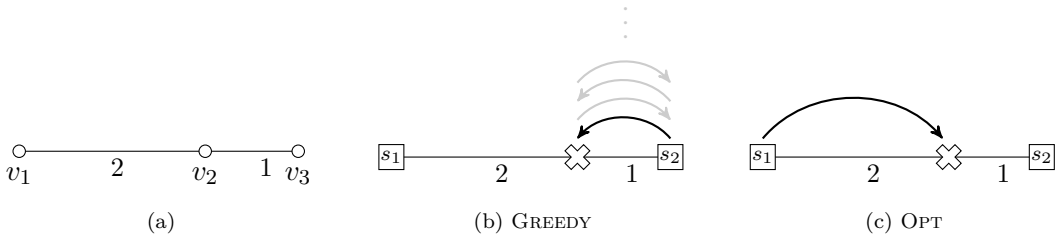
Tatsächlich stellt Definition 3.4 keine Einschränkung dar, wie der folgende Satz zeigt, den wir hier nicht beweisen.

Satz 3.5. *Jeder c-kompetitive Online-Algorithmus für k-Server kann in einen trägen Online-Algorithmus umgewandelt werden, der ebenfalls c-kompetitiv ist.*

Beachten Sie, dass aus Satz 3.5 sofort folgt, dass nie zwei Server auf demselben Punkt positioniert sind, wenn die Startpositionen der Server am Anfang verschieden waren.

3.1 Fundamentale Ergebnisse

Zunächst stellen wir fest, dass k-Server eine Verallgemeinerung von Paging ist. Sei I eine beliebige Instanz von Paging mit einer Cache-Grösse von k und insgesamt m Speicherseiten. Wir erstellen einen vollständigen Graphen mit m Knoten $v_1, \ldots, v_m$ und setzen alle Kantenkosten auf 1. Dann positionieren wir k Server auf den Knoten $v_1, \ldots, v_k$. Die Positionen der Server symbolisieren den Inhalt des Caches. Geschieht ein Seitenfehler, bedeutet dies, dass ein Knoten angefragt wird, auf dem kein Server positioniert ist. Also muss ein Server bewegt werden, was Kosten von 1 verursacht. Der Knoten, von dem der Server wegbewegt wird, entspricht der Seite, die aus dem Cache gelöscht wird.



(a) (b) Greedy (c) Opt

Abbildung 8.

Beachten Sie, dass wir bezüglich der Notation inkonsistent sind; eigentlich sollte Paging analog zu k-Server als k-Paging bezeichnet werden. Wir nehmen dies allerdings in Kauf und bleiben bei den Bezeichnungen, die wir in der Literatur finden.

Satz 3.6. *Es existiert ein metrischer Raum, sodass kein deterministischer Online-Algorithmus für k-Server besser als k-kompetitiv ist.*

Beweis. Dies ist eine direkte Konsequenz aus Satz 1.6. □

Wir wissen, dass diese Schranke für Paging scharf ist, da es diverse deterministische Online-Algorithmen gibt, die k-kompetitiv sind (beispielsweise Fifo, wie in Satz 1.5 gezeigt wurde). Ob dies auch für k-Server gilt, weiss man bis heute nicht. Eine der bekanntesten offenen Fragen im Zusammenhang mit Online-Problemen ist, ob die folgende Behauptung stimmt.

Behauptung 3.7 (k-Server-Vermutung). *Es existiert ein k-kompetitiver deterministischer Online-Algorithmus für k-Server.*

Falls die k-Server-Vermutung wahr ist, würde dies heissen, dass k-Server genau so schwer ist wie Paging. Dies wäre eine sehr interessante Einsicht, da, wie wir gerade gesehen haben, ersteres eine starke Verallgemeinerung des letzteren ist.

Eine offensichtliche Strategie, die ein Online-Algorithmus verfolgen könnte, ist der naive Greedy-Ansatz, also immer den Server zum angefragten Punkt zu bewegen, der diesem am nächsten ist. Diese Strategie ist nicht kompetitiv. Für einen solchen Online-Algorithmus Greedy können wir leicht eine Instanz konstruieren, für die die Kosten mit jeder weiteren Anfrage grösser werden, während ein optimaler Algorithmus Kosten von insgesamt 2 besitzt (eigentlich sogar $1 + \varepsilon$ für jedes $\varepsilon > 0$).

Satz 3.8. *Der Online-Algorithmus Greedy ist nicht kompetitiv für k-Server.*

Beweis. Wir betrachten die folgende Instanz I, die aus einen Graphen mit drei Punkten v_1, v_2 und v_3 besteht. Die Kosten sind gegeben durch $\text{dist}(v_1, v_2) = 2$ und $\text{dist}(v_2, v_3) = 1$. Weiterhin sind zwei Server s_1 und s_2 gegeben, die anfangs auf den Knoten v_1 und v_3 positioniert sind. Die betrachtete Instanz besteht nun aus n Anfragen von abwechselnd v_2 und v_3. Die erste Anfrage v_2 muss beantwortet werden, indem Greedy entweder s_1 oder s_2 zu diesem Punkt bewegt. Da $\text{dist}(v_2, v_3) < \text{dist}(v_1, v_2)$ gilt, wird s_2 ausgewählt, was zu Kosten von 1 führt. Danach wird v_3 angefragt, und wieder ist s_2 der nähere

Server, weswegen GREEDY auch im zweiten Zeitschritt Kosten von 1 hat. Dies wird nun $n/2$-mal wiederholt, sodass GREEDY insgesamt n zahlt. OPT bewegt hingegen bei der ersten Anfrage s_1, hat einmalig Kosten von 2 und keine weiteren (siehe Abbildung 8). Es folgt $\text{cost}(\text{GREEDY}(I))/\text{cost}(\text{OPT}(I)) = n/2$, und somit existiert keine Konstante c, sodass GREEDY c-kompetitiv ist. □

Wir haben gesehen, dass («ein bisschen») Randomisierung für Paging sehr hilfreich ist. Es ist deswegen durchaus denkbar, dass wir auch hier eine exponentielle Verbesserung erreichen können, wenn wir den betrachteten Online-Algorithmen erlauben, ihre Entscheidungen auf Zufallsbits zu basieren. Auch hierfür gibt es eine berühmte Vermutung, die bis heute weder bewiesen noch widerlegt werden konnte.

Behauptung 3.9 (*k*-Server-Vermutung für randomisierte Algorithmen). *Es existiert ein im Erwartungswert $\Theta(\log k)$-kompetitiver randomisierter Online-Algorithmus für k-Server.*

Wir werden im letzten Kapitel hierauf zurückkommen. Vorerst bleiben wir aber bei deterministischen Online-Algorithmen. Es würde leider den Rahmen dieser Vorlesung sprengen, k-Server auf allgemeinen metrischen Räumen zu analysieren. Aus diesem Grund werden wir bei dem oben besprochenen Spezialfall der Linie bleiben, beziehungsweise diesen leicht verallgemeinern.

3.2 Potentialfunktionen

Um obere Schranken zu zeigen, lernen wir nun aber zunächst eine weitere Technik kennen, die uns hilft, die **amortisierten Kosten**, also die durchschnittlichen Kosten, die bei der Bearbeitung einer Eingabe entstehen, eines Online-Algorithmus gegen die der optimalen Lösung abzuschätzen. Dieses Werkzeug ermöglicht es uns manchmal, eine geschickte amortisierte Analyse durchzuführen.

- Wenn wir zeigen wollen, dass ein Online-Algorithmus ALG einen kompetitiven Faktor von c erreicht, muss, für eine Konstante α, die Ungleichung $\text{cost}(\text{ALG}(I)) \leq c \cdot \text{cost}(\text{OPT}(I)) + \alpha$ für jede Instanz $I = (x_1, \ldots, x_n)$ gelten.
- Bezeichnen wir nun mit $\text{cost}(\text{ALG}(x_i))$ die Kosten von ALG auf der Anfrage x_i, $1 \leq i \leq n$.
- Könnten wir zeigen, dass $\text{cost}(\text{ALG}(x_i)) \leq c \cdot \text{cost}(\text{OPT}(x_i))$ für $1 \leq i \leq n$ gilt, wären wir offensichtlich fertig.
- Das funktioniert aber in der Regel nicht. Bislang haben wir deswegen bei Paging Phasen analysiert und nicht einzelne Anfragen.
- Es reicht aus, zu zeigen, dass die Kosten durchschnittlich höchstens c-mal grösser sind als die Kosten einer optimalen Lösung.
- Dies bedeutet, dass ALG in einzelnen Zeitschritten ruhig etwas mehr zahlen darf, wenn er in anderen dafür weniger zahlt.

- Wenn diese Werte sich ausgleichen, ist ALG c-kompetitiv.

Jetzt wollen wir diese einfache Idee formalisieren, indem wir uns für jede Anfrage anschauen, was ALG zu viel oder zu wenig bezahlt hat. Zunächst benötigen wir hierfür den Begriff der **Konfiguration** eines Online-Algorithmus. Diesen Begriff formal zu definieren ist schwierig, deswegen bleiben wir auf einer intuitiven Ebene. Beispielsweise ist eine Konfiguration eines Paging-Algorithmus der Inhalt des Caches, bei k-Server ist es die Position der k Server in Raum. Hier haben wir den Begriff sogar in der Definition benutzt. Wir bemerken, dass eine Konfiguration etwas anderes ist als der Zustand des Algorithmus, mit dem wir seinen internen Speicherinhalt etc. assoziieren oder bei Turingmaschinen etwa die Kopfpositionen auf den Bändern. Für die Konfiguration interessiert uns gewissermassen lediglich, was nach aussen sichtbar ist.

Bezeichne nun also $\mathcal{K}_{\text{ALG}}$ die Menge aller Konfigurationen eines Online-Algorithmus ALG für eine Instanz I und $\mathcal{K}_{\text{OPT}}$ die Menge der Konfigurationen eines beliebigen festen optimalen Algorithmus OPT für I. Eine **Potentialfunktion** Φ ist eine Funktion

$$\Phi\colon \mathcal{K}_{\text{ALG}} \times \mathcal{K}_{\text{OPT}} \to \mathbb{R}\ .$$

Die Berechnung eines festen deterministischen Online-Algorithmus und (einer festen optimalen Lösung) lässt sich als Folge von Konfigurationen darstellen, die eindeutig durch die Anfragen der Eingabe gegeben sind. Deshalb kann Φ auch als Funktion von $\mathcal{I}$ nach $\mathbb{R}$ definiert werden; wir benutzen im Folgenden beides.

Wir bezeichnen $\Phi(x_i)$ als das **Potential** von ALG in Zeitschritt i; dabei ist $\Phi(x_0)$ das Potential vor der ersten Anfrage. Für eine Eingabe $I = (x_1, \ldots, x_n)$ entsteht somit eine Folge von Potentialen $\Phi(x_0), \ldots, \Phi(x_n)$. Das Potential ist also ein Wert, der sich während des Laufs von ALG abhängig von den Konfigurationen von ALG und OPT ändert und zwar im Zeitschritt i um den Wert $\Phi(x_i) - \Phi(x_{i-1})$.

Wir definieren jetzt die **amortisierten Kosten** von ALG auf der Anfrage x_i als

$$\text{amcost}(\text{ALG}(x_i)) := \text{cost}(\text{ALG}(x_i)) + \Phi(x_i) - \Phi(x_{i-1})\ .$$

Dies sind die Kosten, die wir für ALG im Durchschnitt garantieren möchten; $\text{cost}(\text{ALG}(x_i))$ bezeichnet hingegen die **tatsächlichen Kosten** von ALG auf x_i. Bleiben wir kurz auf einer intuitiven Ebene. Wir wollen im Folgenden zeigen, dass

$$\text{amcost}(\text{ALG}(x_i)) \leq c \cdot \text{cost}(\text{OPT}(x_i))$$

und somit

$$\text{cost}(\text{ALG}(x_i)) \leq c \cdot \text{cost}(\text{OPT}(x_i)) - (\Phi(x_i) - \Phi(x_{i-1}))$$

in jedem Zeitschritt i gilt. Zahlen wir nun in einem Zeitschritt zu viel, also mehr als $c \cdot \text{cost}(\text{OPT}(x_i))$, so können wir dies ausgleichen, indem das Potential um den entsprechenden Betrag verringert wird. Hierbei fordern wir allerdings, dass das Potential nicht negativ werden darf. Auf der anderen Seite darf das Potential natürlich auch wachsen, was uns dann in einem späteren Zeitschritt erlaubt, etwas mehr zu bezahlen. Unser Ziel wird also sein, eine Potentialfunktion und ein c zu finden, so dass wir dies garantieren können. Drücken wir dies nun formal aus.

Satz 3.10. *Sei $I = (x_1, \ldots, x_n)$ eine beliebige Eingabe für ein Online-Minimierungsproblem Π und sei Φ eine Potentialfunktion. Für einen beliebigen Online-Algorithmus* Alg *seien die amortisierten Kosten von* Alg *auf I definiert wie oben. Wenn*

- *eine Konstante $\beta \in \mathbb{R}^+$ existiert, sodass für alle i, $1 \leq i \leq n$, $0 \leq \Phi(x_i) \leq \beta$ gilt, und*
- $\text{amcost}(\textsc{Alg}(x_i)) \leq c \cdot \text{cost}(\textsc{Opt}(x_i))$ *für alle i, $1 \leq i \leq n$, dann*

ist Alg *c-kompetitiv für Π.*

Beweis. Für die Kosten von Alg auf I folgt nach obigen Voraussetzungen sofort

$$\begin{aligned}
\text{cost}(\textsc{Alg}(I)) &= \sum_{i=1}^{n} \text{cost}(\textsc{Alg}(x_i)) \\
&= \sum_{i=1}^{n} (\text{amcost}(\textsc{Alg}(x_i)) + \Phi(x_{i-1}) - \Phi(x_i)) \\
&= \Phi(x_0) - \Phi(x_n) + \sum_{i=1}^{n} \text{amcost}(\textsc{Alg}(x_i)) \\
&\leq \Phi(x_0) - \Phi(x_n) + \sum_{i=1}^{n} c \cdot \text{cost}(\textsc{Opt}(x_i)) \\
&= \Phi(x_0) - \Phi(x_n) + c \cdot \text{cost}(\textsc{Opt}(I)) \\
&\leq \Phi(x_0) + c \cdot \text{cost}(\textsc{Opt}(I)) \\
&\leq c \cdot \text{cost}(\textsc{Opt}(I)) + \beta
\end{aligned}$$

und somit ist Alg c-kompetitiv, wobei wir $\alpha := \beta$ in Definition 1.3 setzen. □

Die Schwierigkeit liegt jetzt darin, eine Potentialfunktion für ein gegebenes Problem zu finden, für die die beiden obigen Bedingungen gelten. Wir zeigen nun eine Anwendung für eine Teilklasse von k-Server-Eingaben.

Es sei hier noch erwähnt, dass Satz 3.10 so verallgemeinert werden kann, dass das Potential auch negativ werden darf, solange der Wert durch eine Konstante beschränkt ist. Ausserdem kann die Idee der Potentialfunktion auf randomisierte Online-Algorithmen erweitert werden.

3.3 *k*-Server auf der Linie

Wir haben bereits gesehen, dass schon die sehr einfache Klasse von Instanzen mit drei Punkten auf einer Linie für die Greedy-Strategie schwer ist. Jetzt betrachten wir eine Verallgemeinerung dieser Eingaben, und zwar die reelle Zahlenachse zwischen 0 und 1 beziehungsweise den metrischen Raum $\mathcal{M}_{[0,1]} = ([0,1], \text{dist})$ mit $\text{dist}(x,y) = |x-y|$ für zwei Punkte x und y.

Greedy ist somit ebenfalls beliebig schlecht für diesen Raum. Wir schauen uns deswegen den Online-Algorithmus DC («Double Coverage», Algorithmus 3.1) an, der hier ein gutes Ergebnis erzielt. DC benutzt eine simple Greedy-Strategie, wenn alle Server nur auf einer

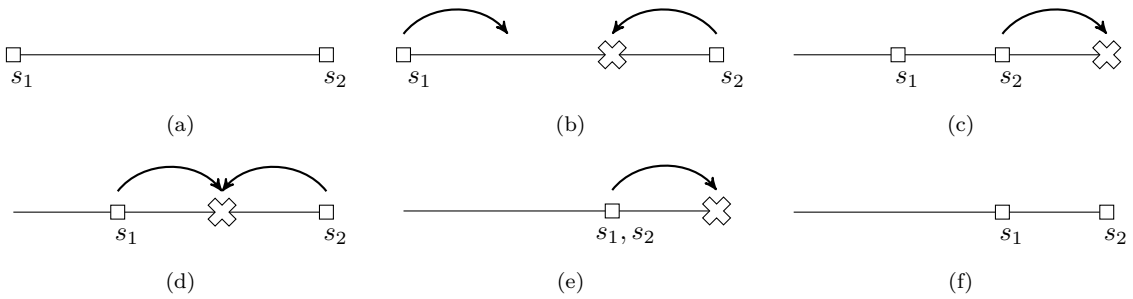


Abbildung 9.

Seite einer Anfrage liegen. Gibt es allerdings sowohl Server, die links als auch welche die rechts der Anfrage positioniert sind, bewegt der Algorithmus die beiden, die am nächsten links und rechts von ihm stehen, um die minimale Distanz auf den angefragten Punkt zu, bis einer der Server auf ihm steht. Beachten Sie, dass DC also kein träger Online-Algorithmus ist.

Zunächst einmal stellen wir fest, dass das schwere Beispiel für GREEDY für DC nicht zu beliebig schlechten Kosten führt, sondern diese konstant bleiben, obwohl sie ebenfalls nicht optimal sind (siehe Abbildung 9). Genauer sind die (konstanten) Kosten 6, also 3-mal höher als die (konstanten) Kosten eines optimalen Algorithmus.

Satz 3.11. *Der Online-Algorithmus DC ist k-kompetitiv für k-Server auf $\mathcal{M}_{[0,1]}$.*

Beweis. Wir definieren nun eine Potentialfunktion Φ, für die die in Satz 3.10 geforderten Eigenschaften gelten. Wie oben bereits besprochen, ist eine Konfiguration eines Online-Algorithmus ALG durch die Positionen aller Server gegeben.

Somit ist die Menge $\mathcal{K}_{\text{DC}}$ gegeben durch alle möglichen Konfigurationen $K_{\text{DC}} = \{p_1^{\text{DC}}, \ldots, p_k^{\text{DC}}\}$, die die Serverpositionen von DC im aktuellen Zeitschritt angeben; für $\mathcal{K}_{\text{OPT}}$ definieren wir $K_{\text{OPT}} = \{p_1^{\text{OPT}}, \ldots, p_k^{\text{OPT}}\}$ analog. Φ setzt sich nun aus zwei Termen zusammen. Zum einen sind dies die mit k multiplizierten Kosten eines Matchings minimaler

Algorithmus 3.1: Der Algorithmus DC für $\mathcal{M}_{[0,1]}$

$s := x_i$;
$s_{\text{rechts}} := \lambda$; $s_{\text{links}} := \lambda$;
$s_{\text{rechts}} :=$ Server direkt rechts neben s;
$s_{\text{links}} :=$ Server direkt links neben s;
if $s_{\text{rechts}} = \lambda$
  output «Bewege s_{links} zu s»;
else if $s_{\text{links}} = \lambda$
  output «Bewege s_{rechts} zu s»;
else
  $d := \min\{\text{dist}(s_{\text{rechts}}, s), \text{dist}(s_{\text{links}}, s)\}$;
  output «Bewege s_{rechts} um d nach links und s_{links} um d nach rechts»;
end

Kosten zwischen den Server-Positionen der beiden Algorithmen, die wir für zwei Konfigurationen K_{DC} und K_{Opt} mit $M_{\min}(K_{\text{DC}}, K_{\text{Opt}})$ bezeichnen. Zum anderen betrachten wir die Summe aller paarweisen Distanzen zwischen den einzelnen Servern von DC und bezeichnen diese mit $D_{\text{DC}}(K_{\text{DC}})$. Dann setzen wir

$$\Phi(K_{\text{DC}}, K_{\text{Opt}}) := k \cdot M_{\min}(K_{\text{DC}}, K_{\text{Opt}}) + D_{\text{DC}}(K_{\text{DC}}) \ .$$

Offensichtlich hängt der Wert von Φ nicht von der Eingabelänge ab, sondern ist konstant; ausserdem ist er positiv. Konkret können wir ihn für beliebige Konfigurationen von DC und Opt abschätzen mit

$$0 \leq \Phi(K_{\text{DC}}, K_{\text{Opt}}) \leq k \cdot k + \binom{k}{2} \leq 2k^2 =: \beta \ ,$$

da die maximale Distanz im betrachteten Raum 1 ist. Damit ist die erste Eigenschaft, die notwendig ist, um Satz 3.10 anzuwenden, bereits gezeigt.

