

## Competitive-Analyse

### Kapitel 4:

## Online-Algorithmen

### Hilfreiche Literatur:

- Borodin, El-Yaniv: Online Computation and Competitive Analysis, Cambridge University Press, 1998.
- Albers: Competitive Online Algorithms, Technical Report, BRICS, University of Aarhus, 1996.  
<http://www.informatik.uni-freiburg.de/~salbers/brics.ps.gz>

Online-Algorithmen sind dadurch gekennzeichnet, dass die Eingabe nicht vorab bekannt ist sondern erst nach und nach aufgedeckt wird. Dem Online-Algorithmus wird eine Anfragesequenz

$$\sigma = \sigma_1, \sigma_2, \dots, \sigma_m$$

präsentiert und der Algorithmus muss die Anfrage  $\sigma_t$  bedienen ohne die Zukunft, also die Anfragen  $\sigma_{t'}$  mit  $t' > t$ , zu kennen. Beim Bedienen der Anfrage entstehen Kosten, die von vorherigen Entscheidungen abhängen. Für einen on-line Algorithmus  $\mathcal{A}$  und eine Anfragesequenz  $\sigma$  bezeichne  $C_{\mathcal{A}}(\sigma)$  die Kosten von  $\mathcal{A}$  auf  $\sigma$ .

Sleator und Tarjan (1985) haben vorgeschlagen, die Qualität von Online-Algorithmen in Form eines Vergleiches mit der optimalen Strategie nachzuweisen. Bezeichne  $C^*(\sigma)$  die Kosten einer optimalen Strategie für die Anfragesequenz  $\sigma$ . Algorithmus  $\mathcal{A}$  wird als c-competitive bezeichnet, wenn es einen festen Wert  $a$  gibt, so dass für jede Anfragesequenz  $\sigma$  gilt

$$C_{\mathcal{A}}(\sigma) \leq c \cdot C^*(\sigma) + a .$$

Der Term  $c$  wird auch als Competitive-Faktor bezeichnet.

## Das File-Allocation-Problem (FAP)

Wir können eine Competitive-Analyse als ein Frage-Antwort-Spiel zwischen einem Online-Algorithmus und einem Gegner auffassen, wobei der Gegner die Anfragen möglichst so stellt, dass der Faktor  $\frac{C_A(\sigma) - a}{C^*(\sigma)}$  möglichst groß ist, und der Online-Algorithmus so antwortet, dass dieser Faktor möglichst klein ist.

Als Einführung in das Gebiet der Online-Algorithmen und Competitive-Analyse werden wir im folgenden exemplarisch zwei Probleme betrachten, nämlich das File-Allocation-Problem (mit uniformen Kosten auf zwei Knoten) und das Paging-Problem. In beiden Fällen geben wir untere und obere Schranken für den Competitive-Faktor.

Beim FAP geht es um die Verwaltung von Dateien in Computernetzwerken. Falls von vielen Nutzern lesend auf eine Datei zugegriffen wird, ist es sinnvoll viele lokale Kopien dieser Datei zu erzeugen, möglichst an denjenigen Netzwerkknoten von denen oder in deren Nähe die Datei gelesen wird. Im Falle von Schreibzugriffen, müssen jedoch alle Kopie aktualisiert oder invalidiert werden, so dass es günstiger ist wenige Kopien zu halten.

In dieser Vorlesung beschränken wir uns auf ein extrem einfaches Netzwerk, nämlich ein Netzwerk aus nur zwei Knoten,  $a$  und  $b$ , die durch eine Leitung miteinander verbunden sind. Eine Datei kann entweder auf dem Knoten  $a$ , dem Knoten  $b$  oder auf beiden Knoten gehalten werden. Es gibt also die Konfigurationen  $[a]$ ,  $[b]$  oder  $[ab]$ . Die Anfragesequenz  $\sigma$  beschreibt in welcher Reihenfolge von welchem Knoten gelesen oder geschrieben wird. Dabei steht  $\sigma_t = r(v)$  für eine Leseanfrage von Knoten  $v \in \{A, B\}$  zum Zeitpunkt  $t \geq 1$  und  $\sigma_t = w(v)$  für eine entsprechende Schreibenanfrage.

Der Online-Algorithmus muss die folgenden Spielregeln einhalten. Nachdem  $\sigma_t$  präsentiert ist, bedient der Algorithmus diese Anfrage, ohne zukünftige Anfragen  $\sigma_{t'}$  mit  $t' > t$  zu kennen. Wir kümmern uns nicht um die Details der Durchführung, sondern definieren stattdessen abstrakte Servicekosten für die einzelnen Zugriffe. Sei  $v \in \{a, b\}$ .

- Bei Anfrage  $\sigma_t = r(v)$  in Konfiguration  $[u]$ ,  $u \in \{a, b\} \setminus \{v\}$ , entstehen Servicekosten in Höhe einer Einheit;
- Bei Anfrage  $\sigma_t = w(v)$  in den Konfigurationen  $[ab]$  oder  $[u]$ ,  $u \in \{a, b\} \setminus \{v\}$ , entsteht eine Einheit Servicekosten.

