Algorithmen 2

Kapitel: Online-Algorithmen

Thomas Worsch

Fakultät für Informatik
Karlsruher Institut für Technologie

Wintersemester 2017/2018

Überblick

Quellen

- Skript zu dieser Vorlesung
- Unterlagen von R. van Stee zur Vorlesung «Approximations- und Online-Algorithmen» (KIT)
- Buch Borodin/El-Yaniv: Online Computation and Competitive Analysis
- Unterlagen von F. Kuhn zur Vorlesung «Algorithm Theory» (Uni Freiburg)
- Unterlagen von G. Schnitger und A. Kovacs (beide Uni Frankfurt/Main) zur Vorlesung «Effiziente Algorithmen»
- Unterlagen zur Vorlesung «Online and Approximation Algorithms» von Susanne Albers (TU München)
- und viele andere mehr …

Eine Reihe von Beispielen

auf manche werden wir genauer eingehen:

- Job-Scheduling
- Skiausleihe
- Speicherverwaltung
- Auswahl von «Experten»
- selbstorganisierende Datenstrukturen

Beispiel Job-Scheduling

ähnlich wie im Kapitel zu Approximationsalgorithmen

- *Maschinen* $M_1, \ldots, M_m$
- *Anfrage:* ein Job J_i, der Zeit $t_j \geq 0$ benötigt
- *Antwort:* Zuordnung von J_i zu einer Maschine M_j
- Wie kann man den *Makespan* minimieren?
  (Wann ist die letzte Maschine fertig?)

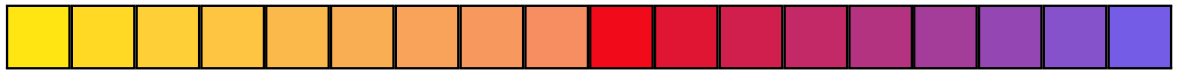
- Entscheidung für jedes J_i ohne Kenntnis der zukünftigen Jobs

Beispiel Skiausleihe

- *Skiurlaub:* solange das Wetter gut ist, jeden Morgen
  Anforderung: «Skier besorgen!»
  - sobald Wetter schlecht, Aufforderung: «Heimfahren!»
- zwei mögliche *«Antworten»*
  - Ski für einen Tag ausleihen; Kosten k €
  - Ski für ganzen Urlaub kaufen; Kosten K €, $K > k$
- Was tun, um Gesamtkosten klein zu halten?
  (in welchem Sinne?)
- Entscheidung ohne Kenntnis des zukünftigen Wetters

Beispiel Speicherverwaltung

- langsamer *Hauptspeicher* Größe N



schneller *Cache* Größe $k < N$

- *Anforderung:* eine Seite P des Hauptspeichers: 
  - P im Cache: alles gut
- P nicht im Cache: *Cache Miss, Strafpunkt*, *Seitenfehler*
  «Antwort» des Systems:
  - verdränge an einer Stelle a im Cache
  - die dort gespeicherte Seite durch P
- Welche Verdrängungsstrategie minimiert die «Kosten»?
- Auswahl der verdrängten Seiten ohne Kenntnis der zukünftigen Anforderungen

Auswahl von Experten

- in jeder von mehreren Runden:
  1. jeder von n Experten: eine Ja/Nein-Empfehlung
     - Experten können irren!
  2. eigene Ja/Nein-Entscheidung
  3. Mitteilung, welche Entscheidung richtig gewesen wäre
- Wie kann man die Anzahl Fehlentscheidungen minimieren? (in welchem Sinne?)
- eigene Entscheidungen ohne Kenntnis der zukünftigen Qualität der Expertenantworten

Beispiel Selbstorganisierende Datenstrukturen

- einfach verkettete Liste
- *Anforderung:* ein Element x der Liste
  Kosten: Position von x in Liste
- mögliche *Reaktionen:*
  - ohne weitere Kosten: angefragtes Element weiter nach vorne
  - mit Kosten 1: Vertauschung zweier aufeinanderfolgender Elemente
- Welche Listen-Verwaltung minimiert die Kosten?
- eventuelle Umordnungen ohne Kenntnis der zukünftigen Anforderungen