Die amortisierten Kosten für eine Anfrage x_i von DC setzen wir nun wie gefordert auf

$$\text{amcost}(\text{DC}(x_i)) = \text{cost}(\text{DC}(x_i)) + \Phi(x_i) - \Phi(x_{i-1}) \ .$$

Für den zweiten Punkt zeigen wir jetzt noch, dass

$$\text{amcost}(\text{DC}(x_i)) \leq k \cdot \text{cost}(\text{Opt}(x_i))$$

und somit

$$\Phi(x_i) - \Phi(x_{i-1}) \leq k \cdot \text{cost}(\text{Opt}(x_i)) - \text{cost}(\text{DC}(x_i))$$

gilt. Hierfür müssen wir sowohl die Bewegungen von Opt als auch die von DC berücksichtigen, denn beide führen zu einer Änderung des Potentials. Für Opt nehmen wir nach Satz 3.5 ohne Beschränkung der Allgemeinheit an, dass er träge ist. DC ist hingegen, wie bereits erwähnt, nach Definition kein träger Algorithmus.

Wir müssen nun untersuchen, was passiert, wenn DC und Opt in einem Zeitschritt die Konfiguration wechseln. Wir stellen uns für die Analyse vor, dass die beiden Algorithmen ihre Züge abwechselnd machen. Zuerst ändert Opt seine Konfiguration und dann DC. In beiden Fällen schätzen wir die Änderung des Potentials ab. Betrachten wir zuerst die Änderung des Potentials, die Opt verursacht.

- Offensichtlich beinträchtigt die Bewegung der Server von Opt nicht den Term $D_{\text{DC}}(K_{\text{DC}})$, sondern nur die Kosten des Matchings $M_{\min}(K_{\text{DC}}, K_{\text{Opt}})$.
- Da Opt ein träger Algorithmus ist, bewegt er höchstens einen Server pro Anfrage.
- Somit legt dieser Server s genau die Distanz $\text{cost}(\text{Opt}(x_i))$ zurück.
- Nachdem Opt den Server s bewegt hat, hat sich die Distanz jedes Servers in der alten Konfiguration von DC zu s um maximal diesen Wert erhöht, also auch zu dem Server, mit dem s gematcht wurde. Die Kosten von $M_{\min}(K_{\text{DC}}, K_{\text{Opt}})$ ändern sich also höchstens um $\text{cost}(\text{Opt}(x_i))$.

- Somit ist die Potentialänderung maximal $k \cdot \text{cost}(\text{OPT}(x_i))$.

Jetzt schauen wir, was durch die Änderung der Konfiguration von DC in diesem Zeitschritt passiert, nachdem OPT seine Konfiguration schon geändert hat. Zu diesem Zweck unterscheiden wir zwei Fälle, und zwar davon abhängig, ob DC einen oder zwei Server bewegt. Nehmen wir zunächst an, der Algorithmus bewegt einen Server und zwar, ohne Beschränkung der Allgemeinheit, den Server s_{rechts} nach links, da kein Server links der Anfrage positioniert ist.

- Der Server s_{rechts} vergrössert seinen Abstand zu allen anderen Servern von DC um jeweils $\text{cost}(\text{DC}(x_i))$. Der zweite Summand $D_{\text{DC}}(K_{\text{DC}})$ des Potentials vergrössert sich also um $(k-1) \cdot \text{cost}(\text{DC}(x_i))$.

- Jetzt wollen wir abschätzen, wie sehr sich der Wert des ersten Summanden, also des Matchings minimaler Kosten, ändert.

- Beachten Sie, dass es uns hier um das Matching vor und nach der Bewegung des Servers s_{rechts} geht, wobei OPT schon vorher den Server s bewegt hat, der jetzt an der Stelle x_i positioniert ist, die der Anfrage entspricht.

- Wir behaupten, dass es, bevor DC den Server s_{rechts} bewegt, ein Matching minimaler Kosten gibt, bei dem s_{rechts} und s gematcht wurden.

- Nehmen wir an, dies sei nicht der Fall. Dann ist s mit einem anderen Server s' von DC gematcht und s' ist weiter rechts von s als s_{rechts}. Andererseits ist s_{rechts} mit einem Server s'' von OPT gematcht.

- Jetzt können wir eine Fallunterscheidung bezüglich der Lage von s'' durchführen. Für jeden der vier möglichen Fälle sehen wir sofort, dass es ein Matching gibt, das nicht teurer ist und bei dem s und s_{rechts} gematcht werden (siehe Abbildung 10).

- Nachdem DC die Anfrage x_i beantwortet hat, sind die Server s und s_{rechts} nun auf demselben Punkt positioniert, wobei s_{rechts} um $\text{cost}(\text{DC}(x_i))$ bewegt wurde.

- Also existiert nun ein Matching, das so teuer ist wie das vorangegangene minus diese Kosten, der erste Term verringert sich also um mindestens $k \cdot \text{cost}(\text{DC}(x_i))$.

- Mit diesen Überlegungen folgt nun

$$\begin{aligned}\Phi(x_i) &\leq \Phi(x_{i-1}) + k \cdot \text{cost}(\text{OPT}(x_i)) + (k-1) \cdot \text{cost}(\text{DC}(x_i)) - k \cdot \text{cost}(\text{DC}(x_i)) \\ &= \Phi(x_{i-1}) + k \cdot \text{cost}(\text{OPT}(x_i)) - \text{cost}(\text{DC}(x_i)) \ ,\end{aligned}$$

  was zu zeigen war.

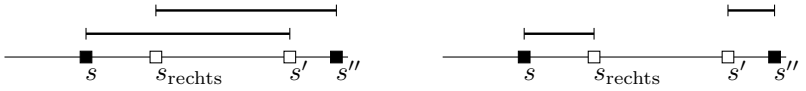
Es bleibt der Fall zu betrachten, dass DC zwei Server s_{rechts} und s_{links} bewegt, von denen mindestens einer auf dem Punkt x_i platziert wird.

- Beide Server bewegen sich jeweils genau um den Wert $\text{cost}(\text{DC}(x_i))/2$.

(a) Fall 1: s'' links von s

(b) Fall 2: s'' zwischen s und s_{rechts}

(c) Fall 3: s'' zwischen s_{rechts} und s'



(d) Fall 4: s'' rechts von s'

Abbildung 10.

- Für jeden Server von DC ausser s_{rechts} und s_{links} wird der Abstand zu einem der beiden vergrössert und der Abstand zum anderen um denselben Wert verringert, da sich beide Server in die entgegengesetzte Richtung bewegen. Somit ändert sich für die Summe der Distanzen zwischen den Servern nur, dass s_{rechts} und s_{links} nun um $\text{cost}(\text{DC}(x_i))$ näher beieinander stehen. Der zweite Summand $D_{\text{DC}}(K_{\text{DC}})$ des Potentials wird also um $\text{cost}(\text{DC}(x_i))$ kleiner.

- Wir können mit einer ähnlichen Argumentation wie oben zeigen, dass entweder s und s_{rechts} oder s und s_{links} gematcht werden, bevor DC seine Konfiguration ändert. Nachdem DC die beiden Server bewegt hat, hat sich dieser Wert von $\text{cost}(\text{DC}(x_i))/2$ auf 0 verringert.

- Auf der anderen Seite wird der andere der beiden Server mit einem Server s''' gematcht. Offensichtlich kann sich der Abstand zwischen diesen beiden Servern höchstens um $\text{cost}(\text{DC}(x_i))/2$ erhöhen.

- Die Kosten eines Matchings minimaler Kosten ändern sich im schlechtesten Fall also nicht beziehungsweise werden sogar kleiner.

- Daraus folgt unmittelbar

$$\Phi(x_i) \leq \Phi(x_{i-1}) + k \cdot \text{cost}(\text{OPT}(x_i)) - \text{cost}(\text{DC}(x_i)) \; .$$

Also sind die Voraussetzungen für Satz 3.10 erfüllt und DC ist k-kompetitiv. □

3.4 Verwendete und weiterführende Literatur

Vorgestellt wurde k-Server von Manasse et al. [30], die auch die k-Server-Vermutung formuliert haben. Eine Übersicht wird von Koutsoupias [27] gegeben. Der bislang beste bekannte deterministische Online-Algorithmus ist der Work-Function-Algorithmus von Papadimitriou und Koutsoupias [28] aus dem Jahr 1995, der versucht, einen optimalen Offline-Algorithmus so gut es geht zu imitieren. Dieser erreicht einen kompetitiven Faktor von $2k - 1$ und, wie 2009 von Emek et al. [12] gezeigt wurde, einen strikten kompetitiven Faktor von $4k - 2$. Wir haben die untere Schranke von k nur für eine besondere Klasse metrischer Räume gezeigt, und zwar die, die wir für Paging erhalten. Es kann allerdings allgemein gezeigt werden, dass diese Schranke für alle metrischen Räume gilt. Damit ist insbesondere DC ein bestmöglicher Online-Algorithmus für $\mathcal{M}_{[0,1]}$. Mit ein paar einfachen Überlegungen kann gezeigt werden, dass DC auch auf Bäumen k-kompetitiv ist [9].

2011 wurde von Bansal et al. [2] ein randomisierter Online-Algorithmus vorgestellt, der einen im Erwartungswert polylogarithmischen kompetitiven Faktor erreicht, solange der betrachtete metrische Raum im Vergleich zur Anzahl der Server nicht zu gross wird. Trotz jahrzehntelanger Forschung ist es bislang nicht gelungen, einen randomisierten Online-Algorithmus zu finden, der besser ist als der deterministische von Papadimitriou und Koutsoupias. Somit ist die randomisierte k-Server-Vermutung nach wie vor offen.

4 Die Advice-Komplexität von Online-Algorithmen

Bevor wir in Kapitel 2 einen randomisierten Online-Algorithmus für Ski-Rental analysiert haben, haben in der Einleitung gesehen, dass deterministische Online-Algorithmen fast doppelt so schlecht sind wie die optimale Lösung. Wir wollen diese Tatsache nun etwas gründlicher untersuchen. Die Fragen, die wir uns zunächst stellen, sind zum einen

Warum sind wir fast doppelt so schlecht?

und etwas genauer

Was fehlt uns?

Die Antwort scheint auf der Hand zu liegen. Was uns fehlt, ist schlicht die Kenntnis der vollständigen Eingabe. Das ist sicher richtig; hätten wir einen verlässlichen beliebig langen Wetterbericht, könnten wir uns immer optimal entscheiden, ob wir die Ski-Ausrüstung direkt am ersten Tag kaufen oder leihen. Aber bei genauerem Hinsehen stellen wir fest, dass dies gar nicht nötig ist. Alles, was uns an Information fehlt, ist ein einziges Bit, das uns sagt, ob wir die Ausrüstung direkt am ersten Tag kaufen sollen oder gar nicht.

Der kompetitive Faktor sagt uns, wieviel wir zahlen, wenn wir ein Online-Problem unter normalen Bedingungen bearbeiten; die **Advice-Komplexität** hingegen sagt uns, «wofür» wir zahlen. Bei Ski-Rental zahlen wir im Worst-Case fast das Zweifache der Kosten der optimalen Lösung. Was uns jedoch fehlt, gewissermassen also das, wofür wir zahlen, ist nur ein einziges Bit, also die kleinste Einheit an Information, die wir uns vorstellen können.

Allgemein ist die Frage nach dem «Wofür» natürlich viel schwerer zu beantworten und wir werden im Folgenden ein paar Beispiele untersuchen, die zeigen, dass sich unterschiedliche Probleme ganz unterschiedlich verhalten hinsichtlich der Informationen, die wir benötigen, um eine gute Ausgabequalität zu erreichen.

Um diese Informationen wiederum messen zu können, benutzen wir ein Modell, in dem wir ein Orakel einführen, das die vollständige Eingabe kennt. Dieses Orakel kann binär kodierte Informationen über diese Eingabe auf ein **Advice-Band** schreiben, das danach von dem diese Eingabe bearbeitenden Online-Algorithmus benutzt werden kann. Informell können wir das Modell wie folgt beschreiben.

- Wir konstruieren nicht mehr einfach einen Online-Algorithmus, sondern einen solchen Algorithmus Alg zusammen mit einem Orakel. Wir nennen Alg einen **Online-Algorithmus mit Advice**.
- Das Orakel schreibt für jede Eingabe eine bestimmte Anzahl sogenannter **Advice-Bits** auf das Advice-Band.
- Der Gegenspieler kennt sowohl Alg als auch das Orakel. Insbesondere weiss er, welchen Advice das Orakel für die jeweils konstruierten Eingaben auf das Advice-Band schreibt.

Den Ablauf können wir dann wie folgt skizzieren.

- Der Gegenspieler konstruiert eine Eingabe I der Länge n, sodass der kompetitive Faktor von ALG, unter Verwendung des Advice-Bandes, möglichst gross ist. Zu diesem Zweck kann er beispielsweise ALG auf jedem Advice-String simulieren. Dabei kennt der Gegenspieler die Anzahl an Advice-Bits $b(n)$, die ALG maximal lesen wird.

- Das Orakel inspiziert anschliessend die vom Gegenspieler erstellte Eingabe I und schreibt einen binären Advice-String auf das Advice-Band, der von I abhängt.

- ALG liest die Eingabe I und berechnet die Ausgabe O unter Verwendung des Advice-Bandes; hierbei liest ALG höchstens $b(n)$ Advice-Bits.

- Erreicht ALG so einen kompetitiven Faktor von c, so sagen wir, dass ALG c-kompetitiv mit Advice-Komplexität $b(n)$ ist, beziehungsweise maximal $b(n)$ Advice-Bits braucht, um c-kompetitiv zu sein.

Hierbei ist wichtig, dass das Advice-Band unendlich lang ist. Dies fordern wir, um zu vermeiden, dass Informationen in die Länge des Advice-Strings hineinkodiert werden können. Ferner muss das Band von ALG sequentiell gelesen werden.

Wir sehen hier eine Analogie zu der bereits besprochenen Randomisierung. Wir verbieten allerdings, dass das Orakel einen Zufallsstring auf das Advice-Band schreibt; der Advice wird deterministisch aus der Eingabe abgeleitet. Auf der anderen Seite können wir uns vorstellen, dass das Orakel einfach für jede Eingabe einen String auf das Advice-Band eines Online-Algorithmus ALG mit Advice schreibt, der dem «besten» Zufallsstring eines randomisierten Online-Algorithmus RAND für diese Eingabe entspricht. Formulieren wir diese Überlegungen in folgender Beobachtung.

Beobachtung 4.1. *Sei Π ein Online-Problem. Wenn ein randomisierter Online-Algorithmus für Π existiert, der c-kompetitiv im Erwartungswert ist und höchstens $b(n)$ Zufallsbits für Eingaben der Länge n benutzt, so existiert auch ein Online-Algorithmus mit Advice für Π, der c-kompetitiv ist und $b(n)$ Advice-Bits verwendet.*

Existiert auf der anderen Seite kein Online-Algorithmus mit Advice, der c-kompetitiv mit Advice-Komplexität $b(n)$ für Π ist, so existiert auch kein randomisierter Online-Algorithmus, der c-kompetitiv im Erwartungswert ist und weniger als $b(n)$ Zufallsbits benötigt.

In Kapitel 6 kommen wir auf den Zusammenhang zwischen Advice und Randomisierung zurück. Definieren wir jetzt einen Online-Algorithmus mit Advice formal; unserer soeben angestellten Diskussion folgend scheint es sinnvoll, diese Definition analog zu Definition 1.7 für randomisierte Online-Algorithmen zu formulieren.

Definition 4.2 (Online-Algorithmus mit Advice). *Sei $I = (x_1, \ldots, x_n)$ eine Eingabe für ein Online-Problem Π. Ein* ***Online-Algorithmus* ALG *mit Advice*** *berechnet die Ausgabe $\textsc{Alg}^{\phi}(I) = (y_1, \ldots, y_n)$, wobei y_i nur von $\phi, x_1, \ldots, x_i$ und $y_1, \ldots, y_{i-1}$ abhängt; ϕ bezeichnet einen* ***binären Advice-String****. ALG ist* ***c-kompetitiv mit Advice-Komplexität b(n)****, wenn eine nicht negative Konstante α existiert, sodass für jede Eingabe I von Π gilt, dass*

$$\text{cost}\Big(\textsc{Alg}^{\phi}(I)\Big) \leq c \cdot \text{cost}(\textsc{Opt}(I)) + \alpha \ ,$$

falls Π ein Online-Minimierungsproblem ist, oder

$$\text{gain}(\textsc{Opt}(I)) \leq c \cdot \text{gain}\Big(\textsc{Alg}^{\phi}(I)\Big) + \alpha \ ,$$

falls Π ein Online-Maximierungsproblem ist, wobei OPT wieder ein optimaler Offline-Algorithmus ist, und ALG höchstens $b(n)$ Advice-Bits für Eingaben der Länge n liest. Gilt obige Ungleichung für $\alpha = 0$, so heisst ALG ***strikt c-kompetitiv mit Advice-Komplexität b(n)****; ist ALG strikt 1-kompetitiv mit Advice-Komplexität $b(n)$, so nennen wir ALG* ***optimal****.*

Auch hier lassen wir ϕ weg, um die Notation einfach zu halten. Für einen solchen Algorithmus ALG fordern wir also, dass es für jede Eingabe einen Advice-String gibt, der ALG erlaubt, einen entsprechenden kompetitiven Faktor zu erzielen.

Wir können jetzt mit den obigen Überlegungen zu Ski-Rental den folgenden einfachen Satz formulieren.

Satz 4.3. *Für Ski-Rental existiert ein Online-Algorithmus mit Advice, der mit einem Advice-Bit eine optimale Ausgabe erzeugt.*

4.1 Die Advice-Komplexität von Paging

Wir kommen zurück zum in Kapitel 1 eingeführten Paging. Wir haben für dieses Problem bewiesen, dass deterministische Online-Algorithmen nicht besser als k-kompetitiv sein können, randomisierte Online-Algorithmen im Erwartungswert jedoch exponentiell besser sind. Wir fragen als erstes, was uns eine kleine Anzahl an Advice-Bits helfen kann. Da wir bereits einen Barely-Random-Algorithmus für das Problem kennen, folgt zusammen mit Beobachtung 4.1, dass es einen Online-Algorithmus mit Advice gibt, der mindestens dieselbe Ausgabequalität erreicht und hierbei eine konstante Anzahl an Advice-Bits verwendet.

Satz 4.4. *Es existiert ein Online-Algorithmus mit Advice für Paging, der b Advice-Bits benutzt, wobei $2^b < k$ gilt, und einen kompetitiven Faktor von*

$$3b + \frac{2(k+1)}{2^b}$$

erreicht.

