Approximations- und Online-Algorithmen

thgoebel@ethz.ch

ETH Zürich, FS 2022

This documents is a **short** summary for the course *Approximations- und Online-Algorithmen* at ETH Zurich. It is intended as a document for quick lookup, e.g. during revision, and as such does not replace attending the lecture, reading the slides or reading a proper book.

We do not guarantee correctness or completeness, nor is this document endorsed by the lecturers. Feel free to point out any errata, either by mail or on Github.

Contents

List of Figures

Credits: images are generally taken from the lecture scripts.

Part I.
Approximations-Algorithmen

1. Approximations-Algorithmen

TODO. Siehe das Skript von letzem Jahr.

Part II.
Online-Algorithmen

2. Einführung und das Paging-Problem

Konzepte

- Online-Problem, Online-Algorithmus, kompetitiver Faktor
- Skirental-Problem
- Paging-Problem
- Randomisierte Online-Algorithmen
- Yaos Prinzip

Motivation Probleme lösen und Entscheidungen fällen ohne alle für eine optimale Lösung relevanten Informationen zu haben. Stattdessen werden die Informationen stückweise zur Laufzeit bekannt.

Beispiel: Skirental-Problem Unendlich langer Urlaub, nur an schönen Tagen Ski fahren. Skier mieten für 1 CHF pro Tag, oder kaufen für k CHF. Erst am Tag selbst wird bekannt ob ein Tag schön ist.

Optimale Lösung: Sei s die Anzahl schöner Tag. Miete bei $s < k$, kaufe bei $s > k$, bei $s = k$ egal.

Problem: s nicht bekannt, erst am Tag selber wird bekannt ob ein Tag schön ist.

Szenario	Worst Case	Approximationsgüte
An Tag 1 kaufen	Ab Tag 2 schlechtes Wetter	$\frac{k}{1}$
Immer mieten	An $x >> k$ Tagen schönes Wetter	$\frac{x}{k}$
An $k-1$ Tagen mieten, dann kaufen	Ab Tag $k+1$ schlechtes Wetter	$\frac{2k-1}{k} = 2 - \frac{1}{k}$

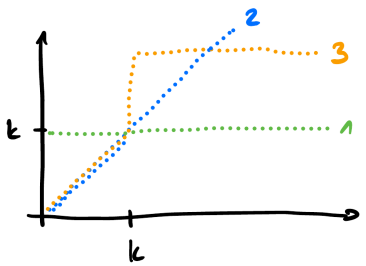

Figure 1: Skirental Szenarios

Online-Problem Ein *Online-Minimierungsproblem* ist $\Pi = (I, O, cost, \min)$. Eine Eingabe $I = (x_1, ..., x_n) \in \mathcal{I}$ ist eine Folge von *Anfragen*, jeweils für *Zeitschritt* i. Eine akzeptierte Lösung $O = (y_1, ..., y_n)$ ist eine Folge von *Antworten*.

Beim analogen Maximierungsproblem spricht man statt von $cost(I, O)$ oft vom *Gewinn* $gain(I, O)$.

Online-Algorithmus Sei Π ein Online-Optimierungsproblem. Ein *Online-Algorithmus* $\mathcal{A}$ berechnet die Ausgabe $\mathcal{A}(I) = (y_1, ..., y_n)$ wobei y_i nur von $(x_1, ..., x_i)$ abhängt. $\mathcal{A}(I)$ ist eine zulässig Lösung für I.

Kompetitiver Faktor (aka. competitive ratio, Wettbewerbsgüte, kompetitive Güte)
Ein Online-Algorithmus $\mathcal{A}$ ist *c-kompetitiv* falls gilt:

$$\exists \alpha \geq 0 \quad \forall I \ : \quad cost(\mathcal{A}(I)) \leq c \cdot cost(Opt(I)) + \alpha$$
$$\frac{cost(\mathcal{A}(I))}{cost(Opt(I))} + \alpha' \leq c$$

für ein Minimierungsproblem und α konstant. *Opt* ist ein optimaler Offline-Algorithmus, d.h. mit vollständiger Information.

Für Maximierungsprobleme:

$$gain(Opt(I)) \leq c \cdot gain(\mathcal{A}(I)) + \alpha$$

Das kleinste c für das dies gilt heisst *kompetitiver Faktor*.
$\mathcal{A}$ heisst *strikt c-kompetitiv* falls $\alpha = 0$.
$\mathcal{A}$ heisst *optimal* falls er strikt 1-kompetitiv ist ($\alpha = 0, c = 1$).

Wir sprechen hierbei von *kompetitiver Analyse*. Der kompetitiver Faktor ist vergleichbar mit der Approximationsgüte von Approximationsalgorithmen.

Ein Online-Algorithmus heisst *kompetitiv* wenn sein kompetitiver Faktor nicht von der Länge der Eingabe abhängt (d.h. es keine Startkosten gibt die amortisiert werden müssen). Die Konstante α ist wichtig da sie erlaubt auf kurze Eingaben schlecht zu sein (und erst auf lange besser zu werden). [1]

Untere Schranken beweisen Für einen strikt kompetitiven Algorithmus: Finde eine Instanz I mit $\frac{\mathcal{A}(I)}{Opt(I)} > c \implies$ nicht strikt-kompetitiv.

Für einen nicht-strikt kompetitiven Algorithmus: Finde eine unendliche Folge $I_1, I_2, ...$ von Instanzen so dass $\frac{\mathcal{A}(I_i)}{Opt(I_i)} > c$ und $Opt(I_i) \overset{i \to \infty}{\longrightarrow} \infty$.