Wir nehmen an, dass der Online-Algorithmus erst die Anfrage  $\sigma_t$  bedient und dann die Konfiguration anpasst. Beim Erzeugen einer neuen Kopie (nicht aber beim Löschen) entstehen zusätzlich Migrationskosten in Höhe von  $D \geq 1$  Einheiten. Der Parameter  $D$  wird damit begründet, dass die Migration einer ganzen Datei häufig wesentlich teurer ist als der Zugriff auf nur ein einzelnes Datum aus dieser Datei. In dieser Vorlesung vereinfachen wir und nehmen an  $D = 1$ , d.h. wir untersuchen eine vereinfachte Variante des allgemeinen FAP mit uniformen Kosten auf zwei Knoten.

## Obere Schranke für FAP

Wir zeigen, dass der folgende Online-Algorithmus 3-competitive ist. Der Algorithmus ändert die Konfiguration nur in den folgenden Fällen.

- Bei Anfrage  $\sigma_t = s(v)$ ,  $s \in \{r, w\}$ , in Konfiguration  $[u]$ ,  $u \neq v$ , gehe über zu Konfiguration  $[ab]$ .
- Bei Anfrage  $\sigma_t = w(v)$  in Konfiguration  $[ab]$  gehe über zu Konfiguration  $[v]$ .

Beachte, im ersten Fall entstehen Service- und Migrationskosten jeweils in Höhe einer Einheit, also werden zwei Kosteneinheiten zugerechnet. Im zweiten Fall entsteht nur eine Einheit Kosten, da das Löschen einer Kopie keine Migrationskosten verursacht. Alle anderen Arten von Zugriffen verursachen keine Konfigurationswechsel und auch keine Servicekosten.

**Satz 1** Der oben beschriebene Algorithmus für das uniforme FAP auf zwei Knoten ist 3-competitive.

**Beweis:** Zunächst beobachten wir, es gibt keine direkten Konfigurationsübergänge von  $[a]$  nach  $[b]$  oder umgekehrt, sondern nur Übergänge der Form  $[u] \rightarrow [ab]$  und  $[ab] \rightarrow [u]$  für  $u \in \{a, b\}$ .

O.B.d.A. starte unser Online-Algorithmus in der Konfiguration  $[ab]$ , und der erste Zugriff sei ein Schreibzugriff. (Falls das nicht der Fall ist, entstehen nur  $O(1)$  zusätzliche Kosten, die wir wegen des additiven Terms in der Definition des Competitive-Faktors vernachlässigen dürfen.) Der erste Zugriff endet dann in Konfiguration  $[a]$  oder  $[b]$ . Es folgen möglicherweise mehrere Zugriffe die in derselben Konfiguration enden, danach folgen Zugriffen, die in Konfiguration  $[ab]$  enden, dann wieder Zugriffe, die alle in  $[a]$  oder alle in  $[b]$  enden, usw.

Wir können somit die Anfragesequenz in disjunkte Phasen maximaler Länge einteilen, so dass die Zugriffe in einer Phase alle in derselben Konfiguration enden. Es folgt immer jeweils eine  $[ab]$ -Phase auf eine  $[a]$ - oder  $[b]$ -Phase. Eine Doppelphase bestehe aus eine  $[u]$ -Phase,  $u \in \{a, b\}$ , gefolgt von einer  $[ab]$ -Phase.

O.B.d.A. betrachte eine Doppelphase bestehend aus einer  $[a]$ -Phase gefolgt einer  $[ab]$ -Phase. Wir zeigen nun der Online-Algorithmus hat Kosten 3 in der Doppelphase:

- Der erste Zugriff in der  $[a]$ -Phase ist ein Schreibzugriff von Knoten  $a$  mit Servicekosten 1, aber ohne Migrationskosten, denn es wird nur eine Kopie gelöscht.
- Alle folgenden Zugriffe in der  $[a]$ -Phase sind Zugriffe von  $a$  und verursachen keine Kosten, denn der erste Zugriff von  $b$  würde bereits zur  $[ab]$ -Phase gehören.
- Der erste Zugriff in der  $[ab]$ -Phase geht vom Knoten  $b$  aus und hat Servicekosten 1 und Migrationskosten 1.
- Alle weiteren Zugriffe in der  $[ab]$ -Phase sind kostenlose Lesezugriffe, denn Schreibzugriffe würden bereits zur nächsten Doppelphase gehören.