Als nächstes leiten wir eine untere Schranke für eine konstante Anzahl an Advice-Bits her, die sehr nah an die soeben gezeigt obere Schranke herankommt.

Satz 4.5. *Jeder Online-Algorithmus mit Advice, der b Advice-Bits verwendet, kann keinen besseren kompetitiven Faktor erreichen als*

$$\frac{k}{2^b} - \mathcal{O}\left(\frac{1}{n}\right).$$

Beweis. Sei ALG also ein Online-Algorithmus mit Advice, der eine konstante Anzahl von b Advice-Bits benutzt. Es reicht auch für diesen Beweis, wenn wir uns wieder auf Instanzen beschränken, die $k+1$ verschiedene Seiten anfragen. Wir strukturieren diese Instanzen nun auf eine bestimmte Art.

- Alle Eingaben starten mit der Anfrage der Seite mit dem Index $k+1$, die bei keinem Algorithmus zu Beginn im Cache ist.
- Wir ordnen alle Instanzen nun in einem k-ären Baum $\mathcal{T}$ der Höhe $n-1$, also mit n Ebenen, an (siehe Abbildung 11).
- Jeder Knoten hat dabei k Nachfolger, die die k möglichen Seiten repräsentieren, die im entsprechenden Zeitschritt angefragt werden können, wobei nie dieselbe Seite in zwei direkt aufeinanderfolgenden Zeitschritten angefragt wird.
- Die Blätter repräsentieren vollständige Eingaben der Länge n; allgemein entspricht jeder Knoten v einem Präfix von genau den Eingaben, die Blätter im Teilbaum mit Wurzel v sind.
- Die Wurzel des Baums entspricht somit genau der Anfrage $k+1$, die Präfix von jeder der betrachteten Eingaben ist.

Für jede Eingabe wählt ALG einen der 2^b vielen Online-Algorithmen aus strat(ALG) aus. Jede Eingabe wird also von einem dieser Algorithmen bearbeitet. Wir geben den Blättern von $\mathcal{T}$ nun Farben und zwar davon abhängig, welcher Algorithmus dieses Blatt bearbeitet. Jedes Blatt erhält also eine eindeutige Farbe zwischen 1 und 2^b.

Sei $d := \lfloor n/2^b \rfloor$. Für jeden Knoten v sei $\mathcal{T}_v$ der Teilbaum von $\mathcal{T}$, der v als Wurzel hat. Wir zeigen jetzt per Induktion über die Anzahl der Farben i, dass für alle Knoten v gilt, dass, wenn $\mathcal{T}_v$ eine Höhe von mindestens $d \cdot i - 1$ besitzt und alle Blätter von $\mathcal{T}_v$ mit höchstens i Farben gefärbt sind, es eine Instanz gibt, sodass ALG mindestens d Seitenfehler auf $\mathcal{T}_v$ verursacht.

Induktionsanfang. Sei $i = 1$ und sei $\mathcal{T}_v$ ein Baum mit einer Höhe von mindestens $d-1$ (das heisst mit mindestens d Ebenen), dessen Blätter also alle mit einer Farbe gefärbt sind. Dies bedeutet aber, dass ALG für alle durch diesen Baum repräsentierten Instanzen denselben Advice benutzt und somit vollständig deterministisch auf diesem Teilbaum arbeitet. Wie wir bereits aus Satz 1.6 wissen, existiert dann eine Eingabe, sodass ALG auf dieser Teilinstanz genau einen Seitenfehler pro Zeitschritt macht und somit insgesamt d Seitenfehler.

Induktionsvoraussetzung. Die Annahme gelte für $i-1$.

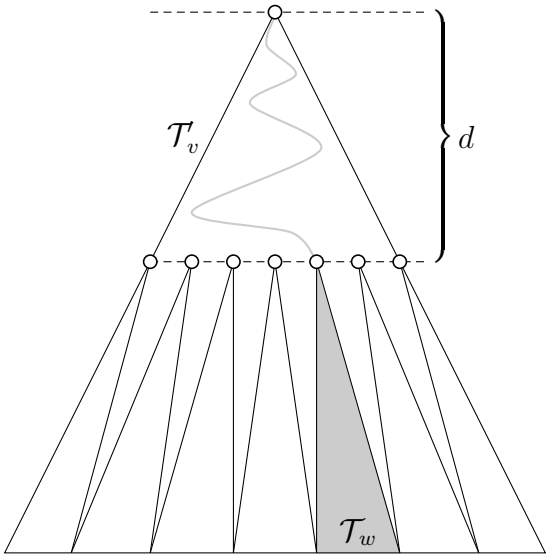


Abbildung 11.

Induktionsschritt. Sei $i > 1$. Wir schneiden $\mathcal{T}_v$ nach d Ebenen ab und erhalten so einen Baum $\mathcal{T}'_v$ der Höhe $d-1$. Ferner ist jedes Blatt von $\mathcal{T}'_v$ Wurzel eines Baums $\mathcal{T}_w$ der Höhe mindestens $d(i-1)-1$. Wir unterscheiden zwei Fälle bezüglich der Farben, die in den Bäumen $\mathcal{T}_w$ benutzt werden.

- Wenn ein Baum $\mathcal{T}_w$ existiert, dessen Blätter mit höchstens $i-1$ Farben gefärbt sind, so gilt nach Induktionsvoraussetzung, dass ALG auf einer Instanz, die von einem Blatt in $\mathcal{T}_w$ repräsentiert wird, mindestens d Fehler macht. Somit existiert offensichtlich auch eine Instanz, die von einem Blatt von $\mathcal{T}_v$ repräsentiert wird.
- Existiert ein solcher Baum nicht, so wissen wir, da alle Blätter von $\mathcal{T}_v$ mit i Farben gefärbt sind, dass eine Farbe F existiert, sodass jeder Teilbaum $\mathcal{T}_w$ ein Blatt hat, das mit F gefärbt ist. Nehmen wir jetzt ein Blatt von jedem Teilbaum, so werden also alle entsprechenden Eingaben mit demselben Advice bearbeitet. Also wird wieder derselbe deterministische Algorithmus für alle diese Instanzen ausgewählt.

  Wegen der Konstruktion von $\mathcal{T}$ decken die Eingaben, die zu den jeweilen Bäumen $\mathcal{T}_w$ führen, alle möglichen Anfragefolgen der Länge d ab. Somit macht ALG wiederum auf einer von ihnen mindestens d Fehler.

Nun können wir dieses Ergebnis für $i = 2^b$ verwenden und erhalten, dass ALG mindestens

$$\left\lfloor \frac{n}{2^b} \right\rfloor$$

Fehler auf Eingaben macht, die durch einen Baum mit $n \geq \lfloor n/2^b \rfloor \cdot 2^b$ Ebenen repräsentiert werden beziehungsweise Länge n haben. Auf der anderen Seite wissen wir bereits, dass ein optimaler Algorithmus höchstens alle k Anfragen einen Seitenfehler verursacht. Somit sind die optimalen Kosten für Eingaben der Länge n höchstens n/k.

Jetzt setzen wir die beiden Schranken in Definition 1.3 des kompetitiven Faktors ein und erhalten

$$\frac{n}{2^b} - 1 \leq \left\lfloor \frac{n}{2^b} \right\rfloor \leq c \cdot \frac{n}{k} + \alpha \,,$$

weswegen der kompetitive Faktor von ALG durch

$$c \geq \frac{k}{2^b} - \frac{(\alpha+1)k}{n} = \frac{k}{2^b} - \mathcal{O}\left(\frac{1}{n}\right)$$

beschränkt werden kann. □

Zuletzt interessieren wir uns für die Anzahl an Advice-Bits, die wir benötigen, um optimal zu sein. Eine triviale obere Schranke für die Anzahl der Advice-Bits ist $n\lceil \log_2 k \rceil$.

Satz 4.6. *Es existiert ein optimaler Online-Algorithmus mit Advice für Paging, der für Eingaben der Länge n höchstens $n\lceil \log_2 k \rceil$ Advice-Bits verwendet.*

Beweis. Für jede Anfrage kann einem Online-Algorithmus mit Advice einfach der Index der Seite mitgeteilt werden, der von einem festen optimalen Algorithmus im entsprechenden Zeitschritt verdrängt wird. □

Geht es besser? Die Antwort ist «Ja». Diese obere Schranke können wir um einen Faktor von fast $\lceil \log_2 k \rceil$ verbessern.

Satz 4.7. *Es existiert ein optimaler Online-Algorithmus* LIN *mit Advice für Paging, der für Eingaben der Länge n höchstens $n + k$ Advice-Bits verwendet.*

Beweis. Sei OPT ein fester optimaler Algorithmus für Paging. Wir nennen eine Seite im Cache von OPT «aktiv», wenn sie noch einmal angefragt wird, bevor OPT sie verdrängt.

- LIN ist so konstruiert, dass er jede aktive Seite im entsprechenden Zeitschritt auch in seinem Cache hat.
- Um dies zu garantieren, speichert LIN für jede Seite im Cache ein Bit, das diese Seite als aktiv oder nicht markiert.
- Für jede Anfrage, die einen Seitenfehler verursacht, verdrängt LIN eine beliebige nicht aktive Seite.
- Wird jetzt also eine Seite s angefragt, für die LIN einen Seitenfehler verursacht, so kann dies keine aktive Seite sein, denn LIN hat alle aktiven Seiten in jedem Zeitschritt ebenfalls im Cache.
- Ferner kann s auch nicht im Cache von OPT sein, denn dies würde sofort der Definition von nicht aktiven Seiten widersprechen.
- Somit verursacht s auch einen Seitenfehler für OPT.

Jetzt können wir die Anzahl an benötigten Advice-Bits abschätzen.

- Pro Anfrage liest LIN ein Bit vom Advice-Band, das angibt, ob die aktuell angefragte Seite aktiv ist.
- Vor der ersten Anfrage müssen die k im Cache befindlichen Seiten noch initialisiert werden, wofür offensichtlich ein Bit pro Speicherzelle benötigt wird.
- Insgesamt werden also $n + k$ Advice-Bits gebraucht.

Somit verursacht LIN nicht mehr Seitenfehler für eine beliebige Anfrage als OPT. Damit ist er aber selbst ein optimaler Online-Algorithmus. □

4.2 Die Advice-Komplexität von *k*-Server

Für das k-Server Problem wollen wir untersuchen, wie viele Advice-Bits nötig und ausreichend sind, um eine optimale Lösung zu berechen. Eine obere Schranke kann leicht gezeigt werden.

Satz 4.8. *Für k-Server existiert ein optimaler Online-Algorithmus KSOPT mit Advice, der $n\lceil \log_2 k \rceil$ Advice-Bits für Eingaben der Länge n benutzt.*

Beweis. Mit jeder Anfrage liest KSOPT genau $\lceil \log_2 k \rceil$ Bits vom Advice-Band, die den Index des Servers kodieren, der im entsprechenden Zeitschritt von einem fixen optimalen Algorithmus OPT verwendet wird. Offensichtlich berechnet KSOPT somit dieselbe Ausgabe wie OPT. Da k bekannt ist, ist keine weitere Information nötig. □

Wir sehen hier sofort die Analogie zu Satz 4.6, in welchem wir einen optimalen Online-Algorithmus mit Advice für Paging untersucht haben. Geht es auch hier wieder besser wie in Satz 4.7? Hier ist die Antwort «Nein» und wir zeigen, dass die in Satz 4.8 gezeigte Schranke asymptotisch auch notwendig ist. Wir beschränken uns hier zunächst auf Eingaben der Länge k und bezeichnen mit $\mathrm{e} = 2.718\ldots$ wieder die Eulersche Zahl.

Satz 4.9. *Für eine Instanz der Länge k muss jeder optimale Online-Algorithmus mit Advice für k-Server mindestens $k(\log_2 k - \log_2 \mathrm{e})$ Advice-Bits verwenden.*

Beweis. Wir konstruieren einen vollständigen bipartiten Graphen $G = (U \cup W, E, \mathrm{dist})$, wobei $U = \{u_1, \ldots, u_k\}$ und $W = \{w_1, \ldots, w_{2^k}\}$ ist, mit einer Kostenfunktion $\mathrm{dist}\colon E \to \{1, 2\}$.

- Weil $|W| = 2^k$ ist, können wir eine bijektive Funktion $\mathrm{set}\colon W \to \mathcal{P}(U)$ definieren, die jeden Knoten aus W auf eine eindeutige Teilmenge der Knoten in U abbildet.
- Jetzt können wir die Kantenkosten definieren. Für $u \in U$ und $w \in W$ setzen wir

$$\mathrm{dist}(\{u, w\}) := \begin{cases} 2 & \text{wenn } u \in \mathrm{set}(w), \\ 1 & \text{sonst .} \end{cases}$$

- Der so erhaltene Graph ist natürlich noch keine zulässige Eingabe für k-Server; wir müssen ihn noch vollständig machen. Wir setzen alle Kosten der Kanten aus $(U \times U) \cup (W \times W)$ auf 2.

- Nach Beobachtung 3.2 wissen wir, dass G metrisch ist.

- Sei $\overline{G}_i \subseteq W$ die Menge aller Knoten aus W, die genau den Teilmengen von U mit i, Elementen entsprechen, $0 \leq i \leq k$, also

  $$\overline{G}_i = \{w \in W \mid |\text{set}(w)| = i\}$$

  und ferner ist

  $$|\overline{G}_i| = \binom{k}{i} .$$

  Da die Menge G_k für die Beweisführung unwichtig ist, ignorieren wir sie im Folgenden.

Wir nennen Kanten mit Kosten 1 fortan «billig» und Kanten mit Kosten 2 «teuer». Ein Beispiel für 4-Server ist in Abbildung 12 gegeben. Der Übersicht halber haben wir hier die Kanten $(U \times U) \cup (W \times W)$ weggelassen; billige Kanten sind mit dünnen Linien dargestellt, teure mit dicken.

Wir konstruieren nun eine Klasse $\mathcal{I}'$ von Instanzen auf dem Graphen G. Die Idee ist, diese so zu definieren, dass jede Instanz eine eindeutige optimale Lösung besitzt, die mit einer Permutation der Menge $\{1, \ldots, k\}$ korrespondiert.

- Zu Beginn stehen alle Server auf den Knoten in U; konkret ist der Server s_i auf dem Knoten u_i, $1 \leq i \leq k$ positioniert (siehe Abbildung 12).

- Die Anfragen sind gegeben durch $(x_1, \ldots, x_k)$, sodass, für $1 \leq j \leq k$,

  - $x_j \in \overline{G}_{j-1}$ und
  - $\text{set}(x_j) \subseteq \text{set}(x_{j+1})$.

Erklären wir diese Idee kurz intuitiv.

- Die erste Anfrage x_1 fragt den eindeutigen Knoten aus W (nämlich den Knoten aus $\overline{G}_0$) an, der nur billige Kanten zu allen Knoten in U hat.

- Jede weitere Anfrage x_j fragt einen der Knoten aus $\overline{G}_{j-1}$ an. Allerdings geschieht dies nicht willkürlich, sondern derart, dass alle Knoten aus U, die zur Anfrage x_{j-1} teure Kanten hatten, auch zu x_j teure Kanten haben. Es kommt also eine weitere teure Kante zu einem Knoten in U pro Anfrage hinzu.

- Die Anfrage x_k hat schliesslich nur noch eine billige Kante zu einem Server in U.

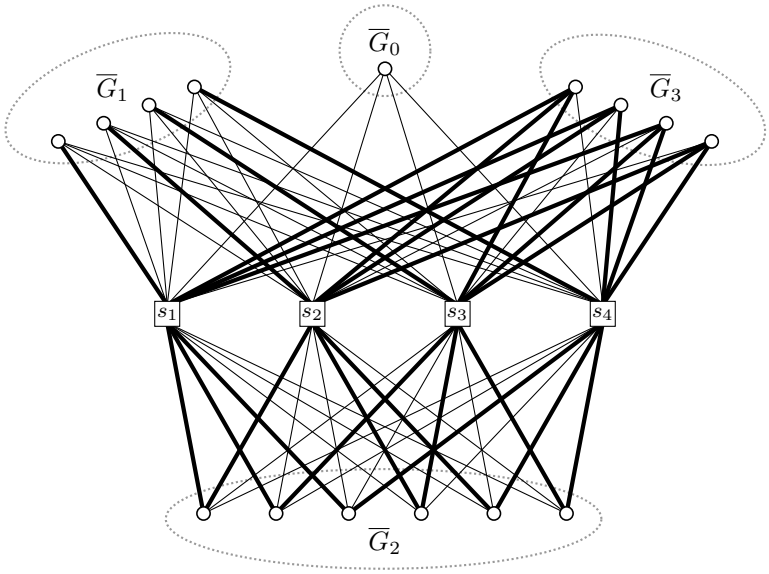


Abbildung 12.

Sei nun $I \in \mathcal{I}'$ eine feste Instanz, die Anfragen nach obigem Vorgehen macht. Um die Beschreibung im Folgenden einfach zu halten, identifizieren wir alle Knoten aus U mit ihren Indizes. Wir können I nun als eine Permutation π_I von $\{1, \ldots, k\}$ interpretieren, wobei

$$\pi_I(j) := \text{set}(x_{j+1}) \setminus \text{set}(x_j), \text{ für } 1 \leq j \leq k-1 \text{ und}$$
$$\pi_I(k) := U \setminus \{\pi_I(j) \mid 1 \leq j \leq k-1\} = U \setminus \text{set}(x_k)$$

ist. Erklären wir es wieder intuitiv; $\pi_I(j)$ bezeichnet genau den Knoten aus U, der von der Anfrage x_{j+1} an nur noch mit teuren Kanten verbunden ist. Jetzt beschreiben wir einen eindeutigen optimalen Algorithmus Opt für I, der eine Lösung berechnet, die Kosten von k besitzt.

- Wir können Opt ebenfalls durch π_I beschreiben.
- Für jede Anfrage x_j benutzt Opt genau den Server, der in U positioniert ist, der den Index $\pi_I(j)$ besitzt. Dies ist also genau der Server, der mit allen folgenden Anfragen mit Kanten der Kosten 2 verbunden ist, mit dem aktuellen aber noch mit einer Kante mit Kosten 1.
- Folglich hat diese Lösung in jedem Zeitschritt Kosten von 1.

Es ist einfach zu sehen, dass Opt tatsächlich eine optimale Lösung berechnet.

- Alle Server starten auf Knoten der Menge U.
- Alle Anfragen sind unterschiedliche Knoten aus der Menge W.
- Alle Kantenkosten sind mindestens 1.
- Also hat jede Lösung Kosten von mindestens k.

Um zu beweisen, dass die von Opt berechnete Lösung tatsächlich eindeutig ist, müssen wir etwas genauer hinschauen.

- Betrachten wir die Offline-Version des Problems, in der alle Anfragen von Anfang an bekannt sind und ein Algorithmus Off sie in einer beliebigen Reihenfolge beantworten kann.
- Die letzte Anfrage x_k fragt einen Knoten aus $\overline{G}_{k-1}$ an. Dieser Knoten ist mit genau einem Server, der auf dem Knoten $\pi_I(k)$ positioniert ist, in U mit Kosten 1 verbunden; alle weiteren Kanten zu Knoten in U haben Kosten von 2. Wenn Off optimal sein will, darf er in keinem Zeitschritt Kosten von mehr als 1 verursachen, denn kein Knoten wird zweimal angefragt. Er ist also gezwungen, diesen Server für x_k zu benutzen.
- Anschliessend beantwortet er Anfrage x_{k-1} aus der Gruppe $\overline{G}_{k-2}$. Der angefragte Knoten hat ebenfalls eine Kante mit Kosten 1 zu dem Knoten $\pi_I(k)$, auf dem jetzt aber kein Server mehr positioniert ist. Dieser Server darf nicht verwendet werden, denn in G gilt $\text{dist}(w_1, w_2) = 2$ für alle $w_1, w_2 \in W$. Will Off also wieder Kosten von 1 garantieren, so muss er den Server benutzen, der auf dem zweiten Knoten von U positioniert ist, der eine billige Kante zu x_{k-1} besitzt; dies ist genau der Knoten $\pi_I(k-1)$.
- Dieses Argument können wir nun iterieren und dabei sehen wir, dass die von Off berechnete Lösung genau der von Opt berechneten entspricht.