Online-Algorithmus

Formalisierung (Borodin/El-Yaniv folgend)

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

zeige: $c_{\text{ALG}} = c^{\inf}$ 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 $c_{\text{ALG}} \geq c^{\inf}$

Wettbewerbsfaktor und strikte Kompetitivität

$C = \{\, c \mid \text{ALG ist strikt } c\text{-kompetitiv} \,\}$ und $c^{\inf} = \inf C$
zeige: $c_{\text{ALG}} = c^{\inf}$ 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 $c_{\text{ALG}} \geq c^{\inf}$

$\leq$: indirekt: angenommen $c_{\text{ALG}} - c^{\inf} = d > 0$
also $c' = c^{\inf} + d/2 \in C$
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 *strikter* c-Kompetitivität erlaubt man Ausnahmen

- Achtung:
  - wir *vermeiden* hier den Begriff Wettbewerbsfaktor bzw. competitive ratio
  - andere benutzen ihn weiter, *in unterschiedlichen Bedeutungen*

Überblick

listScheduling ist (fast) ein Onlinealgorithmus

Approximationsalgorithmus

```
1  listScheduling(n, m, t_{1...n})
2  each L_i ← 0                          ▷ load of machine i, 1 ≤ i ≤ m
3  each S_j ← 0                          ▷ machine for job j, 1 ≤ j ≤ n
4  for each j in range(1, n)
5       do pick k from {i | L_i is currently minimal }
6          S_j ← k
7          L_k ← L_k + t_j
8
9  return S
```

frühere Analyse: Wettbewerbsfaktor höchstens 2

listScheduling ist (fast) ein Onlinealgorithmus

Onlinealgorithmus

```
1  listScheduling(   m      )
2  each L_i ← 0                    ▷ load of machine i, 1 ≤ i ≤ m
3
4  for each t_j in σ
5       do pick k from {i | L_i is currently minimal }
6          a_j ← k
7          L_k ← L_k + t_j
8          assign Job j to machine a_j
9
```

frühere Analyse: Wettbewerbsfaktor höchstens 2

Überblick

Skiausleihe

- Ski für einen Tag ausleihen kostet k €
- Ski für ganzen Urlaub kaufen kostet K €, $K > k$
- Man weiß nicht, wie lange der Urlaub dauert.
- Am wievielten Tag sollte man die Ski kaufen?

Ähnlich (was ist anders?)

- Nach wievielen Bahnfahrten eine Bahncard kaufen?

Optimale Kosten?

- Anforderungsfolge: t mal «Skier ausleihen!»
- Optimale Kosten:
  - falls Urlaubsdauer $t \leq K/k$ Tage: t mal Ausleihe
    - Kosten tk
  - falls Urlaubsdauer $t \geq K/k$ Tage: sofort kaufen
    - Kosten K

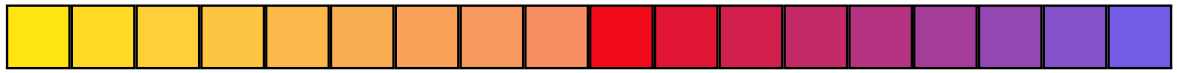
Deterministische Entscheidung für Skikauf

- Entscheidung hängt nur von k und K ab.
- *Wettbewerbsfaktor* 2
  - wenn Kauf an Tag K/k
  - besser gehts nicht
  - Details in der Übung

Überblick

Problemstellung

- langsamer *Hauptspeicher* Größe N



schneller *Cache* Größe $k < N$

- *Anforderung:* eine Seite P des Hauptspeichers: 
  - P im Cache: alles gut
- P nicht im Cache: *Cache Miss, Page Fault, Strafpunkt*
  Reaktion 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

Plan für diesen Abschnitt

- Offlinealgorithmus LFD und seine Optimalität
- deterministische Onlinealgorithmen:
  untere Schranke k für Wettbewerbsfaktor
- wird von LRU erreicht
- kurz: resource augmentation
- randomisierter Onlinealgorithmus:
  RANDMARK hat Wettbewerbsfaktor $2H_k$

