

Randomisierte Algorithmen

12. Randomisierte Online-Algorithmen am Beispiel des Seitenwechselproblems

Thomas Worsch

Fakultät für Informatik
Karlsruher Institut für Technologie

Wintersemester 2018/2019

Überblick

Allgemeines zu (deterministischen) Online-Algorithmen

Seitenwechselproblem und deterministische Algorithmen

Randomisierte Online-Algorithmen und Widersacher

Seitenwechsel gegen unwissende Widersacher

Einschub: amortisierte Analyse

Seitenwechsel gegen adaptive Widersacher

Überblick

Allgemeines zu (deterministischen) Online-Algorithmen

Seitenwechselproblem und deterministische Algorithmen

Randomisierte Online-Algorithmen und Widersacher

Seitenwechsel gegen unwissende Widersacher

Einschub: amortisierte Analyse

Seitenwechsel gegen adaptive Widersacher

Online-Algorithmus

Formalisierung (Borodin/El-Yaniv folgend)

- ▶ «Eingabe» als *Folge von Anforderungen* (engl. *requests*)
 $\sigma = (r_1, r_2, \dots, r_n) \in R^n$
- ▶ auf Anforderung r_i muss bearbeitet werden
 - ▶ *Antwort* $a_i = g_i(r_1, \dots, r_i) \in A$
 - ▶ *ohne Wissen der Zukunft* (r_{i+1}, \dots)
 - ▶ auch keine probabilistischen Annahmen
 - ▶ *unwiderruflich*
- ▶ also ALG festgelegt durch g_1, g_2, \dots
- ▶ *Kosten* festgelegt durch $\text{cost}_n : R^n \times A^n \rightarrow \mathbb{R}_{>0}$ (nie 0)
- ▶ für $\sigma \in R^n$ produziert ALG Ausgabe
 $\text{ALG}[\sigma] = (g_1(r_1), g_2(r_1, r_2), \dots, g_n(r_1, \dots, r_n))$
mit Kosten $\text{ALG}(\sigma) = \text{cost}_n(\sigma, \text{ALG}[\sigma])$

Competitive Analysis

Warum man nicht einfach die Kosten schlimmster Instanzen betrachten kann

- ▶ Beispiel Speicherverwaltung
 - ▶ schlimmste Anforderungsfolgen $\sigma \in R^n$ erzwingen
 - ▶ *in jedem Schritt* Seitenfehler
 - ▶ *für jeden Onlinealgorithmus*
- ▶ schlimmste Instanzen sind immer gleich schlimm

Competitive Analysis

Betrachte alle Instanzen und vergleiche mit «optimalem» Wettbewerber

- ▶ betrachte «*optimalen Offline-Algorithmus*» OPT »
- ▶ (jedenfalls, falls A endlich ist, existiert)
für $\sigma \in R^n$ $\text{OPT}(\sigma) = \min\{\text{cost}_n(\sigma, \tau) \mid \tau \in A^n\}$
- ▶ nur bei Kenntnis der *ganzen* Folge σ bestimmbar
- ▶ Aufgabe: Vergleiche ALG mit OPT!
- ▶ Frage: Wie?

- ▶ Beachte: hier nur *Minimierungsprobleme*

Wettbewerbsfaktor

für Onlinealgorithmus ALG heist

$$c_{\text{ALG}} = \sup\{\text{ALG}(\sigma)/\text{OPT}(\sigma) \mid \sigma \in R^+\}$$

der *Wettbewerbsfaktor* (engl. *competitive ratio*) von ALG

- ▶ sofern nicht ∞

Strikte c -Kompetitivität

der technisch angenehmere Fall

- ▶ Onlinealgorithmus ALG *strikt c -kompetitiv*, falls

$$\forall \sigma \in R^+ : \text{ALG}(\sigma) \leq c \cdot \text{OPT}(\sigma)$$

- ▶ da wir stets $\text{cost}(\sigma) > 0$ voraussetzen, äquivalent zu

$$\forall \sigma \in R^+ : \text{ALG}(\sigma) / \text{OPT}(\sigma) \leq c$$

- ▶ Wenn ALG strikt c -kompetitiv, dann
 - ▶ $c \geq 1$
 - ▶ ALG strikt c' -kompetitiv für jedes $c' \geq c$
- ▶ Wie klein kann man c machen?

Wettbewerbsfaktor und strikte Kompetitivität

Lemma

Es sei ALG ein Onlinealgorithmus und

$C = \{ c \mid \text{ALG ist strikt } c\text{-kompetitiv} \}$ sei nicht leer.

Dann ist

$$c_{\text{ALG}} = \inf C \quad \text{und} \quad c_{\text{ALG}} \in C .$$

Wettbewerbsfaktor und strikte Kompetitivität

$C = \{ c \mid \text{ALG ist strikt } c\text{-kompetitiv} \}$ und $c^{\text{inf}} = \inf C$

zeige: $c_{\text{ALG}} = c^{\text{inf}}$ und nebenbei auch $c_{\text{ALG}} \in C$

Beweis

\geq : für alle σ ist $\text{ALG}(\sigma)/\text{OPT}(\sigma) \leq c_{\text{ALG}}$

also $\text{ALG}(\sigma) \leq c_{\text{ALG}} \cdot \text{OPT}(\sigma)$

also ist ALG auch c_{ALG} -kompetitiv

also ist $c_{\text{ALG}} \in C$ und $c_{\text{ALG}} \geq c^{\text{inf}}$

Wettbewerbsfaktor und strikte Kompetitivität

$C = \{ c \mid \text{ALG ist strikt } c\text{-kompetitiv} \}$ und $c^{\text{inf}} = \inf C$

zeige: $c_{\text{ALG}} = c^{\text{inf}}$ und nebenbei auch $c_{\text{ALG}} \in C$