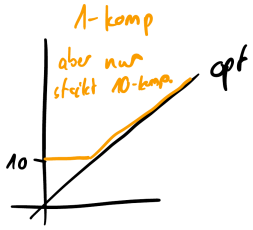

Figure 2: *Opt* in schwarz. $\mathcal{A}$ in orange, 1-kompetitiv und strikt-10-kompetitiv.

2.1. Das Paging-Problem

Paging

- Eingabe: $I = (x_1, ..., x_n)$ mit Speicher-Indizes $x_i \in \mathbb{N}$
- Hauptspeicher mit m Seiten: $(s_1, ..., s_m)$

[1] Warum brauchen wir bei der Approximationsgüte keine vergleichbare Konstante?

- Cache-Speicher mit k Seiten: $B = (s_{j_1}, ..., s_{j_k})$, initialisiert mit $(s_1, ..., s_k)$ [2]
- Zeitschritt i:
  - Index x_i wird angefragt
  - Falls x_i im Cache (d.h. $s_{x_i} \in B$): return $y_i = 0$
  - Andernfalls: return $y_i = j$, und setze $B = B \backslash \{s_j\} \cup \{s_{x_i}\}$, d.h. lösche Seite s_j aus dem Cache und ersetze sie durch s_{x_i}. [3]
- $cost(\mathcal{A}(I)) := |\{i \mid y_i > 0\}|$
- goal := min

Strategien bei *Seitenfehlern (page faults)* zum *Verdrängen* von Seiten: First-in-First-Out (FIFO, wie eine Queue), Last-in-First-Out (LIFO, wie ein Stack), Least-Recently-Used (LRU), Longest-Forward-Distance (LFD, offline-only!).

Satz (FIFO) Ein Online-Algorithmus für Paging der FIFO nutzt ist strikt-k-kompetitiv.

Beweis: Gruppiere Zeitschritte in *Phasen*. Phase 1 endet nach dem ersten Seitenfehler. Phase $P \geq 2$ endet nach $1 + (P-1)k$ Seitenfehlern, d.h. alle k Fehler endet eine Phase und beginnt eine neue.

In Phase 1 machen Opt und $Fifo$ je genau einen Fehler (warum?).

Sei s die Seite die den letzten Seitenfehler von Phase $P-1$ verursacht (d.h. sie kommt neu in den Cache, und wird dank FIFO als letztes in Phase P verdrängt werden).
$\Longrightarrow$ Zu Beginn von Phase P ist s im Cache von Opt und von $Fifo$.
$\Longrightarrow$ Es gibt $\leq k-1$ Seiten die im Cache von Opt sind, aber nicht in dem von $Fifo$.
Während Phase P macht $Fifo$ genau k Fehler.
$\Longrightarrow$ Während P muss Opt mindestens einen Seitenfehler machen.
$\Longrightarrow$ $Fifo$ ist k-kompetitiv.

LRU ist in der Theorie ebenfalls k-kompetitiv, in der Praxis allerdings tendenziell besser als FIFO.

Satz (untere Schranke) Kein Online-Algorithmus für Paging kann eine besseren kompetitiven Faktor als k erreichen.

Beweis: Sei k die Grösse vom Cache und $k+1$ die Grösse vom Hauptspeicher. [4] Betrachte die "worst case" Eingabe $I = (k+1, s_{y_1}, s_{y_2}, , ..., s_{y_{n-1}})$, d.h. in Zeitschritt i wird die Seite angefragt die $\mathcal{A}$ zuvor erst verdrängt hat. $\mathcal{A}$ verursacht also exakt k Seitenfehler, und Opt nur einen in Zeitschritt 1.

Für alle Strategien von $\mathcal{A}$ lässt sich eine worst-case Eingabe konstruieren (siehe Idee eines *Gegenspielers* der die Strategie/den Quellcode kennt). Durch Wiederholen solcher k-langen Phasen lässt sich ausserdem eine unendlich lange Eingabe konstruieren. Eingabelänge n, $\mathcal{A}$ mit n Fehlern, Opt mit n/k Fehlern $\Longrightarrow$ k-kompetitiv. [5]

[2] Der Vorsprung eines selbstgewählten Startinhalts kann in α versteckt werden.
[3] Zusätzliches, proaktives Entfernen bringt keinen Vorteil.
[4] $k+1$ macht die Aussage nur stärker. Warum?
[5] Mit etwas Glück (abhängig davon was $\mathcal{A}$ in zukünftigen Phasen verdrängt) macht Opt sogar nur den Fehler in Zeitschritt 1, und macht danach nie wieder einen Fehler.

2.2. Randomisierte Online-Algorithmen

Motivation Randomisierung verunmöglicht es dem Gegenspieler die genaue Strategie von $\mathcal{A}$ zu kennen, d.h. es verunmöglicht ihm eine worst case Instanz zu konstruieren.

Randomisierter Online-Algorithmus Bekommt als Eingabe zusätzlich ein unendliche langes Zufallsband ϕ mit Zufallsbits (die u.a.r. 0 oder 1 sind). Jede Antwort y_i darf nur von $\phi, x_1, ..., x_i, y_1, ..., y_{i-1}$ abhängen.

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

Erwarteter kompetitiver Faktor Ein Online-Algorithmus *Rand* ist *c-kompetitiv im Erwartungswert* falls

$$\exists \alpha \geq 0 \quad \forall I \ : \quad \mathbb{E}[cost(Rand(I))] \leq c \cdot cost(Opt(I)) + \alpha$$

Das kleinste c für das dies gilt heisst *erwarteter kompetitiver Faktor*.
Rand heisst *strikt c-kompetitiv im Erwartungswert* falls $\alpha = 0$.

