12 Online-Algorithmen am Beispiel des Seitenwechselproblems

Dieses Kapitel des Skripts ist noch nicht an die aktualisierten Folien vom Wintersemester 2017/2018 angepasst.

12.1 Unter *Online-Algorithmen* versteht man Algorithmen, denen nicht schon zu Beginn der Berechnung die ganze Eingabe vollständig zur Verfügung gestellt wird, sondern Schritt für Schritt in einer Folge $P = (r_1, r_2, \ldots, r_n)$ von „Anforderungen“. Auf jede Anforderung muss der Online-Algorithmus mit einer Aktion reagieren (ohne zu wissen, wie die weiteren Anforderungen aussehen werden). Einmal gefällte Entscheidungen können nicht zurück genommen werden.

Im ersten Abschnitt werden wir kurz auf deterministische Online-Algorithmen für ein konkretes Problem eingehen, das des „Seitenwechsels“ bei schnellen Zwischenspeichern. Dabei werden wir grundlegende Begriffe einführen und prinzipielle Unterschiede zur klassischen Art algorithmischer Aufgabenstellung kennen lernen. Im zweiten Abschnitt werden randomisierte Online-Algorithmen eingeführt und das Konzept der „Widersacher“ oder „Adversaries“. Im dritten Abschnitt stellen wir einen randomisierten Online-Algorithmus vor, der (gegen „unwissende“ Widersacher) signifikant besser ist als es jeder deterministische sein kann. Seitenwechsel gegen „adaptive“ Widersacher ist Gegenstand des vierten Abschnittes.

Wer genauer an diesem Thema interessiert ist, dem sei das Buch von Borodin und El-Yaniv (1998) empfohlen.

12.1 Das Seitenwechselproblem und deterministische Algorithmen

12.2 Dem *Seitenwechselproblem* liegt die folgende Aufgabenstellung zu Grunde: Ein Rechner ist mit einem *Cache* der Größe k und einem Hauptspeicher größeren Umfangs N ausgestattet. Der Zugriff auf Daten im Cache ist schneller und daher zu bevorzugen. Jede Anforderung r wird durch die Adresse einer Hauptspeicherzelle repräsentiert, deren Inhalt man lesen möchte. Die Aufgabe besteht darin, dafür zu sorgen, dass möglichst oft die zu lesenden Daten schon im Cache liegen und möglichst selten ein *Cache Miss* auftritt, bei dem doch auf den Hauptspeicher zugegriffen werden muss. Soll das dabei gelesene Datum im Cache abgelegt werden, ergibt sich bei bereits gefülltem Cache das Problem, ein früher dort abgelegtes Datum auszuwählen und durch das neue zu ersetzen.

Die Qualität eines Algorithmus A für das Seitenwechselproblem ist die Anzahl $f_A(r_1, \ldots, r_n)$ der bei einer Anforderungsfolge $(r_1, \ldots, r_n)$ insgesamt auftretenden Cache Misses.

12.3 Sogenannte *Offline-Algorithmen* für das Seitenwechselproblem dürfen für die Auswahl eines Datums, das bei einem Cache Miss bei Anforderung r_i aus einem vollen Cache entfernt werden soll, die Kenntnis auch aller noch folgenden Anforderungen r_j mit $j > i$ benutzen.

Man kann zeigen, dass die Anzahl Cache Misses minimiert wird, wenn immer (sofern nötig) das Datum aus dem Cache entfernt wird, das am spätesten in der Zukunft jemals wieder benötigt

wird. Dieser Algorithmus heißt üblicherweise **LFD** (engl. *longest forward distance*). Der Beweis seiner Optimalität ist nicht trivial (Belady 1966; Mattison u. a. 1971).

12.4 Definition Die Anzahl der bei einer Anforderungsfolge $(r_1, \ldots, r_n)$ insgesamt auftretenden Cache Misses eines optimalen Offline-Algorithmus bezeichnen wir mit $f_O(r_1, \ldots, r_n)$. ◇

12.5 Betrachten wir für einen Moment den Fall, dass es nur ein Datum mehr gibt als in den Cache passen. Man kann zeigen, dass *in diesem Fall* die Anzahl der Cache Misses des Offline-Algorithmus **LFD** für Folgen von n Anforderungen schlimmstenfalls n/k ist.

Algorithmus **LFD** ist in der Praxis völlig unbrauchbar, weil eine CPU bei der Cacheverwaltung eben *nicht* weiß, welche Anforderungen in der Zukunft kommen werden. Wir wenden uns daher nun Online-Algorithmen zu, die im Falle eines Cache Miss bei Anforderung r_i nur auf Grund der Kenntnis von $r_1, \ldots, r_i$ entscheiden, welches Datum aus dem Cache verdrängt wird.

12.6 Das Verhalten eines Online-Algorithmus bei der Verarbeitung einer Anforderung r_i ist unabhängig von r_{i+1}. Ist der Cache gefüllt, kann man folglich (in dem einzig interessanten Fall $N > k$) erzwingen, dass *jede* Anforderung zu einem Cache Miss führt.

Daher gibt es im Fall $N = k + 1$ auch beliebig lange Anforderungsfolgen, für die jeder Online-Algorithmus k mal mehr Cache Misses produziert als der optimale Offline-Algorithmus **LFD**.

12.7 Üblicherweise beschreibt man die Qualität eines Algorithmus, indem man für jedes n den Aufwand für die „schlimmsten" Eingaben dieser Länge angibt (*worst case complexity*). Aus der vorangegangenen Bemerkung ergibt sich, dass beim Seitenwechselproblem diese vergröbernde Sichtweise sinnlos ist. Denn man kann beliebig lange Anforderungsfolgen konstruieren, bei denen *in jedem Schritt* ein Cache Miss auftritt.

Ein sinnvoller Vergleich von Algorithmen ist so also nicht möglich. Der naheliegende Ausweg ist, jede Probleminstanz (i. e. Anforderungsfolge) einzeln zu betrachten.

Für den Vergleich von deterministischen (Online-)Algorithmen hat es sich als sinnvoll erwiesen, die folgenden Definition zu Grunde zu legen.

12.8 Definition Ein deterministischer Online-Algorithmus A heißt C-*kompetitiv*, falls es ein b gibt, das von C abhängen darf, aber nicht von n, so dass für alle Anforderungsfolgen $(r_1, \ldots, r_n)$ gilt:

$$f_A(r_1, \ldots, r_n) - C \cdot f_O(r_1, \ldots, r_n) \leqslant b \; .$$

Der *Wettbewerbsfaktor* C_A von A ist das Infimum der C, so dass A C-kompetitiv ist. ◇

Man beachte, dass dies insofern eine strenge Forderung ist, als die Ungleichung für *alle* Anforderungsfolgen gelten muss.