Beweis

\geq : für alle σ ist $\text{ALG}(\sigma)/\text{OPT}(\sigma) \leq c_{\text{ALG}}$

also $\text{ALG}(\sigma) \leq c_{\text{ALG}} \cdot \text{OPT}(\sigma)$

also ist ALG auch c_{ALG} -kompetitiv

also ist $c_{\text{ALG}} \in C$ und $c_{\text{ALG}} \geq c^{\text{inf}}$

\leq : indirekt: angenommen $c_{\text{ALG}} - c^{\text{inf}} = d > 0$

also $c' = c^{\text{inf}} + d/2 \in C$

also $\forall \sigma: \text{ALG}(\sigma) \leq c' \cdot \text{OPT}(\sigma)$

also $\sup_{\sigma} \text{ALG}(\sigma)/\text{OPT}(\sigma) \leq c'$

im Widerspruch zu $c' = c_{\text{ALG}} - d/2 < c_{\text{ALG}}$

Nicht-strikte c -Kompetitivität

- ▶ Onlinealgorithmus ALG *c -kompetitiv* für ein $c \in \mathbb{R}_{>0}$, falls Konstante $b \in \mathbb{R}$ (unabhängig von σ) existiert mit

$$\forall \sigma \in R^+ : \text{ALG}(\sigma) \leq c \cdot \text{OPT}(\sigma) + b$$

- ▶ im Vergleich zu *striker* c -Kompetitivität erlaubt man Ausnahmen
- ▶ Achtung:
 - ▶ wir *vermeiden* hier den Begriff Wettbewerbsfaktor bzw. competitive ratio
 - ▶ andere benutzen ihn weiter, *in unterschiedlichen Bedeutungen*

Überblick

Allgemeines zu (deterministischen) Online-Algorithmen

Seitenwechselproblem und deterministische Algorithmen

Randomisierte Online-Algorithmen und Widersacher

Seitenwechsel gegen unwissende Widersacher

Einschub: amortisierte Analyse

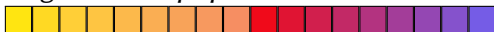
Seitenwechsel gegen adaptive Widersacher

12.2 Seitenwechselproblem

- ▶ Rechner mit *Cache der Größe k* und Hauptspeicher der Größe $N > k$.
- ▶ Anforderung r : Adresse einer Hauptspeicherzelle
Antwort a : aus Cache entferntes Element oder «nichts»
- ▶ Problem: Cache „zu klein“
 - ⇒ Konflikte
 - ⇒ Cache-Elemente werden verdrängt
 - ⇒ *Cache Misses* («Strafpunkt»)
- ▶ Ziel: deren Zahl soll klein bleiben
- ▶ Kosten bzw. Qualität eines Algorithmus A :
Anzahl $f_A(r_1, \dots, r_n)$ der Cache Misses für r_1, \dots, r_n


Problemstellung

- ▶ langsamer *Hauptspeicher* Größe N



- schneller *Cache* Größe $k < N$



- ▶ *Anforderung*: eine Seite r des Hauptspeichers: 
 - ▶ r im Cache: alles gut
 - ▶ r nicht im Cache: *Cache Miss, Page Fault, Strafpunkt*
 - ▶ *Antwort* des Systems:
 - ▶ verdränge an einer Stelle a im Cache
 - ▶ dort gespeicherte Seite durch P
- ▶ Welche Verdrängungsstrategie minimiert die «Kosten»?
- ▶ Online-Entscheidungen ohne Kenntnis der Zukunft

12.3 Offline-Algorithmen für das Seitenwechselproblem

► *Offline:*

Für die Auswahl des zu entfernenden Datums bei Cache-Miss für r_i kennt man auch alle noch folgenden Anforderungen r_j mit $j > i$.

12.3 Offline-Algorithmen für das Seitenwechselproblem

- ▶ *Offline:*
Für die Auswahl des zu entfernenden Datums bei Cache-Miss für r_i kennt man auch alle noch folgenden Anforderungen r_j mit $j > i$.
- ▶ Algorithmus LFD (*longest forward distance*):
Blick in die Zukunft
 - ▶ Entferne das Datum,
 - ▶ das am spätesten wieder benötigt wird.
- ▶ Dieser Algorithmus ist optimal.
(Belady 1967; Beweis nicht ganz banal)

12.4 Erinnerung

Anzahl insgesamt auftretender Cache Misses eines optimalen Offline-Algorithmus:

$$f_O(r_1, \dots, r_n)$$

12.5

- ▶ Betrachten den Fall $N = k + 1$.
- ▶ *In diesem Fall ist*

$$f_O(r_1, \dots, r_n) = f_{\text{LFD}}(r_1, \dots, r_n) \leq n/k$$

12.6 Deterministische Online-Algorithmen

- ▶ kein Blick in die Zukunft.
 - ▶ Verarbeitung von r_i ist unabhängig von r_{i+1}, \dots
- ▶ Da $N > k$, kann man bei vollem Cache bei *jedem* Zugriff einen Cache Miss erzwingen.
- ▶ Im Fall $N = k + 1$ gibt es also lange Anforderungsfolgen, bei denen jeder det. Online-Algorithmus k mal mehr Cache Misses produzieren *muss* als LFD.

12.9 Beispiele deterministischer Online-Algorithmen

12.9 Beispiele deterministischer Online-Algorithmen

- ▶ LRU
 - ▶ *least recently used*: verdränge das Datum, für das am längsten keine Anforderung mehr auftrat.
 - ▶ bei vielen Prozessoren mit mehrfach assoziativen First Level Caches benutzt
- ▶ FIFO
 - ▶ verdränge das Datum, das von den derzeit im Cache vorhandenen am frühesten angefordert wurde.