Aus dieser Argumentation folgt, dass ein Online-Algorithmus, der für eine gegebene Instanz in einem Zeitschritt einen Server benutzt, den Opt nicht benutzt, nicht optimal sein kann. In einfachen Worten, einmal gemachte Fehler können nicht in späteren Zeitschritten wieder gut gemacht werden.

Jede Instanz in $\mathcal{I}'$ korreliert also mit einer Permutation von $\{1, \ldots, k\}$ und besitzt ferner eine eindeutige optimale Lösung mit Kosten k, die wir ebenfalls mit dieser Permutation beschreiben können. Jetzt müssen wir noch zeigen, dass wir tatsächlich einen eindeutigen Advice-String für jede Instanz aus $\mathcal{I}'$ benötigen.

- Nehmen wir an, es existiere ein optimale Online-Algorithmus Alg mit Advice, der weniger als $\log_2(k!)$ Advice-Bits benötigt, und somit weniger als $k!$ veschiedene Instanzen aus $\mathcal{I}'$ unterscheiden kann.
- Dann existieren zwei Instanzen I_1 und I_2 in $\mathcal{I}'$, die verschiedene eindeutige optimale Lösungen besitzen, für die Alg aber denselben Advice benutzt. Somit verhält sich Alg auf beiden Instanzen wie ein deterministischer Online-Algorithmus. Wir nehmen an, Alg sei optimal für beide.
- Sei i die erste Stelle, an der sich die entsprechenden Permutationen π_{I_1} und π_{I_2} das erste Mal unterscheiden.
- Ab dem nächsten Zeitschritt $i + 1$ sind somit zwei verschiedene Knoten aus U mit teuren Kanten mit allen folgenden Anfragen verbunden. Alg muss somit im Zeitschritt i zwei verschiedene Server benutzen, um optimal für beide Instanzen zu sein.

- Dies ist aber ein Widerspruch dazu, dass ALG sich auf beiden Instanzen bis zu diesem Zeitpunkt gleich verhält, was daraus folgt, dass er denselben Advice gelesen hat und ein gemeinsames Präfix der Eingaben I_1 und I_2.

Jetzt können wir die benötigte Anzahl an Advice-Bits abschätzen. Da jeder Online-Algorithmus $k!$ Instanzen aus $\mathcal{I}'$ unterscheiden muss, um auf jeder von ihnen optimal zu sein, muss er mindestens

$$\begin{aligned}\log_2(k!) &\geq \log_2\left(\sqrt{2\pi k}\left(\frac{k}{\mathrm{e}}\right)^k\right)\\ &= \frac{1}{2}(\log_2(2\pi) + \log_2 k) + k(\log_2 k - \log_2 \mathrm{e})\\ &\geq k(\log_2 k - \log_2 \mathrm{e})\end{aligned}$$

Advice-Bits lesen, wobei wir für die Abschätzung der Fakultät die Stirling-Formel benutzt haben. □

Zwar ist Optimalität als ein strikter kompetitiver Faktor von 1 definiert, aber trotzdem sind wir mit dem Ergebnis nicht ganz zufrieden, da die konstruierten Instanzen nicht beliebig lang sein können. Wir können die in Satz 4.9 verwendete Idee aber iterieren, wie der folgende Satz zeigt.

Satz 4.10. *Jeder optimale Online-Algorithmus mit Advice für k-Server muss mindestens*

$$n\left(\frac{1}{2}\log_2 k - \frac{1}{2}\log_2 \mathrm{e}\right)$$

Advice-Bits verwenden.

Beweis. Sei ohne Beschränkung der Allgemeinheit n ein ungerades Vielfaches von k, also ist $n = (2t-1)k$ für ein $t \in \mathbb{N}^+$. Wir konstruieren jetzt einen Graphen, der aus

$$t = \frac{n-k}{2k}$$

Teilgraphen G_j besteht, die jeweils genau so aufgebaut sind wie der Graph G im Beweis von Satz 4.9. Seien diese Graphen gegeben durch $(U_j \cup W_j, E_j, \mathrm{dist}_j)$ für $1 \leq j \leq t$. Wir verbinden sie untereinander wie folgt (siehe Abbildung 13).

- Analog zum Beweis von Satz 4.9 definieren wir die Gruppen $\overline{G}_{j,i}$ von Knoten, für $1 \leq j \leq t$, $1 \leq i \leq k-1$.
- Jetzt verbinden wir alle Knoten aus W_j, $1 \leq j \leq t-1$, mit den Knoten von $U_{j+1} = \{u_{j+1,1}, \ldots, u_{j+1,k}\}$, sodass, für $1 \leq i \leq k-1$, die Kanten von allen Knoten in $\overline{G}_{j,i}$ zum Knoten $u_{j+1,i+1}$ Kosten von 1 haben und die übrigen Kosten von 2. Die Knoten jeder Gruppe im Graphen G_j besitzen also jeweils billige Kanten zu genau einem der Startknoten des Graphen G_{j+1}.

- Alle weiteren neu hinzugefügten Kanten erhalten Kosten von 2.
- Nach Beobachtung 3.2 ist auch dieser Graph metrisch.

Zu Beginn sind alle Server auf Knoten von U_1 positioniert und zwar wieder analog zum Beweis von Satz 4.9. Jetzt beschreiben wir die n Anfragen für Instanzen aus der Klasse $\mathcal{I}$.

- Die ersten k Anfragen sind aus der Menge W_1 und korrespondieren mit einer Permutation π_1 wie im Beweis von Satz 4.9.
- Danach werden alle k Knoten aus U_2 jeweils einmal angefragt.
- Als nächstes werden k Knoten aus W_2 angefragt, die wiederum einer Permutation π_2 entsprechen.
- Dieses Vorgehen wiederholen wir, bis am Ende die Knoten aus W_t entsprechend einer Permutation π_t angefragt werden.

Wir können die Eingabe somit beschreiben durch

$$\underset{G_1}{\pi_1}, \underbrace{u_{2,1}, \ldots, u_{2,k}, \pi_2}_{G_2}, \underbrace{u_{3,1}, \ldots, u_{3,k}, \pi_3}_{G_3}, \ldots, \underbrace{u_{t,1}, \ldots, u_{t,k}, \pi_t}_{G_t},$$

wobei, für $1 \leq j \leq t$, π_j eine Permutation der Knotenindizes von W_j repräsentiert. Wir behaupten jetzt, dass ein optimaler Algorithmus Opt für eine Instanz aus $\mathcal{I}$ eindeutig wie folgt bestimmt ist.

- Wenn Opt, für $1 \leq j \leq t$, einen Server von U_j nach W_j bewegt, so tut er dies immer anhand der entsprechenden Permutation π_j.
- Wenn Opt, für $1 \leq j \leq t-1$, einen Server von W_j nach U_{j+1} bewegt, so benutzt er, wenn der angefragte Knoten $u_{j+1,i+1}$ ist, genau den Server, der sich in der Gruppe $\overline{G}_{j,i}$ befindet für $0 \leq i \leq k-1$.

Wir halten fest, dass Opt in jedem Zeitschritt Kosten von 1 hat. Sei I eine fixe Instanz aus $\mathcal{I}$. Jetzt zeigen wir, dass Opt tatsächlich ein eindeutiger optimaler Algorithmus für I ist. Zu diesem Zweck nehmen wir an, es existiere ein weiterer Algorithmus Alg, der ebenfalls optimal ist, aber eine andere Lösung als Opt berechnet. Wir dürfen folgende Annahmen machen.

- Wir gehen nach Satz 3.5 davon aus, dass Opt und Alg träge Algorithmen sind.
- Beachten Sie, dass in I kein Knoten mehrmals angefragt wird.
- Deswegen hat kein träger Algorithmus in irgendeinem Zeitschritt Kosten von 0.
- Da Opt in jedem Zeitschritt Kosten von 1 hat, kann somit kein Online-Algorithmus, der in mindestens einem Zeitschritt Kosten von 2 hat, optimal sein.

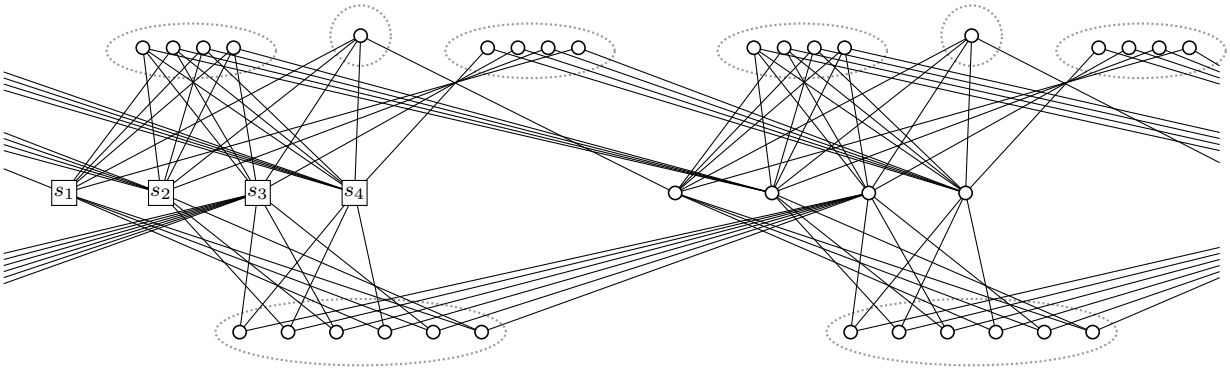


Abbildung 13.

Sei i der erste Zeitschritt, in dem OPT und ALG einen unterschiedlichen Server benutzen. Wir unterscheiden zwei Fälle bezüglich der von OPT berechneten Ausgabe und beweisen, dass ALG in einem folgenden Zeitschritt Kosten von 2 hat.

Betrachten wir als erstes den Fall, dass OPT im Zeitschritt i einen Server von U_j nach W_j, $1 \leq j \leq t$, bewegt.

- Wir wissen bereits aus dem Beweis von Satz 4.9, dass, wenn ALG einen anderen Server aus U_j benutzt als OPT, dies zu einmaligen Kosten von 2 führt und zwar spätestens nach der letzten Anfrage aus W_j

- Benutzt ALG einen Server, der bereits in W_j positioniert ist, führt dies unmittelbar zu Kosten von 2 in diesem Zeitschritt.

- Da OPT und ALG bis zu diesem Zeitpunkt dieselbe Lösung berechnet haben, sind ferner keine Server in $U_{j'}$ oder $W_{j'}$, $1 \leq j' \leq t$, $j' \neq j$, positioniert.

Betrachten wir als nächstes den Fall, dass OPT im Zeitschritt i einen Server von W_j nach U_{j+1}, $1 \leq j \leq t-1$ bewegt.

- Vor der ersten Anfrage aus U_{j+1} haben OPT und ALG alle Server auf gleiche Weise auf die Gruppen $\overline{G}_{j,i}$, $1 \leq i \leq l-1$ verteilt; und dabei genau denselben Server in der derselben Gruppe positioniert.

- Es folgt sofort, dass ALG Kosten von 2 in diesem Zeitschritt hat, wenn er nicht denselben Server wie OPT verwendet, da jeder Knoten aus U_{j+1} genau mit einer Gruppe mit billigen Kanten verbunden ist.

- Einen Server, der bereits in U_{j+1} positioniert ist, zu nehmen, führt ebenfalls direkt zu Kosten von 2.

- In diesem Fall können keine Server in $U_{j'}$ oder $W_{j''}$, $1 \leq j' \leq t-1$, $j' \neq j+1$, $1 \leq j'' \leq t$, $j'' \neq t$, positioniert sein.

Da es also eine eindeutige Lösung für jede Instanz in $\mathcal{I}$ gibt, die entsprechend der Permutationen $\pi_1, \ldots, \pi_t$ die Server auswählt, können wir wieder analog zum Beweis von Satz 4.9 argumentieren und folgern, dass

$$t \log_2(k!) = \frac{n+k}{2k} \log_2(k!) \geq \frac{n+k}{2}(\log_2 k - \log_2 \mathrm{e})$$

Advice-Bits notwendig sind, um optimal zu sein. □

4.3 Verwendete und weiterführende Literatur

Orakel werden in der theoretischen Informatik oft benutzt, um die Performanz von Algorithmen zu untersuchen. Hier sei beispielsweise auf die Komplexitätstheorie von Offline-Algorithmen verwiesen [1] oder diverse Anwendungen in der Kryptologie [32, 33, 37]. Online-Algorithmen mit Advice wurden von Dobrev et al. [10] eingeführt und unter anderem auf Paging angewendet. Ein Kritikpunkt an diesem ersten Modell war die Tatsache, dass der Advice in einer Art und Weise kodiert wurde, die es ermöglichte, mehr Informationen zu transportieren als mit der entsprechenden Anzahl an Bits eigentlich möglich ist.

Das hier verwendete Modell wurde von Böckenhauer et al. [8] und Hromkovič et al. [17] eingeführt. Dobrev et al. [10] untersuchten die Advice-Komplexität von Paging als erste in dem von ihnen aufgestellten Modell. Die hier präsentierten Resultate stammen von Böckenhauer et al. [8], wobei die in Satz 4.7 vorgestellte obere Schranke für die Advice-Komplexität eines optimalen Online-Algorithmus mit Advice bereits von Dobrev et al. beobachtet wurde. Emek et al. [11] haben zuerst die Advice-Komplexität von k-Server untersucht. Die hier vorgestellten Ergebnisse stammen wiederum von Böckenhauer et at. [7]. Komm und Královič [26] haben den Zusammenhang zwischen einer konstanten Advice-Komplexität und Barely-Random-Algorithmen erstmals explizit untersucht.

5 Das Online-Rucksack-Problem

Wir betrachten in diesem Kapitel ein Maximierungsproblem, dessen Offline-Version wir bereits kennengelernt haben. Wie bereits beschrieben, sprechen wir hier im Folgenden nicht von den Kosten eines Online-Algorithmus, die minimiert werden sollen, sondern vom Gewinn, den wir maximieren sollen (siehe Definition 1.3). Den Gewinn einer Lösung $\textsc{Alg}(I)$ eines Online-Algorithmus $\textsc{Alg}$ auf einer Instanz I bezeichnen wir mit $\text{gain}(\textsc{Alg}(I))$.

Beim **Online-Rucksack-Problem** (wir reden zukünftig einfach vom Rucksack-Problem) werden einem Online-Algorithmus in aufeinanderfolgenden Zeitschritten Objekte angeboten, die er in einen Rucksack packen kann, wobei eine Gewichtsschranke des Rucksacks eingehalten werden muss. Ziel ist, den Wert aller eingepackten Objekte zu maximieren. Wir betrachten hier die einfache Version des Rucksack-Problems, in der alle Objekte einen Parameter, ihr «Gewicht», besitzen (Gewicht und Wert sind hier also dasselbe). Anders als bei der Offline-Version sind die Gewichte bzw. Werte keine natürlichen, sondern reelle Zahlen. Die Kapazität des Rucksacks ist dem Online-Algorithmus vor der Bearbeitung bekannt. Wir setzen sie deswegen ohne Beschränkung der Allgemeinheit auf 1 und definieren das Problem formal wie folgt.

Definition 5.1 (Rucksack-Problem). *Eine Eingabe $I = (w_1, \ldots, w_n)$ für das Rucksack-Problem ist eine Folge von n Objekten, die wir mit ihrem Gewicht w_i, $1 \leq i \leq n$, $0 \leq w_i \leq 1$, identifizieren. Eine zulässige Lösung ist eine Menge von Indizes $S \subseteq \{1, \ldots, n\}$, sodass*

$$\sum_{i \in S} w_i \leq 1 \, .$$

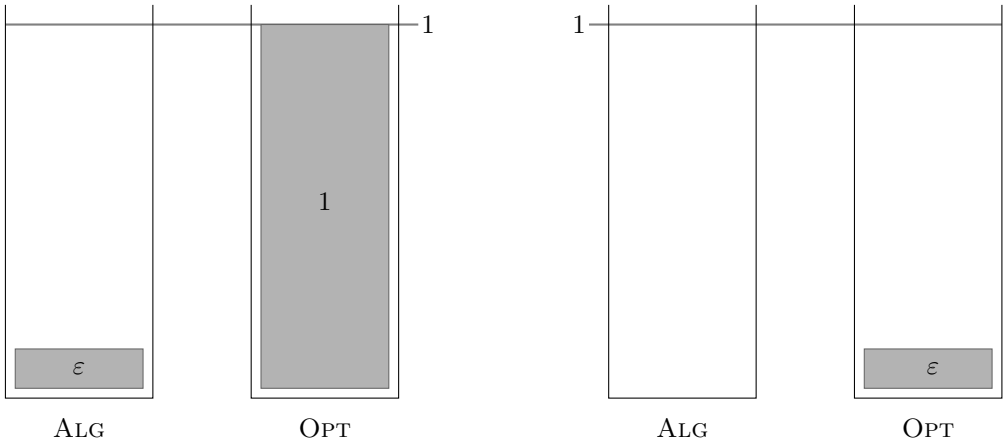
Das Ziel ist, diese Summe zu maximieren. Die Objekte werden hierbei nacheinander, eines pro Zeitschritt, angeboten. Ein Online-Algorithmus muss nach jedem angebotenen Objekt entscheiden, ob es in den Rucksack gepackt wird oder nicht, und diese Entscheidung kann nicht rückgängig gemacht werden.

Da der Wert jeder Lösung nach oben durch die Rucksackgrösse 1 beschränkt ist, betrachten wir für dieses Problem lediglich strikte kompetitive Faktoren. Beachten Sie, dass die Anzahl der angebotenen Objekte einem Online-Algorithmus jedoch nicht bekannt ist. Die Algorithmen für das Offline-Rucksack-Problem, die wir kennengelernt haben, haben die Objekte zum Teil der Grösse nach sortiert, um die Eingabe bearbeiten zu können. Eine solche Strategie können wir in einer Online-Situation natürlich nicht mehr verfolgen, denn wir wissen zu einem gegebenen Zeitpunkt nicht, welche Objekte noch angeboten werden.

5.1 Deterministische Online-Algorithmen

Aus diesen Überlegungen folgt die nächste Behauptung, die aussagt, dass jeder Online-Algorithmus einen beliebig schlechten kompetitiven Faktor haben kann.

Satz 5.2. *Jeder deterministische Online-Algorithmus für das Rucksack-Problem hat einen beliebig grossen strikten kompetitiven Faktor.*



(a) Fall 1 (b) Fall 2

Abbildung 14.

Beweis. Wir konstruieren einen Gegenspieler, der zunächst ein Objekt w mit dem Gewicht $\varepsilon > 0$ anbietet.

- Falls ein deterministischer Online-Algorithmus ALG sich entscheidet, w zu akzeptieren, wird ihm danach ein Objekt mit einem Gewicht von 1 angeboten, das offensichtlich nicht mehr in den Rucksack passt.
- Entscheidet sich ALG auf der anderen Seite, w nicht zu akzeptieren, wird ihm kein weiteres Objekt mehr angeboten.

Im ersten Fall ist der kompetitive Faktor von ALG genau $1/\varepsilon$, im zweiten sogar unendlich gross (siehe Abbildung 14); genau genommen ist der kompetitive Faktor in diesem Fall gar nicht definiert. □

Jetzt untersuchen wir einen einfachen Greedy-Algorithmus GREEDY, der jedes Objekt in den Rucksack packt, wenn noch Platz ist. Wie wir gerade gesehen haben, ist auch diese Strategie nicht erfolgreich. Der wesentliche Punkt ist aber, dass GREEDY auf gewissen Eingaben sehr wohl eine gute Ausgabequalität erreicht, wie der folgende Satz zeigt.