12.9 Beispiel. Ein Online-Algorithmus, der zum Beispiel bei vielen Prozessoren mit mehrfach assoziativen First Level Caches (siehe z. B. Ungerer 1995) benutzt wird, ist **LRU**. Diese Abkürzung steht für *least recently used*. D. h., wenn ein Datum aus dem Cache verdrängt werden muss, wird jeweils das ausgewählt, für das am längsten keine Anforderung mehr auftrat.

Eine andere naheliegende Vorgehensweise ist **FIFO**, d. h. man verdrängt das Datum, das von den derzeit im Cache vorhandenen am frühesten angefordert wurde.

12.10 Daniel D. Sleator und Robert E. Tarjan (1985) haben gezeigt, dass **LRU** und **FIFO** k-kompetitiv sind. Andererseits folgt aus dem in Punkt 12.5 und Punkt 12.6 Gesagten, dass kein deterministischer Online-Algorithmus besser als k-kompetitiv sein kann. Die genannten Algorithmen sind also optimal.

12.2 Randomisierte Online-Algorithmen und Widersacher

12.11 Bei einem randomisierten Online-Algorithmus R geht in die Wahl des aus dem Cache zu verdrängenden Datums eine zufällige Komponente mit ein. Folglich ist die Zahl der Cache Misses nun eine *Zufallsvariable* $f_R(r_1, \ldots, r_n)$.

Im deterministischen Fall gibt der Kompetitivitätsfaktor eines Algorithmus A an, um wieviel mal A schlechter als der optimale Algorithmus schlimmstenfalls ist. Etwas Ähnliches möchte man auch im randomisierten Fall.

Eine einfache Möglichkeit bestände darin, z. B. den Erwartungswert von f_R wie in Definition 12.8 mit f_O von **LFD** zu vergleichen. In Satz 12.29 werden wir (ohne Beweis) sehen, warum man das *nicht* einfach so machen möchte.

12.12 Die Vorstellung, die sich für den Nachweis der Existenz solch ungünstiger Fälle eingebürgert hat, ist die, dass ein „böser Widersacher“ (engl. *adversary*) eine schlechte Anforderungsfolge erzeugt, dessen Länge er als Eingabe bekommt. Für den Widersacher selbst ist auch ein Verfahren festgelegt, nach dem die für ihn entstehenden Kosten $f_W(r_1, \ldots, r_n)$ bestimmt werden. Sie übernehmen die Rolle der Kosten von **LFD** im deterministischen Fall.

Es gibt verschiedene Varianten von Widersachern. Alle kennen den Programmtext des randomisierten Algorithmus R, gegen den sie arbeiten sollen. Es stellt sich nun aber auch die Frage, wieviel Information ihnen über die Werte von Zufallsbits zur Verfügung steht.

- Ein *unwissender Widersacher* W (der englische Begriff *oblivious adversary* heißt eigentlich vergesslicher Widersacher) ist einer, der *kein* Wissen über die Zufallsbits hat. Zu vorlegtem randomisierten Algorithmus R und n wird W also immer die gleiche Folge $(r_1, \ldots, r_n)$ erzeugen.
- Im Gegensatz dazu arbeitet ein sogenannter adaptiver Widersacher gegen eine konkrete Abarbeitung eines randomisierten Algorithmus R. Hat er bereits $(r_1, \ldots, r_i)$ erzeugt, so kann er für die Bestimmung von r_{i+1} auch auf die Information zurückgreifen, welche Zufallsbits R gewürfelt hat und welche Daten sich in Folge dessen momentan im Cache von R befinden.

Man stellt sich nun auch die Frage, womit man die Anzahl der Cache Misses des zu beurteilenden Algorithmus R vergleicht. Es hat sich eingebürgert, die folgenden Varianten zu betrachten:

- Bei einem unwissenden Widersacher werden die Kosten für die Bearbeitung der Anforderungsfolge mit dem optimalen deterministischen Algorithmus mit nur $f_O(r_1, \ldots, r_n)$ Cache Misses in Rechnung gestellt.
- Bei adaptiven Widersachern unterscheidet nun zwei Varianten.
  - Ein *adaptiver Online-Widersacher* (engl. *adaptive online adversary*) muss nach der Erzeugung jedes r_i sofort, also unter Benutzung eines Online-Algorithmus, entscheiden, welches andere Datum dadurch aus dem Cache verdrängt werden soll (falls das nötig ist).
  - Ein *adaptiver Offline-Widersacher* (engl. *adaptive offline adversary*) kann zunächst die ganze Anforderungsfolge $(r_1, \ldots, r_n)$ erzeugen. Für f_W werden die Kosten für die Bearbeitung dieser Anforderungsfolge durch den optimalen Offline-Algorithmus in Rechnung gestellt.

In beiden Fällen ist nun auch $f_O(r_1, \ldots, r_n)$ eine Zufallsvariable, denn die erzeugte Anfragefolge hängt von der Wahl der Zufallsbits von R ab. (Man mache sich klar, dass dies auch für adaptive Offline-Widersacher gilt.)

12.13 Definition

- Ein randomisierter Online-Algorithmus R ist C-*kompetitiv gegen unwissende Widersacher*, wenn es ein von n unabhängiges b gibt, so dass für jede Anforderungsfolge $(r_1, \ldots, r_n)$ gilt:

$$\mathbf{E}[f_R(r_1, \ldots, r_n)] - C \cdot f_O(r_1, \ldots, r_n) \leqslant b$$

  Das Infimum solcher C wird auch mit C_R^{obl} bezeichnet.
- Ein randomisierter Online-Algorithmus R ist C-*kompetitiv gegen einen adaptiven Online-Widersacher*, wenn es ein von n unabhängiges b gibt, so dass gilt:

$$\mathbf{E}[f_R(r_1, \ldots, r_n) - C \cdot f_W(r_1, \ldots, r_n)] \leqslant b$$

  Das Infimum solcher C wird auch mit C_R^{aon} bezeichnet.
- Ein randomisierter Online-Algorithmus R ist C-*kompetitiv gegen einen adaptiven Offline-Widersacher*, wenn es ein von n unabhängiges b gibt, so dass gilt:

$$\mathbf{E}[f_R(r_1, \ldots, r_n) - C \cdot f_O(r_1, \ldots, r_n)] \leqslant b$$

  Das Infimum solcher C wird auch mit C_R^{aof} bezeichnet. ◇

Bei den Online-Widersachern ergeben sich aus den Zufallsentscheidungen des randomisierten Algorithmus jeweils zufällige Anforderungsfolgen $(r_1, \ldots, r_n)$. Sie sind in den beiden zuletzt genannten Ungleichungen jeweils zweimal erwähnt, um deutlich zu machen, dass davon die Werte der Zufallsvariablen abhängen.