Überblick

Longest Forward Distance (LFD)

- Aus dem Cache wird ein Eintrag verdrängt, der *am spätesten in der Zukunft* wieder angefordert wird.
- Beispiel:
  vor Bearbeitung der ersten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	4	2				

Longest Forward Distance (LFD)

- Aus dem Cache wird ein Eintrag verdrängt, der *am spätesten in der Zukunft* wieder angefordert wird.
- Beispiel:
  vor Bearbeitung der ersten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	4	2				

  nach Bearbeitung der ersten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	3	2				

Longest Forward Distance (LFD)

- Beispiel:
  nach Bearbeitung der ersten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	3	2				

Longest Forward Distance (LFD)

- Beispiel:
  nach Bearbeitung der ersten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	3	2				

  nach Bearbeitung der zweiten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	3	2				

Longest Forward Distance (LFD)

- Beispiel:
  nach Bearbeitung der zweiten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	3	2				

Longest Forward Distance (LFD)

- Beispiel:
  nach Bearbeitung der zweiten Anforderung

σ	3	1	5	3	2	4	1
Cache	1	3	2				

  nach Bearbeitung der dritten Anforderung

σ	3	1	5	3	2	4	1
Cache	5	3	2				

Optimalität von LFD

Satz

LFD ist optimal.

Optimalität von LFD — Beweisskizze (1)

Satz

LFD ist optimal.

Beweisidee

- zeige: Ein optimaler Algorithmus OPT kann schrittweise zu LFD umgebaut werden.
- für induktiven Schritt folgendes Lemma

Lemma

Gegeben ALG, $\sigma = (r_1, \ldots, r_n)$ und $i \leq n$ arbeite $\text{LFD}_i(\text{ALG})$ so:

- $\text{LFD}_i(\text{ALG})$ verarbeitet $(r_1, \ldots, r_{i-1})$ wie ALG
- Bei Cache Miss für r_i entfert $\text{LFD}_i(\text{ALG})$ den am spätestens in der Zukunft angeforderten Eintrag.

Dann gilt: $\text{LFD}_i(\text{ALG})(\sigma) \leq \text{ALG}(\sigma)$.

Optimalität von LFD: Beweisskizze (2)

- angenommen, Lemma wäre gezeigt, also $\text{LFD}_i(\text{ALG})(\sigma) \leq \text{ALG}(\sigma)$
- zu gegebenem optimalem Algorithmus OPT definiere:
  - $\text{OPT}_0 = \text{OPT}$
  - $\text{OPT}_1 = \text{LFD}_1(\text{OPT}_0)$
  - $\text{OPT}_2 = \text{LFD}_2(\text{OPT}_1)$
  - ...
  - d. h. allgemein für $1 \leq i \leq n$: $\text{OPT}_i = \text{LFD}_i(\text{OPT}_{i-1})$
- dann per Induktion
  - alle OPT_i sind optimal
  - $\text{OPT}_n = \text{LFD}$

Optimalität von LFD: Beweisskizze (3)

bleibt zu zeigen: $\text{LFD}_i(\text{ALG})(\sigma) \leq \text{ALG}(\sigma)$

nicht hier

Überblick

Deterministische Onlinealgorithmen

FIFO	first in first out
LIFO	last in first out
LRU	least recently used
LFU	least frequently used

- LIFO ist nicht kompetitiv: betrachte
  $\langle 1, 2, \ldots, k-1, k, k+1, k, k+1, k, k+1, k, k+1, \ldots \rangle$
- LFU ist nicht kompetitiv: betrachte
  $\langle 1^m, 2^m, \ldots, (k-1)^m \rangle, \langle k, k+1 \rangle^{m-1}$
- LRU und FIFO sind k-kompetitiv
  - in der Praxis nimmt man LRU

Untere Schranke für den Wettbewerbsfaktor (det. Seitenwechsel)

Satz

Jeder deterministische Onlinealgorithmus für das Seitenwechselproblem mit Cachegröße k hat einen Wettbewerbsfaktor $c \geq k$.