Wahrscheinlichkeitsverstärkung Einen randomisierten Offline-Algorithmus der mit Wahrscheinlichkeit $\frac{1}{2}$ korrekt ist, kann man k Mal wiederholen um $\frac{1}{2^k}$ zu erreichen. Online ist dies nicht möglich, da wir direkt eine Antwort auf jede Anfrage geben müssen.

Randomisierter Paging-Algorithmus RMark Eine Phase endet/beginnt wenn nach einem Seitenfehler alle Seiten unmarkiert werden.

Algorithm 1 RMark

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

Satz $RMark$ hat einen erwarteten kompetitiven Faktor von $2H_k$. [6] D.h. $RMark$ ist im Erwartungswert $\mathcal{O}(\log k)$-kompetitiv.

Beweis: Siehe auch Skript S.14ff.

Betrachte eine einzelne Phase P. O.B.d.A werden k veschiedene Seiten angefragt (eventuell auch mehrmals). Im worst case werden zuerst l "neue" Seiten angefragt, und danach $k-l$ "alte". Mit Wahrscheinlichkeit $(k-l)/k$ ist die erste alte Seite noch im Cache, dann mit Wahrscheinlichkeit $(k-l-1)/(k-1)$ die zweite alte, usw. Umgekehrt ist die i-te alte Seite mit Wahrscheinlichkeit $1-\frac{k-l-(i-1)}{k-(i-1)} = \frac{l}{k-(i-1)}$ nicht mehr im Cache. Die erwarteten Kosten während P sind also

$$l + \sum_{i=1}^{k-l} \frac{l}{k-(i-1)} = \cdots = l(H_k - H_l + 1) \leq lH_k$$

Ausserdem gilt $l \geq 1$ da jede Phase per Definition mit einer neuen Seite beginnt.

Betrachte die Kosten von Opt. Betrachte zwei aufeinanderfolgende Phasen P_{j-1}, P_j. In diesen wurden $\geq k + l_j$ verschiedene Seiten angefragt. $\implies$ Opt macht $\geq l_j$ Seitenfehler. $RMark$ und Opt machen in P_1 beide l_1 Fehler (da sie mit demselben Cache starten).

Durch unterschiedliches Gruppieren $((P_1,P_2),(P_3,P_4),...$ vs $P_1,(P_2,P_3),(P_4,P_5),...)$ erhalten wir:

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

Der kompetitive Faktor ist also

$$c \geq \frac{\sum_{i=1}^{N} H_k l_i}{\sum_{i=1}^{N} \frac{1}{2} l_i} = 2H_k$$

Verbesserung Für Paging existiert kein deterministischer Online-Algorithmus mit kompetitivem Faktor k (s.o.). Mit Randomisierung können wir im Erwartungswert aber $\mathcal{O}(\log k)$ erreichen! D.h. asymptotisch expontentieller Speedup! Dies ist asymptotisch optimal für randomisierte Algorithmen (s.u.).

2.3. Yaos Prinzip

Motivation Untere Schranke für kompetitiven Faktor von deterministischen OAs $\implies$ Untere Schranke für erwarteten kompetitiven Faktor von randomisierten OAs

Wahrscheinlichkeitsverteilung über Instanzen $\Pr_{Adv}$ $\implies$ W'keitsverteilung über Algorithmen $\Pr_{Rand}$.

Limitierung: konstante Anzahl von Instanzen $\mathcal{I} = \{I_1, ..., I_m\}$ und Algorithmen $strat(Rand) = \{A_1, ..., A_l\}$ (d.h. Eingabelänge n begrenzt).

[6] Für jedes $l \in \mathbb{N}^+$ heisst H_l die *l-te Harmonische Zahl* und $H_l := 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{l} = \sum_{i=1}^{l} \frac{1}{i}$.

Lemma 1 (1.13) Sei Π ein Optimierungsproblem, sei $\mathcal{I}$ eine Klasse von Instanzen. Sei $\Pr_{Adv}$ eine W'keitsverteilung so dass gilt:

$$\forall A \in strat(Rand) \ : \ \mathbb{E}_{Adv}[cost(A(\mathsf{I}))] \geq c \cdot \mathbb{E}_{Adv}[cost(Opt(\mathsf{I}))]$$

Dann gilt:

$$\forall Rand \ \exists I \in \mathcal{I} \ : \ \mathbb{E}_{Rand}[cost(\mathsf{A}(I)))] \geq c \cdot cost(Opt(I))$$

wobei A, I Zufallsvariablen sind aus den Wahrscheinlichkeitsräumen $\Pr_{Rand}, \Pr_{Adv}$.

Beweis: Siehe Skript S.18f.

Lemma 2 (1.14) Seien $\Pi, \mathcal{I}, \Pr_{Adv}$ wie oben. Sei $\forall$ det. OAs A_j der erwartete kompetitive Faktor $\geq c$, d.h.

$$\mathbb{E}_{Adv}\left[\frac{cost(A_j(\mathsf{I}))}{cost(Opt(\mathsf{I}))}\right] \geq c$$

Dann gilt: $\forall$ rand. OAs ist der erwartete kompetitive Faktor $\geq c$, d.h. $\exists I \in \mathcal{I}$ so dass

$$\frac{\mathbb{E}_{Rand}[cost(\mathsf{A}(I))]}{cost(Opt(I))} \geq c$$

Beweis: Siehe Skript S.20f.