12.3 Seitenwechsel gegen einen unwissenden Widersacher

Wir beschreiben nun einen Algorithmus R, von dem gezeigt werden wird, dass er $2H_k$-kompetitiv gegen unwissende Widersacher ist. Anschließend werden wir nachweisen, dass er damit bis auf einen konstanten Faktor 2 an eine untere Schranke herankommt.

12.14 Algorithmus. (Marker-Algorithmus von Fiat u. a. (1991))

```
⟨Cache: Speicherstellen cache[i] mit 1 ⩽ i ⩽ k,⟩
⟨      die jeweils mit einem Markierungsbit mark[i] versehen sind.⟩

⟨Initialisierung:⟩
for i ← 1 to k do mark[i] ← 0 od

⟨Abarbeitung der Anforderungen:⟩
while ⟨noch weitere Anforderungen⟩ do
    r ← ⟨nächste Anforderung⟩
    if ⟨memory[r] ist nicht im Cache⟩ then
        if ⟨alle mark[i] = 1⟩ then ⟨alle mark[i] ← 0⟩ fi
```



```
        i ← ⟨zufällig gleichverteilt gewähltes j derer mit mark[j] = 0⟩
        cache[i] ← memory[r]
    else
        i ← ⟨Index mit cache[i] = memory[r]⟩
    fi
    mark[i] ← 1
od
```

12.15 Satz. *Algorithmus 12.14 ist* $2H_k$*-kompetitiv gegen unwissende Widersacher.*

12.16 Beweis. Der Ablauf des obigen Algorithmus zerfällt in sogenannte *Phasen*. Jede Phase beginnt mit dem Zurücksetzen aller Markierungsbits auf 0 und endet unmittelbar vor Beginn der nächsten Phase.

Wir untersuchen nun die Arbeit des optimalen Offline-Algorithmus und des Markeralgorithmus für eine Anforderungsfolge $r_1, r_2, \ldots, r_n$. Zu Beginn seien die Cacheinhalte für beide Algorithmen gleich und es führe r_1 zu einem Cache Miss. Folglich beginnt jede Phase mit einem Cache Miss. Geschieht dieser für Anforderung r_i, so ist die Anforderungsteilfolge für die gesamte Phase die maximale Teilfolge $r_i, \ldots, r_j$, während der genau k mal ein Markierungsbit auf 1 gesetzt wird.

Wir betrachten nun eine einzelne beliebige Phase. Ein Datum heiße *markiert*, wenn es *zum betrachteten Zeitpunkt* an einer markierten Stelle des Caches liegt. Der Markeralgorithmus entfernt bei einem Cache Miss stets ein nicht markiertes Datum aus dem Cache, und das den Cache Miss verursachende Datum ist ab dem Zeitpunkt, zu dem es in den Cache geladen wird, markiert.

Ein Datum heiße *veraltet*, wenn es *zu Beginn der Phase* im Cache ist und es heiße *sauber*, wenn es *zu Beginn der Phase nicht* im Cache ist. Die Anforderung eines sauberen Datums führt also auf jeden Fall zu einem Cache Miss. Die Anforderung eines veralteten Datums kann ebenfalls zu einem Cache Miss führen, nämlich dann, wenn es zwischenzeitlich (aufgrund eines anderen Cache Miss) aus dem Cache entfernt worden war. Die Anforderung eines veralteten Datums kann aber *nur einmal* innerhalb einer Phase zu einem Cache Miss führen, da es beim erneuten Laden markiert wird. Außerdem wird hierdurch wieder ein (nicht markiertes, also) veraltetes Datum verdrängt.

Es sei ℓ die Anzahl sauberer Datenelemente, die durch den Marker-Algorithmus im Laufe der Phase (unter Umständen mehrfach) angefordert werden.

Wir zeigen nun zweierlei:

1. Der optimale Offline-Algorithmus führt im Mittel pro Phase zu mindestens $\ell/2$ Cache Misses.
2. Beim Marker-Algorithmus ist die erwartete Anzahl von Cache Misses pro Phase ℓH_k.

Hieraus folgt die Behauptung des Satzes.

zu 1. Es bezeichne S_O die Menge der vom optimalen Algorithmus und S_M die der vom Marker-Algorithmus im Cache gehaltenen Elemente. Es sei d_a die Größe von $S_O \setminus S_M$ zu Beginn und d_e die Größe am Ende der Phase. Es sei m_O die Anzahl der Cache Misses des optimalen Algorithmus während der Phase.

Zu Beginn der Phase sind die ℓ später durch den Marker-Algorithmus angeforderten sauberen Datenelementen nicht in S_M. Und höchstens d_a von ihnen sind in S_O. Also ist $m_O \geqslant \ell - d_a$.

Am Ende der Phase besteht S_M genau aus den k markierten, also insbesondere auch während der Phase angeforderten Elementen. Da davon d_e am Ende nicht mehr in S_O sind, muss der optimale Algorithmus sie wieder verdrängt haben. Das kann nur durch den Cache Miss eines anderen Elementes geschehen sein. Als muss der optimale Algorithmus mindestens d_e Cache Misses erzeugt haben: $m_O \geqslant d_e$.

Insgesamt ist also $m_O \geqslant \max\{\ell - d_a, d_e\} \geqslant (\ell - d_a + d_e)/2$.

Der Wert d_e am Ende einer Phase ist der Wert d_a zu Beginn der nächsten. Über alle Phasen hinweg gesehen heben sich von den Brüchen die Summanden für d_a und d_e auf. Ausnahme sind der Wert d_a zu Beginn der ersten Phase, der nach Voraussetzung $d_a = 0$ ist, und der Wert d_e am Ende der letzten Phase. Man kann sich vorstellen, dass entsprechende Cache misses „benachbarten Phasen“ angerechnet werden. Dann bleiben in jeder Phase immer noch mindestens $m_O \geqslant \ell/2$ Cache misses.

zu 2. Die jeweils erste Anforderung eines der ℓ sauberen Elemente führt zu einem Cache Miss; die übrigen Anforderungen sauberer Elemente führen nicht zu einem Cache Miss. Alle anderen Anforderungen betreffen veraltete Elemente, von denen am Ende der Phase $k - \ell$ im Cache sind. Die erwartete Anzahl der hierdurch erzwungenen Cache Misses wird maximiert, wenn zuerst alle sauberen Elemente angefordert werden. Danach fehlen im Cache ℓ veraltete Elemente. Dies bleibt bis zum Ende der Phase so, denn durch Anforderung eines nicht mehr vorhandenen veralteten Datums x wird stets ein anderes veraltetes Datum x' verdrängt. Aber x wird bis zum Ende der Phase nicht mehr verdrängt.