Untere Schranke für den Wettbewerbsfaktor (det. Seitenwechsel) — Beweisskizze (1)

Satz

Jeder deterministische Onlinealgorithmus für das Seitenwechselproblem mit Cachegröße k hat einen Wettbewerbsfaktor $c \geq k$.

Beweisskizze

- Hauptspeicher Größe $N = k + 1$
- ALG beliebiger Onlinealgorithmus für Speicherverwaltung
- betrachte σ mit der Eigenschaft:
  - Jede Anforderung verursacht Seitenfehler bei ALG und
  - $|\sigma| > k$
- Zeige: LFD hat höchstens alle k Anforderungen einen Seitenfehler.

Beweisskizze (2)

- o. B. d. A.
  - zu Beginn ALG und LFD mit gleichem Cache-Inhalt
  - Cache Miss bei erster Anforderung
- *Phase:*
  - beginnt mit Anforderung, bei der LFD Cache Miss hat
  - endet unmittelbar vor der nächsten
- Länge einer Phase: *mindestens k*, denn
  - wenn LFD einen Cache Miss hat,
  - dann wird die Seite verdrängt,
  - vor der jede der anderen $k - 1$ Seiten,
  - die gerade im Cache sind,
  - mindestens einmal angefordert wird.
- Anzahl Cache Misses pro Phase:
  - ALG Länge der Phase viele, also $\geq k$
  - LFD nur 1 zu Beginn der Phase

Überblick

LRU — Beispiel

live

LRU ist k-kompetitiv

Satz

LRU ist k-kompetitiv.

LRU ist k-kompetitiv — Beweisskizze (1)

Satz

LRU ist k-kompetitiv.

Beweisskizze

- σ beliebige Anforderungsfolge
- Aufteilung von σ in Phasen P_0, P_1, etc.
  - P_0 beginnt mit erster Anforderung
  - jedes P_{i+1} beginnt unmittelbar nach Ende von P_i
  - jede P_i umfasst maximal viele Anforderungen, die nur k Seiten betreffen
- sei außerdem Q_i die Folge P_i ohne deren erstes Element, aber mit dem ersten Element von P_{i+1} angehängt

Beweisskizze (2)

- zeige:
  - LRU hat höchstens k Cache Misses pro Phase P_i
  - LFD hat mindestens 1 Cache Miss pro Q_i
- dann bei insgesamt m Phasen
  - $\text{LRU}(\sigma) \leq km$
  - $\text{LFD}(\sigma) \geq m - 1$
  - also $\text{LRU}(\sigma) \leq k \cdot \text{LFD}(\sigma) + k$

Beweisskizze (3)

zeige: LRU hat höchstens k Cache Miss pro P_i

- andernfalls hätte LRU in einer Phase
  für eine Seite p *zweimal* einen Cache Miss
- wegen «least recently used» hätten zwischen den beiden Cache Misses k *andere* Seiten angefordert werden müssen
- d. h. es wären in einer Phase $k + 1$ verschiedene Seiten angefordert worden
- im Widerspruch zur Definition der Phasen

Beweisskizze (4)

zeige: LFD hat mindestens 1 Cache Miss pro Q_i

- seien $p_1^i, \ldots, p_k^i$ die der Reihe nach in P_i auftretenden verschiedenen Seiten
- betrachte $p_2^i, \ldots, p_k^i, p_1^{i+1}$
- das sind k verschiedene Seiten
- wenn von $p_2^i, \ldots, p_k^i$ keine einen Cache Miss verursacht, dann p_1^{i+1} (weil $p_1^i, \ldots, p_k^i$ im Cache sind)
- also hat LRU mindestens in jeder Phase außer der ersten mindestens einen Cache Miss