Satz 5.3. *Für jede Instanz des Rucksack-Problems, bei der alle Objekte ein Gewicht von höchstens β haben, erreicht* GREEDY *einen Gewinn von mehr als $1 - \beta$ oder ist optimal.*

Beweis. Wir unterscheiden zwei Fälle bezüglich des Gesamtgewichts der Objekte in der Eingabe.

- Wenn dieses Gesamtgewicht kleiner als 1 ist, ist GREEDY offensichtlich optimal, da er jedes Objekt in den Rucksack packt.
- Ist das Gesamtgewicht grösser, so muss der Platz, der in der von GREEDY berechneten Lösung nicht gefüllt wurde, kleiner als β sein. Andernfalls wäre für ein weiteres Objekt Platz. Also ist der Rucksack zu einem Anteil gefüllt, der grösser ist als $1 - \beta$.

Somit ist der Gewinn von GREEDY mindestens $1 - \beta$ oder er ist optimal. □

Für eine Teilklasse von Instanzen ist GREEDY sogar optimal.

Satz 5.4. *Für jede Instanz des Rucksack-Problems, für die eine optimale Lösung höchstens einen Gewinn von* $1/2$ *besitzt, ist* GREEDY *optimal.*

Beweis. Wenn $\text{gain}(\text{OPT}(I)) \leq 1/2$ für eine Instanz I gilt, so ist offensichtlich auch $\text{gain}(\text{GREEDY}(I)) \leq 1/2$. Falls ferner $\text{gain}(\text{GREEDY}(I)) < \text{gain}(\text{OPT}(I))$ ist, so muss dies daran liegen, dass GREEDY ein Objekt nicht mit in den Rucksack gepackt hat, das ein Gewicht hat, welches grösser ist als $1/2$. Dies ist aber ein direkter Widerspruch. □

Wir werden diese Tatsache im weiteren Verlauf benutzen, um Online-Algorithmen mit Advice zu konstruieren, die eine gute Ausgabequalität erzielen.

5.2 Online-Algorithmen mit Advice

Wir untersuchen jetzt die Advice-Komplexität des Rucksack-Problems. Zunächst betrachten wir die Anzahl an Advice-Bits, die ausreichend und nötig ist, um eine optimale Lösung zu konstruieren. Wir beginnen mit einer einfachen oberen Schranke.

Satz 5.5. *Es existiert ein optimaler Online-Algorithmus mit Advice für das Rucksack-Problem, der* n *Advice-Bits benutzt.*

Beweis. Für jedes der n Objekte liest dieser Algorithmus ein Bit vom Advice-Band, das ihm sagt, ob das aktuelle Objekt Teil einer beliebigen, aber fixen optimalen Lösung ist oder nicht. Folglich gibt er genau diese optimale Lösung aus. □

Spannender wird es, wenn wir nun zeigen, dass diese Schranke scharf (bis auf ein Bit) ist.

Satz 5.6. *Jeder Online-Algorithmus mit Advice für das Rucksack-Problem muss mindestens* $n - 1$ *Advice-Bits benutzen, um optimal sein zu können.*

Beweis. Wir konstruieren eine Klasse von Eingaben $\mathcal{I}$, die wie folgt aussieht. Für jedes n betrachten wir die Eingabe

$$\frac{1}{2}, \frac{1}{4}, \ldots, \frac{1}{2^{n-1}}, w_b \ ,$$

wobei w_b definiert ist als

$$w_b := 1 - \sum_{i=1}^{n-1} b_i 2^{-i} \ ,$$

für einen Binärstring $b = b_1 \ldots b_{n-1}$. Für $n = 8$ und $b = 1101101$ erhalten wir also beispielsweise

$$\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{64}, \frac{1}{128}, w_b = 1 - \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{16} + \frac{1}{32} + \frac{1}{128}\right) = \frac{109}{128} \ ,$$

was einer optimalen Lösung mit Kosten 1 entspricht. Jetzt betrachten wir die ersten $n-1$ angebotenen Objekte in der Eingabe, die für jedes n dieselben sind. Zwei verschiedene Teilmengen dieser Objekte führen offensichtlich zu zwei verschiedenen Teilgewichten. Für jeden Wert von w_b beziehungsweise b existiert also eine eindeutige optimale Lösung. Damit ein Online-Algorithmus diese ausgeben kann, muss er in den ersten $n-1$ Zeitschritten die richtigen Objekte akzeptieren, sodass im letzten Zeitschritt das Objekt w_b schliesslich den Rucksack zu 1 auffüllt.

Falls ein Algorithmus weniger als $n-1$ Advice-Bits benutzt, kann er zwischen weniger als 2^{n-1} Instanzen aus $\mathcal{I}$ unterscheiden. Nach dem Schubfachprinzip verhält er sich somit gleich für zwei verschiedene Instanzen und kann nicht optimal für beide sein. □

Wir brauchen also eine grosse Menge an Zusatzinformationen, um optimal zu sein. Ohne Zusatzinformationen sind wir auf der anderen Seite beliebig schlecht. Aber was geschieht zwischen diesen Extremwerten? Überraschenderweise ermöglicht uns ein einziges Bit an Information bereits 2-kompetitiv zu sein. Betrachten wir den Online-Algorithmus KPone, der am Anfang der Berechnung ein Advice-Bit benutzt, um zu entscheiden, ob er entweder eine einfache Greedy-Strategie realisiert, oder auf ein Objekt wartet, das ein Gewicht hat, das grösser als $1/2$ ist.

Satz 5.7. *Der Online-Algorithmus KPone mit Advice für das Rucksack-Problem ist strikt 2-kompetitiv.*

Beweis. Wir unterscheiden wieder zwei Fälle, diesmal davon abhängig, ob ein Objekt mit einem Gewicht $> 1/2$ angeboten wird oder nicht.

- Existiert ein solches Objekt, wird dies KPone mit einem Bit mitgeteilt. Das erste Objekt mit einem ausreichenden Gewicht wird dann in den Rucksack gepackt. Da ein optimaler Algorithmus Opt höchstens einen Gewinn von 1 erzielen kann, ist der Gewinn von KPone also mindestens halb so gross wie der optimale.
- Existiert ein solches Objekt nicht, so folgt die Behauptung sofort nach Satz 5.3.

In beiden Fällen ist KPone also 2-kompetitiv. □

Ein Bit ist also alles, was uns fehlt, um aus einer beliebig schlechten Ausgabequalität eine 2-kompetitive zu machen. Es liegt nun nahe, zu fragen, wie viel uns weitere Bits helfen, was also beispielsweise passiert, wenn wir einem Online-Algorithmus mit Advice erlauben, 2 Bits anstatt einem zu benutzen. Auch hier können wir ein kontraintuitives Ergebnis zeigen; und zwar helfen weitere Bits nicht, solange wir keine logarithmische Anzahl benutzen.

Satz 5.8. *Sei $\varepsilon > 0$. Kein Online-Algorithmus mit Advice für das Rucksack-Problem, der weniger als $\log_2(n-1)$ Advice-Bits benutzt, kann besser als strikt $(2-\varepsilon)$-kompetitiv sein.*

Beweis. Sei im Folgenden

$$\delta := \frac{\varepsilon}{4-2\varepsilon}$$

und sei ALG ein Online-Algorithmus, der $b < \log_2(n-1)$ Advice-Bits benutzt. Wir konstruieren wieder eine Klasse $\mathcal{I}$ von Instanzen. Diese besteht aus Eingaben I_j, $1 \leq j \leq n-1$, die jeweils die Form

$$\frac{1}{2} + \delta^2, \frac{1}{2} + \delta^3, \ldots, \frac{1}{2} + \delta^{j+1}, \frac{1}{2} - \delta^{j+1}, \frac{1}{2} + \delta, \ldots, \frac{1}{2} + \delta$$

haben, wobei das Objekt mit dem Gewicht $1/2 + \delta$ am Ende $(n-j-1)$-mal angeboten wird. Wir sehen sofort, dass

$$|\mathcal{I}| = n - 1 = 2^{\log_2(n-1)} > 2^b$$

gilt, also existieren in $\mathcal{I}$ mehr Eingaben als es Advice-Strings gibt, die ALG benutzen kann. Wir können nun erneut mit dem Schubfachprinzip argumentieren, dass es mindestens zwei verschiedene Eingaben gibt, für die ALG denselben Advice liest.

- Um optimal für eine Eingabe I_j zu sein, muss ALG genau das j-te und das $(j+1)$-te Objekt in den Rucksack packen und diese Lösung ist eindeutig. Der Gewinn einer optimalen Lösung ist ferner immer 1.
- Seien $I_{j'}$ und $I_{j''}$ zwei verschiedene Eingaben aus $\mathcal{I}$, für die ALG denselben Advice liest, wobei ohne Beschränkung der Allgemeinheit $j' < j''$ gilt.
- Diese beiden Instanzen sind in den ersten j' Zeitschritten gleich und somit packt ALG das j'-te Objekt für beide entweder ein oder nicht.
- Packt ALG das j'-te Objekt ein, so kann er zwar optimal für $I_{j'}$ sein, allerdings kann er kein weiteres Objekt mehr für $I_{j''}$ einpacken, da

  $$\frac{1}{2} + \delta^{j'} + \frac{1}{2} - \delta^{j''} > 1$$

  gilt.
- Packt ALG das j'-te Objekt wiederum nicht ein, so kann er optimal für $I_{j''}$ sein, jedoch nicht für $I_{j'}$.
- In beiden Fällen hat ALG also höchstens einen Gewinn von $1/2 + \delta$.

Dies bedeutet wiederum, dass der erreichte kompetitive Faktor nicht besser als

$$\frac{1}{\frac{1}{2} + \delta} = 2 - \varepsilon$$

ist. □

Als nächstes beweisen wir eine asymptotisch scharfe obere Schranke, die zeigt, dass wir mit einer logarithmischen Anzahl an Advice-Bits beliebig nahe an eine optimale Lösung herankommen können. Mit $\mathcal{O}(\log n)$ Advice-Bits sind wir also «fast» optimal. Vorher lernen wir aber noch eine Technik kennen, die wir zum Kodieren der Zusatzinformationen benötigen.

5.3 Selbstbeschränkte Bitstrings

Fangen wir zunächst mit einer einfachen Beobachtung an. Wie wir wissen, können wir mit m Bits 2^m verschiedene Zahlen darstellen. Oftmals wollen wir die Anzahl an benötigten Bits mit der maximal zu kodierenden Zahl n abschätzen; wir nehmen an, n sei eine natürliche Zahl, die grösser als 2 ist.

- Um eine Zahl a, $0 \leq a \leq n$, binär zu kodieren, brauchen wir $\lceil \log_2(n+1) \rceil$ Bits, da a potentiell $n+1$ Werte annehmen kann.
- Ist a allerdings nie 0, so brauchen wir lediglich $\lceil \log_2 n \rceil$ Bits; anstatt a können wir in diesem Fall $a-1$ auf das Advice-Band schreiben.

Bislang konnten die konstruierten Online-Algorithmen mit Advice selbstständig erkennen, wie sie den Advice zu benutzen hatten. Im Folgenden müssen wir allerdings mehrere Zahlen auf dem Advice-Band kodieren, über deren Grösse von Beginn an nichts bekannt ist. Das verwendete Modell erlaubt es uns nicht, ein Symbol zu verwenden, das wir als Trenner benutzen könnten, da der String binär sein muss. Ein besonderes Muster, wie etwa 111, als Trenner zu verwenden, ist offensichtlich auch nicht möglich, da es Teil des kodierten Advice sein könnte.

Die Idee ist, den String in einer selbstbeschränkten (auf englisch «self-delimiting») Art zu kodieren. Nehmen wir an, wir wollen zwei Zahlen 42 und 9 auf das Advice-Band schreiben. Kodieren wir diese Zahlen binär, so erhalten wir ein Präfix

1	0	1	0	1	0	1	0	0	1	...

42 9

auf dem Advice-Band. Allerdings ist die Kodierung nicht eindeutig; wir könnten den Bitstring auch als

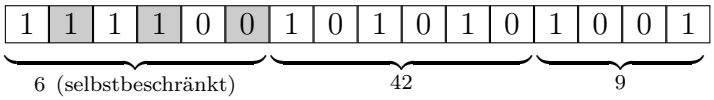
1	0	1	0	1	0	1	0	0	1	...

10 41

interpretieren. Eine Lösung des Problems ist, dem Algorithmus mitzuteilen, wie viele Bits zur Kodierung der ersten Zahl, nennen wir sie wieder a, gehören. Wenn a wieder eine Zahl zwischen 0 und n ist, brauchen wir maximal $\lceil \log_2(n+1) \rceil$ Bits, um a zu kommunizieren; ausserdem können wir mit zusätzlichen $\lceil \log_2 \lceil \log_2(n+1) \rceil \rceil$ Bits angeben, wie gross $\lceil \log_2(n+1) \rceil$ ist, wie viel Platz also gebraucht wird, um a zu kodieren. Wegen $n > 2$ ist $\lceil \log_2(n+1) \rceil$ dabei nie 0 (auch dann, wenn a nie 0 ist).

Allerdings stehen wir nun wieder vor einem ähnlichen Problem, denn jetzt muss der Algorithmus wissen, wo diese ersten $\lceil \log_2 \lceil \log_2(n+1) \rceil \rceil$ Bits aufhören und die Kodierung von a anfängt. Der Trick ist, dass wir dem Algorithmus nach einem festen Muster mitteilen, wo die ersten $\lceil \log_2 \lceil \log_2(n+1) \rceil \rceil$ enden, indem wir hinter jedes Bit, das noch dazu gehört, eine 1 schreiben und hinter das letzte eine 0. Somit brauchen wir $2\lceil \log_2 \lceil \log_2(n+1) \rceil \rceil$ Bits am Anfang und anschliessend ist dem Algorithmus bekannt, wie viele weitere Bits er lesen muss, um a zu erhalten. Möchten wir also wieder die Zahlen 42 und 9 kodieren,

so stellen wir zunächst fest, dass wir $\lceil \log_2(42+1) \rceil = 6$ Bits brauchen, um 42 binär zu kodieren. Dementsprechend schreiben wir zuerst 6 selbstbeschränkt auf das Band, dann 42 und hiernach 9 und erhalten



als Präfix. Natürlich hätten wir diese Kodierung auch direkt auf die Zahl a anwenden können, aber dann würden wir für deren Darstellung $2\lceil \log_2(n+1) \rceil$ Bits benötigen. Wir tauschen also die multiplikative Konstante 2 gegen einen additiven Term, der asymptotisch kleiner ist als der restliche Inhalt des Bandes.

Diese Ideen benutzen wir jetzt, um den folgenden Satz zu beweisen, der eine logarithmische obere Schranke für die Anzahl an Advice-Bits angibt, um das Rucksackproblem fast optimal zu lösen.

Satz 5.9. *Sei $\varepsilon > 0$. Es existiert ein Online-Algorithmus KPlog mit Advice für das Rucksack-Problem, der einen strikten kompetitiven Faktor von $1+\varepsilon$ erreicht und höchstens*

$$\left\lceil \frac{3\varepsilon+3}{\varepsilon} \right\rceil \cdot \lceil \log_2 n \rceil + 2 \cdot \left\lceil \log_2 \left(\left\lceil \frac{3\varepsilon+3}{\varepsilon} \right\rceil + 1 \right) \right\rceil + 2 \cdot \lceil \log_2 \lceil \log_2 n \rceil \rceil + 1$$

Advice-Bits verwendet.

Beweis. Sei im Folgenden

$$\delta := \frac{\varepsilon}{3+3\varepsilon} .$$

Da KPlog die Konstante ε kennt, kennt er auch δ. Sei I eine beliebige Eingabe für das Rucksack-Problem. Wir machen wieder eine Fallunterscheidung, diesmal bezüglich der Objekte, die ein beliebiger aber fixer optimaler Algorithmus Opt in den Rucksack packt.

- Nehmen wir zunächst an, es existiere kein Objekt mit einem Gewicht grösser als δ in der von Opt berechneten Lösung; dies kann mit einem Bit kommuniziert werden.
- Dann realisiert KPlog die Greedy-Strategie und hat nach Satz 5.3 einen kompetitiven Faktor von höchstens

$$\frac{1}{1-\delta} = 1 + \frac{\delta}{1-\delta} = 1 + \frac{\varepsilon}{3+2\varepsilon} \leq 1+\varepsilon .$$

Nehmen wir jetzt also an, dies sei nicht der Fall und somit gebe es mindestens ein Objekt mit einem Gewicht von mindestens δ.

- Die von Opt berechnete Lösung besteht also aus zwei disjunkten Teilmengen S_1 und S_2, wobei S_1 alle Objekte mit einem Gewicht von mindestens δ enthält (diese Objekte nennen wir «schwer») und S_2 solche Objekte, die ein kleineres Gewicht besitzen (die sogenannten «leichten» Objekte).

- Sei $|S_1| = i$.
- Sei s_1 die Summe aller Gewichte der Objekte in S_1 und s_2 analog definiert für S_2.
- Die Indizes (bezüglich I) der schweren Objekte werden auf das Advice-Band geschrieben, sodass KPlog alle Objekte aus S_1 einpackt, wenn sie angeboten werden.

Jetzt müssen wir uns noch um die leichten Objekte kümmern, aber wir haben nicht genug Advice, um ihre Position ebenfalls genau zu kodieren. Die Idee ist, eine untere Schranke für den Teil des Rucksacks anzugeben, der von Opt mit diesen Objekten gepackt wird und KPlog leichte Objekte mit der Greedy-Strategie akzeptieren zu lassen, solange ihr Gesamtgewicht nicht über die Schranke hinausgeht.

- Das Orakel kodiert eine Zahl k auf dem Advice-Band, sodass

  $$k\delta \leq s_2 < (k+1)\delta$$

  gilt. Offensichtlich ist $k \leq 1/\delta$, da $s_2 \leq 1$ ist.
- KPlog berechnet nun $k\delta$ und kennt somit eine untere Schranke für den Teil des Rucksacks, den Opt mit leichten Objekten füllt.
- Jedes leichte Objekt wird dann in den Rucksack gepackt, solange die Summe der Gewichte der bereits gepackten leichten Objekte kleiner als $k\delta$ ist.

Wir können eine weitere Fallunterscheidung bezüglich der von Opt berechneten Lösung machen.

- Wenn Opt keine leichten Objekte in den Rucksack packt, so ist KPlog optimal, da alle schweren Objekte eingepackt werden. Wir gehen im Folgenden also davon aus, dass Opt mindestens ein leichtes Objekt in den Rucksack packt.
- Wenn Opt einen Gewinn von weniger als $1 - \delta$ besitzt, so folgt wiederum, dass Opt alle leichten Objekte, die angeboten werden, in den Rucksack packt. In dem Fall schreibt das Orakel $k = \lceil 1/\delta \rceil$ auf das Advice-Band und KPlog ist wieder optimal.
- Also beschränken wir den Gewinn von Opt von unten mit $1 - \delta$.

Jetzt schätzen wir ab, wie viel wir durch die Tatsache, dass wir leichte Objekte nicht exakt kodieren, maximal verlieren können.

- Wir können uns vorstellen, dass KPlog ein Greedy-Algorithmus ist, der mit einer Rucksackgrösse von $k\delta$ arbeitet, wobei jedes angebotene Objekt eine Grösse von höchstens δ besitzt.
- Wieder benutzen wir Satz 5.3, aus dem folgt, dass KPlog auf dieser Teilinstanz einen Gewinn von mindestens $k\delta - \delta \geq s_2 - 2\delta$ besitzt oder optimal ist.