Satz (Yaos Prinzip) Folgt aus Lemma 1 und 2. Seien $\Pi, \mathcal{I}, \Pr_{Adv}$ wie oben. Für jeden randomisierten Online-Algorithmus existiert dann eine Eingabe I so dass

$$\frac{\mathbb{E}_{Rand}[cost(\mathsf{A}(I))]}{cost(Opt(I))} \geq \max\left\{\left\{\min_j \frac{\mathbb{E}_{Adv}[cost(A_j(\mathsf{I}))]}{\mathbb{E}_{Adv}[cost(Opt(\mathsf{I}))]}\right\}, \left\{\min_j \mathbb{E}_{Adv}\left[\frac{cost(A_j(\mathsf{I}))}{cost(Opt(\mathsf{I}))}\right]\right\}\right\}$$

Anders formuliert (laut Wikipedia):

$$\max_{I \in \mathcal{I}} \mathbb{E}_{Rand}[cost(\mathsf{A}(I))] \geq \min_{A \in strat(Rand)} \mathbb{E}_{Adv}[cost(A(\mathsf{I}))]$$

wobei A, I Zufallsvariablen sind.

Spieltheoretische Interpretation Yaos Minimax Prinzip. Spezialfall von Von Neumanns Minimax Theorem (in Nullsummenspielen mit 2 Spielern und gemischten Strategien gibt es ein Gleichgewicht). Zero-sum game, Spieler A wählt den det. Algorithmus, Spieler B wählt die Instanz, der payoff ist $cost(A_j(I))$.

Für jeden Spieler ist "zufällig wählen" eine Strategie. Aus Yao folgt: für eine fixe Eingabe zufällig eine Algo wählen ist nicht schlechter als für einen fixen Algo zufällig eine Eingabe wählen.

Satz (Untere Schranke für randomisiertes Paging) Kein randomisierter Online-Algorithmus für Paging kann einen besseren (= kleineren) erwarteten kompetitiven Faktor als H_k erreichen.

Beweis: Siehe Skript S.27ff.

Analog zu Paging: k Cache, $k+1$ Hauptspeicher, frage zuerst s_{k+1} an, danach (neu!) jede der nicht gerade angefragten Seiten mit Wahrscheinlichkeit $\frac{1}{k}$. Eine Phase endet nach k Fehlern, d.h. nachdem alle $k+1$ Seiten mind. einmal angefragt wurden.

Betrachte einzelne Phase. Zeige dass für alle deterministischen A_j die erwarteten Kosten circa H_k-mal höher sind als die von *Opt*. Es gilt für die Eingabe während Phase P:

$$\frac{\mathbb{E}_{Adv}[cost(A_j(P))]}{\mathbb{E}_{Adv}[cost(Opt(P))]} \geq \frac{|P| \cdot \frac{1}{k}}{1} = \frac{|P|}{k}$$

Schätze ab (siehe Skript, siehe Coupon Collector): $\mathbb{E}_{Adv}[\,|P|\,] = 1 + k \cdot H_k$.

Wende Yaos Prinzip an:

$$\frac{\mathbb{E}_{Rand}[cost(\mathsf{A}(I))]}{cost(Opt(I))} \geq H_k$$

3. k-Server-Problem

Konzepte

- k-Server-Problem
- Potentialfunktionen, amortisierte Kosten
- Greedy, Double Coverage

Motivation Bewege Objekte in einem Raum zu bestimmten Punkten. Z.B. Polizisten von Dienststellen zu crime scenes, oder Taxis zu Kunden.

Metrischer Raum Sei S eine Menge von Punkten, sei dist : $S \times S \mapsto \mathbb{R}$ eine Distanzfunktion. $\mathcal{M}(S, \text{dist})$ ist ein *metrischer Raum* falls gilt: Definitheit, Symmetrie, Dreiecksungleichung.

Beispiel: Euklidischer Raum. Vollständige, gewichtete, ungerichtete Graphen mit Dreiecksungleichung.
Beobachtung: Alle Graphen mit Kastenkosten $\in \{1, 2\}$ erfüllen die Dreiecksungleichung.

k-Server Sei $\mathcal{M}(S, \text{dist})$ ein metrischer Raum. Sei $s_1, ..., s_k$ Server als Punkte in S. Sei eine Multimenge $C_i \subseteq S$ mit $|C_i| = k$ eine *Konfiguration* von Servern in Zeitschritt i.
Die *Distanz* [7] zwischen C_r und C_t sind die Kosten eines minimalen Matchings zwischen ihnen.

Eine Instanz $I = (x_1, ..., x_n)$ fragt Punkte an, so dass in Zeitschritt i ein Server nach x_i bewegt werden muss (falls dort noch keiner steht).

Ziel: $\min \sum_i costMinMatching(C_i, C_{i+1})$

Träge Ein Online-Algorithmus für k-Server heisst *träge* wenn er nur dann einen Server bewegt, wenn auf x_i noch kein Server steht. Auch bewegt er pro Zeitschritt maximal einen Server.

Dies erleichtert die Analyse. Gleichzeitig gilt (Satz):
Jeder c-kompetitive OA für k-Server kann in einen trägen OA umgewandelt werden der auch c-kompetitiv ist.

k-Server als Verallgemeinerung von Paging Cache k, Hauptspeicher $m \longrightarrow$ vollständiger Graph mit m Knoten und initial Servern auf $(v_1, ..., v_k)$. Angefragte Punkte = angefragte Seiten.