Überblick

Speicherverwaltung

- Offline: LFD ist optimal
- Deterministisch: bestenfalls k-kompetitiv
- Deterministisch: LRU ist k-kompetitiv
- Resource Augmentation: (h, k)-Seitenwechsel
- Randomisiert: RANDMARK ist $2H_k$-kompetitiv

Resource Augmentation

- *(h, k)-Seitenwechselproblem*
- vergleiche Onlinealgorithmus ALG_k mit Cachegröße k mit
- LFD_h mit Cachegröße $h < k$

- Onlinealgorithmus heißt *konservativ*, wenn er
  - für Anforderungsfolgen mit höchsten k verschiedenen Seiten
  - höchstens k Cache Misses hat
- Beispiele: LRU, FIFO

Resource Augmentation

Satz

Jeder konservative Onlinealgorithmus ist $\frac{k}{k-h+1}$-kompetitiv.

Beweisskizze

- Verallgemeinerung von «LRU ist k-kompetitiv».
- konservativer Alg. hat in P_i höchstens k Cache Misses
- Nach erster Anforderung von P_i
- enthält der Cache von LFD_h noch $h-1$ andere Seiten,
- die in P_{i-1} angefordert wurden.
- In Q_i folgen aber noch k andere verschiedene Seiten,
- von denen $\geq k-(h-1)$ einen Cache Miss verursachen

Beispiel $h = k/2$: $\rightsquigarrow \text{ALG}_k(\sigma) \leq 2\text{LFD}_h(\sigma) + b$

Überblick

Randomisierte Onlinealgorithmen

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

etwas andere Sicht auf c-Kompetitivität sinnvoll

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 $(r_1, \ldots, r_n)$.
  - *adaptiver Widersacher*:
    - arbeitet gegen eine konkrete Abarbeitung von R,
    - kennt die von R bei Abarbeitung von $(r_1, \ldots, r_i)$ erzeugten Zufallsbits,
    - und folglich auch immer den aktuellen Cachezustand von R.

Widersacher (2)

Womit vergleicht man die Zahl der Cache Misses von R?

- *unwissende Widersacher:* $\text{OPT}(r_1, \ldots, r_n)$
- *adaptive Widersacher:* zwei Varianten
  - adaptiver Online-Widersacher:
    - muss erzeugte Anforderungsfolge selbst auch irgendwie online verwalten
    - muss also sofort nach Erzeugung eines r_i entscheiden, wie er in seinem Cache damit umgeht $\rightsquigarrow$ Kosten
  - adaptiver Offline-Widersacher:
    - kann zunächst vollständig $(r_1, \ldots, r_n)$ erzeugen und
    - bekommt nur $\text{OPT}(r_1, \ldots, r_n)$ in Rechnung gestellt
  - In beiden Fällen ist Anzahl der Cache Misses von W Zufallsvariable (Abhängigkeit von den Zufallsbits von R)

Wettbewerbsfaktor

- 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:

$$\mathbf{E}\left[f_R(r_1, \ldots, r_n)\right] - c \cdot \mathrm{OPT}(r_1, \ldots, r_n) \leq b$$

  nur das werden wir hier betrachten

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

$$\mathbf{E}\left[f_R(r_1, \ldots, r_n) - c \cdot f_W(r_1, \ldots, r_n)\right] \leq b$$

resp.
$$\mathbf{E}\left[f_R(r_1, \ldots, r_n) - c \cdot f_O(r_1, \ldots, r_n)\right] \leq b$$

randMark Algorithmus (Fiat et al., 1991)

randMark Algorithmus (Fiat et al., 1991)

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

randMark Algorithmus (Fiat et al., 1991)

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

Kompetitivität von randMark

Satz

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

Beachte:

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

Beweis

- Algorithmus zerfällt *Phasen*:
  - Phase beginnt mit Zurücksetzen aller Markierungsbits auf 0
  - und endet unmittelbar vor der nächsten.
- Betrachte LFD (optimal) und RANDMARK für $r_1, r_2, r_3 \ldots$
- anfangs gleiche Cacheinhalte und r_1 führe zu Cache Miss
- also
  - jede Phase beginnt mit Cache Miss,
  - umfasst maximale Teilfolge $r_i, \ldots, r_j$
  - 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. RANDMARK hat erwartet pro Phase ℓH_k Cache Misses.