Wir müssen nun nur noch zeigen, dass jeder Algorithmus Kosten mindestens 1 in der Doppelphase hat. Zum Zwecke des Widerspruchs, sei  $\mathcal{A}$  ein Algorithmus mit Kosten 0. Vor dem ersten Zugriff in der  $[a]$ -Phase, hält  $\mathcal{A}$  nur eine Kopie auf  $a$ , denn sonst würden Kosten für diesen Schreibzugriff von  $a$  anfallen.  $\mathcal{A}$  kann keine Kopien migrieren. Also ist der erste Zugriff in der  $[ab]$ -Phase nicht kostenlos, da dieser Zugriff vom Knoten  $b$  ausgeht. Ein Widerspruch!  $\square$

## Untere Schranke für FAP

**Satz 2** Es gibt keinen deterministischen Online-Algorithmus mit Competitive-Faktor besser als 3 für uniformes FAP auf zwei Knoten.

**Beweis** Fixiere einen beliebigen Online-Algorithmus  $\mathcal{A}$ . Die jeweils zuletzt durch  $\mathcal{A}$  erzeugte Kopie bezeichnen wir als neueste Kopie. Sei  $\sigma$  die Eingabesequenz, die immer auf demjenigen Knoten eine Schreibanfrage stellt, auf dem die neueste Kopie *nicht* liegt. Wir werden zeigen, dass gilt

$$C_{\mathcal{A}}(\sigma) \geq 3 \cdot C^*(\sigma) .$$

Hieraus folgt der Satz.

Für den Nachweis der obigen Ungleichung definieren wir die folgenden drei Strategien, die immer nur auf einem Knoten eine Kopie halten.

- Strategie 1 hält eine Kopie auf Knoten  $a$ .
- Strategie 2 hält eine Kopie auf Knoten  $b$ .
- Strategie 3 hält immer eine Kopie auf demjenigen Knoten, auf dem sich *nicht* die neueste Kopie von  $\mathcal{A}$  befindet.

*Vergleich der Migrationskosten:* Die ersten beiden Strategien verursachen keine Migrationskosten. Die Migrationskosten der dritten Strategie entsprechen genau den Migrationskosten von  $\mathcal{A}$ . Also haben die Strategien 1 bis 3 zusammen dieselben Migrationskosten wie Algorithmus  $\mathcal{A}$ .

*Vergleich der Servicekosten:* Algorithmus  $\mathcal{A}$  verursacht in jedem Schritt eine Einheit Servicekosten. In jedem Schritt verursacht entweder Strategie 1 oder Strategie 2 eine Einheit Servicekosten. Strategie 3 verursacht keinerlei Servicekosten. Also haben die Strategien 1 bis 3 zusammen auch dieselben Servicekosten wie Algorithmus  $\mathcal{A}$ .

Es folgt

$$C_{\mathcal{A}}(\sigma) = C_1(\sigma) + C_2(\sigma) + C_3(\sigma) \geq 3C^*(\sigma) ,$$

wobei die letzte Ungleichung sich aus  $C_i(\sigma) \geq C^*(\sigma)$ ,  $1 \leq i \leq 3$ , ergibt.  $\square$

## Das Paging-Problem

Wir betrachten ein Zwei-Schichten Speichersystem bestehend aus einem langsamen Speicher (Hauptspeicher) und einem schnellen Speicher (Cache). Der Cache ist wesentlich kleiner als der Hauptspeicher und hat nur Platz für  $k$  Speicherseiten (bzw. Cache-Lines). Eine Anfrage  $\sigma_t$  entspricht jeweils einer Seite. Falls diese Seite nicht im Cache vorhanden ist, so liegt ein Seitenfehler vor, d.h. die Seite muss aus dem Cache nachgeladen werden und gegebenenfalls muss eine andere Seite verdrängt werden. Ein Online-Algorithmus muss entscheiden, welche Seite verdrängt wird, ohne die zukünftigen Anfragen zu kennen.

Wir werden zunächst deterministische Verfahren untersuchen und dabei feststellen, dass keine wirklich überzeugenden Competitive-Faktoren möglich sind. Um den Gegner „auszutricksen“ benötigt man Randomisierung. Wir werden sehen, dass randomisierte Online-Algorithmen einen wesentlich besseren Competitive-Faktor erreichen als deterministische Verfahren. Dabei ist wichtig, dass sich die Online-Verfahren nicht vom Gegner „in die Karten schauen lassen“, d.h. die Zufallsbits sind geheim und dürfen nicht in die gegnerische Eingabesequenz einfließen.

### Bekanntes Paging-Verfahren (Online-Algorithmen)

- LRU (Least Recently Used): verdränge diejenige Seite, deren letzter Zugriff am längsten zurückliegt
- LFU (Least Frequently Used): verdränge diejenige Seite, die am seltensten nachgefragt wurde
- FIFO (First In First Out): verdränge diejenige Seite, die sich am längsten im Cache befindet
- LIFO (Last In First Out): verdränge diejenige Seite, die als letztes in den Cache geladen wurde
- RANDOM: verdränge eine uniform zufällig ausgewählte Seite aus dem Cache
- FWF (Flush When Full): entleere den Cache vollständig bei jedem Seitenfehler

### Ein optimales Verfahren (Offline-Algorithmus)

- LFD (Longest Forward Distance): verdränge diejenige Seite, deren nächster Zugriff am weitesten in der Zukunft liegt

## Obere Schranke für deterministisches Paging

Wir teilen die Anfragesequenz in sogenannte  $k$ -Phasen ein:

- Phase 0 bezeichnet die leere Teilsequenz vor  $\sigma_1$ .
- Phase  $i \geq 1$  ist die maximale Teilsequenz, die direkt im Anschluss an Phase  $i - 1$  startet, und in der auf höchstens  $k$  viele verschiedene Seiten zugegriffen wird.