Daraus folgt eine untere Schranke (Satz): Es existiert ein metrischer Raum so dass kein deterministischer OA für k-Server besser als k-kompetitiv ist.
Frage: für Paging ist die Schranke scharf, d.h. wir kennen einen Algo (z.B. FIFO). Können wir für k-Server auch einen Algo konstruieren?

k-Server Vermutung(en)

- Es existiert ein k-kompetitiver deterministischer OA für k-Server.
- Es existiert ein im Erwartungswert $\Theta(\log k)$-kompetitiver randomisierter OA für k-Server.

Wenn dies wahr ist, dann ist k-Server genauso schwer wie Paging!

[7] Achtung Verwechslungsgefahr!

Greedy-Algorithmus Bewege immer den Server der am nähesten dran ist.

Satz: *Greedy* ist nicht kompetitiv für k-Server.

Beweis: Siehe Instanz in Figure 3. Hier gilt $\frac{cost(Greedy(I))}{cost(I)} = \frac{n}{2}$, d.h. es gibt keine Konstante c so dass *Greedy* c-kompetitiv wäre.

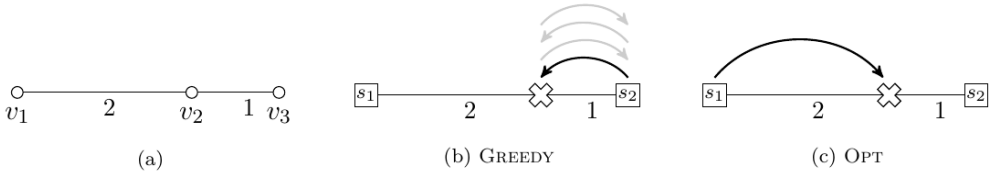

Figure 3: k-Server: *Greedy* versus *Opt*

3.1. Potentialfunktionen

Motivation Kompetivität c abschätzen über die *amortisierten Kosten*. Statt dass $cost(A(I)) \leq c \cdot cost(Opt(I)) + \alpha$ für alle I gelten muss, wollen wir zeigen dass $cost(A(x_i)) \leq c \cdot cost(Opt(x_i)) + \alpha$ für alle x_i gilt.
Dann können wir erlauben dass A in einigen Zeitschritten mehr als c-mal schlechter ist als Opt, solange er in anderen wieder weniger schlecht ist.

Potentialfunktion Sei $\mathcal{K}_{Alg}$ die Menge aller *Konfigurationen* von A auf Instanz I und sei $\mathcal{K}_{Opt}$ die Menge aller Konfigurationen eines beliebigen, aber festen, *Opt*. [8]

Dann ist eine *Potentialfunktion* Φ:

$$\Phi \ : \ \mathcal{K}_{Alg} \times \mathcal{K}_{Opt} \mapsto \mathbb{R} \qquad \text{oder} \qquad \Phi \ : \ \mathcal{I} \mapsto \mathbb{R}$$

Die Konfigurationen sind eindeutig durch die Eingabe gegeben, daher die beiden Darstellungen.

Das *Potential* in Zeitschritt i ist $\Phi(x_i)$.

Die *amortisierten Kosten* (vgl. *tatsächliche Kosten*) sind:

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

Satz Falls

$$\exists \beta \in \mathbb{R}^{+} \text{konstant } \forall i \in [1, n] \ : \ 0 \leq \Phi(x_i) \leq \beta \quad \wedge \quad amcost(A(x_i)) \leq c \cdot cost(Opt(x_i))$$

dann ist A c-kompetitiv für Π.

Dies lässt sich verallgemeinern dass Φ negativ werden darf, solange es trotzdem durch eine Konstante beschränkt ist.

Beweis: Siehe Skript S.48. Kurz:

$$cost(A(I)) = \sum_{i=1}^{n} cost(A(x_i)) = \cdots \leq c \cdot cost(Opt(I)) + \beta$$

Amortisierte Kosten einsetzen, dann Potentiale auscanceln. Dann $\alpha := \beta$ setzen.

[8] Konfiguration: nach aussen sichtbar, nicht der interne state der Turingmaschine. Z.B. Position der Server, Seiten im Cache.

3.2. k-Server auf der Linie

Die Line Betrachte den metrischen Raum $\mathcal{M}_{[0,1]} = ([0,1], \text{dist})$ mit $\text{dist}(x,y) = |x-y|$, d.h. den Zahlenstrahl der reellen Zahlen zwischen 0 und 1.

Double Coverage-Algorithmus Idee: bewege von beiden Seiten eine Server je um die selbe Distanz in Richtung x_i. Nicht träge!

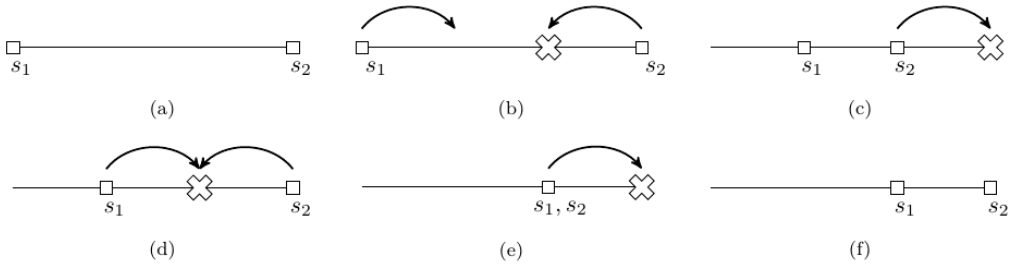

Figure 4: k-Server: *DoubleCoverage* anhand von *Greedys* worst-case Beispiel

Algorithm 2 Double Coverage (ein Zeitschritt)

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

Satz *DoubleCoverage* ist k-kompetitiv für k-Server auf $\mathcal{M}_{[0,1]}$.