Definition von ℓ kommt gleich

Beweis (3)

betrachte einzelne Phase

ein Datum heiße

- *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)

- randMark 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 randMark 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:

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

Beweis (8)

Abschätzung Cache Misses von LFD (3)

Insgesamt:

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

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 $x_1, \ldots, x_{k-\ell}$ 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

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

- also erwartete Anzahl Cache Misses kleiner oder gleich

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

Überblick

Überblick

Problemstellung

- in jeder von mehreren Runden:
  1. jeder von k «Experten»: eine Ja/Nein-Empfehlung
     - Experten können irren!
  2. eigene Ja/Nein-Entscheidung
  3. Mitteilung, welche Entscheidung richtig gewesen wäre

Ziel:

- *Anforderung:* k-Tupel, J/N-Empfehlungen der Experten
- *Antwort:* eigene J/N-Entscheidung
- *Kosten:* Anzahl eigener Fehlentscheidungen
- Ziel: eigene Kosten nahe an Kosten der besten Experten

Auswahl von Experten

Wie man es nicht machen sollte:

- Wähle die Antwort eines Experten, der bisher am besten war.

denn

- für jedes k Menge von k Experten konstruierbar, für die gilt:
  - Die obige Strategie irrt *immer*.
  - Jeder Experte hat nach t Runden höchsten t/k Fehler gemacht.
- $\rightsquigarrow$ Übung

Wie dann?

Auswahl von Experten: der deterministische Weighted Majority Algorithm (WMA)

- Experte i hat ein «Gewicht» w_i
- initial: alle $w_i \leftarrow 1$
- in jeder Runde
  1. Experten geben Empfehlungen $x_i \in \{ \text{ja}, \text{nein} \}$
  2. eigene Entscheidung fällen:

$$\begin{cases} \text{ja}, & \text{falls } \sum_{i, x_i = \text{ja}} w_i \geq \sum_{i, x_i = \text{nein}} w_i \\ \text{nein}, & \text{falls } \sum_{i, x_i = \text{ja}} w_i < \sum_{i, x_i = \text{nein}} w_i \end{cases}$$

  3. Expertengewichte anpassen

$$w_i^{t+1} = \begin{cases} w_i^t, & \text{falls } x_i \text{ richtig war} \\ w_i^t / 2, & \text{falls } x_i \text{ falsch war} \end{cases}$$

Qualität von WMA

Satz

WMA ist $1/\log_2(4/3)$-kompetitiv. Genauer

$$\text{WMA}(\sigma) \leq \frac{1}{\log_2(4/3)} (\text{OPTXPRT}(\sigma) + \log_2 k)$$

Beweis

in einer Runde t

- W_t: Summe der w_i zu Beginn der Runde
  - initial $W_1 = n$
- x_i die Expertenempfehlungen, y die richtige Antwort
- $R_t = \sum_{i,x_i=y} w_i$ und $F_t = \sum_{i,x_i \neq y} w_i$
  - $W_t = R_t + F_t$

wenn Entscheidung falsch, dann

- $R_t < W_t/2$
- $W_{t+1} = \frac{F_t}{2} + R_t = \frac{F_t}{2} + \frac{R_t}{2} + \frac{R_t}{2} \leq \frac{W_t}{2} + \frac{W_t}{4} = \frac{3}{4} W_t$

Beweis (2)

bis zu Beginn von Runde t

- wenn eigene Entscheidungen f mal falsch
  dann $W_t \leq \left(\frac{3}{4}\right)^f \cdot W_1 = \left(\frac{3}{4}\right)^f \cdot k$
- wenn bester Experte i nur f_{opt} mal falsch
  dann $W_t \geq w_i = \left(\frac{1}{2}\right)^{f_{opt}}$
- also $\left(\frac{1}{2}\right)^{f_{opt}} \leq \left(\frac{3}{4}\right)^f \cdot k$
- also $f \leq \frac{1}{\log_2(4/3)} \left(f_{opt} + \log_2 k\right)$
  - $\frac{1}{\log_2(4/3)} \approx 2.409$