12.10 Bemerkung

- ▶ LRU und FIFO sind k -kompetitiv (Sleator/Tarjan, 1985).
- ▶ Wegen Punkt 12.5 und Punkt 12.7 kann kein deterministischer Online-Algorithmus besser als k -kompetitiv sein.
- ▶ Die Algorithmen sind also beide optimal, jedenfalls aus dieser (vereinfachten!) Sicht.
 - ▶ in der Praxis: LRU

Überblick

Allgemeines zu (deterministischen) Online-Algorithmen

Seitenwechselproblem und deterministische Algorithmen

Randomisierte Online-Algorithmen und Widersacher

Seitenwechsel gegen unwissende Widersacher

Einschub: amortisierte Analyse

Seitenwechsel gegen adaptive Widersacher

12.11

- ▶ Bei randomisierten Online-Algorithmus R ist die Zahl der Cache Misses eine **Zufallsvariable** $f_R(r_1, \dots, r_n)$.

12.12 Widersacher: verschieden miese Typen

- ▶ Widersacher
 - ▶ bekommen Eingabe n
 - ▶ erzeugen für R „schlimme“ Anforderungsfolgen der Länge n
 - ▶ die sie aber auch selbst verarbeiten müssen
- ▶ Wieviel Information ist über die Zufallsbits bekannt?
 - ▶ *unwissender Widersacher* W (engl. *oblivious adversary*)
 - ▶ *kein Wissen über erzeugte Zufallsbits*
 - ▶ Zu R und n erzeugt W immer gleiches (r_1, \dots, r_n) .
 - ▶ *adaptiver Widersacher*:
 - ▶ arbeitet gegen eine konkrete Abarbeitung von R ,
 - ▶ *kennt die von R erzeugten Zufallsbits* bei Abarbeitung von (r_1, \dots) ,
 - ▶ und folglich auch immer den aktuellen Cachezustand von R .

12.12 Widersacher: verschieden miese Typen (2)

Womit vergleicht man die Zahl der Cache Misses von R ?

- ▶ *unwissende Widersacher*: $f_O(r_1, \dots, r_n)$
- ▶ *adaptive Widersacher*: zwei Varianten
 - ▶ adaptiver Online-Widersacher:
 - ▶ muss auch selbst sofort nach Erzeugung eines r_i entscheiden, welches andere Datum ggf. verdrängt werden soll.
 - ▶ adaptiver Offline-Widersacher:
 - ▶ kann zunächst vollständig (r_1, \dots, r_n) erzeugen und
 - ▶ bekommt nur die Kosten des optimalen Offline-Algorithmus in Rechnung gestellt
- ▶ In beiden Fällen ist Anzahl der Cache Misses von W eine Zufallsvariable wegen Abhängigkeit von den Zufallsbits von R .

12.13 Definition (Wettbewerbsfaktor)

- ▶ R ist *C-kompetitiv gegen unwissende Widersacher*, wenn es ein von n unabhängiges b gibt so, dass für jede Anforderungsfolge (r_1, \dots, r_n) gilt:

$$\mathbb{E}[f_R(r_1, \dots, r_n)] - C \cdot f_O(r_1, \dots, r_n) \leq b$$

(manchmal Infimum der C mit C_R^{obl} bezeichnet)

- ▶ R ist *C-kompetitiv gegen einen adaptiven Offline- resp. Online-Widersacher*, wenn es ein von n unabhängiges b gibt so, dass gilt:

$$\begin{aligned} & \mathbb{E}[f_R(r_1, \dots, r_n) - C \cdot f_O(r_1, \dots, r_n)] \leq b \\ \text{resp.} \quad & \mathbb{E}[f_R(r_1, \dots, r_n) - C \cdot f_W(r_1, \dots, r_n)] \leq b \end{aligned}$$

(manchmal Infimum der C mit C_R^{aof} resp. C_R^{aon} bezeichnet)

Überblick

Allgemeines zu (deterministischen) Online-Algorithmen

Seitenwechselproblem und deterministische Algorithmen

Randomisierte Online-Algorithmen und Widersacher

Seitenwechsel gegen unwissende Widersacher

Einschub: amortisierte Analyse

Seitenwechsel gegen adaptive Widersacher

12.14 RANDMARK Algorithmus (Fiat et al., 1991)

12.14 RANDMARK Algorithmus (Fiat et al., 1991)

⟨Cache: cache[i], Markierungsbits mark[i], $1 \leq i \leq k$ ⟩
⟨Initialisierung:⟩
for $i \leftarrow 1$ **to** k **do** $mark[i] \leftarrow 0$ **od**
⟨Abarbeitung der Anforderungen:⟩
while *⟨noch weitere Anforderungen⟩* **do**
 $r \leftarrow$ *⟨nächste Anforderung⟩*
 if *⟨memory[r] nicht in cache⟩* **then**
 if *⟨alle mark[i] = 1⟩* **then** *⟨alle mark[i] \leftarrow 0⟩* **fi**
 $i \leftarrow$ *⟨zufälliges j mit mark[j] = 0⟩*
 $cache[i] \leftarrow memory[r]$
 else
 $i \leftarrow$ *⟨Index mit cache[i] = memory[r]⟩*
 fi
 $mark[i] \leftarrow 1$
od

12.15 Satz

RANDOMARK ist $2H_k$ -kompetitiv gegen unwissende Widersacher.