Beweis: Siehe Skript S.49ff.

Ziel: Definiere Potentialfunktion Φ so dass die Bedingungen vom Satz gelten.
Sei $K_{DC} = \{p_1^{DC}, \ldots, p_k^{DC}\}$ eine Konfigurationen von DC (d.h. die Positionen seiner Server). Sei K_{Opt} analog. Seien $M_{\min}(K_{DC}, K_{Opt})$ die Kosten eines minimalen Matchings und sei $DC(K_{DC})$ die Summe der paarweisen Distanzen aller Server von DC. Wir definieren:

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

Beobachte: Φ ist positiv, konstant, und hängt nicht von n ab. Konkret (Bedingung 1): [9]

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

[9] Recall that wir uns in $\mathcal{M}_{[0,1]}$ bewegen, d.h. alle Distanzen zwischen zwei Punkten sind ≤ 1.

Zeige nun dass $\forall i$ gilt ($\star$): $\Phi(x_i) - \Phi(x_{i-1}) \leq k \cdot cost(Opt(x_i)) - cost(DC(x_i))$ (Bedingung 2).
Schätze dazu ab wie sich das Potential verändert (durch die Änderung der Konfiguration) wenn erst *Opt* und dann *DC* einen Zug machen.

Der Zug von *Opt* vergrössert Φ um $\leq k \cdot cost(Opt(x_i))$ (maximal ein Server wird um $cost(Opt(x_i))$ bewegt, nur $k \cdot M_{\min}$ ist affected).

Der Zug von *DC* verändert Φ um:

Fall 1: x_i ist "ganz aussen". OBdA wird s_{rechts} nach links verschoben. Der zweite Summand vergrössert das Potential um $\leq (k-1) \cdot cost(DC(x_i))$. OBdA existiert ein minimales Matching das s (von *Opt* bereits nach x_i bewegt) und s_{rechts} matched – siehe Fallunterscheidung). D.h. die Kosten von $k \cdot M_{\min}$ verringern sich um $k \cdot cost(DC(x_i))$. $\implies$ insgesamt gilt ($\star$).

Fall 2: x_i ist zwischen s_{links} und s_{rechts}. Der zweite Summand wird um $cost(DC(x_i))$ kleiner (da sich s_{links}, s_{rechts} näher kommen). OBdA sind vor dem Zug von *DC* s und s_{links} (oder s und s_{rechts}) gematched. Dies verringert die Kosten von $M_{\min}$ um $cost(DC(x_i))/2$. Der andere wird mit einem s''' von *Opt* gematched. Hier erhöhen sich die Kosten um $\leq cost(DC(x_i))/2$. D.h. der erste Summand bleibt gleich oder wird kleiner. $\implies$ insgesamt gilt ($\star$).

$\implies$ Voraussetung vom Satz erfüllt $\implies$ *DC* ist k-kompetitiv.

4. Advice-Komplexität

Konzepte

- Advice-Komplexität, Online-Algorithmus mit Advice
- Advice-Komplexität von Paging und k-Server
- Advice und Randomisierung

Motivation Welche Information fehlt uns? Z.B. bei Ski-Rental: müssen wir die gesamte Eingabe kennen (n), oder die Anzahl guter Tage ($\log_2 k$)? Nein – ein einzelnes Bit (kaufen oder mieten) reicht aus um optimal zu sein!

Der kompetitive Faktor sagt *wie viel* wir zahlen. Die Advice-Komplexität sagt *wofür* wir zahlen.

Modell: ein Orakel, das für eine Eingabe I deterministisch ein *Advice-Band* schreibt, von welchem der Online-Algorithmus sequentiell bis zu $b(n)$ Bits lesen kann.
Intuitiv: das Orakel sieht die gesamte Eingabe im Voraus, und kann bis zu $b(n)$ Bits leaken.

Online-Algorithmus mit Advice Sei $I = (x_1, ..., x_n)$ eine Eingabe. Ein *Online-Algorithmus* $\mathcal{A}$ *mit Advice* berechnet eine Ausgabe $\mathcal{A}^\phi(I) = (y_1, ..., y_n)$ wobei y_i nur von $\phi, x_1, ..., x_n, y_1, ..., y_{i-1}$ abhängt. ϕ ist ein binärer *Advice-String*.

$\mathcal{A}$ ist *c-kompetitiv mit Advice-Komplexität* $b(n)$ falls $\mathcal{A}$ maximal $b(n)$ Advice-Bits liest und falls gilt:

$$\exists \alpha \in \mathbb{R}^+ \text{konstant } \forall i \in [1, n] \ : \ cost(\mathcal{A}^\phi(I)) \leq c \cdot cost(Opt(I)) + \alpha$$

Falls $\alpha = 0$, heisst $\mathcal{A}$ *strikt* c-kompetitiv mit Advice-Komplexität $b(n)$.
Falls $\mathcal{A}$ strikt 1-kompetitiv mit Advice-Komplexität $b(n)$ ist, heisst $\mathcal{A}$ *optimal*.

Intuitiv: für jede Eingabe soll ein Advice-String ϕ existieren, der es $\mathcal{A}$ erlaubt einen kompetitiven Faktor c zu erreichen.