(Wir erinnern uns,  $k$  bezeichnet die Größe des Caches.)

Ein Marking-Algorithmus assoziiert (implizit oder explizit) ein Bit mit jeder Seite, das angibt, ob die jeweilige Seite markiert oder unmarkiert ist. Zu Beginn einer Phase werden alle Seiten unmarkiert. Sobald in einer Phase auf eine Seite zugegriffen wird, so wird sie markiert. Ein Marking-Algorithmus ist nun dadurch gekennzeichnet, dass er niemals eine markierte Seite verdrängt.

**Satz 3** Jeder Marking-Algorithmus  $\mathcal{A}$  ist  $k$ -competitive.

**Beweis:**

Für jede Sequenz  $\sigma$  hat  $\mathcal{A}$  die folgenden Eigenschaften.

- Zu jeder Zeit sind alle markierten Seiten im Cache.
- Jede Markierung verursacht höchstens einen Fehler.
- In jeder Phase gibt es somit höchstens  $k$  Seitenfehler.

Andererseits hat jede Strategie pro Phase (bis auf die letzte Phase) mindestens einen Seitenfehler. Warum?

- Sei  $p$  die erste in Phase  $i$  angefragte Seite.
- Betrachte die Teilsequenz die mit dem zweiten Zugriff aus Phase  $i$  anfängt und mit dem ersten Zugriff aus Phase  $i + 1$  aufhört.
- Zu Beginn der Teilsequenz sind neben der Seite  $p$  noch höchstens  $k - 1$  andere Seiten im Cache.
- In der Teilsequenz gibt es Zugriffe auf  $k$  verschiedene Seiten, die alle auch verschieden von  $p$  sind.
- Also muß mindestens eine dieser Seiten pro Phase (bis auf die letzte Phase) nachgeladen werden.

Es folgt  $C_{\mathcal{A}}(\sigma) \leq k \cdot (C^*(\sigma) + 1) = k \cdot C^*(\sigma) + k$ . □

## Nicht-competitive Algorithmen

**Satz 4** LRU und FWF sind Marking-Algorithmen.

**Beweis:** Bei FWF werden genau die markierten Seiten im Cache gehalten. FWF ist also so etwas wie ein minimaler Marking-Algorithmus.

Um zu zeigen, dass LRU ein Marking-Algorithmus ist, betrachte eine beliebige  $k$ -Phase.

- Zu jedem Zeitpunkt in dieser Phase gibt es eine Menge von höchstens  $k$  markierten Seiten.
- LRU berechnet diese Markierungen nicht explizit, aber die markierten Seiten entsprechen genau denjenigen Seiten, auf die seit Beginn der Phase zugegriffen wurde.
- Die Markierungen kennzeichnen also diejenigen Seiten auf die zuletzt zugegriffen wurde.

Offensichtlich werden diese Seiten von LRU nicht verdrängt. Also ist LRU ein Marking-Algorithmus.  $\square$

**Korollar 5** LRU und FWF sind  $k$ -kompetitive.

Ein Online-Algorithmus wird als competitive bezeichnet, falls er einen Competitive-Faktor hat, der nicht von der Länge der Eingabesequenz abhängt.

**Satz 6** LFU ist nicht competitive.

**Beweis:** Betrachte die Sequenz

$$\sigma = p_1^\ell, p_2^\ell, \dots, p_{k-1}^\ell, (p_k, p_{k+1})^{\ell-1} .$$

Es gibt eine einfache Strategie für diese Sequenz mit höchstens  $k + 1$  Seitenfehlern. LFU hingegen verursacht in jedem Zugriff nach den ersten  $(k - 1)\ell$  vielen Zugriffen einen Fehler, und verursacht somit insgesamt mindestens  $2(\ell - 1)$  Fehler. Wir können den Competitive-Faktor von LFU also beliebig nach oben treiben, indem wir  $\ell$  genügend groß wählen.  $\square$

*Übungsaufgaben:*

- Beweise, FIFO ist *kein* Marking-Algorithmus.
- Beweise, FIFO ist (dennoch)  $k$ -competitive.
- Beweise, dass LIFO hingegen nicht competitive ist.

## Untere Schranke für deterministisches Paging

**Satz 7** Es gibt keinen deterministischen Online-Algorithmus für das Paging-Problem mit competitive-Faktor besser als  $k$ .