Insgesamt hat KPlog also einen Gewinn von mindestens $s_1+s_2-2\delta$ und Opt einen Gewinn von s_1+s_2. Für den kompetitiven Faktor folgt damit, wegen $1-\delta \leq \text{gain}(\text{Opt}(I)) \leq 1$, unmittelbar

$$\frac{s_1+s_2}{s_1+s_2-2\delta} \leq \frac{1}{1-3\delta} = 1+\frac{3\delta}{1-3\delta} = 1+\varepsilon\ .$$

Zuletzt schätzen wir noch die Anzahl der verwendeten Advice-Bits ab.

- Wir benötigen als erstes ein Bit, das angibt, ob schwere Objekte Teil der optimalen Lösung sind.
- Wie bereits erwähnt, ist $k \leq 1/\delta$, da $s_2 \leq 1$ ist. Also ist k eine natürliche Zahl zwischen 0 und $\lceil 1/\delta \rceil$, die mit $\lceil \log_2(\lceil 1/\delta \rceil + 1) \rceil$ Bits kodiert werden kann.
- Wir stellen ausserdem fest, dass keine Lösung mehr als $1/\delta$ viele schwere Objekte akzeptieren kann, da ihr Gesamtgewinn sonst grösser als 1 wäre. Da es mindestens ein schweres Objekt gibt, i also nicht 0 werden kann, lässt sich i mit $\lceil \log_2 \lceil 1/\delta \rceil \rceil$ vielen Advice-Bits kodieren.
- Beachten Sie, dass i und k nicht selbstbeschränkt kodiert werden müssen, da KPlog den Wert δ kennt und somit obere Schranken für diese Werte ausrechnen kann. Die tatsächlichen Darstellungen der beiden Zahlen werden vom Orakel durch führende Nullen auf die richtige Länge gebracht.
- Für jedes der i schweren Objekte muss dessen Index mit jeweils $\lceil \log_2 n \rceil$ Advice-Bits auf das Advice-Band geschrieben werden.
- Um den Advice dekodieren zu können, muss KPlog die Zahl $\lceil \log_2 n \rceil$ kennen. Nach den obigen Beobachtungen benötigen wir hierfür $2\lceil \log_2 \lceil \log_2 n \rceil \rceil$ zusätzliche Bits.

Insgesamt ist die Anzahl an Advice-Bits also nach oben durch

$$\begin{aligned} &1 + i \cdot \lceil \log_2 n \rceil + 2 \cdot \left\lceil \log_2 \left(\left\lceil \frac{1}{\delta} \right\rceil + 1 \right) \right\rceil + 2 \cdot \lceil \log_2 \lceil \log_2 n \rceil \rceil \\ &\quad \leq 1 + \left\lceil \frac{3\varepsilon+3}{\varepsilon} \right\rceil \cdot \lceil \log_2 n \rceil + 2 \cdot \left\lceil \log_2 \left(\left\lceil \frac{3\varepsilon+3}{\varepsilon} \right\rceil + 1 \right) \right\rceil + 2 \cdot \lceil \log_2 \lceil \log_2 n \rceil \rceil \end{aligned}$$

beschränkt, was in $\mathcal{O}(\log n)$ ist. □

Wir fassen die bisherigen Ergebnisse kurz zusammen. Ohne Advice ist jeder Online-Algorithmus beliebig schlecht; jedoch erlaubt bereits ein Advice-Bit einen kompetitiven Faktor von 2. Jedes weitere Advice-Bit hilft wiederum nicht, bis eine Anzahl erreicht wird, die logarithmisch in der Eingabelänge ist. Ist dieser Schwellenwert (asymptotisch) erreicht, so lässt sich ein Online-Algorithmus mit Advice konstruieren, der fast optimal ist. Um allerdings tatsächliche Optimalität zu erreichen, muss die Anzahl an Advice-Bits wiederum exponentiell vergrössert werden (siehe Abbildung 15 für eine schematische Darstellung).

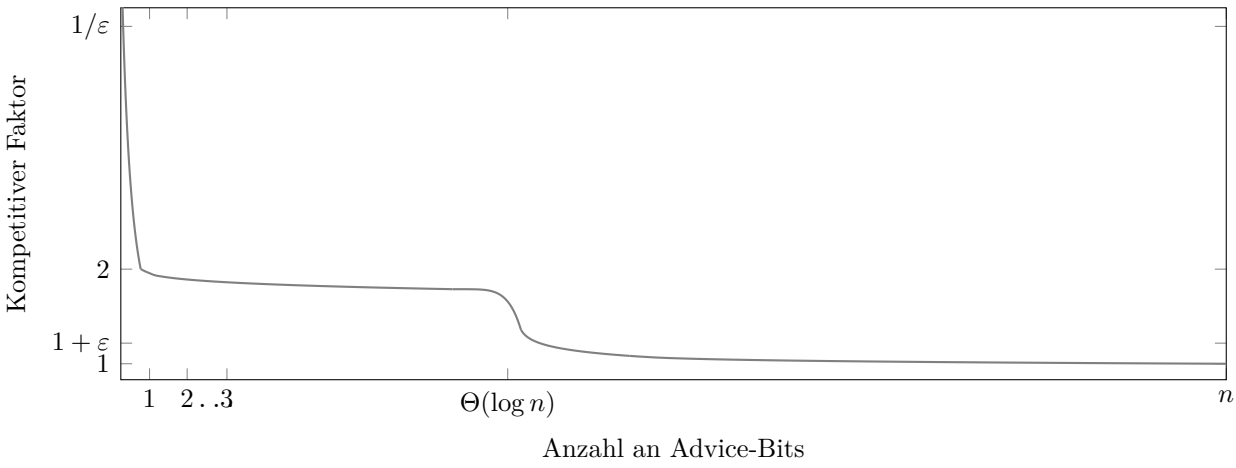


Abbildung 15.

5.4 Ein Barely-Random-Algorithmus

Wir wissen jetzt, dass ein Advice-Bit für das Rucksack-Problem sehr mächtig ist. Nun wollen wir uns die Frage stellen, was ein Zufallsbit bewirken kann. Betrachten wir zunächst den Barely-Random-Algorithmus RKPone′, der analog zum oben besprochenen Online-Algorithmus KPone mit Advice funktioniert, das eine Bit aber nicht von einem Orakel mitgeteilt bekommt, sondern rät.

Satz 5.10. *Der Barely-Random-Algorithmus RKPone′ für das Rucksack-Problem ist strikt 4-kompetitiv im Erwartungswert.*

Beweis. Wir machen wieder dieselbe Fallunterscheidung wie im Beweis von Satz 5.7.

- Existiert ein Objekt in der Eingabe, das ein Gewicht von mehr als $1/2$ besitzt, so wird dieses mit einer Wahrscheinlichkeit von $1/2$ genommen, da RKPone′ auf es wartet. Entscheidet sich der Algorithmus hingegen, eine Greedy-Strategie zu verfolgen, packt er im schlimmsten Fall ein Objekt mit sehr kleinem Gewicht $\varepsilon > 0$ ein. Der erwartete Gewinn ist also grösser als $1/2 \cdot 1/2 = 1/4$.

- Existiert kein solches Objekt, so ist der Gewinn von RKPone′ offensichtlich 0 mit einer Wahrscheinlichkeit von $1/2$.

  Ist das Gesamtgewicht aller Objekte in der Eingabe höchstens $1/2$, so ist der Algorithmus mit einer Wahrscheinlichkeit von $1/2$ optimal nach Satz 5.3 und somit 2-kompetitiv.

  Ist der Gesamtgewinn grösser als $1/2$, so ist sein Gewinn nach Satz 5.3 mit Wahrscheinlichkeit von $1/2$ mindestens $1/2$. Somit folgt auch in diesem Fall eine untere Schranke von $1/4$ für den Gewinn von RKPone′.

Da die optimale Lösung höchstens einen Gewinn von 1 hat, folgt sofort, dass der kompetitive Faktor nicht schlechter als 4 ist. □

Für diesen Algorithmus ist diese Schranke dicht.

Satz 5.11. *Der Barely-Random-Algorithmus RKP*ONE′ *für das Rucksack-Problem kann nicht besser als strikt* 4*-kompetitiv im Erwartungswert sein.*

Beweis. Sei im Folgenden $\varepsilon < 1/6$ und

$$\delta := \frac{\varepsilon}{4(6-\varepsilon)} .$$

Wir betrachten eine Eingabe, die aus drei Objekten

$$\frac{1}{2} - \delta, 3\delta, \frac{1}{2} - \delta$$

besteht. Wieder unterscheiden wir zwei Fälle.

- Eine Greedy-Strategie akzeptiert die ersten beiden Objekte und hat einen Gewinn von $1/2 + 2\delta$.
- Da kein Objekt mit einem Gewicht von mehr als $1/2$ existiert, führt die Strategie, auf ein solches Objekt zu warten, zu einem Gewinn von 0.

Es folgt, dass der kompetitive Faktor von RKPONE′ nicht besser als

$$\frac{1-2\delta}{\frac{1}{2}\left(\frac{1}{2}+2\delta\right)+\frac{1}{2}\cdot 0} = 4 \cdot \frac{1-2\delta}{1+4\delta} = 4-\varepsilon$$

sein kann. □

Intuitiv scheint uns dieses Ergebnis logisch. Ein Advice-Bit, das uns immer sagt, was die beste Strategie ist, ist doppelt so mächtig wie ein Zufallsbit, das nur mit einer Wahrscheinlichkeit von $1/2$ eine solche Strategie auswählt. Umso erstaunlicher ist das folgende Ergebnis. Hierzu betrachten wir den Online-Algorithmus mit Advice RKPONE, der uniform einen deterministischen Online-Algorithmus aus strat(RKPONE) = $\{\mathit{Greedy}_1, \mathit{Greedy}_2\}$ wählt. Greedy_1 realisiert hierbei wieder eine einfache Greedy-Strategie; Greedy_2 simuliert zu Beginn hingegen Greedy_1, ohne Objekte zu akzeptieren. Sobald er allerdings feststellt, dass das Objekt, das im aktuellen Zeitschritt angeboten wird, nicht mehr in den von Greedy_1 gepackten Rucksack passt, akzeptiert er dieses Objekt und realisiert eine Greedy-Strategie von diesem Zeitschritt an.

Satz 5.12. *Der Barely-Random-Algorithmus RKP*ONE *für das Rucksack-Problem ist strikt* 2*-kompetitiv im Erwartungswert.*

Beweis. Wir unterscheiden diesmal zwei Fälle bezüglich des Gesamtgewichts der Objekte in der Eingabe I.

- Falls das Gesamtgewicht aller angebotenen Objekte in der Eingabe nicht grösser als 1 ist, so ist Greedy_1 optimal, während Greedy_2 einen Gewinn von 0 erzielt. Der erwartete Gewinn ist also genau $1/2\,\mathrm{gain}(\textsc{Opt}(I))$.

Abbildung 16.

- Falls das Gesamtgewicht grösser als 1 ist, so ist der Gesamtgewinn von beiden Algorithmen auch grösser als 1, weswegen für den erwarteten Gewinn von RKPone

$$\begin{aligned}&\frac{1}{2}\operatorname{gain}(Greedy_1(I)) + \frac{1}{2}\operatorname{gain}(Greedy_2(I))\\ &\quad= \frac{1}{2}\left(\operatorname{gain}(Greedy_1(I)) + \operatorname{gain}(Greedy_2(I))\right)\\ &\quad\geq \frac{1}{2}\end{aligned}$$

  folgt. Beachten Sie auch hier, dass Opt höchstens einen Gewinn von 1 besitzt.

Folglich ist ist der erwartete kompetitive Faktor in beiden Fällen also höchstens 2 (siehe Abbildung 16). □

In Satz 5.8 haben wir gezeigt, dass eine konstante Anzahl von Advice-Bits es nicht erlaubt, besser als 2-kompetitiv zu sein. Dies gilt somit auch für Zufallsbits. In Satz 5.9 zeigten wir wiederum, dass logarithmischer Advice es erlaubt $(1+\varepsilon)$-kompetitiv zu sein. Dies ist bei Randomisierung nicht der Fall.

5.5 Eine untere Schranke für randomisierte Online-Algorithmen

Wir zeigen im folgenden Satz, dass ein Zufallsbit genau so mächtig ist wie jede beliebige andere Anzahl an Randomisierung.

Satz 5.13. *Kein randomisierter Online-Algorithmus für das Rucksack-Problem ist besser als strikt 2-kompetitiv im Erwartungswert.*

Beweis. Sei $\varepsilon > 0$. Wir betrachten die folgende Klasse $\mathcal{I}$ von Instanzen, die aus zwei Eingaben besteht. Bei beiden wird zunächst ein Objekt mit dem Gewicht von ε angeboten. Hiernach wird entweder nichts mehr angeboten oder ein weiteres Objekt mit einem Gewicht von 1. Sei nun Rand ein beliebiger randomisierter Online-Algorithmus.

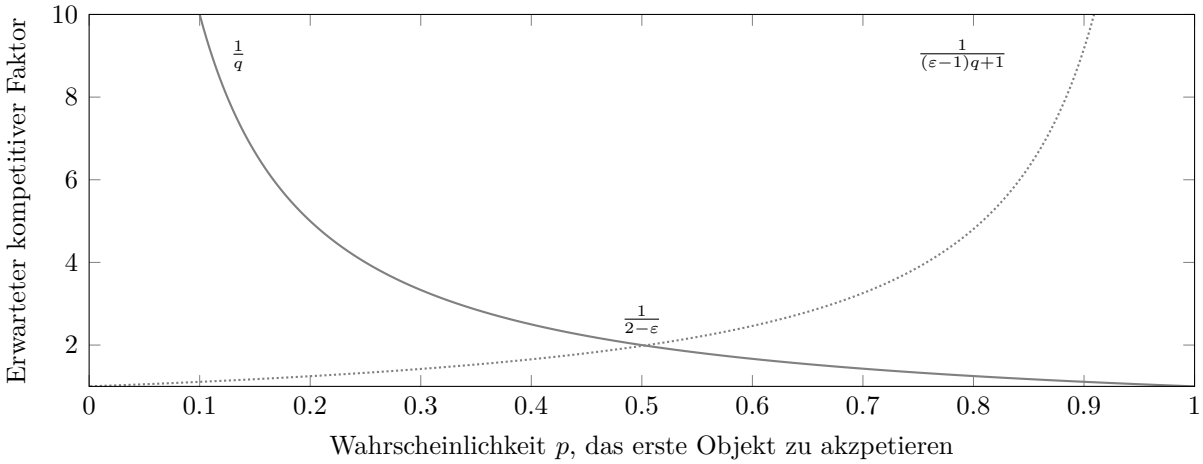


Abbildung 17.

- RAND akzeptiert das erste Objekt mit einer Wahrscheinlichkeit von p. Wir stellen fest, dass p nicht 0 sein darf, da RAND sonst einen Gewinn von 0 auf der Instanz besitzt, die nur aus dem ersten Objekt besteht.
- Akzeptiert RAND das erste Objekt, so ist kein Platz mehr in seinem Rucksack für das zweite Objekt, wenn es angeboten wird.
- Auf der anderen Seite ist sein Gewinn 0 auf der Instanz, in der kein zweites Objekt angeboten wird, wenn er das erste nicht akzeptiert. Die Wahrscheinlichkeit hierfür ist offensichtlich $1 - p$.

Nehmen wir an, das zweite Objekt wird angeboten, was zu einem optimalen Gewinn von 1 führt. Der erwartete kompetitive Faktor von RAND ist somit

$$\frac{1}{p \cdot \varepsilon + (1-p) \cdot 1} = \frac{1}{(\varepsilon - 1) \cdot p + 1} .$$

Wird das zweite Objekt nicht angeboten, hat eine optimale Lösung einen Gewinn von ε. Für den kompetitiven Faktor von RAND folgt also

$$\frac{\varepsilon}{p \cdot \varepsilon + (1-p) \cdot 0} = \frac{1}{p} .$$

Durch die Wahl der konkreten Instanz kann ein Gegenspieler immer dafür sorgen, dass der erwartete kompetitive Faktor dem Maximum dieser beiden Werte entspricht. RAND will also p so wählen, dass dieses Maximum minimiert wird. Da $((\varepsilon - 1)p + 1)^{-1}$ monoton in p steigt, während $1/p$ monoton in p fällt, ist die beste Strategie für RAND, p so zu wählen, dass die beiden erwarteten kompetitiven Faktoren gleich sind (siehe Abbildung 17). Es folgt

$$\frac{1}{(\varepsilon - 1) \cdot p + 1} = \frac{1}{p} \iff p = \frac{1}{2 - \varepsilon}$$

und, wenn wir dies wieder in eine der beiden Gleichungen für den erwarteten kompetitiven Faktor einsetzen, folgt, dass dieser mindestens $2 - \varepsilon$ ist. □

5.6 Resource-Augmentation

In diesem Kapitel wollen wir untersuchen, wie sehr es Online-Algorithmen hilft, wenn wir ihnen einen kleinen Vorteil verschaffen. Konkret weichen wir die Gewichtsschranke des Rucksacks ein bisschen auf, sodass, für ein $\gamma > 0$, ein Online-Algorithmus ein Gesamtgewicht von $1+\gamma$ einpacken darf, wohingegen der optimale Algorithmus Opt weiterhin die Kapazität von 1 nicht überschreiten darf. Ferner nehmen wir an, dass die Gewichte der Objekte in der Eingabe nach wie vor nicht grösser als 1 sind. Die Online-Algorithmen haben also gewissermassen mehr Ressourcen zur Verfügung, weswegen wir bei diesem Modell von **Resource-Augmentation** (auf Deutsch in etwa «Ressourcen-Vergrösserung») sprechen. Bei dem hier vorgestellten Problem sprechen wir vom γ-Rucksack-Problem. Wenn wir deterministische Online-Algorithmen betrachten, so stellen wir fest, dass Greedy hier das beste Ergebnis erzielt.

Satz 5.14. *Greedy ist strikt $1/\gamma$-kompetitiv für das γ-Rucksack-Problem.*

Beweis. Wenn Greedy ein Objekt nicht mehr in den Rucksack packen kann, so ist dieser schon mindestens mit Objekten mit einem Gesamtgewicht von γ gefüllt, da alle Objekte ein Gewicht von höchstens 1 besitzen. Mit dem maximalen Gewinn von 1 für Opt folgt die Behauptung. □

Es folgt, dass $\gamma = 1$ es Greedy erlaubt, optimal für das γ-Rucksack-Problem zu sein; wir wollen interessante Instanzen untersuchen und gehen deswegen davon aus, dass γ eine kleine Konstante ist mit $0 < \gamma < 1$. Die in Satz 5.14 vorgestellte Schranke ist dicht.

Satz 5.15. *Kein deterministischer Online-Algorithmus für das γ-Rucksack-Problem ist besser als strikt $(1/\gamma + \varepsilon)$-kompetitiv.*

Beweis. Der Beweis kann analog zu dem von Satz 5.2 geführt werden. Sei $\varepsilon > 0$ und sei Alg ein deterministischer Online-Algorithmus für das γ-Rucksack-Problem. Zunächst wird ein Objekt w mit einem Gewicht von $\gamma + \varepsilon$ angeboten.

- Packt Alg w nicht ein, wird kein weiteres Objekt mehr angeboten.
- Wird w hingegen eingepackt, wird ein zweites Objekt mit dem Gewicht 1 angeboten.

Damit erhalten wir sofort eine untere Schranke von $1/(\gamma+\varepsilon)$ auf den strikten kompetitiven Faktor von Alg. □

Für kleine Werte von γ hilft Resource-Augmentation den konstruierten Online-Algorithmen nicht viel. Anders sieht es allerdings aus, wenn wir dieses Konzept mit dem eines Advice-Bandes kombinieren. In diesem Fall ermöglicht uns eine konstante, von γ abhängige Anzahl an Advice-Bits, sehr nah an eine optimale Lösung heranzukommen. Der erreichte kompetitive Faktor hängt hier ebenfalls von γ ab.