Beachte:

- ▶ $2H_k \in \Theta(\log k)$
- ▶ jeder det. Online-Alg. ist bestenfalls k -kompetitiv

12.16 Beweis

- ▶ Algorithmus zerfällt *Phasen*:
 - ▶ Jede Phase beginnt mit Zurücksetzen aller Markierungsbits auf 0
 - ▶ und endet unmittelbar vor der nächsten.
- ▶ Betrachte LFD und RANDMARK für Anforderungsfolge r_1, r_2, \dots
- ▶ Anfangs gleiche Cacheinhalte und r_1 führe zu Cache Miss.
- ▶ Also:
 - ▶ jede Phase beginnt mit Cache Miss,
 - ▶ umfasst maximale Teilfolge r_i, \dots, r_j
 - ▶ mit genau k verschiedenen Adressen,
 - ▶ denn jede in der Teilfolge „neue“ Adresse führt dazu,
 - ▶ dass ein Markierungsbit auf 1 gesetzt wird, und
 - ▶ Phase endet direkt vor $(k + 1)$ -ter neuer Adresse.

Beweis (2)

Zeige:

1. LFD hat im Mittel pro Phase $\geq \ell/2$ Cache Misses.
2. RANDOMMARK hat erwartet pro Phase ℓH_k Cache Misses.

Definition von ℓ kommt gleich

Beweis (3)

betrachte einzelne Phase

ein Datum heie

- ▶ *veraltet*, wenn es zu *Beginn der Phase* im Cache ist
- ▶ *sauber*, wenn es zu Beginn der Phase *nicht* im Cache ist
- ▶ *markiert*, wenn es *zum betrachteten Zeitpunkt* an einer markierten Stelle des Caches liegt

Beweis (4)

- ▶ **RANDOM** entfernt aus dem Cache stets ein nicht markiertes, veraltetes Datum und
- ▶ das den Cache Miss verursachende Datum ist ab dem Zeitpunkt, zu dem es geladen wird, markiert.
- ▶ Innerhalb einer Phase führt die erste Anforderung jedes sauberen Datums zu Cache Miss.
- ▶ ℓ : Anzahl sauberer Datenelemente, die durch **RANDOM** im Laufe der Phase (evtl. mehrfach) angefordert werden.

Beweis (5)

Anforderung eines veralteten Datums kann zu Cache Miss führen

- ▶ wenn es zwischenzeitlich (aufgrund eines anderen Cache Miss) aus dem Cache entfernt worden war.
- ▶ Das 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.

Beweis (6)

Abschätzung Cache Misses bei LFD

- ▶ S_O : Menge der Cacheelemente von LFD
 S_R : Menge der Cacheelemente von RANDMARK
- ▶ d_a : Größe von $S_O \setminus S_R$ am Anfang der Phase
 d_e : Größe von $S_O \setminus S_R$ am Ende der Phase
- ▶ m_O : Anzahl der Cache Misses von LFD während der Phase.

Beweis (6)

Abschätzung Cache Misses bei LFD

- ▶ S_O : Menge der Cacheelemente von LFD
 S_R : Menge der Cacheelemente von RANDMARK
- ▶ d_a : Größe von $S_O \setminus S_R$ am Anfang der Phase
 d_e : Größe von $S_O \setminus S_R$ am Ende der Phase
- ▶ m_O : Anzahl der Cache Misses von LFD während der Phase.

Zu Beginn der Phase:

- ▶ ℓ später angeforderte saubere Datenelemente nicht in S_R
- ▶ höchstens d_a von ihnen sind in S_O
- ▶ also $m_O \geq \ell - d_a$

Beweis (7)

Abschätzung Cache Misses von LFD (2)

Am Ende der Phase:

- ▶ S_R enthält genau die k markierten, also insbesondere auch während der Phase angeforderten Elemente.
- ▶ Davon sind d_e am Ende nicht mehr in S_O , sie sind also vom optimalen Algorithmus verdrängt worden.
- ▶ Das kann nur durch den Cache Miss eines anderen Elementes geschehen sein.
- ▶ Als muss der optimale Algorithmus mindestens d_e Cache Misses erzeugt haben.
- ▶ Also $m_O \geq d_e$

Beweis (8)

Abschätzung Cache Misses von LFD (3)

Insgesamt:

$$\blacktriangleright m_O \geq \max\{\ell - d_a, d_e\} \geq (\ell - d_a + d_e)/2.$$

Beweis (8)

Abschätzung Cache Misses von LFD (3)

Insgesamt:

$$\blacktriangleright m_O \geq \max\{\ell - d_a, d_e\} \geq (\ell - d_a + d_e)/2.$$

Summation über alle Phasen:

- ▶ d_e am Ende einer Phase ist d_a zu Beginn der nächsten.
- ▶ die Summanden für d_a und d_e heben sich auf,
 - ▶ außer zu Beginn der ersten Phase: $d_a = 0$
 - ▶ und am Ende der letzten Phase d_e
- ▶ Vorstellung:
 - ▶ schiebe Cache Misses in „benachbarte Phasen“
 - ▶ immer noch $m_O \geq \ell/2$ Cache Misses in jeder Phase

Beweis (9)

Abschätzung Cache Misses von RANDMARK (1)

- ▶ jeweils erste Anforderung eines der ℓ sauberen Elemente
 - ▶ führt zu Cache Miss
 - ▶ übrige Anforderungen sauberer Elemente nicht
- ▶ alle anderen Anforderungen: veraltete Elemente
- ▶ am Ende der Phase sind $k - \ell$ davon im Cache.
- ▶ erwartete Anzahl dadurch erzwungener Cache Misses maximieren: erst alle sauberen Elemente angefordern.
- ▶ danach fehlen im Cache ℓ veraltete Elemente
- ▶ das bleibt bis zum Ende der Phase so, denn
 - ▶ durch Anforderung eines fehlenden veralteten Datums x wird stets ein anderes veraltetes Datum x' verdrängt,
 - ▶ aber x wird bis zum Ende der Phase nicht mehr verdrängt.