### Beweis:

Fixiere einen beliebigen Online-Algorithmus  $\mathcal{A}$ .  $\mathcal{A}$  ist  $c$ -competitive falls gilt, es gibt einen Term  $a$ , so dass für jede Anfragesequenz  $\sigma$  gilt

$$C_{\mathcal{A}}(\sigma) \leq c \cdot C^*(\sigma) + a .$$

Wir müssen nachweisen, dass  $\mathcal{A}$  nicht  $c$ -competitive ist für  $c < k$ . Dazu ist zu zeigen, dass für jeden Term  $a$  eine Anfragesequenz  $\sigma$  und ein Offline-Algorithmus  $\mathcal{B}$  existiert, so dass gilt

$$C_{\mathcal{A}}(\sigma) > c \cdot C_{\mathcal{B}}(\sigma) + a .$$

Die gegnerische Sequenz  $\sigma$  und der Offline-Algorithmus  $\mathcal{B}$  sehen wie folgt aus.

- Es gebe  $k + 1$  Seiten. Die ersten  $k$  Zugriffe gehen auf verschiedene Seiten. Danach haben  $\mathcal{A}$  und  $\mathcal{B}$  dieselben Seiten im Cache.
- Ab Zugriff  $k$  fordert der Gegner immer genau diejenige Seite an, die nicht im Cache von  $\mathcal{A}$  ist. Bei einer Sequenz der Länge  $k + m$  hat  $\mathcal{A}$  somit genau  $k + m$  Fehler.
- $\mathcal{B}$  hingegen, verdrängt immer die Seite, die in den nächsten  $k - 1$  Schritte nicht nachgefragt wird, und verursacht somit höchstens alle  $k$  Schritte einen Seitenfehler. Insgesamt macht das  $k + \lceil \frac{m}{k} \rceil$  Fehler für  $\mathcal{B}$ .

Für jedes vorgegebene  $c < k$  und jeden beliebigen Term  $a$  können wir offensichtlich  $m$  groß genug wählen, so dass gilt

$$k + m > c \cdot \left( k + \left\lceil \frac{m}{k} \right\rceil \right) + a .$$

Also ist  $\mathcal{A}$  nicht  $c$ -competitive für  $c < k$ . □

## Competitive-Analyse für randomisierte Online-Algorithmen

Die untere Schranke hat verdeutlicht, wo die Schwierigkeit bei der Entwicklung von Online-Algorithmen liegt. Wie können wir einen Gegner austricksen, der den Online-Algorithmus kennt? – Wir verwenden Zufallsbits und machen dadurch den Online-Algorithmus aus des Sicht des Gegners unvorhersehbar. Es ist wichtig, die Reihenfolge festzulegen, in der der Online-Spieler und der Gegner agieren.

- 1) Der randomisierte Online-Algorithmus  $\mathcal{A}$  wird bekanntgegeben, ohne die Zufallsbits zu bestimmen.
- 2) Dann präsentiert der Gegner eine Anfragesequenz  $\sigma$ .
- 3) Erst danach werden die Zufallsbits ausgewürfelt.

Ein derartiger Gegner, der den Algorithmus kennt aber nicht die Zufallsbits, wird als oblivious bezeichnet. Das Wort „oblivious“ ist schwer zu übersetzen und bedeutet so viel wie „außer acht lassen“. Beachte, wenn man die Reihenfolge der Schritte 2) und 3) vertauscht ist die Randomisierung bedeutungslos.

Wir müssen die Definition des Competitive-Faktors auf die neue Situation anpassen. Wir sagen ein randomisierter Online-Algorithmus  $\mathcal{A}$  ist  $c$ -competitive, falls es einen festen Wert  $a$  gibt, so dass für jede Anfragesequenz  $\sigma$ , die durch einen oblivious Gegner bestimmt ist, gilt

$$\mathbf{E} [C_{\mathcal{A}}(\sigma)] \leq c \cdot C^*(\sigma) + a .$$

Wir werden sehen, dass eine sehr einfache Randomisierung im Falle des Paging-Problems zu signifikant besseren Ergebnissen führt.

## Obere Schranke für randomisiertes Paging

Wir verwenden einen *randomisierte Marking-Algorithmus*, den wir MARK nennen. Bei einem Seitenfehler verdrängt MARK eine uniform zufällig ausgewählte nicht markierte Seite aus dem Cache.

**Satz 8** MARK ist  $(2H_k)$ -competitive.

### Beweis:

Betrachte eine beliebige  $k$ -Phase.

Zu jedem Zeitpunkt in der Phase unterscheiden wir die folgenden Arten von unmarkierten Seiten.

- Eine Seite, die unmarkiert ist, aber auf die in der vorherigen Phase zugegriffen wurde, wird als vorherige Seite bezeichnet.
- Eine Seite, die weder markiert noch vorherig ist, wird als frisch bezeichnet.

Sei  $f$  die Anzahl der frischen Seiten, die in dieser Phase nachgefragt werden.