Überblick

Verallgemeinerte Problemstellung

- in Runde t:
  1. Experten: Empfehlungen $x_1, \ldots, x_k$
  2. eigene Wahl einer Empfehlung
  3. Mitteilung, welche Entscheidung richtig gewesen wäre
  4. Beurteilung der Empfehlungen durch «Noten» c_i^t
     - beste Note: 0
     - schlechteste Note: 1
  5. eigene *Kosten:* Note der gewählten Empfehlung

Randomisiert: RandWMA$_\varepsilon$

- es sei $0 < \varepsilon < 1/2$
- Experte i hat ein «Gewicht» w_i
- initial: alle $w_i \leftarrow 1$
- in jeder Runde $t = 1, 2, \ldots$
  1. wähle Empfehlung von Experte i mit Wahrscheinlichkeit $p_i = \frac{w_i}{\sum_1^k w_i}$
  2. Benotung
  3. setze jedes $w_i \leftarrow w_i(1 - \varepsilon c_i^t)$

Qualität von RANDWMA$_\varepsilon$

Satz

Es sei k die Anzahl der Experten und $0 < \varepsilon < 1/2$.
Wenn RANDWMA$_\varepsilon$ nach einer Anzahl Runden erwartete Gesamtkosten K hat und die besten Experten Empfehlungen mit (minimalen) Gesamtkosten K_{opt} gegeben haben, dann ist

$$K \leq (1 + \varepsilon) \cdot K_{opt} + \frac{\ln n}{\varepsilon}$$

Beweis

- setze
  - w_i^t Gewicht von Experte i zu Beginn von Runde t
  - $W_t = \sum_{i=1}^{k} w_i$ Gesamtgewicht
  - $K_t = \sum_{i=1}^{k} \frac{w_i^t}{W_t} c_i^t$ erwartete eigene Kosten in Runde t
- $$W_{t+1} = \sum_{i=1}^{k} w_i^t(1 - \varepsilon c_i^t) = \sum_{i=1}^{k} w_i^t - \varepsilon \sum_{i=1}^{k} w_i^t c_i^t$$
  $$= W_t - \varepsilon K_t W_t = W_t(1 - \varepsilon K_t)$$
- also $W_{t+1} = W_1 \cdot \prod_{\tau=1}^{t}(1 - \varepsilon K_\tau)$
- also $\ln W_{t+1} \leq \ln n - \sum_{\tau=1}^{t} \varepsilon K_\tau = \ln n - \varepsilon K$
  - wegen $\ln(1 - x) \leq -x$ für $0 \leq x \leq 1/2$

Beweis (2)

- andererseits für jeden Experten i (auch den opt.):
  $W_{t+1} \geq w_i^{t+1} = \prod_{\tau=1}^{t}(1 - \varepsilon c_i^\tau)$
- wegen $\ln(1-x) \geq -x - x^2$ für $0 \leq x \leq 1/2$ daher
  $\ln W_{t+1} \geq \sum_{\tau=1}^{t} \ln(1 - \varepsilon c_i^\tau) \geq -\sum_{\tau=1}^{t}(\varepsilon c_i^\tau + (\varepsilon c_i^\tau)^2)$
- wegen $(c_i^\tau)^2 \leq c_i^\tau$ weiter:
  $\ln W_{t+1} \geq -\sum_{\tau=1}^{t}(\varepsilon c_i^\tau + \varepsilon^2 c_i^\tau) = -(\varepsilon + \varepsilon^2)K(i)$
- insgesamt insbesondere für K_{opt}:

$$-(\varepsilon + \varepsilon^2)K_{opt} \leq \ln W_{t+1} \leq \ln k - \varepsilon \cdot K$$

$$\text{also} \qquad K \leq (1+\varepsilon) \cdot K_{opt} + \frac{\ln k}{\varepsilon}$$