Beweis (10)

Abschätzung Cache Misses von RANDMARK (2)

- ▶ Seien $x_1, \dots, x_{k-\ell}$ die am Ende der Phase im Cache befindlichen veralteten Elemente.
- ▶ x_1 zuerst angefordert, dann x_2 , usw.
- ▶ W.keit für Cache Miss bei 1. Anforderung von $x_i, i \leq k - \ell$:
gleich der W.keit, ein nicht im Cache vorhandenes veraltetes Element auszuwählen aus den veralteten Elementen, die in dieser Phase noch nicht angefordert wurden.
- ▶ stets ℓ veraltete Elemente nicht im Cache
- ▶ bei erster Anforderung von x_i wurden $k - (i - 1)$ veraltete Elemente noch nicht angefordert
- ▶ relative Häufigkeit eines Cache Miss also $\ell / (k - i + 1)$

Beweis (11)

Abschätzung Cache Misses von RANDMARK (3)

- ▶ Es ist

$$\sum_{i=1}^{k-\ell} \frac{\ell}{(k-i+1)} = \ell \left(\frac{1}{k} + \frac{1}{k-1} + \dots + \frac{1}{\ell+1} \right) = \ell(H_k - H_\ell) .$$

- ▶ also erwartete Anzahl Cache Misses kleiner oder gleich

$$\ell + \ell(H_k - H_\ell) = \ell H_k - (H_\ell - 1)\ell \leq \ell H_k .$$

12.17 Satz

Ist R ein C -kompetitiver Algorithmus für das Seitenwechselproblem gegen unwissende Widersacher, dann ist

$$C \geq H_k .$$

12.18 Beweis

Wir gehen in zwei Schritten vor:

- ▶ eine Variante des Minimax-Prinzips von Yao
- ▶ Benutzung derselben

12.18 Beweis (2)

Variante des Minimax-Prinzips von Yao

- ▶ \mathbf{p} : Wahrscheinlichkeitsverteilung für Folgen von Anforderungen
- ▶ für det. Online-Algorithmus A sei $C_A^{\mathbf{p}}$ unter \mathbf{p} das Infimum aller C , so dass eine von n unabhängige Konstante b existiert mit

$$\mathbf{E}[f_A(r_1, \dots, r_n)] - C \cdot \mathbf{E}[f_O(r_1, \dots, r_n)] \leq b$$

für alle Anforderungsfolgen (r_1, \dots, r_n) .

- ▶ ähnlich der Minimax-Methode von Yao gilt:

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

12.18 Beweis (3)

Benutzung dieser Idee:

- ▶ Konstruiere Wahrscheinlichkeitsverteilung für Anforderungsfolgen so, dass für die Erwartungswerte gilt, dass **der Offline-Algorithmus LFD H_k mal weniger Cache Misses hat als jeder deterministische Online-Algorithmus.**
- ▶ Die Größe des Cache: k
- ▶ $k + 1$ verschiedene Anforderungen $I = \{a_1, \dots, a_{k+1}\}$.
- ▶ Verteilung \mathbf{p} durch folgende „zufällige Konstruktion“ gegeben:
 - ▶ r_1 wird zufällig gleichverteilt aus I gewählt.
 - ▶ Zu r_1, \dots, r_{i-1} wird r_i zufällig gleichverteilt aus $I \setminus \{r_{i-1}\}$ gewählt.

12.18 Beweis (4)

- ▶ jede Anforderungsfolge in *Runden* aufgeteilt:
 - ▶ Erste Runde beginnt mit r_1 und jede weitere Runde beginnt unmittelbar nach Ende der vorangegangenen.
 - ▶ Jede Runde endet unmittelbar **bevor zum ersten Mal jedes** der $k + 1$ existierenden $a_j \in I$ mindestens einmal angefordert worden ist.
- ▶ Algorithmus LFD entfernt aus seinem Cache immer dasjenige Element, das am spätesten in der Zukunft wieder angefordert wird.
- ▶ O. B. d. A. verursache r_1 einen Cache Miss.
- ▶ Induktion: In jeder Runde erfährt LFD genau einen Cache Miss, nämlich bei der jeweils ersten Anforderung.

12.18 Beweis (5)

Cache Misses eines det. Online-Algorithmus A in einer Runde?
(während bei LFD nur 1 Cache Miss)

12.18 Beweis (5)

Cache Misses eines det. Online-Algorithmus A in einer Runde?
(während bei LFD nur 1 Cache Miss)

- ▶ Dauer einer Runde:
 - ▶ $a_j \in I$ Knoten des vollständigen Graphen K_{k+1} :
Anforderungsfolge entspricht Random Walk in K_{k+1}
 - ▶ Erwartungswert für Länge einer Runde ist die sogenannte *Überdeckungszeit*.
 - ▶ Übungsaufgabe: für K_{k+1} ist das kH_k
- ▶ W.keit für Cache Miss pro Anforderung:

12.18 Beweis (5)

Cache Misses eines det. Online-Algorithmus A in einer Runde?
(während bei LFD nur 1 Cache Miss)