Es seien nun $x_1, \ldots, x_{k-\ell}$ die am Ende der Phase im Cache befindlichen veralteten Elemente. Dabei sei x_1 dasjenige, das von allen x_i als erstes während der Phase angefordert wurde, x_2 dasjenige, das als zweites angefordert wurde, und so weiter. Die Wahrscheinlichkeit, dass bei der ersten Anforderung von x_i ($1 \leqslant i \leqslant k - \ell$) ein Cache Miss auftritt, ist gleich der Wahrscheinlichkeit, dass ein nicht im Cache vorhandenes veraltetes Element aus den veralteten Elementen, die in dieser Phase noch nicht angefordert wurden, ausgewählt wird. Es sind stets ℓ veraltete Elemente nicht im Cache. Und bei der ersten Anforderung von x_i wurden $k - (i-1)$ veraltete Elemente noch nicht angefordert. Die relative Häufigkeit eines Cache Miss ist also $\ell/(k-i+1)$.

Es ist $\sum_{i=1}^{k-\ell} \ell/(k-i+1) = \ell\left(\frac{1}{k} + \frac{1}{k-1} + \cdots + \frac{1}{l+1}\right) = \ell(H_k - H_\ell)$. Folglich ist die erwartete Anzahl Cache Misses kleiner oder gleich $\ell + \ell(H_k - H_\ell) = \ell H_k - (H_\ell - 1)\ell \leqslant \ell H_k$.

■

Der durch Algorithmus 12.14 gegebenen oberen Schranke für C_R^{obl} stellen wir nun eine untere Schranke gegenüber, die zuerst von Fiat u. a. (1991) gezeigt wurde.

12.17 Satz. *Es sei R ein randomisierter Algorithmus für das Seitenwechselproblem. Dann ist $C_R^{obl} \geqslant H_k = \sum_{i=1}^{k} 1/i \in \Theta(\log k)$.*

12.18 Beweis. Wir gehen in zwei Schritten vor:

1. Es sei $\mathbf{p}$ eine Wahrscheinlichkeitsverteilung für Folgen von Anforderungen. Für einen deterministischen Online-Algorithmus A für das Seitenwechselproblem sei seine Kompetitivität $C_A^{\mathbf{p}}$ unter $\mathbf{p}$ das Infimum aller C, so dass eine von n unabhängige Konstante b existiert mit $\mathbf{E}[f_A(r_1, \ldots, r_n)] - C \cdot \mathbf{E}[f_O(r_1, \ldots, r_n)] \leqslant b$ für alle Anforderungsfolgen $(r_1, \ldots, r_n)$.

Mit Hilfe von Überlegungen ganz ähnlich denen, die der Minimax-Methode von Yao zu Grunde liegen, kann man zeigen:

$$\inf_R C_R^{obl} = \sup_{\mathbf{p}} \inf_A C_A^{\mathbf{p}} .$$

Diese hier nicht bewiesene Tatsache werden wir nun benutzen,

2. und zwar so: Es wird eine Wahrscheinlichkeitsverteilung $\mathbf{p}$ für Anforderungsfolgen konstruiert, so dass für die Erwartungswerte gilt, dass der Algorithmus **LFD** (Offline!) H_k mal weniger Cache Misses hat als jeder deterministische Online-Algorithmus. Der Ausdruck $\inf_A C_A^{\mathbf{p}}$ auf der rechten Seite der Gleichung ist für dieses $\mathbf{p}$ also mindestens H_k.

   Die Größe des Cache werde wie immer mit k bezeichnet, und es werden $k+1$ verschiedene Anforderungen $I = \{a_1, \ldots, a_{k+1}\}$ benutzt. Die Wahrscheinlichkeitsverteilung $\mathbf{p}$ für die Anforderungsfolgen $(r_1, \ldots)$ ergibt sich durch die folgende „zufällige Konstruktion“ einer solchen Folge: Liegen $r_1, \ldots, r_{i-1}$ schon fest, so ergibt sich r_i, indem es zufällig gleichverteilt aus $I \setminus \{r_{i-1}\}$ ausgewählt wird. Außerdem werde r_1 zufällig gleichverteilt aus I ausgewählt.

   Jede Anforderungsfolge kann wie folgt in *Runden* aufgeteilt werden[1]: Die erste Runde beginnt mit r_1 und jede weitere Runde beginnt unmittelbar nach Ende der vorangegangenen Runde. Jede Runde endet unmittelbar *bevor zum ersten Mal jedes* der $k+1$ existierenden $a_j \in I$ mindestens einmal angefordert worden ist. Innerhalb einer Runde finden sich also stets Anforderungen von k verschiedenen a_j und unter diesen befindet sich *nicht* das erste angeforderte Element der nächsten Runde.

   O. B. d. A. führe r_1 zu einem Cache Miss. Algorithmus **LFD** entfernt aus seinem Cache immer dasjenige Element, das am spätesten in der Zukunft wieder angefordert wird. Dies ist also das Element, das als erstes in der zweiten Runde angefordert wird. Per Induktion ergibt sich, dass bei **LFD** in jeder Runde nur genau ein Cache Miss auftritt, nämlich bei der ersten Anforderung.

   Wie lange dauert eine Runde? Stellt man sich die $a_j \in I$ als Knoten des vollständigen Graphen K_{k+1} vor, dann entspricht wegen der oben beschriebenen Wahl der r_i jede Anforderungsfolge einem Random Walk in K_{k+1}. Der Erwartungswert für die Länge einer Runde ist gerade der Erwartungswert für die Anzahl Schritte, bis ein Random Walk, der an einem Knoten beginnt, jeden Knoten mindestens einmal besucht hat (also die sogenannte Überdeckungszeit). Dieser Erwartungswert ist für vollständige Graphen gerade kH_k.

   Wieviele Cache Misses erzeugt ein deterministischer Online-Algorithmus A? Zu jedem Zeitpunkt gibt es genau ein a_j, das nicht im Cache von A ist. Dieses a_j ist mit Wahrscheinlichkeit $1/k$ die nächste Anforderung. Also ist der Erwartungswert für die Anzahl Cache Misses während einer Runde (deren Länge Erwartungswert kH_k hat) dann gerade H_k, während in der gleichen Zeit **LFD** nur 1 Cache Miss erzeugt.

■

Ein Algorithmus, der noch um einen Faktor 2 besser ist als Algorithmus 12.14 wurde von McGeoch und Daniel Dominic Sleator (1991) angegeben. Er ist wegen des eben gezeigten Satzes 12.17 gegen unwissende Widersacher optimal.

[1] Dies ist mal wieder eine Stelle, an der wir bewusst vom von Motwani und Raghavan (1995, S. 375) gegebenen Beweis abweichen.