Satz 5.16. *Sei $1/4 > \gamma > 0$. Es existiert ein Online-Algorithmus* Aug *mit Advice für das γ-Rucksack-Problem, der einen strikten kompetitiven Faktor von*

$$1 + \frac{3\gamma}{1 - 4\gamma}$$

erreicht und höchstens

$$\left\lceil 2 \log_2 \left\lceil \frac{1}{\gamma} \right\rceil + \frac{1}{\gamma} \log_2 \left\lceil \frac{1}{\gamma^2} \right\rceil \right\rceil + 1$$

Advice-Bits verwendet.

Beweis. Als erstes beschränken wir den Gewinn des fixen optimalen Algorithmus Opt von unten, indem wir nach Satz 5.4 annehmen, dass der Gewinn von Opt mindestens $1/2$ ist; ansonsten verfolgt Aug eine simple Greedy-Strategie und ist optimal. Dies wird wieder mit dem ersten Bit auf dem Advice-Band kommuniziert.

Wie im Beweis von Satz 5.9 unterteilen wir die Objekte, die Opt in den Rucksack packt, in schwere und leichte. Die schweren Objekte seien mit $x_1, \ldots, x_k$ und die leichten mit $y_1, \ldots, y_m$ bezeichnet. Für eine Instanz I ist also

$$\textsc{Opt}(I) := \{x_1, \ldots, x_k\} \cup \{y_1, \ldots, y_m\} .$$

Als schwer bezeichnen wir hier Objekte mit einen Gewicht von mindestens γ und die leichten Objekte haben ein Gewicht, das echt kleiner als γ ist; offensichtlich gilt für die Anzahl der schweren Objekte $k \leq 1/\gamma$.

- Aug kennt γ und er berechnet die ungefähren Gewichte dieser Objekte, die auf ein Vielfaches von γ^2 abgerundet sind.
- Ausserdem berechnet Aug eine Schranke für den Teil des Rucksacks, der von Opt mit leichten Objekten gefüllt wird.
- Solange diese Schranke nicht überschritten wird, akzeptiert Aug leichte Objekte mit einer Greedy-Strategie.

Als nächstes zeigen wir, wie die Gewichte der schweren Objekte kodiert werden. Zu diesem Zweck sei

$$\overline{x}_i := j, \text{ sodass } j\gamma^2 \leq x_i < (j+1)\gamma^2 ,$$

für jedes schwere Objekt x_i, $1 \leq i \leq k$. Alle $\overline{x}_i$s werden nacheinander auf das Advice-Band geschrieben und von Aug am Anfang der Berechnung gelesen. Wird nun ein Objekt x' angeboten, so überprüft Aug, ob das entsprechende $\overline{x}_i$ als Advice gegeben wurde. Dies bedeutet, der Algorithmus schaut nach, ob ein $\overline{x}_i$ Teil des Advice ist, sodass

$$\overline{x}_i\gamma^2 \leq x' < (\overline{x}_i + 1)\gamma^2 .$$

Ist dies der Fall, so wird x' akzeptiert. Dieses Objekt bezeichnen wir dann mit x'_i, da es mit x_i in der Lösung von Opt korrespondiert. Andernfalls wird x' verworfen. Wenn wir jetzt x_i gegen x'_i abschätzen, so stellen wir fest, dass wegen der Rundung

$$x_i - \gamma^2 \leq x'_i < x_i + \gamma^2$$

gilt. Addieren wir nun alle diese Werte, so erhalten wir

$$-\gamma + \sum_{i=1}^{k} x_i \leq \sum_{i=1}^{k} x'_i \leq k\gamma^2 + \sum_{i=1}^{k} x_i \leq \gamma + \sum_{i=1}^{k} x_i \ .$$

Dies bedeutet, dass Aug für jedes schwere Objekt ein Objekt einpackt, sodass die Summe der Gewichte dieser eingepackten Objekte höchstens γ mehr Platz braucht als die von Opt eingepackten schweren Objekte. Auf der anderen Seite wird aber auch nicht mehr als ein Anteil von γ des Rucksacks nicht genutzt.

Wir unterscheiden wieder zwei Fälle bezüglich der Grösse der optimalen Lösung. Nehmen wir zunächst einmal an $\text{gain}(\text{Opt}(I)) < 1 - \gamma$.

- Dies bedeutet, dass Opt alle leichten Objekte, die in der Eingabe vorhanden sind, in den Rucksack packt, denn sonst wäre er nicht optimal.
- Wie wir gerade gesehen haben, benutzt Aug aufgrund der x'_is höchstens γ mehr Platz im Rucksack.
- Aufgrund der Resource-Augmentation hat Aug also mindestens so viel Platz wie Opt im Rucksack für leichte Objekte und kann diese alle einpacken.
- Auf der anderen Seite haben wir auch gesehen, dass

  $$-\gamma + \sum_{i=1}^{k} x_i \leq \sum_{i=1}^{k} x'_i$$

  ist.
- Somit hat Aug einen Gewinn von mindestens $\text{gain}(\text{Opt}(I)) - \gamma$ und es folgt

  $$\frac{\text{gain}(\text{Opt}(I))}{\text{gain}(\text{Opt}(I)) - \gamma} = 1 + \frac{\gamma}{\text{gain}(\text{Opt}(I)) - \gamma} \leq 1 + \frac{2\gamma}{1 - 2\gamma} \leq 1 + \frac{3\gamma}{1 - 4\gamma} \ ,$$

  wobei wir für die erste Ungleichung benutzt haben, dass $\text{gain}(\text{Opt}(I)) \geq 1/2$ ist.

Nehmen wir also jetzt an, dass $1 - \gamma \leq \text{gain}(\text{Opt}(I)) \leq 1$ ist.

- Betrachten wir zunächst die leichten Objekte. Sei

  $$g := 1 - \sum_{i=1}^{k} x_i$$

  der Platz im Rucksack, der bei der von Opt berechneten Lösung für leichte Objekte übrig ist (siehe Abbildung 18).

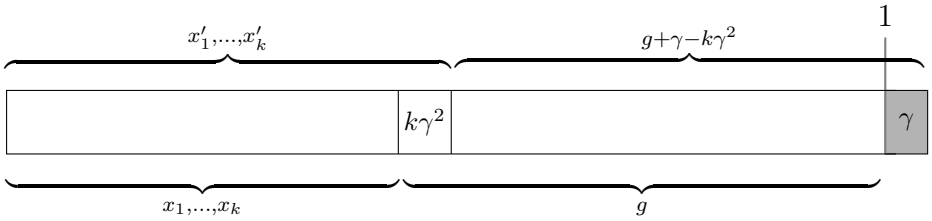


Abbildung 18.

- AUG kennt g nicht, aber er ist in der Lage, eine Näherung

  $$g' := 1 - \sum_{i=1}^{k} (\overline{x}_i + 1)\gamma^2$$

  zu berechnen.

- Es gilt

  $$g - \gamma \leq g' \leq g$$

  und wir wissen, dass sowohl AUG und OPT alle leichten Objekte zur Verfügung haben.

- Sei nun I_g die Instanz I eingeschränkt auf einen Rucksack der Grösse g und lediglich leichte Objekte. Sei ferner $\text{OPT}(I_g)$ eine fixe optimale Lösung für I_g. Analog definieren wir $I_{g'}$ und $\text{OPT}(I_{g'})$.

- OPT hat einen Gewinn von $\text{gain}(\text{OPT}(I_g)) \leq g$ auf I_g.

- Da AUG eine Greedy-Strategie auf $I_{g'}$ realisiert, ist er nach Satz 5.3 entweder optimal oder er erzielt einen Gewinn, der mindestens

  $$g' - \gamma \geq g - 2\gamma \geq \text{gain}(\text{OPT}(I_g)) - 2\gamma$$

  ist.

- Mit diesen Überlegungen folgt schliesslich, dass der kompetitive Faktor von AUG höchstens

  $$\begin{aligned}
  \frac{\text{gain}(\text{OPT}(I))}{\sum_{i=1}^{k} x'_i + g' - \gamma} &\leq \frac{\text{gain}(\text{OPT}(I))}{\sum_{i=1}^{k} x'_i + g - 2\gamma} \\
  &\leq \frac{\text{gain}(\text{OPT}(I))}{\sum_{i=1}^{k} x_i + g - 3\gamma} \\
  &\leq \frac{\text{gain}(\text{OPT}(I))}{\sum_{i=1}^{k} x_i + \text{gain}(\text{OPT}(I_g)) - 3\gamma}
  \end{aligned}$$

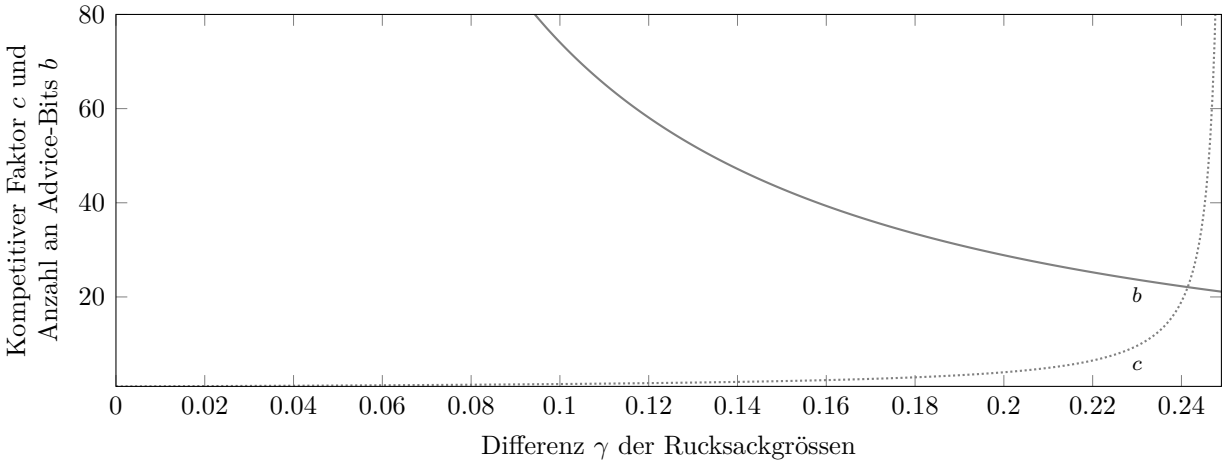


Abbildung 19.

$$= \frac{\text{gain}(\textsc{Opt}(I))}{\text{gain}(\textsc{Opt}(I)) - 3\gamma}$$
$$\leq 1 + \frac{3\gamma}{1 - 4\gamma}$$

ist.

Jetzt schätzen wir noch die Anzahl der verwendeten Advice-Bits ab.

- Ein Advice-Bit gibt zunächst an, ob $\text{gain}(\textsc{Opt}(I)) \leq 1/2$ oder nicht.
- Offensichtlich gibt es höchstens k verschiedene $\overline{x}_i$s (so viele wie es schwere Objekte in der von Opt berechneten Lösung gibt).
- Jeder dieser Werte ist höchstens $1/\gamma^2$.
- Um alle $\overline{x}_i$s auf das Advice-Band zu schreiben, werden also

  $$k \log_2 \left\lceil \frac{1}{\gamma^2} \right\rceil \leq \left\lceil \frac{1}{\gamma} \log_2 \left\lceil \frac{1}{\gamma^2} \right\rceil \right\rceil$$

  Bits benötigt.
- Um den Advice zu dekodieren, muss Aug den Wert k kennen. Dies wird wieder selbstbeschränkt mit $2\lceil \log_2 k \rceil \leq 2\lceil \log_2 \lceil 1/\gamma \rceil \rceil$ Advice-Bits getan.
- Eine obere Schranke für die Länge der binären Kodierung der $\overline{x}_i$s kann von Aug ohne weitere Information berechnet werden.

Insgesamt können wir die Anzahl der benötigten Advice-Bits also mit

$$\left\lceil \frac{1}{\gamma} \log_2 \left\lceil \frac{1}{\gamma^2} \right\rceil \right\rceil + 2 \left\lceil \log_2 \left\lceil \frac{1}{\gamma} \right\rceil \right\rceil + 1$$

abschätzen. □

Die Anzahl an benötigten Advice-Bits ist monoton fallend in γ, während der erreichte kompetitive Faktor mit γ steigt (siehe Abbildung 19).

5.7 Verwendete und weiterführende Literatur

Die Entscheidungsversion des Rucksack-Problems ist unter Richard Karps 21 $\mathcal{NP}$-vollständigen Problemen [23]. Für das Optimierungsproblem existiert ein pseudo-polynomieller Algorithmus, der auf dynamischer Programmierung basiert. Ibarra und Kim [19] benutzten diesen Ansatz, um ein FPTAS zu konstruieren. Somit ist die Offline-Version des Problems unter den einfacheren $\mathcal{NP}$-schweren Optimierungsproblemen.

Die untere Schranke für deterministische Online-Algorithmen stammt von Marchetti-Spaccamela und Vercellis [31], die die Online-Version als erstes untersuchten. Die hier präsentierten Ergebnisse über Randomisierung und die Advice-Komplexität des Problems stammen von Böckenhauer et al. [5,6]. In dieser Arbeit wurde auch das allgemeine Rucksack-Problem untersucht, bei dem die Objekte Gewichte und Werte haben, die sich unterscheiden. Auch hier lässt sich ein Online-Algorithmus mit Advice konstruieren, der $\mathcal{O}(\log n)$ Advice-Bits benutzt und $(1+\varepsilon)$-kompetitiv ist, für jedes $\varepsilon > 0$; jedoch gilt dies nur, wenn die Werte und Gewichte mit polynomiellem Platz dargestellt werden können.

Die Idee der Resource-Augmentation stammt von Kalyanasundaram und Pruhs [21]; seit ihrer Einführung wurde sie für eine Vielzahl von Online-Problemen angewandt. Für Varianten des Rucksack-Problems wurde Resource-Augmentation beispielsweise von Han und Makino [15] und Iwama und Zhang [20], von denen auch die untere Schranken für deterministische Online-Algorithmen stammt, verwendet. Die hier präsentierten Resultate für Resource-Augmentation mit Advice gehen ebenfalls auf Böckenhauer et al. [5,6] zurück.

6 Advice und Randomisierung

In diesem Kapitel wollen wir uns dem Zusammenhang zwischen Advice- und Zufallsbits widmen. Fassen wir zunächst kurz zusammen, was wir bislang gelernt haben (siehe Beobachtung 4.1).

- Existiert für ein Online-Problem Π ein randomisierter Online-Algorithmus, der mit b Zufallsbits einen erwarteten kompetitiven Faktor von c erreicht, so existiert auch ein Online-Algorithmus mit Advice, der c-kompetitiv ist und b Advice-Bits benutzt.

- Existiert auf der anderen Seite kein Online-Algorithmus mit Advice für Π, der c-kompetitiv ist, so existiert auch kein randomisierter Online-Algorithmus, der c-kompetitiv im Erwartungswert ist und b Zufallsbits braucht.

Im letzten Kapitel haben wir gesehen, dass dazwischen vieles offen ist. Für das Rucksack-Problem ist ein Zufallsbit so mächtig wie ein Advice-Bit, aber weitere Zufallsbits helfen dann nicht, während beispielsweise eine lineare Anzahl an Advice-Bits erlaubt, einen optimalen Online-Algorithmus mit Advice anzugeben.

Generell erscheint uns der Advice viel mächtiger als zufällige Entscheidungen. Deswegen drängt sich die Frage auf, ob wir nicht Bits sparen können, wenn diese von einem Orakel anstatt von einem Zufallsgenerator stammen. Der folgende Satz zeigt, dass dies für Online-Minimierungsprobleme in gewissen Schranken tatsächlich der Fall ist. Genauer zeigen wir, dass wir, wenn wir einen randomisierten Online-Algorithmus RAND gegeben haben, beliebig nahe an den von ihm erreichten kompetitiven Faktor herankommen, indem wir eine Anzahl von Advice-Bits benutzen, die nicht von der Anzahl der von RAND benutzten Zufallsbits abhängt; allerdings müssen wir hierfür die Anzahl der möglichen Eingaben für jede gegebene Eingabelänge einschränken. Wir können den folgenden Satz dann benutzen, um untere Schranken für randomisierte Online-Algorithmen zu zeigen.

Satz 6.1. *Sei Π ein Online-Minimierungsproblem, für das $m(n)$ verschiedene Eingaben mit einer Eingabelänge von n existieren. Ausserdem existiere für Π ein randomisierter Online-Algorithmus RAND mit einem erwarteten kompetitiven Faktor von c. Dann können wir einen Online-Algorithmus ALG mit Advice für Π konstruieren, der $((1+\varepsilon)c)$-kompetitiv ist, für jedes $\varepsilon > 0$, und der höchstens*

$$\lceil \log_2 n \rceil + 2\lceil \log_2 \lceil \log_2 n \rceil \rceil + \left\lceil \log_2 \left(\left\lfloor \frac{\log_2(m(n))}{\log_2(1+\varepsilon)} \right\rfloor + 1 \right) \right\rceil$$

Advice-Bits benötigt.

Beweis. Wir nehmen an, der betrachtete Online-Algorithmus RAND benutzt $b(n)$ Zufallsbits für Eingaben der Länge n; wie wir wissen (siehe Beobachtung 1.10), bedeutet dies, dass RAND uniform zufällig einen deterministischen Online-Algorithmus aus einer Menge strat(RAND) =

$\{A_1, \ldots, A_{2^{b(n)}}\}$ wählt. Wir konstruieren jetzt wieder eine Matrix T wie in Kapitel 1.6.

	A_1	A_2	A_3	$\ldots$
I_1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$\ldots$
I_2	$c_{2,1}$	$c_{2,2}$	$c_{2,3}$	
I_3	$c_{3,1}$	$c_{3,2}$	$c_{3,3}$	
$\vdots$	$\vdots$			$\ddots$

Der Eintrag in der i-ten Zeile und j-ten Spalte gibt also wieder an, wie der kompetitive Faktor des Online-Algorithmus A_j auf der Eingabe I_i ist. Unsere Idee ist jetzt zu zeigen, dass wir geschickt ein paar Spalten auswählen können, sodass der kompetitive Faktor der entsprechenden Online-Algorithmen für möglichst viele der Eingaben niedrig ist. Diese Algorithmen fassen wir in einer Menge $\mathcal{S}$ zusammen und als Advice wird ALG der Index des zu benutzenden Algorithmus für die aktuelle Eingabe gegeben.

- Da der erwartete kompetitive Faktor von RAND höchstens c ist, gilt für eine fixe Zeile (und somit Eingabe) i nach Definition

$$\begin{aligned} c &\geq \frac{\mathbb{E}[\text{cost}(\text{RAND}(I_i))]}{\text{cost}(\text{OPT}(I_i))} \\ &= \frac{\sum_{j=1}^{2^{b(n)}} \frac{1}{2^{b(n)}} \cdot \text{cost}(A_j(I_i))}{\text{cost}(\text{OPT}(I_i))} \\ &= \frac{1}{2^{b(n)}} \sum_{j=1}^{2^{b(n)}} \frac{\text{cost}(A_j(I_i))}{\text{cost}(\text{OPT}(I_i))} \\ &= \frac{1}{2^{b(n)}} \sum_{j=1}^{2^{b(n)}} c_{i,j} \ . \end{aligned}$$

  Damit folgt für die Summe aller Einträge in einer Zeile

$$\frac{1}{2^{b(n)}} \sum_{j=1}^{2^{b(n)}} c_{i,j} \leq c \iff \sum_{j=1}^{2^{b(n)}} c_{i,j} \leq c \cdot 2^{b(n)} \ .$$

- Also ist die Summe aller Einträge in T höchstens

$$\sum_{i=1}^{m(n)} \sum_{j=1}^{2^{b(n)}} c_{i,j} \leq \sum_{i=1}^{m(n)} c \cdot 2^{b(n)} \leq c \cdot 2^{b(n)} \cdot m(n) \ .$$

- Da es $2^{b(n)}$ Spalten in T gibt, existiert eine Spalte (Online-Algorithmus) j', sodass

$$\sum_{i=1}^{m(n)} c_{i,j'} \leq c \cdot m(n)$$

  ist.