- ▶ Dauer einer Runde:
 - ▶ $a_j \in I$ Knoten des vollständigen Graphen K_{k+1} :
Anforderungsfolge entspricht Random Walk in K_{k+1}
 - ▶ Erwartungswert für Länge einer Runde ist die sogenannte *Überdeckungszeit*.
 - ▶ Übungsaufgabe: für K_{k+1} ist das kH_k
- ▶ W.keit für Cache Miss pro Anforderung:
 - ▶ stets genau ein a_j nicht im Cache von A ,
das mit Wahrscheinlichkeit $1/k$ nächste Anforderung
 - ▶ also Erwartungswert für Anzahl Cache Misses pro Runde:
 $kH_k \cdot 1/k = H_k$

Überblick

Allgemeines zu (deterministischen) Online-Algorithmen

Seitenwechselproblem und deterministische Algorithmen

Randomisierte Online-Algorithmen und Widersacher

Seitenwechsel gegen unwissende Widersacher

Einschub: amortisierte Analyse

Seitenwechsel gegen adaptive Widersacher

Amortisierte Analyse

- ▶ Idee von Tarjan
- ▶ Methode, um für ganze Folge von Operationen eine obere Schranke für die „Ausführungskosten“ erhalten.
- ▶ Ziel: schärfere Schranke als Anzahl der Operationen mal schlimmste Kosten einer Operation

Beispiel: Stack

- ▶ Operationen
 - ▶ PUSH: 1 Schritt
 - ▶ POP: 1 Schritt
 - ▶ „multi-pop“ MPOP(j):
 - ▶ macht j POP Operationen
 - ▶ braucht j Schritte

sehr naive Worst-Case-Analyse:

- ▶ n Operationen brauchen $O(n \cdot n)$ Schritte

Stack: bessere Laufzeitabschätzung

- ▶ D_i : Zustand der Datenstruktur nach i Operationen
- ▶ definiere sogenannte *Potenzialfunktion* $\Phi(i)$ ($= \Phi(D_i)$)
- ▶ mit folgenden Eigenschaften:
 - ▶ $\Phi(0) = 0$
 - ▶ für alle i ist $\Phi(i) \geq 0$

- ▶ Finden eines geeigneten Φ ist die große Kunst

Reale und amortisierte Kosten

- ▶ r_i : die *realen* Kosten der i -ten Operation
- ▶ a_i : die *amortisierten* Kosten

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

- ▶ Vorstellung:
 - ▶ $\Phi(i)$: Guthaben auf einem Konto (nicht überziehbar)
 - ▶ wenn
 - ▶ $\Phi(i) > \Phi(i - 1)$: neben realen Kosten
Einzahlung von $\Phi(i) - \Phi(i - 1)$ auf das Konto
 - ▶ $\Phi(i) < \Phi(i - 1)$: Teil der realen Kosten
vom Guthaben auf Konto beglichen

Amortisierte Gesamtkosten

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

Wegen $\Phi(0) = 0$ und $\Phi(i) \geq 0$ für alle i :

$$\sum_{i=1}^n r_i \leq \sum_{i=1}^n a_i$$

Amortisierte Gesamtkosten

- ▶ reale Gesamtkosten *höchstens* so hoch wie amortisierte Gesamtkosten
 - ▶ aber amortisierte Kosten manchmal einfacher und besser abzuschätzen
- ~> gute obere Schranke für reale Kosten

Amortisierte Gesamtkosten: Beispiel Stack

- ▶ wähle: $\Phi(i) = \text{Größe des Stacks}$
- ▶ PUSH: Stack wächst von ℓ auf $\ell + 1$
 - ▶ also $a_i = 1 + (\ell + 1) - \ell = 2$.
 - ▶ für jedes PUSH zusätzlich ein Euro auf Konto
- ▶ POP: Stack schrumpft von ℓ auf $\ell - 1$
 - ▶ also $a_i = 1 + (\ell - 1) - \ell = 0$.
 - ▶ Sparen hat sich gelohnt: für POP genug auf dem Konto
- ▶ MPOP(j): Der Stack schrumpft von ℓ auf $\max(\ell - j, 0)$
 - ▶ $a_i = r_i + \max(\ell - j, 0) - \ell$

$$= \begin{cases} j + \ell - j - \ell & \text{falls } \ell \geq j \\ \ell - \ell & \text{falls } \ell < j \end{cases} = 0$$
 - ▶ auch hier hat sich das Sparen gelohnt
- ▶ amortisierte Kosten *jeder* Operation konstant
- ▶ also Gesamtkosten linear

Überblick

Allgemeines zu (deterministischen) Online-Algorithmen

Seitenwechselproblem und deterministische Algorithmen

Randomisierte Online-Algorithmen und Widersacher

Seitenwechsel gegen unwissende Widersacher

Einschub: amortisierte Analyse

Seitenwechsel gegen adaptive Widersacher

12.19 Seitenwechselproblem mit Gewichten

- ▶ Jedes Element x hat ein *Gewicht* $w(x) > 0$
- ▶ Kosten für Laden von x in den Cache: $w(x)$.
- ▶ Gesamtkosten für eine Anforderungsfolge
= Summe der Einzelkosten
- ▶ alle Gewichte gleich
⇒ ursprüngliches Seitenwechselproblem

12.20 Reziprok-Algorithmus

- ▶ Sind x_1, \dots, x_k die Elemente im Cache und muss verdrängt werden, dann wählt der Reziprok-Algorithmus Element x_i mit Wahrscheinlichkeit

12.20 Reziprok-Algorithmus

- ▶ Sind x_1, \dots, x_k die Elemente im Cache und muss verdrängt werden, dann wählt der Reziprok-Algorithmus Element x_i mit Wahrscheinlichkeit

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

- ▶ „Leichte“ Elemente werden bevorzugt hinausgeworfen.