12.4 Einschub: Amortisierte Analyse

Solange dieser Abschnitt noch nicht (fertig) geschrieben ist, lese man folgende Dokumente:

- Robert Endre Tarjan (1985). „Amortized Computational Complexity“. In: *SIAM Journal Alg. Disc. Meth.* 6.2, S. 306–318
- Das „Handout“ http://www.ugrad.cs.ubc.ca/~cs320/2010W2/handouts/aa-nutshell.pdf zu der Vorlesung „CPSC 320: Intermediate Algorithm Design and Analysis“ von Patrice Belleville
- http://www.cs.princeton.edu/~fiebrink/423/AmortizedAnalysisExplained_Fiebrink.pdf
- Die “Lecture Notes” zu Lecture 16 (Amortized Analysis) der Vorlesung „Algorithms“ (CS357) gehalten von Vijaya Ramachandran im Frühjahr 2006, zu finden noch über www.archive.org durch Suche nach http://www.cs.utexas.edu/~vlr/s06.357/notes/lec16.pdf (und unter http://www.coursehero.com/file/4742/Lecture-16/).

Amortisierte Analyse ist eine Methode, um für eine ganze Folge von Operationen eine obere Schranke für die „Kosten“ für ihre Ausführung erhalten. Das Ziel ist dabei eine bessere Schranke zu finden als die Anzahl der Operationen mal schlimmste Kosten einer einzelnen Operation. Als Kosten wird häufig die Laufzeit herangezogen; im vorliegenden Kapitel geht es natürlich um die Anzahl Cache Misses.

Gegeben sei ein Stack, auf die üblichen Operationen Push und Pop sowie für $j \in \mathbb{N}_0$ „Multi-Pop-Operationen“ Mpop(j) ausgeführt werden können. Push und Pop mögen jeweils einen Schritt benötigen. Mpop(j) macht j Pop -Operationen (sofern der Stack groß genug ist) und benötigt daher im allgemeinen j Schritte.

Eine zugegebenermaßen *sehr* naive Worst-Case-Analyse für den Zeitbedarf einer Folge von n Operationen würde $n \cdot n$ liefern, da es Mpop -Operationen geben kann, die $\Omega(n)$ Zeit brauchen.

Der Zustand der Datenstruktur nach i Operationen wird mit D_i bezeichnet. Wir definieren nun eine sogenannte *Potenzialfunktion* $\Phi(i)$ $(= \Phi(D_i))$ für die Datenstruktur, die die folgenden Eigenschaften hat:

- $\Phi(0) = 0$
- für alle i ist $\Phi(i) \geqslant 0$

Im Falle des Stacks wähle man als $\Phi(i)$ die Größe des Stacks.

Es bezeichne nun r_i die *realen* Kosten der i-ten Operation und a_i ihre sogenannten *amortisierten* Kosten. Dabei ist per definitionem

$$a_i = r_i + \Phi(i) - \Phi(i-1)$$

Man kann sich $\Phi(i)$ als ein Guthaben auf einem Konto vorstellen (das man nicht überziehen darf). Ist $\Phi(i) > \Phi(i-1)$ dann zahlt man zusätzlich zu den realen Kosten etwas auf das Konto ein, und im Fall $\Phi(i) < \Phi(i-1)$ wird ein Teil der realen Kosten vom Guthaben abgehoben.

Die amortisierten Gesamtkosten sind dann

$$\begin{aligned}\sum_{i=1}^{n} a_i &= \sum_{i=1}^{n} (r_i + \Phi(i) - \Phi(i-1)) \\ &= \sum_{i=1}^{n} r_i + \sum_{i=1}^{n} \Phi(i) - \sum_{i=1}^{n} \Phi(i-1) \\ &= \sum_{i=1}^{n} r_i + \sum_{i=1}^{n} \Phi(i) - \sum_{i=0}^{n-1} \Phi(i) \\ &= \sum_{i=1}^{n} r_i + \Phi(n) - \Phi(0)\end{aligned}$$

Hieraus ergibt sich mit $\Phi(0) = 0$:

$$\sum_{i=1}^{n} r_i = \sum_{i=1}^{n} a_i - \Phi(n) \leqslant \sum_{i=1}^{n} a_i$$

Die realen Gesamtkosten sind also *höchstens* so hoch wie die amortisierten Kosten. Wie man gleich sehen wird, gibt es durchaus Fälle, in denen letztere einfacher und besser abzuschätzen sind, so dass man eine bessere obere Schranke für die realen Kosten erhält. Das Kunststück besteht darin, eine für den jeweiligen Anwendungsfall passende Potenzialfunktion zu finden.

Wie verändert sich bei den einzelnen Operationen das Potenzial im Beispiel mit dem Stack und wie groß sind die amortisierten Kosten?

- PUSH: Der Stack wächst von ℓ auf $\ell + 1$, also ist $a_i = 1 + (\ell + 1) - \ell = 2$.

  Der „Trick“ besteht hier also darin, für jedes PUSH zusätzlich zu den realen Kosten eine Einheit auf das Guthabenkonto „einzuzahlen“. Wofür das gut ist, sieht man gleich:

- POP: Der Stack schrumpft von ℓ auf $\ell - 1$, also ist $a_i = 1 + (\ell - 1) - \ell = 0$.

  Das Sparen hat sich gelohnt: Es ist bestimmt genug auf dem Konto, um das POP zu bezahlen. Das Gleiche gilt im letzten Fall:

- MPOP(j): Der Stack schrumpft von ℓ auf $\max(\ell - j, 0)$, also ist

$$\begin{aligned}a_i &= r_i + \max(\ell - j, 0) - \ell \\ &= \begin{cases} j + \ell - j - \ell & \text{falls } \ell \geqslant j \\ \ell - \ell & \text{falls } \ell < j \end{cases} \\ &= 0\end{aligned}$$

Damit sind also die amortisierten Kosten *aller* Operationen durch eine Konstante beschränkt. Folglich ist $\sum_{i=1}^{n} a_i \in O(n)$. Da außerdem $\Phi(n) \leqslant n$ gilt ist auch $\sum_{i=1}^{n} r_i \in O(n)$.

12.5 Seitenwechsel gegen adaptive Widersacher

12.19 Im folgenden wird die folgende Verallgemeinerung des Seitenwechselproblems betrachtet: Jedem Element x ist ein *Gewicht* $w(x) > 0$ zugeordnet. Wird ein Element x in den Cache geladen,

verursacht das Kosten in Höhe von $w(x)$. Der Gesamtaufwand für eine Anforderungsfolge ist die Summe der Kosten für die einzelnen Elemente.

Wählt man alle Gewichte identisch (z. B. zu 1), dann ergibt sich offensichtlich das weiter vorne betrachtete einfache Seitenwechselproblem.

Bei dem neuen Problem sind unter Umständen neue Algorithmen angebracht, denn es erscheint plausibel, bevorzugt Elemente mit kleinem Gewicht aus dem Cache zu verdrängen, da deren erneutes Laden „billiger“ ist.