Den Algorithmus $A_{j'}$ nehmen wir in die Menge $\mathcal{S}$ auf und er wird verwendet für jede Instanz I_i, für die

$$\frac{\text{cost}(A_{j'}(I_i))}{\text{cost}(\textsc{Opt}(I_i))} = c_{i,j'} \leq (1+\varepsilon) \cdot c$$

gilt. Ein Beispiel ist in Abbildung 20 dargestellt. Sei s die Anzahl dieser Instanzen. Nun interessiert uns natürlich, wie gross s ist. Offensichtlich ist der kompetitive Faktor von $A_{j'}$ auf $m(n)-s$ Instanzen grösser als $(1+\varepsilon)c$. In der Summe ergibt dies für die entsprechenden Zeilen $(m(n)-s)(1+\varepsilon)c$ und es gilt

$$(m(n)-s)(1+\varepsilon)c < \sum_{i=1}^{m(n)} c_{i,j'} \ .$$

Aus diesen Überlegungen folgt

$$(m(n)-s)(1+\varepsilon)c < m(n) \cdot c \iff s > \frac{\varepsilon}{1+\varepsilon} \cdot m(n) \ .$$

Mit anderen Worten können wir den Online-Algorithmus $A_{j'}$ für einen Anteil von $\varepsilon/(1+\varepsilon)$ aller Eingaben verwenden, da wir wissen, dass sein kompetitiver Faktor auf dieser Anzahl nicht grösser ist als $(1+\varepsilon)c$.

Nachdem $A_{j'}$ der Menge $\mathcal{S}$ hinzugefügt wurde, löschen wir die Spalte j' und alle Zeilen, die mit Eingaben korrespondieren, auf denen $A_{j'}$ einen ausreichend kleinen kompetitiven Faktor erreicht, aus T. Übrig bleiben also höchstens

$$\left(1 - \frac{\varepsilon}{1+\varepsilon}\right) \cdot m(n) = \left(\frac{1}{1+\varepsilon}\right) \cdot m(n)$$

Zeilen, für die wir noch einen guten Algorithmus finden müssen. Für jede dieser Zeilen war der entfernte Eintrag in Spalte j' grösser als c. Somit ist nach dem Entfernen dieser Spalte der Durchschnitt aller Einträge jeder übrig gebliebenen Zeile weiterhin nicht grösser als c. Wir können also das soeben gezeigte Vorgehen mit den verbliebenen

$$\left(\frac{1}{1+\varepsilon}\right) m(n)$$

Zeilen wiederholen und erneut einen Algorithmus $A_{j''}$ finden, der einen Anteil von

$$\frac{\varepsilon}{1+\varepsilon}$$

von den entsprechenden Instanzen mit einem genügend kleinen kompetitiven Faktor bearbeitet.

Wir schätzen jetzt ab, wie oft wir dieses Vorgehen wiederholen müssen bis für jede Eingabe ein Algorithmus gefunden ist. Konkret wollen wir die natürliche Zahl r finden, sodass

$$\left(\frac{1}{1+\varepsilon}\right)^r m(n) < 1$$

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	⌀
I_1	5	4	4	2	3	1	4	1	3
I_2	3	1	1	3	5	5	2	4	3
I_3	4	6	4	1	2	4	1	2	3
I_4	1	1	5	5	4	2	3	3	3
I_5	2	2	4	2	5	2	2	5	3
I_6	1	5	1	8	1	2	4	2	3
I_7	2	1	4	1	4	3	5	4	3
I_8	1	3	1	7	2	2	3	5	3
I_9	3	3	6	2	1	4	1	4	3

	A_1	A_3	A_4	A_5	A_6	A_7	A_8	⌀
I_1	5	4	2	3	1	4	1	2.86
I_2								
I_3	4	4	1	2	4	1	2	2.58
I_4								
I_5								
I_6	1	1	8	1	2	4	2	2.72
I_7								
I_8								
I_9								

Abbildung 20.

ist. Wir erhalten

$$\left(\frac{1}{1+\varepsilon}\right)^r < \frac{1}{m(n)} \iff (1+\varepsilon)^r > m(n) \iff r > \log_{1+\varepsilon}(m(n)) \ ,$$

was bedeutet, dass wir nicht mehr als

$$\left\lfloor \frac{\log_2(m(n))}{\log_2(1+\varepsilon)} \right\rfloor + 1$$

Wiederholungen beziehungsweise deterministische Algorithmen aus strat(RAND) benötigen. Das gibt uns sofort eine obere Schranke für die Kardinalität von $\mathcal{S}$.

Jetzt schätzen wir die nötigen Advice-Bits ab.

- Zunächst muss ALG die konkrete Eingabelänge kommuniziert werden. Dies passiert selbstbeschränkt, wie es in Kapitel 5 vorgestellt wurde, und benötigt somit maximal

  $$2\lceil \log_2 \lceil \log_2 n \rceil \rceil + \lceil \log_2 n \rceil$$

  Advice-Bits.

- Hieraus erstellt ALG dann die Matrix T durch Simulation des randomisierten Online-Algorithmus RAND auf jeder möglichen Eingabe. Nach obigem Vorgehen konstruiert ALG die Menge $\mathcal{S}$ von deterministischen Online-Algorithmen; die Algorithmen werden schliesslich nach einer festen Ordnung durchnummeriert. Mit weiteren

  $$\left\lceil \log_2 \left(\left\lfloor \frac{\log_2(m(n))}{\log_2(1+\varepsilon)} \right\rfloor + 1 \right) \right\rceil$$

  Advice-Bits wird schliesslich einer dieser Algorithmen ausgewählt.

Es folgt, dass der kompetitive Faktor von ALG auf jeder Instanz höchstens c ist. □

Beachten Sie, dass der Online-Algorithmus ALG mit Advice nicht in Polynomzeit läuft, wenn $m(n)$ oder $b(n)$ gross bezüglich der Eingabelänge sind, da ALG die gesamte Matrix T berechnen muss.

6.1 Grenzen des Ansatzes

Der in Satz 6.1 konstruierte Online-Algorithmus mit Advice ist schlechter als der ursprüngliche randomisierte Online-Algorithmus, auch wenn die Differenz ihrer kompetitiven Faktoren sehr klein ist. Wir fragen jetzt, ob es möglich ist, das Resultat zu verbessern und einen Online-Algorithmus zu konstruieren, der denselben kompetitiven Faktor erreicht wie der gegebene randomisierte Online-Algorithmus.

Die Antwort fällt negativ aus. Wir können wieder ein Online-Minimierungsproblem konstruieren, für das Advice-Bits und Zufallsbits in gewissem Sinne gleichmächtig sind.

- Die Eingabe $I = (x_1, \ldots, x_n)$ beginnt mit einer Anfrage $x_1 = 0$.
- Alle weiteren Anfragen sind Bits, also $x_i \in \{0, 1\}$, $2 \leq i \leq n$; ferner müssen auch Bits als Antworten gegeben werden, also $y_i \in \{0, 1\}$, $1 \leq i \leq n - 1$.
- Wenn $y_i = x_{i+1}$ für alle i, $1 \leq i \leq n - 1$, gilt, sind die Gesamtkosten der berechneten Lösung 1, sonst 2.
- Ein optimaler Algorithmus zahlt also 1 und jede weitere Lösung 2.
- Offensichtlich wählt ein bester randomisierter Online-Algorithmus jede Antwort so, dass er mit einer Wahrscheinlichkeit von $1/2$ jeweils 0 oder 1 ausgibt.
- Dieser Algorithmus benutzt $n - 1$ Zufallsbits und sein erwarteter kompetitiver Faktor ist nicht grösser als

$$\frac{\frac{2^{n-1}-1}{2^{n-1}} \cdot 2 + \frac{1}{2^{n-1}} \cdot 1}{1} = 2 - \frac{1}{2^{n-1}} \, .$$

- Auf der anderen Seite ist jeder deterministische Online-Algorithmus mit Advice, der weniger als $n - 1$ Advice-Bits benutzt, offensichtlich bestenfalls 2-kompetitiv.

Es existieren also Probleme, für die ein Online-Algorithmus mit Advice, um nicht schlechter zu sein als ein gegebener randomisierter Online-Algorithmus, ebenso viele Advice-Bits braucht wie dieser Zufallsbits.

6.2 Eine Anwendung für *k*-Server

In Kapitel 3 haben wir die k-Server-Vermutung für randomisierte Online-Algorithmen vorgestellt, die aussagt, dass es einen im Erwartungswert $\Theta(\log k)$-kompetitiven randomisierten Online-Algorithmus für dieses Problem gibt. Der folgende Satz zeigt, wie wir die Advice-Komplexität als Technik verwenden könnten, um diese Vermutung zu widerlegen.

Satz 6.2. *Wenn jeder Online-Algorithmus mit Advice für k-Server mindestens $\omega(\log n)$ Advice-Bits benötigt, um $\mathcal{O}(\log k)$-kompetitiv zu sein, gilt die randomisierte k-Server-Vermutung nicht.*

Beweis. Wir betrachten nur Eingaben, bei denen wir die Grösse des metrischen Raums $\mathcal{M}$ nach oben durch 2^n beschränken, wobei n die Länge der Eingabe ist. Sei $m(n)$ wieder die Anzahl an Eingaben der Länge n. In jedem Zeitschritt wird ein Punkt angefragt, somit ist

$$m(n) = (2^n)^n .$$

Existiert ein randomisierter Online-Algorithmus RAND, der auf allen betrachteten Eingaben $\mathcal{O}(\log k)$-kompetitiv ist, so gilt nach Satz 6.1, dass auch ein $\mathcal{O}(\log k)$-kompetitiver Online-Algorithmus mit Advice existiert, der höchstens

$$\lceil \log_2 n \rceil + 2\lceil \log_2 \lceil \log_2 n \rceil \rceil + \log_2 \left(\left\lfloor \frac{\log_2((2^n)^n)}{\log_2(1+\varepsilon)} \right\rfloor + 1 \right) \in \mathcal{O}(\log n)$$

Advice-Bits benötigt.

Können wir also zeigen, dass jeder Online-Algorithmus mit Advice asymptotisch mehr Advice benötigt, ist dies ein Widerspruch zur Existenz von RAND. □

6.3 Verwendete und weiterführende Literatur

Satz 6.2 wurde von Böckenhauer et al. [7] bewiesen. Bianchi et al. [3] verwendeten ihn anschliessend, um eine untere Schranke für den kompetitiven Faktor für randomisierte Online-Algorithmen für Online-$L(1,2)$-Färbung auf Pfaden und Kreisen zu zeigen.

Ein weiterer Schritt ist, Randomisierung und Advice zu verbinden. Emek et al. [13] haben untere Schranken für randomisierte Online-Algorithmen mit Advice für metrische Task-Systeme vorgestellt. Böckenhauer et al. [4] haben ferner solche Algorithmen für das sogenannte Boxen-Problem entworfen. Weiterführende Überlegungen zum Verhältnis zwischen Advice und Randomisierung werden von Komm [24] und Komm und Královič [26] gegeben. In diesen Arbeiten wird ein Barely-Random-Algorithmus für das Job-Shop-Scheduling-Problem mit zwei Jobs mit Einheitsgrösse [8, 18] vorgestellt.

Literaturverzeichnis

[1] Theodore P. Baker, John Gill und Robert Solovay. Relativizations of the $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ question. *SIAM Journal on Computing*, 4(4):431–442, Society for Industrial and Applied Mathematics, 1975.

[2] Nikhil Bansal, Niv Buchbinder, Aleksander Mądry und Joseph Naor. A polylogarithmic-competitive algorithm for the k-server problem (extended abstract). In *Proceedings of the 52th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2011)*, Seiten 267–276. IEEE Computer Society, 2011.

[3] Maria Paola Bianchi, Hans-Joachim Böckenhauer, Juraj Hromkovič, Sacha Krug und Björn Steffen. On the advice complexity of the online $L(2,1)$-coloring problem on paths and cycles. In *Proceedings of the 18th Annual International Conference on Computing and Combinatorics (COCOON 2013)*, Band 7936 von *Lecture Notes in Computer Science*, Seiten 53–64. Springer-Verlag, 2013.

[4] Hans-Joachim Böckenhauer, Juraj Hromkovič, Dennis Komm, Richard Královič und Peter Rossmanith. On the power of randomness versus advice in online computation. In *Languages Alive, Essays Dedicated to Jürgen Dassow on the Occasion of His 65th Birthday*, Band 7300 von *Lecture Notes in Computer Science*, Seiten 30–43. Springer-Verlag, 2012.

[5] Hans-Joachim Böckenhauer, Dennis Komm, Richard Královič und Peter Rossmanith. On the advice complexity of the knapsack problem. Technischer Bericht 740, Department of Computer Science, ETH Zurich, 2011.

[6] Hans-Joachim Böckenhauer, Dennis Komm, Richard Královič und Peter Rossmanith. On the advice complexity of the knapsack problem. In *Proceedings of the 10th Latin American Symposium on Theoretical Informatics (LATIN 2012)*, Band 7256 von *Lecture Notes in Computer Science*, Seiten 61–72. Springer-Verlag, 2012.

[7] Hans-Joachim Böckenhauer, Dennis Komm, Rastislav Královič und Richard Královič. On the advice complexity of the k-server problem. In *Proceedings of the 38th International Colloquium on Automata, Languages and Programming (ICALP 2011)*, Band 6755 von *Lecture Notes in Computer Science*, Seiten 207–218. Springer-Verlag, 2011.

[8] Hans-Joachim Böckenhauer, Dennis Komm, Rastislav Královič, Richard Královič und Tobias Mömke. On the advice complexity of online problems. In *Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC 2009)*, Band 5878 von *Lecture Notes in Computer Science*, Seiten 331–340. Springer-Verlag, 2009.

[9] Allan Borodin und Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[10] Stefan Dobrev, Rastislav Královič und Dana Pardubská. How much information about the future is needed? In *Proceedings of the 34th Conference on Current Trends in*

Theory and Practice of Computer Science (SOFSEM 2008), Band 4910 von *Lecture Notes in Computer Science*, Seiten 247–258. Springer-Verlag, 2008.

[11] Yuval Emek, Pierre Fraigniaud, Amos Korman und Adi Rosén. Online computation with advice. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP 2009)*, Band 5555 von *Lecture Notes in Computer Science*, Seiten 427–438. Springer-Verlag, 2009.

[12] Yuval Emek, Pierre Fraigniaud, Amos Korman und Adi Rosén. On the additive constant of the k-server work function algorithm. *Information Processing Letters*, 110(24):1120–1123, Elsevier Science Publishers, 2010.

[13] Yuval Emek, Pierre Fraigniaud, Amos Korman und Adi Rosén. Online computation with advice. *Theoretical Computer Science*, 412(24):2642–2656, Elsevier Science Publishers, 2011.

[14] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator und Neal E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, Elsevier Science Publishers, 1991.

[15] Xin Han und Kazuhisa Makino. Online removable knapsack with limited cuts. *Theoretical Computer Science*, 411(44-46):3956–3964, Elsevier Science Publishers, 2010.

[16] Juraj Hromkovič. *Design and Analysis of Randomized Algorithms*. Springer-Verlag, 2005.

[17] Juraj Hromkovič, Rastislav Královič und Richard Královič. Information complexity of online problems. In *Proceedings of the 35th International Symposium on Mathematical Foundations of Computer Science (MFCS 2010)*, Band 6281 von *Lecture Notes in Computer Science*, Seiten 24–36. Springer-Verlag, 2010.

[18] Juraj Hromkovič, Tobias Mömke, Kathleen Steinhöfel und Peter Widmayer. Job shop scheduling with unit length tasks: Bounds and algorithms. *Algorithmic Operations Research*, 2(1):1–14, Preeminent Academic Facets, 2007.

[19] Oscar H. Ibarra und Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, Association for Computing Machinery, 1975.

[20] Kazuo Iwama und Guochuan Zhang. Online knapsack with resource augmentation. *Information Processing Letters*, 110(22):1016–1020, Elsevier Science Publishers, 2010.

[21] Bala Kalyanasundaram und Kirk Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47(4):617–643, Association for Computing Machinery, 2000.

[22] Anna R. Karlin, Mark S. Manasse, Lyle A. McGeoch und Susan Owicki. Competitive randomized algorithms for non-uniform problems. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1990)*, Seiten 301–309. Society for Industrial and Applied Mathematics, 1990.

[23] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a Symposium on the Complexity of Computer Computations*, Seiten 85–103. Plenum Press, 1972.

[24] Dennis Komm. *Advice and Randomization in Online Computation.* Dissertation, ETH Zürich, 2012.

[25] Dennis Komm. *An Introduction to Online Computation: Determinism, Randomization, Advice.* Springer-Verlag, 2016.

[26] Dennis Komm und Richard Královič. Advice complexity and barely random algorithms. *Theoretical Informatics and Applications (RAIRO)*, 45(2):249–267, EDP Sciences, 2011.

[27] Elias Koutsoupias. The k-server problem. *Computer Science Review*, 3(2):105–118, Elsevier Science Publishers, 2009.

[28] Elias Koutsoupias und Christos H. Papadimitriou. On the k-server conjecture. *Journal of the ACM*, 42(5):971–983, Association for Computing Machinery, 1995.

[29] Lynn H. Loomis. On a theorem of von Neumann. *Proceedings of the National Academy of Sciences of the United States of America*, 32(8):213–215, National Academy of Sciences, 1946.

[30] Mark S. Manasse, Lyle A. McGeoch und Daniel D. Sleator. Competitive algorithms for on-line problems. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC 1988)*, Seiten 322–333. Association for Computing Machinery, 1988.

[31] Alberto Marchetti-Spaccamela und Carlo Vercellis. Stochastic on-line knapsack problems. *Mathematical Programming*, 68:73–104, Springer-Verlag, 1995.

[32] Ueli M. Maurer. On the oracle complexity of factoring integers. *Computational Complexity*, 5(3/4):237–247, Birkhäuser-Verlag, 1995.

[33] Ueli M. Maurer und Stefan Wolf. The relationship between breaking the Diffie-Hellman protocol and computing discrete logarithms. *SIAM Journal on Computing*, 28(5):1689–1721, Society for Industrial and Applied Mathematics, 1999.

[34] Lyle A. McGeoch und Daniel D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(6):816–825, Springer-Verlag, 1991.

[35] Rajeev Motwani und Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[36] John F. Nash. Equilibrium points in N-person games. *Proceedings of the National Academy of Sciences of the United States of America*, 36(1):48–49, National Academy of Sciences, 1950.

[37] Ronald L. Rivest und Adi Shamir. Efficient factoring based on partial information. In *Proceedings of the 4th Workshop on the Theory and Application of Cryptographic Techniques (EUROCRYPT 1985)*, Band 219 von *Lecture Notes in Computer Science*, Seiten 31–34. Springer-Verlag, 1985.

[38] Daniel D. Sleator und Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, Association for Computing Machinery, 1985.

[39] Philip D. Straffin. *Game Theory and Strategy*. Mathematical Association of America Textbooks, 1993.

[40] John Von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, Springer-Verlag, 1928.

[41] Andrew C.-C. Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, Seiten 222–227. IEEE Computer Society, 1977.

[42] Neal E. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, Springer-Verlag, 1994.