12.21 Satz

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

12.22 Beweis

- ▶ R : Reziprok-Algorithmus
- ▶ W : adaptiver Online-Widersacher
- ▶ S_i^R : Cacheelemente von R nach der i -ten Anforderung
- ▶ S_i^W : Cacheelemente von W nach der i -ten Anforderung
- ▶ f_i^R : bei R verursachte Kosten bei Abarbeitung von r_i
- ▶ f_i^W : bei W verursachte Kosten bei Abarbeitung von r_i
- ▶ Zeige:

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

durch Konstante beschränkt

12.22 Beweis (2)

- ▶ Definiere *Potenzialfunktion*

$$\Phi_i =$$

- ▶ Betrachte Zufallsvariablen $X_i = f_i^R - k f_i^W - (\Phi_i - \Phi_{i-1})$.

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

12.22 Beweis (2)

- ▶ Definiere *Potenzialfunktion*

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

- ▶ Betrachte Zufallsvariablen $X_i = f_i^R - k f_i^W - (\Phi_i - \Phi_{i-1})$.

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

12.22 Beweis (2)

- ▶ Definiere *Potenzialfunktion*

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

- ▶ Betrachte Zufallsvariablen $X_i = f_i^R - k f_i^W - (\Phi_i - \Phi_{i-1})$.

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

- ▶ genügt, zu zeigen: $\mathbf{E}[\sum_i X_i] \leq 0$.
- ▶ sogar: $\forall i : \mathbf{E}[X_i] \leq 0$.

12.22 Beweis (3)

- ▶ Idee: Anforderungen erst von W und dann von R bearbeitet.
- ▶ Untersuche die Veränderungen, die X_i dabei erfährt.
- ▶ Definiere:

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

- ▶ Es ist:

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

- ▶ Zeige: $\mathbf{E} [-k f_i^W - (\Psi_i - \Phi_{i-1})] \leq 0$
- ▶ und $\mathbf{E} [f_i^R - (\Phi_i - \Psi_i)] \leq 0$

12.22 Beweis (4)

- ▶ eine wenig erfreuliche Rechnung; erst für Widersacher W
- ▶ Cache Miss: Element x' werde durch x ersetzt.
O. B. d. A.: Kosten $w(x')$ in Rechnung gestellt, also $f_i^W = w(x')$.
- ▶ $S_i^W = S_{i-1}^W \setminus \{x'\} \cup \{x\}$.
- ▶ definiere Menge $A = S_{i-1}^R \setminus (S_{i-1}^W \cup \{x\})$

$$\begin{aligned}
 -\Psi_i + \Phi_{i-1} &= - \sum_{z \in S_{i-1}^R} w(z) + k \sum_{z \in S_{i-1}^R \setminus S_i^W} w(z) \\
 &\quad + \sum_{z \in S_{i-1}^R} w(z) - k \sum_{z \in S_{i-1}^R \setminus S_{i-1}^W} w(z) \\
 &= k \sum_{z \in S_{i-1}^R \setminus S_i^W} w(z) - k \sum_{z \in S_{i-1}^R \setminus S_{i-1}^W} w(z)
 \end{aligned}$$

12.22 Beweis (5)

$$-\Psi_i + \Phi_{i-1}$$

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

12.22 Beweis (6)

Bei jedem Cache Miss von W gilt folglich stets:

$$-kf_i^W - (\Psi_i - \Phi_{i-1}) \leq 0$$

also erst recht

$$\mathbf{E} \left[-kf_i^W - (\Psi_i - \Phi_{i-1}) \right] \leq 0 .$$

12.22 Beweis (7)

- ▶ nun Algorithmus Reziprok R
- ▶ Cache Miss: x'' werde von x verdrängt. $f_i^R = w(x)$.
- ▶ Beachte: $x \in S_i^W$.

12.22 Beweis (8)

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

12.22 Beweis (9)

$E[-\Phi_i + \Psi_i]$:

R verdrängt x'' mit W.keit $(1/w(x''))/\sum_y 1/w(y)$. Also:

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

12.22 Beweis (10)

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

- ▶ vor Cache Miss von R :
 $x \in S_i^W \setminus S_{i-1}^R$, also $|S_i^W \setminus S_{i-1}^R| \geq 1$.
- ▶ Beide Caches gleich groß, also auch
 $|S_{i-1}^R \setminus S_i^W| \geq 1$

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

12.23 k -Server-Problem

- ▶ Metrischer Raum (M, d)
- ▶ Auf k Punkten befindet sich je ein *Server*.
- ▶ Anforderung: Angabe eines Punktes $x \in M$.
- ▶ Bearbeitung:
 - ▶ Steht gerade ein Server auf x , ist nichts zu tun.
 - ▶ Andernfalls muss ein Server von seinem y nach x bewegt werden.
 - ▶ Kosten: $d(x, y)$.
- ▶ Aufgabe: wähle Server so, dass die Gesamtkosten minimiert werden.
- ▶ Seitenwechselproblem mit Gewichten ist Spezialfall hiervon

12.24 Satz

Löst ein randomisierter Online-Algorithmus R das k -Server-Problem in jedem metrischen Raum, dann ist $C_R^{aon} \geq k$.

- ▶ trivial, falls $k = 1$
- ▶ o. B. d. A. $k \geq 2$

12.25 Beweis

- ▶ R randomisierter Online-Algorithmus
- ▶ H Menge von $k + 1$ Punkten des Raumes, mit den k Punkten, auf denen R seine Server positioniert hat.
- ▶ Betrachte Anforderungsfolge r_1, r_2, \dots bei der jeweils der Punkt aus H als nächstes gewählt wird, auf dem R gerade *keinen* Server hat.
- ▶ Was sind die Kosten von R für Bedienung von Punkt r_j ?