12.20 Algorithmus. (Reziprok-Algorithmus) Sind $x_1, \ldots, x_k$ die zu einem Zeitpunkt im Cache befindlichen Elemente und muss eines verdrängt werden, dann wählt dieser Algorithmus dafür mit Wahrscheinlichkeit

$$\frac{1/w(x_i)}{\sum_{j=1}^{k} 1/w(x_j)}$$

Element x_i aus. „Leichte“ Elemente werden also bevorzugt.

Wir wollen nun zeigen:

12.21 Satz. *Der Reziprok-Algorithmus ist k-kompetitiv gegen adaptive Online-Widersacher.*

12.22 Beweis. (von Satz 12.21) Es bezeichne R den Reziprok-Algorithmus und W einen adaptiven Online-Widersacher. S_i^R sei die Menge der Elemente, die sich nach der Bearbeitung der i-ten Anforderung im Cache von R befinden, und S_i^W die Menge der Elemente, die sich nach der Bearbeitung der i-ten Anforderung im Cache von W befinden.

Es bezeichne f_i^R resp. f_i^W die durch den jeweiligen Algorithmus bei der Abarbeitung der i-ten Anforderung verursachten Kosten. Die Aufgabe besteht also darin, zu zeigen, dass

$$\sum_i \left(\mathbf{E}\left[f_i^R\right] - k\mathbf{E}\left[f_i^W\right] \right)$$

beschränkt ist. Dazu definieren wir zunächst eine sogenannte Potenzialfunktion

$$\Phi_i = \sum_{z \in S_i^R} w(z) - k \sum_{z \in S_i^R \setminus S_i^W} w(z)$$

und betrachten die Zufallsvariablen $X_i = f_i^R - k f_i^W - (\Phi_i - \Phi_{i-1})$ für $1 \leqslant i \leqslant n$. Für sie gilt:

$$\sum_i X_i = \Phi_0 - \Phi_n + \left(\sum_i f_i^R - k f_i^W \right) .$$

Daher genügt es, zu zeigen, dass $\mathbf{E}\left[\sum_i X_i\right] \leqslant 0$ ist.

Wir zeigen, dass sogar für jedes einzelne i gilt: $\mathbf{E}[X_i] \leqslant 0$. Dazu stellen wir uns vor, dass jede Anforderung zuerst von W und dann von R bearbeitet wird, und untersuchen die Veränderungen, die X_i bei der Bearbeitung der Anforderung durch W bzw. R nacheinander erfährt. Um die Veränderung von Φ nach dem ersten Teilschritt zu erfassen, definieren wir noch

$$\Psi_i = \sum_{z \in S_{i-1}^R} w(z) - k \sum_{z \in S_{i-1}^R \setminus S_i^W} w(z)$$

Man beachte die Indizes in den Definitionen von Φ_i und Ψ_i.

Es ist:

$$\begin{aligned} X_i &= f_i^R - k f_i^W - (\Phi_i - \Phi_{i-1}) \\ &= f_i^R - k f_i^W - ((\Phi_i - \Psi_i) + (\Psi_i - \Phi_{i-1})) \\ &= \left(f_i^R - (\Phi_i - \Psi_i)\right) + \left(-k f_i^W - (\Psi_i - \Phi_{i-1})\right) \end{aligned}$$

Um zu zeigen, dass $\mathbf{E}[X_i] \leqslant 0$ ist, beweisen wir nacheinander:

1. $\mathbf{E}\left[-k f_i^W - (\Psi_i - \Phi_{i-1})\right] \leqslant 0$
2. $\mathbf{E}\left[f_i^R - (\Phi_i - \Psi_i)\right] \leqslant 0$

1. Widersacher W: Bei einem Cache Hit ist $-k f_i^W + (\Psi_i - \Phi_{i-1}) = 0$.

Betrachten wir nun den Fall eines Cache Miss von W. Die Gesamtkosten ändern sich nur um einen konstanten Betrag, wenn man bei der Ersetzung eines Elementes x' durch durch ein neues Element x im Cache jeweils nicht die Kosten $w(x)$ sondern die Kosten $w(x')$ für W in Rechnung stellt. Wir nehmen daher an, dass in diesem Fall $f_i^W = w(x')$ und folglich $-k f_i^W = -k w(x')$ ist.

Welchen Wert hat $-(\Psi_i - \Phi_{i-1})$? Es ist $S_i^W = S_{i-1}^W \setminus \{x'\} \cup \{x\}$. Wir definieren noch die Menge $A = S_{i-1}^R \setminus (S_{i-1}^W \cup \{x\})$. Dann ist

$$\begin{aligned} -\Psi_i + \Phi_{i-1} &= -\sum_{z \in S_{i-1}^R} w(z) + k \sum_{z \in S_{i-1}^R \setminus S_i^W} w(z) + \sum_{z \in S_{i-1}^R} w(z) - k \sum_{z \in S_{i-1}^R \setminus S_{i-1}^W} w(z) \\ &= k \sum_{z \in S_{i-1}^R \setminus S_i^W} w(z) - k \sum_{z \in S_{i-1}^R \setminus S_{i-1}^W} w(z) \\ &= k \sum_{z \in A} w(z) + k \left\{ \begin{array}{ll} w(x') & \text{falls } x' \in S_{i-1}^R \\ 0 & \text{sonst} \end{array} \right\} - k \sum_{z \in A} w(z) - k \left\{ \begin{array}{ll} w(x) & \text{falls } x \in S_{i-1}^R \\ 0 & \text{sonst} \end{array} \right\} \\ &= k \left\{ \begin{array}{ll} w(x') & \text{falls } x' \in S_{i-1}^R \\ 0 & \text{sonst} \end{array} \right\} - k \left\{ \begin{array}{ll} w(x) & \text{falls } x \in S_{i-1}^R \\ 0 & \text{sonst} \end{array} \right\} \end{aligned}$$

Auch bei einem Cache Miss von W gilt folglich: $-k f_i^W - (\Psi_i - \Phi_{i-1}) \leqslant 0$.