Bezeichne OPT eine optimale Strategie.

**Lemma 9** Die amortisierten Kosten von OPT, die wir der betrachteten Phase zurechnen können, sind mindestens  $\frac{f}{2}$ .

**Beweis:** Die Seitenmengen, die in MARKs bzw. OPTs Cache abgespeichert sind, bezeichnen wir mit  $S$  bzw.  $S^*$ . Sei  $d$  der Wert von  $|S^* \setminus S|$  zu Beginn der Phase und  $d'$  dieselbe Größe am Ende der Phase.

- In der betrachteten Phase muss OPT mindestens  $f - d$  Seiten in den Cache nachladen, weil höchstens  $d$  der nachgefragten frischen Seiten zu Beginn der Phase in OPTs Cache sind.  
(Hier geht ein, dass der Cache von MARK am Anfang der Phase keine frischen Seiten enthält.)
- Außerdem verdrängt OPT mindestens  $d'$  Seiten, weil  $d'$  der in der Phase nachgefragten Seiten am Ende der Phase nicht im Cache von OPT sind.  
(Hier geht ein, dass der Cache von MARK am Ende der Phase alle in dieser Phase nachgefragten Seiten enthält.)

O.B.d.A sei OPT ein *fauler* Algorithmus, der Seiten nur dann verdrängt, wenn neue Seiten geladen werden.

Dann ist die Anzahl der Seitenfehler mindestens

$$\max\{f - d, d'\} \geq \frac{1}{2}(f - d + d') = \frac{f}{2} - \frac{d}{2} + \frac{d'}{2} .$$

Aufsummiert über alle Phasen verrechnen sich die Terme  $\frac{d}{2}$  und  $\frac{d'}{2}$  bis auf für die erste und letzte Phase. Also können wir der betrachteten Phase mindestens  $\frac{f}{2}$  Fehler zurechnen.

□

**Lemma 10** Die erwartete Anzahl der Fehler von MARK in der betrachteten Phase ist höchstens  $f \cdot H_k$ .

**Beweis:** MARK macht  $f$  Fehler für Zugriffe auf frische Seiten. Wieviele Fehler macht MARK für Zugriffe auf vorherige Seiten?

Am Anfang der Phase gibt es genau  $k$  vorherige Seiten. MARK muss  $s = k - f \leq k - 1$  Zugriffe auf verschiedene vorherige Seiten bedienen. Für  $i = 1, \dots, s$  sei  $p_i$  die  $i$ -te vorherige Seite, auf die zugegriffen wird, und  $t_i$  der Zeitpunkt unmittelbar vor dem Zugriff auf  $p_i$ .

- Zum Zeitpunkt  $t_i$  gibt es nur noch  $k - i + 1$  vorherige Seiten, weil die Seiten  $p_1, \dots, p_{i-1}$  ja bereits markiert wurden und somit nicht mehr als vorherige Seiten zählen.
- Es sind höchstens  $f$  Speicherplätze im Cache mit frischen Seiten gefüllt, d.h. höchstens  $f$  der vorherigen Seiten wurden verdrängt.

Zum Zeitpunkt  $t_i$  gibt es also  $k - i + 1$  vorherige Seiten von denen höchstens  $f$  viele Seiten verdrängt wurden. Jede vorherige Seite hat dieselbe Wkeit verdrängt zu werden. Deshalb wurde  $p_i$  höchstens mit Wkeit  $\frac{f}{k-i+1}$  verdrängt.

Somit sind die erwarteten Kosten beim Zugriff auf  $p_i$  höchstens

$$1 \cdot \frac{f}{k-i+1} + 0 \cdot \left(1 - \frac{f}{k-i+1}\right) = \frac{f}{k-i+1} .$$

Damit können wir die Kosten für Zugriffe auf vorherige Seiten folgendermaßen nach oben abschätzen.

$$\sum_{i=1}^s \frac{f}{k+1-i} \leq \sum_{i=2}^k \frac{f}{i} = f \cdot (H_k - 1) .$$

Zusammen mit den Kosten für Zugriffe auf frische Seiten macht das höchstens  $f \cdot H_k$  Seitenfehler.  $\square$

Satz 8 folgt jetzt unmittelbar aus den beiden Lemmas.  $\square$

Ein anderer randomisierter Paging-Algorithmus erreicht sogar den Competitive-Faktor  $H_k$ . Allerdings ist dieser Algorithmus wesentlich komplizierter. Eine untere Schranke zeigt, dass  $H_k$  tatsächlich der bestmögliche Competitive-Faktor für das Paging-Problem ist.

## Untere Schranke für randomisiertes Paging

**Satz 11** Es gibt keinen deterministischen Online-Algorithmus mit Competitive-Faktor besser als  $H_k$ .

**Beweis:** Fixiere einen randomisierten Online-Algorithmus  $\mathcal{A}$ . Es gebe  $k+1$  verschiedene Seiten. Die gegnerische Anfragesequenz  $\sigma$  besteht aus beliebig vielen  $k$ -Phasen. Beachte, da es insgesamt nur  $k+1$  viele Seiten gibt, macht ein optimaler Algorithmus nur einen Seitenfehler pro  $k$ -Phase (abgesehen von der ersten  $k$ -Phase, die wir ignorieren können).