Satz (Ski-Rental) Für Ski-Rental existiert ein optimaler Online-Algorithmus mit Advice der 1 Advice-Bit verwendet.

Triviale Schranke Für Paging und für k-Server existieren optimale OAs mit Advice die $n\lceil \log_2 k \rceil$ Advice-Bits verwenden.

Idee: enkodiere den Index der Seite die *Opt* verdrängt bzw. des Servers den *Opt* bewegt.

Satz (Paging) Es existiert ein OA mit Advice *Lin* für Paging der $\leq n + k$ Advice-Bits verwendet.

Beweis: Idee: nähere das Verdänge-Verhalten von *Opt* an. *Lin* hat für jede Cache-Seite ein Bit das sie als "aktiv" markiert. Eine Seite ist "aktiv" wenn sie noch einmal angefragt wird bevor *Opt* sie verdrängt.
$\implies$ Bei Seitenfehlern werden nur Seiten verdrängt die *Opt* auch verdrängen wird bevor sie wieder angefragt werden. Ein Seitenfehler geschieht nur für Seiten die *Opt* auch nicht im Cache hat.

Anzahl Advice-Bits: k für die Initialisierung, n um für jede angefragte Seite anzuzeigen ob sie aktiv sein wird. $\implies n + k$

Anzahl Seitenfehler: nicht mehr als *Opt*, d.h. *Lin* ist optimal!

Satz (k-Server) Für eine Instanz der Länge k muss jeder optimale OA mit Advice für k-Server $\geq k(\log_2 k - \log_2 e)$ Advice-Bits verwenden.

Beweis: Konstruiere einen vollständigen Graph wie in Figure 5. Server $s_1, ..., s_k$ und Gruppen von Knoten $\overline{G_0}, ..., \overline{G_{k-1}}$. Kantenkosten 2 für dicke Kanten, 1 für dünne (einige teure Kanten fehlen der Übersicht halber).

Eingabe $I = (x_1, ..., x_k)$ wobei x_i ein beliebiger Knoten aus $\overline{G_{i-1}}$ ist.

Idee: es existiert genau eine Reihenfolge nur die billigen Kanten zu nutzen. Wird ein Server "falsch" bewegt, muss in einem späteren Zeitschritt mind. einmal eine teure Kante benutzt werden. Insbesondere haben alle s_i die zur Anfrage x_j teure Kanten hatten auch zu x_{j+1} teure Kanten. x_k hat zu genau einem s_i eine billige Kante. [10]

$\implies$ Jede Instanz I korreliert mit einer Permutation von $\{1, ..., k\}$. Es existiert eine optimale Lösung mit Kosten k.

Wir brauchen einen eindeutigen Advice-String für jede Instanz I, d.h. wir brauchen

$$\log_2(k!) \geq ... \text{ Stirling-Formel } ... \geq k(\log_2 k - \log_2 e)$$

viele Advice-Bits.
Falls wir weniger Advice-Bits haben, dann muss es zwei Instanzen $I_1 \neq I_2$ geben auf denen sich $\mathcal{A}$ gleich verhält (da er deterministisch ist und den gleichen Advice liest). Dann kann $\mathcal{A}$ aber nicht auf beide Instanzen optimal sein, da er in einem Fall eine teure Kante benutzen muss.

Dies lässt sich verallgemeinern für Instanzen der Länge n.

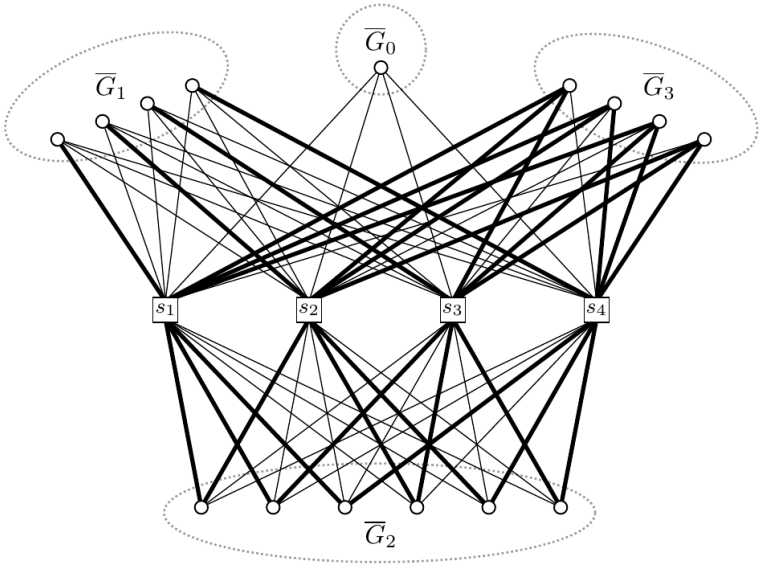

Figure 5: Konstruktion Graph für Beweis Advice-Komplexität von k-Server

Advice und Randomisierung Idee: Orakel muss zwar deterministisch sein, kann aber die Advice-Bits als die "besten" Zufallsbits wälen.

Wenn ein c-kompetitiver randomisierter OA existiert der $b(n)$ Zufallsbits liest, so existiert auch ein c-kompetitiver OA mit Advice der $b(n)$ Advice-Bits liest, und vice versa.

Wenn *kein* OA mit Advice existiert der $b(n)$ Advice-Bits liest, so existiert auch *kein* randomisierter OA der $b(n)$ Zufallsbits liest.

[10] Mit s_i sind hier die Startpositionen der Server gemeint, nicht die Server selbst.