12.25 Beweis

- ▶ R randomisierter Online-Algorithmus
- ▶ H Menge von $k + 1$ Punkten des Raumes, mit den k Punkten, auf denen R seine Server positioniert hat.
- ▶ Betrachte Anforderungsfolge r_1, r_2, \dots bei der jeweils der Punkt aus H als nächstes gewählt wird, auf dem R gerade *keinen* Server hat.
- ▶ Was sind die Kosten von R für Bedienung von Punkt r_j ?
 - ▶ $d(r_{j+1}, r_j)$

12.25 Beweis

- ▶ R randomisierter Online-Algorithmus
- ▶ H Menge von $k + 1$ Punkten des Raumes, mit den k Punkten, auf denen R seine Server positioniert hat.
- ▶ Betrachte Anforderungsfolge r_1, r_2, \dots bei der jeweils der Punkt aus H als nächstes gewählt wird, auf dem R gerade *keinen* Server hat.
- ▶ Was sind die Kosten von R für Bedienung von Punkt r_j ?
 - ▶ $d(r_{j+1}, r_j)$
- ▶ Denn:
 - ▶ zu dem Zeitpunkt, zu dem Anforderung r_j kommt, steht ein Server auf r_{j+1} ,
 - ▶ aber das ist genau der, der sich nach r_j bewegt, denn nur wenn r_{j+1} zum nächsten Zeitpunkt frei ist, wird es angefordert.

12.25 Beweis (2)

- ▶ Gesamtkosten von R für (r_1, r_2, \dots, r_n) sind also

$$\begin{aligned}M_R(r_1, \dots, r_n) &= \sum_{j=1}^{n-1} d(r_{j+1}, r_j) + x \\ &= \sum_{j=1}^{n-1} d(r_j, r_{j+1}) + x .\end{aligned}$$

- ▶ Zeige, dass es k Online-Algorithmen B_1, \dots, B_k gibt mit Gesamtkosten

$$\sum_{i=1}^k M_{B_i}(r_1, \dots, r_n) \leq M_R(r_1, \dots, r_n) .$$

- ▶ Also gibt es einen Algorithmus B_i mit Kosten von höchstens

$$1/k \cdot \sum M_{B_i}(r_1, \dots, r_n) \leq 1/k \cdot M_R(r_1, \dots, r_n) .$$

12.25 Beweis (3)

- ▶ Sei $H = \{r_1, u_1, \dots, u_k\}$.
- ▶ Algorithmus B_i beginnt mit seinen Servern auf allen Punkten außer u_i .
- ▶ Wenn B_i keinen Server auf r_j hat, bewegt er den von r_{j-1} nach r_j .
- ▶ Es bezeichne S_i die Menge der Punkte, an denen B_i seine Server positioniert hat.
- ▶ Zeige:
 1. 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. $\sum_{i=1}^k M_{B_i}(r_1, \dots, r_n) \leq M_R(r_1, \dots, r_n)$.

12.25 Beweis (4)

1. Für $i \neq m$ ist $S_i \neq S_m$.

Induktion über die Zeit.

- ▶ Vor der ersten Anforderung gilt die Behauptung nach Konstruktion.
- ▶ Gelte die Behauptung vor Anforderung r_j .
- ▶ Dann gibt es nicht zwei B_i, B_m , die beide keinen Server auf r_j haben.
- ▶ Also:
 - ▶ entweder $r_j \in S_i \cap S_m$:
Dann ändern sich weder S_i noch S_m .
 - ▶ Oder etwa $r_j \in S_i \setminus S_m$: Dann:
 - B_m bewegt Server von r_{j-1} nach r_j
 - B_i hat Server schon auf r_j ... und auf r_{j-1} , da $k \geq 2$ hinterher $r_{j-1} \in S_i \setminus S_m$.

12.25 Beweis (4)

2. Bei jedem r_j bewegt nur *ein* B_i einen Server dorthin:

- ▶ 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.
- ▶ Widerspruch zu Teil 1.

3. $\sum_{i=1}^k M_{B_i}(r_1, \dots, r_n) \leq M_R(r_1, \dots, r_n)$:

- ▶ r_1 verursacht für kein B_i Kosten.
- ▶ Da für jedes weitere 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 $\sum_{i=1}^k M_{B_i}(r_1, \dots, r_n)$.
- ▶ Dieser Wert ist also $\sum_{j=2}^n d(r_{j-1}, r_j) = \sum_{j=1}^{n-1} d(r_j, r_{j+1})$.

12.25 Beweis (5)

Wenn der Widersacher gleichwahrscheinlich eines der B_i als seine Strategie wählt, dann hat er erwartete Kosten $M_R(r_1, \dots)/k$.

12.26

- ▶ Für die Praxis ist es sinnvoll zu berücksichtigen, dass die Widersacher manchmal nicht ganz so böse sind.
 - ▶ access graphs (Borodin et al. 1995, Irani et al. 1996)
 - ▶ Markov paging (Karlin et al. 2000)
 - ▶ Lokalität (Albers et al. 2005)

12.27

- ▶ C^{xyz} : Infimum der C_R^{xyz} (über randomisierte Algorithmen R für das Seitenwechselproblem)
- ▶ Nach Definition:

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

- ▶ Es gilt aber auch:

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

12.28 Satz

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

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.

Zusammenfassung

- ▶ Betrachtung von Widersachern statt Worst-Case-Analyse
- ▶ Manchmal sind geschickt gewählte Potenzialfunktionen hilfreich.
- ▶ Wettbewerbsfaktoren gegen unwissende Widersacher
 - ▶ randomisiert in $O(\log n)$, aber
 - ▶ deterministisch nur $O(n)$