Wir werden zeigen, dass der Gegner die Sequenz  $\sigma$  derart konstruieren kann, dass die erwarteten Kosten von  $\mathcal{A}$  pro  $k$ -Phase mindestens  $H_k$  beträgt. Wegen der Linearität des Erwartungswertes folgt somit der Satz. Zu jedem Zeitpunkt kann der Gegner bestimmen welche Seite der Algorithmus  $\mathcal{A}$  mit welcher Wahrscheinlichkeit im Cache hält. Bezeichne  $p_j$  die Wahrscheinlichkeit, dass die  $j$ -te Seite *nicht* im Cache von  $\mathcal{A}$  ist. Die Konstruktion von  $\sigma$  darf von diesen Wahrscheinlichkeiten nicht aber von den Zufallsbits abhängen.

Wir zeigen nun wie der Gegner eine  $k$ -Phase konstruiert. Der erste Zugriff sei beliebig vorgegeben. Die anderen Zugriffe und der erste Zugriff der nachfolgenden Phase werden wie folgt konstruiert. Wir teilen diese Zugriffe in  $k$  Subphasen ein. Wir stellen uns vor, dass der Gegner die Seiten genau wie ein Markingalgorithmus markiert. Zu Beginn der  $i$ -ten Subphase gibt es genau  $k - i + 1$  unmarkierte Seiten. Am Ende der  $k$ -ten Subphase soll die  $k + 1$  Seite markiert werden und die nächste Phase beginnt. Wir konstruieren die Subphasen derart, dass die erwarteten Kosten von  $\mathcal{A}$  für die  $i$ -te Subphase  $\frac{1}{k-i+1}$  betragen. Damit sind die erwarteten Kosten von  $\mathcal{A}$  in dieser Phase

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

Die einzelnen Subphasen werden wie folgt konstruiert. Jede Subphase besteht aus null oder mehreren Anfragen nach markierten Seiten gefolgt von einer Anfrage nach einer unmarkierten Seite. Sei  $M$  die Menge der markierten Seiten zu Beginn der  $i$ -ten Subphase. Es gilt  $|M| = i$  und die Anzahl der unmarkierten Seiten ist  $k - i + 1$ .

Sei  $\gamma = \sum_{j \in M} p_j$ . In Abhängigkeit von  $\gamma$  unterscheiden wir zwei Fälle.

*Fall 1:* Falls  $\gamma = 0$  ist, so sind mit Wahrscheinlichkeit 1 alle  $i$  markierten Seiten im Cache von  $\mathcal{A}$ . Der Cache enthält also höchstens  $k - i$  der  $k - i + 1$  unmarkierten Seiten. Also muss eine unmarkierte Seite  $a$  existieren, für die gilt

$$p_a \geq 1 - \frac{k-i}{k+1-i} = \frac{1}{k+1-i} .$$

Der Gegner fragt diese Seite an und die Subphase endet.

## Beweismethode: Potentialfunktion

*Fall 2:* Falls  $\gamma > 0$  ist, dann gibt es eine Seite  $m \in M$  mit  $p_m > 0$ . Setze  $\epsilon = p_m$ . Als erstes wird nun in dieser Subphase die Seite  $m$  nachgefragt. Die erwarteten Kosten für diesen Zugriff sind  $\epsilon$ . Solange die erwarteten Kosten von  $\mathcal{A}$  noch nicht  $\frac{1}{k+1-i}$  betragen und solange  $\gamma > \epsilon$ , wird dann diejenige Seite aus  $M$  angefragt, die mit größter Wahrscheinlichkeit nicht im Cache von  $\mathcal{A}$  ist.

Wir stellen zunächst fest, dass die obige Schleife terminiert. Da  $\gamma > \epsilon$  ist, und somit jede Iteration mit  $\frac{\gamma}{|M|} > \frac{\epsilon}{|M|}$  zu den erwarteten Kosten von  $\mathcal{A}$  beiträgt. Wenn die Schleife terminiert, sind entweder die erwarteten Kosten von  $\mathcal{A}$  groß genug oder  $\gamma \leq \epsilon$ . Im letzteren Fall fragt der Gegner diejenige unmarkierte Seite  $b$  an, die mit größter Wahrscheinlichkeit nicht im Cache ist. Es gilt  $p_b \geq \frac{1-\gamma}{k+1-i}$ . Damit sind die erwarteten Kosten von  $\mathcal{A}$  mindestens

$$\epsilon + p_b \geq \epsilon + \frac{1-\gamma}{k+1-i} \geq \epsilon + \frac{1-\epsilon}{k+1-i} \geq \frac{1}{k+1-i} .$$

□

Bisher basierten unsere Analysen im wesentlichen darauf, dass wir die Anfragesequenz in Phasen eingeteilt haben und dann die Kosten des Online- und des optimalen Offline-Algorithmus phasenweise verglichen haben. Teilweise mussten wir die Phasen jedoch ein wenig verschieben oder die Kosten über verschiedene Phasen amortisieren. Eine elegante Methode um Kosten zeitlich zu verlagern ist die sogenannte Potentialfunktionsmethode.