2. Reziprokalgorithmus R**:** Bei einem Cache Miss ist $f_i^R = w(x)$. Das durch x aus dem Cache von R verdrängte Datum werde mit x'' bezeichnet. Dann gilt (man beachte, dass $x \in S_i^W$ ist):

$$\begin{aligned} -\Phi_i + \Psi_i &= -\sum_{z \in S_i^R} w(z) + k \sum_{z \in S_i^R \setminus S_i^W} w(z) + \sum_{z \in S_{i-1}^R} w(z) - k \sum_{z \in S_{i-1}^R \setminus S_i^W} w(z) \\ &= \sum_{z \in S_{i-1}^R} w(z) - \sum_{z \in S_i^R} w(z) + k \sum_{z \in S_i^R \setminus S_i^W} w(z) - k \sum_{z \in S_{i-1}^R \setminus S_i^W} w(z) \\ &= w(x'') - w(x) - k \left\{ \begin{array}{ll} w(x'') & \text{falls } x'' \in S_{i-1}^R \setminus S_i^W \\ 0 & \text{sonst} \end{array} \right\} \end{aligned}$$

Für die Bestimmung von $\mathbf{E}[-\Phi_i + \Psi_i]$ ist zu beachten, dass R ein Element x'' mit Wahrscheinlichkeit $(1/w(x''))/\sum_y 1/w(y)$ aus dem Cache verdrängt. Also ist

$$\begin{aligned} \mathbf{E}[-\Phi_i + \Psi_i] &= -w(x) + \sum_{x'' \in S_{i-1}^R} w(x'') \frac{1/w(x'')}{\sum_y 1/w(y)} - k \sum_{x'' \in S_{i-1}^R \setminus S_i^W} w(x'') \cdot \frac{1/w(x'')}{\sum_y 1/w(y)} \\ &= -w(x) + k \frac{1}{\sum_y 1/w(y)} - k \frac{|S_{i-1}^R \setminus S_i^W|}{\sum_y 1/w(y)} \end{aligned}$$

Unmittelbar vor dem Cache Miss von R ist $x \in S_i^W \setminus S_{i-1}^R$, also ist $\left|S_i^W \setminus S_{i-1}^R\right| \geqslant 1$. Da beide Caches gleich groß sind, ist folglich auch $\left|S_{i-1}^R \setminus S_i^W\right| \geqslant 1$ und damit

$$\mathbf{E}\left[f_i^R - \Phi_i + \Psi_i\right] = w(x) - w(x) + k\frac{1 - \left|S_{i-1}^R \setminus S_i^W\right|}{\sum_y 1/w(y)} \leqslant 0\,.$$

■

Der Wettbewerbsfaktor k im vorangegangenen Satz ist optimal. Im folgenden wird noch skizziert, auf welchem Wege man das zum Beispiel einsehen kann. Dazu betrachtet man ein weiteres Problem:

12.23 Problem. (k-Server-Problem) Auf k Punkten eines metrischen Raumes (M, d) befinden sich zu jedem Zeitpunkt je ein *Server*. Eine Anforderung besteht in der Angabe eines Punktes $x \in M$. Steht gerade ein Server auf x, ist nichts zu tun. Andernfalls muss ein Server von seinem momentanen Standpunkt y nach x bewegt werden. Die hierfür anfallenden Kosten sind gerade $d(x, y)$. Die Aufgabe eines Algorithmus besteht darin, den jeweils zu bewegenden Server so auszuwählen, dass die Gesamtkosten möglichst gering bleiben.

Koutsoupias (2009) gibt einen guten und aktuellen Überblick über das k-Server-Problem.

Man kann zeigen, dass das Seitenwechselproblem mit Gewichten als Spezialfall des k-Server-Problems aufgefasst werden kann. Für letzteres kann man die folgende untere Schranke zeigen, aus der auch folgt, dass der Reziprok-Algorithmus gegen adaptive Online-Widersacher optimal ist:

12.24 Satz. *Ist* R *ein randomisierter Online-Algorithmus, der das* k*-Server-Problem in jedem metrischen Raum löst, dann ist* $C_R^{aon} \geqslant k$.

12.25 Beweis. Es sei R ein randomisierter Online-Algorithmus und H eine Menge von $k+1$ Punkten des Raumes, die zu Beginn (und wie man sehen wird auch danach zu jedem Zeitpunkt) diejenigen k Punkte umfasst, auf denen R seine Server positioniert hat. Wir betrachten im folgenden die Anforderungsfolge $r_1, r_2, \ldots$ bei der jeweils der Punkt aus H als nächstes gewählt wird, auf dem R gerade *keinen* Server positioniert hat. Auf r_1 habe R zu Beginn keinen Server.

Die Kosten von R für die Bedienung von Punkt r_j sind also gerade $d(r_{j+1}, r_j)$. Die Gesamtkosten von R für eine Anforderungsfolge $(r_1, r_2, \ldots, r_n)$ sind also

$$M_R(r_1, \ldots, r_n) = \sum_{j=1}^{n-1} d(r_{j+1}, r_j) + x = \sum_{j=1}^{n-1} d(r_j, r_{j+1}) + x\,,$$

wobei hier x die Kosten für die Bearbeitung von r_n bezeichnet.

Um den Satz zu beweisen, zeigen wir nun, dass es k Online-Algorithmen $B_1, \ldots, B_k$ gibt, deren Kosten $\sum_{i=1}^{k} M_{B_i}(r_1, \ldots, r_n)$ für die Bearbeitung der Anforderungsfolge *zusammen höchstens* ebenfalls nur $M_R(r_1, \ldots, r_n)$ ist. Also gibt es jedenfalls mindestens ein i, für das die Kosten $M_{B_i}(r_1, \ldots, r_n) \leqslant 1/k \cdot M_R(r_1, \ldots, r_n)$ sind. Dies ist folglich auch eine Schranke für die Kosten des Widersachers.

Es sei $H = \{r_1, u_1, \ldots, u_k\}$. Für $1 \leqslant i \leqslant k$ beginnt Algorithmus B_i so, dass seine Server auf allen Punkten außer u_i positioniert sind. Wann immer B_i keinen Server auf r_j hat, bewegt er den von r_{j-1} nach r_j.

Wir zeigen nun:

1. Es bezeichne S_i die Menge der Punkte, an denen B_i seine Server positioniert hat. Dann gilt: Zu jedem Zeitpunkt ist für $i \neq m$ auch $S_i \neq S_m$.

2. Bei jeder Anforderung r_j muss nur einer der k Algorithmen einen Server nach r_j bewegen.
3. $\sum_{i=1}^{k} M_{B_i}(r_1, \ldots, r_n) \leqslant M_R(r_1, \ldots, r_n)$.

zu 1. Die Behauptung wird durch Induktion über die Zeit bewiesen. Bevor die erste Anforderung kommt, gilt die Behauptung nach Konstruktion.

Gelte die Behauptung nun, bevor irgendeine Anforderung r_j kommt. Das heißt dann aber, dass es nicht zwei B_i, B_m geben kann, die keinen Server auf r_j positioniert haben, denn sonst wäre $S_i = S_m$. Also ist entweder $r_j \in S_i \cap S_m$, so dass sich weder S_i noch S_m ändert, sie also verschieden bleiben. Oder es ist etwa $r_j \in S_i \setminus S_m$. Dann bewegt B_m seinen Server von r_{j-1} nach r_j, B_i aber nicht. Also ist hinterher $r_{j-1} \in S_i \setminus S_m$, so dass S_i und S_m wieder verschieden sind.

zu 2. Müssten zwei verschiedene Algorithmen, etwa B_i und B_m für ein r_j einen Server dorthin bewegen, dann wären S_i und S_m gleich, was wie eben gesehen nicht der Fall ist.

zu 3. Da für jedes r_j nur *ein* B_i einen Server dorthin bewegen muss, und zwar von r_{j-1}, liefert r_j tatsächlich nur einen Beitrag von $d(r_{j-1}, r_j)$ zu $\sum_{i=1}^{k} M_{B_i}(r_1, \ldots, r_n)$. Dieser Wert ist also $\sum_{j=2}^{n} d(r_{j-1}, r_j) = \sum_{j=1}^{n-1} d(r_j, r_{j+1})$.