Satz (6.1) Sei Π ein Online-Minimierungsproblem für das $m(n)$ verschiedene Instanzen der Länge n existieren. [11] Sei *Rand* ein randomisierter OA mit erwartetem kompetitiven Faktor c.

Dann existiert ein OA mit Advice der $((1+\varepsilon)c)$-kompetitiv ist und Advice-Komplexität

$$\log\log(m(n))$$

hat. [12] D.h. die Anzahl benötigter Advice-Bits hängt nur von der Anzahl Instanzen ab.

Umgekehrt: falls ein OA mit Advice mindestens mehr Advice-Bits braucht, dann kann kein randomisierter OA existieren.

[11] Wir müssen die Anzahl der möglichen Eingaben einschränken.
[12] Stark vereinfacht im Vergleich zum Skript.

5. Online-Rucksackproblem

Konzepte

- Online-Rucksackproblem
- Ansatz: deterministisch, mit Advice, randomisiert

Online-Rucksackproblem Eingabe $I = (w_1, ..., w_n)$, wobei wir vereinfachen mit Gewicht = Wert. $w_i \in \mathbb{R}, 0 \leq w_i \leq 1$ (nicht $w_i \in \mathbb{N}$ wie bei Offline!). In jedem Zeitschritt i muss A entscheiden ob er w_i einpacken will. Dies kann nicht aufgeschoben oder revidiert werden.

Zulässige Lösung: Menge von Indizes $S \subseteq \{1, ..., n\}$ so dass $gain(A(I)) := \sum_{i \in S} w_i \leq 1$.
Ziel: $gain(A(I))$ maximieren.

5.1. Deterministisch

Satz Jeder deterministische OA für das Rucksackproblem hat einen beliebig grossen (d.h schlechten) kompetitiven Faktor.

Beweis: Schritt 1: Gegenspieler bietet Objekt w mit Gewicht $\varepsilon > 0$ an. Schritt 2: Falls w akzeptiert, biete Objekt mit Gewicht 1 an. Falls w verworfen, breche ab.
$\Longrightarrow$ kompetitiver Faktor von $\frac{1}{\varepsilon}$ bzw. von $\frac{\varepsilon}{0}$/unendlich/undefiniert.

Trotzdem ist *Greedy* auf bestimmte Klassen von Eingaben gut:

Satz Für jede Instanz mit Gewichten $\leq \beta$ ist *Greedy* optimal oder erreicht einen Gewinn von $> 1 - \beta$.

Beweis: Fallunterscheidung: Gesamtgewicht aller angebotenen Objekte ≤ 1 (optimal), oder > 1 (dann ist in *Greedy*'s Sack nur $< \beta$ Platz leer).

Satz Für jede Instanz für die $gain(Opt(I)) \leq \frac{1}{2}$ ist, ist *Greedy* optimal.

Beweis: Offensichtlich gilt $gain(Greedy(I)) \leq \frac{1}{2}$. Falls $gain(Greedy(I)) < gain(Opt(I))$, dann ... TODO

5.2. Mit Advice

Satz (Triviale untere Schranke) Es existiert ein optimaler OA mit Advice für das Rucksackproblem der n Advice-Bits benutzt.

Beweis: Left as an exercise to the reader.

Satz (Scharfe Schranke) Jeder optimale OA mit Advice für das Rucksackproblem muss mindestens $n-1$ Advice-Bits benutzen.

Beweis: Konstruiere Klasse von Instanzen: $I = (\frac{1}{2}, \frac{1}{4}, ..., \frac{1}{2^{n-1}}, w_b)$ wobei $w_b = 1 - \sum_{i=1}^{n-1} b_i 2^{-i}$ für einen beliebigen Binärstring b der Länge $n-1$. Für $n = 8$ z.B. führt $b = 1101101$ zu

$$w_b = 1 - \left(\frac{1}{2} + \frac{1}{4} + 0 + \frac{1}{16} + \frac{1}{32} + 0 + \frac{1}{128}\right)$$

Die eindeutige optimale Lösung hat Gewinn 1, und füllt mit den ersten $n-1$ Objekten die "Lücken" von w_b auf.

Es gibt 2^{n-1} veschiedene Binärstrings der Länge $n-1$, d.h. es gibt ebenso viele Instanzen. Ein optimaler OA mit Advice muss alle unterscheiden können, d.h. er braucht $n-1$ Advice-Bits.
Mit weniger Bits verhält er sich auf zwei Instanzen gleich (Schubfachprinzip), und kann nicht auf beide optimal sein.

Algorithmus: KPone Lese ein Advice-Bit. Es entscheidet zwischen: Greedy von Beginn an, oder Warten auf ein Objekt mit Gewicht $> \frac{1}{2}$ und ab dann Greedy.

Satz Der OA mit Advice $KPone$ für das Rucksackproblem ist strikt 2-kompetitiv.

Beweis:
Fall 1: ein Objekt mit Gewicht $> \frac{1}{2}$ existiert. $KPone$ packt dieses ein, $\implies gain(KPone(I)) > \frac{1}{2}$.
Fall 2: Setze $\beta = 2$ und wende obigen Satz für det. OAs an $\implies gain(KPone(I)) \geq 1 - \frac{1}{2} = \frac{1}{2}$ (oder optimal).

Dann gilt:

$$\frac{gain(Opt(I))}{gain(KPone(I))} \leq \frac{1}{gain(KPone(I))} \leq \frac{1}{\frac{1}{2}} = 2$$

5.3. Randomisiert