In einigen Schritten verursacht der Online-Algorithmus weniger Kosten, als der Competitive-Faktor erlaubt. Wir können uns vorstellen, diese gesparten Ausgaben auf einem Konto anzusammeln, um sie dann eventuell später für teurere Schritte einzusetzen. Die Potentialfunktion entspricht einer Invariante, die in jedem Schritt den Kontostand (das sogenannte Potential) beschreibt.

Die eigentliche Kunst bei der Analyse ist es die richtige Potentialfunktion zu finden. Kennt man die Funktion, so ist die Analyse häufig sehr einfach. Es ist Zeit für ein Anwendungsbeispiel.

**Asymmetrische Competitive-Modelle:** Um den Vorteil der Online-Strategie auszugleichen, schränken wir die optimale Strategie künstlich ein. Zu diesem Zweck verkleinern wir ihren Speicher um einen Faktor  $m$ , oder anders herum gesehen, wir vergrößern den Speicher des Online-Algorithmus um den Faktor  $m$ .

Wir sagen ein Online-Algorithmus mit einem Cache der Größe  $mk$  ist  $(m, c)$ -competitive, falls er  $c$ -competitive im Vergleich zu einer optimalen Strategie mit einem Cache der Größe  $k$  ist.

**Satz 12** RANDOM ist  $(2, 2)$ -competitive.

Diesen Satz werden wir mit der Potentialfunktionsmethode beweisen.

### Beweis von Satz 12:

Sei OPT eine faule, optimale Strategie.

Wir definieren das Potential  $\phi$  als die Anzahl derjenigen Seiten im Cache von OPT, die nicht auch im Cache von RANDOM gespeichert sind.  $C$  bezeichne die Kosten von RANDOM und  $C^*$  die Kosten von OPT. Als Invariante zeigen wir, nach Abarbeitung jeder Anfrage gilt:

$$\mathbf{E}[C] + 2\mathbf{E}[\phi] \leq 2C^* .$$

Da  $\phi$  per Definition nicht negativ ist, folgt  $\mathbf{E}[C] \leq 2C^*$ . Also ist RANDOM  $(2, 2)$ -competitive.

Betrachte eine beliebige Anfrage  $\sigma_t$ . Mit  $\Delta C, \Delta C^*, \Delta\phi$  bezeichnen wir die additiven Veränderungen der jeweiligen Kostenwerte sowie der Potentialfunktion durch die Abarbeitung von  $\sigma_t$ . Wir zeigen

$$\mathbf{E}[\Delta C] + 2\mathbf{E}[\Delta\phi] \leq 2\Delta C^* .$$

Da dieses für jede einzelne Anfrage gilt, folgt die Invariante. (Beachte  $\mathbf{E}[\Delta\phi]$  kann durchaus negativ sein.)

Wir betrachten vier Fälle in Abhängigkeit davon, ob RANDOM und OPT die angefragte Seite  $\sigma_t$  im Cache haben.

- Beide Strategien haben  $\sigma_t$  im Cache. Dann gilt  $\Delta C = \Delta\phi = \Delta C^* = 0$ .
- Keine der Strategien hat  $\sigma_t$  im Cache. Dann gilt  $\Delta C = \Delta C^* = 1$  und  $\mathbf{E}[\Delta\phi] \leq \frac{1}{2}$ , weil RANDOM mit Wkeit höchstens  $\frac{1}{2}$  eine der Seiten, die auch in OPTs Cache gespeichert sind, verdrängt.
- OPT hält  $\sigma_t$  im Cache, RANDOM nicht. Dann gilt  $\Delta C = 1$ ,  $\Delta C^* = 0$  und  $\mathbf{E}[\Delta\phi] \leq -\frac{1}{2}$ , weil einerseits der Unterschied zwischen den Caches von RANDOM und OPT um die Seite  $\sigma_t$  verringert wird, andererseits aber RANDOM mit Wkeit höchstens  $\frac{1}{2}$  eine andere der Seiten aus OPTs Cache verliert.
- RANDOM hält  $\sigma_t$  im Cache, OPT nicht. Dann gilt  $\Delta C = 0$ ,  $\Delta C^* = 1$  und  $\mathbf{E}[\Delta\phi] \leq 0$ .

In allen vier Fällen gilt  $\Delta C + 2\mathbf{E}[\Delta\phi] \leq 2\Delta C^*$ . Also folgt die Invariante und somit das Theorem.  $\square$