Wenn der Widersacher mit gleicher Wahrscheinlichkeit eines der B_i als seine Cacheverwaltungsstrategie wählt, hat er also erwartete Kosten $M_R(r_1, \ldots)/k$. ∎

12.26 In der Praxis beobachtet man, dass verschiedene Online-Algorithmen z. B. für Seitenwechsel, die den gleichen Wettbewerbsfaktor haben, trotzdem unterschiedlich gut geeignet sind.

Das liegt salopp gesprochen daran, dass in der Realität auftretende Anforderungsfolgen nicht so willkürlich sind, wie sie ein Widersacher wählt. Es gibt mehrere Ansätze, dies zu modellieren. Karlin, S. J. Phillips und Raghavan (2000) verfolgen den Anzsatz, die Anforderungsfolgen von einer Markov-Kette „erzeugen zu lassen“. Ein verwandter, aber anderer Ansatz ist die Verwendung von *access graphs*; hierzu lese man die Arbeiten von Borodin, Irani u. a. (1995) und Irani, Karlin und S. Phillips (1996). Albers, Favrholdt und Giel (2005) berücksichtigen bei ihrer Analyse die Tatsache, dass aufeinander folgende Speicherzugriffe üblicherweise eine gewisse *Lokalität* aufweisen.

Wir beschließen dieses Kapitel mit einigen allgemeinen Ergebnissen zu adaptiven Widersachern, ohne noch auf die Beweise einzugehen. Daran Interessierte seien auf das Buch von Motwani und Raghavan (1995) und die dort angegebenen Literaturstellen verwiesen.

12.27 Die im folgenden auftretenden Konstanten C^{xyz} seien definiert als die Infima der C_R^{xyz}, genommen über alle randomisierten Algorithmen R für das Seitenwechselproblem.

Zunächst einmal ist aufgrund der Definitionen klar:

$$C^{obl} \leqslant C^{aon} \leqslant C^{aof} \leqslant C^{det} .$$

Weniger klar ist:

$$C^{aon} \geqslant \frac{C^{det}}{C^{obl}} = \Omega(k/\ln k) .$$

Das kann man folgern aus:

12.28 Satz. *Wenn R ein randomisierter Algorithmus ist, der α-kompetitiv gegen adaptive Online-Widersacher ist, und wenn es einen β-kompetitiven randomisierten Algorithmus gegen unwissende Widersacher gibt, dann ist R auch $\alpha\beta$-kompetitiv gegen adaptive Offline-Widersacher.*

Der letzte Satz kann so aufgefasst werden, dass adaptive Offline-Widersacher sehr stark sind. Gegen sie hilft Randomisierung nicht:

12.29 Satz. *Wenn es einen randomisierten Algorithmus gibt, der α-kompetitiv gegen jeden adaptiven Offline-Widersacher ist, dann gibt es auch einen deterministischen Algorithmus, der α-kompetitiv ist.*

Das erklärt dann auch, warum man bei randomisierten Algorithmen überhaupt die anderen Widersacher betrachtet: Andernfalls bringt Randomisierung nämlich gar nichts.

Zusammenfassung

1. Zur Beurteilung der Qualität von Onlinealgorithmen für das Seitenwechselproblem (und anderen) ist eine Worst-Case-Analyse sinnlos. Stattdessen ist die Betrachtung von Widersachern hilfreich.
2. Für kompetitive Analyse sind manchmal geschickt gewählte Potenzialfunktionen hilfreich.
3. Gegen unwissende Widersacher erreicht man randomisiert Wettbewerbsfaktoren in $O(\log n)$ während deterministisch $O(n)$ das Optimum ist.

Literatur

Albers, Susanne, Lene M. Favrholdt und Oliver Giel (2005). „On paging with locality of reference“. In: *Journal of Computer and System Sciences* 70.2, S. 145–175 (siehe S. 107).

Belady, L. A. (1966). „A study of replacement algorithms for virtual storage computers“. In: *IBM Systems Journal* 5.2, S. 78–101 (siehe S. 96).

Borodin, Allan, Sandy Irani u. a. (1995). „Competitive Paging with Locality of Reference“. In: *Journal of Computer and System Sciences* 50.2, S. 244–258 (siehe S. 107).

Borodin, Allan und Ran El-Yaniv (1998). *Online Computation and Competitive Analysis*. Cambridge Unviersity Press. ISBN: 0-521-56392-5 (siehe S. 95).

Fiat, A. u. a. (1991). „Competitive Paging Algorithms“. In: *Journal of Algorithms* 12.4, S. 685–699 (siehe S. 98, 100).

Irani, Sandy, Anna R. Karlin und Steven Phillips (1996). „Strongly Competitive Algorithms for Paging with Locality of Reference“. In: *SIAM Journal on Computing* 25.3, S. 477–497 (siehe S. 107).

Karlin, Anna R., Steven J. Phillips und Prabhakar Raghavan (2000). „Markov Paging“. In: *SIAM Journal on Computing* 30.3, S. 906–922 (siehe S. 107).

Koutsoupias, Elias (2009). „The k-server problem“. In: *Computer Science Review* 3.2, S. 105–118 (siehe S. 106).

Mattison, R. L. u. a. (1971). „Evaluation techniques for storage hierarchies“. In: *IBM Systems Journal* 9.2 (siehe S. 96).

McGeoch, Lyle A. und Daniel Dominic Sleator (1991). „A Strongly Competitive Randomized Paging Algorithm“. In: *Algorithmica* 6, S. 816–825 (siehe S. 101).

Motwani, Rajeev und Prabhakar Raghavan (1995). *Randomized Algorithms*. Cambridge University Press. ISBN: 0-521-47465-5 (siehe S. 101, 107).

Sleator, Daniel D. und Robert E. Tarjan (1985). „Amortized Efficiency of List Update and Paging Rules“. In: *Communications of the ACM* 28, S. 202–208 (siehe S. 96).

Tarjan, Robert Endre (1985). „Amortized Computational Complexity“. In: *SIAM Journal Alg. Disc. Meth.* 6.2, S. 306–318 (siehe S. 102).

Ungerer, Theo (1995). *Mikroprozessortechnik*. International Thomson Publishing (siehe S. 96).

