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 taken from the lecture scripts, and drawn by Jan Kleine and the authors.

Part I.
Approximations-Algorithmen

Siehe auch das Skript zu *Algorithmik für Schwere Probleme* [ASP], insbesondere Kapitel 5.

1. Approximations-Algorithmen

Konzepte

- Optimierungsproblem, Approximations-Algorithmus, Approximationsgüte
- (Metrisches) TSP, Spannbaum-Algorithmus, Christofides
- Min-VCP, 2-Approximation
- Min-SCP, $\ln(n)$-Approximation (Greedy)
- Weighted-VCP, 2-Approximation (LP, Relaxieren)

Definition Optimierungsproblem, Approximationsgüte Siehe [ASP].

Travelling Salesperson Problem (TSP) Normales und Metrisches TSP (Δ-TSP). Siehe [ASP].

Spannbaum-Algorithmus für Δ-TSP 2-Approximation. Siehe [ASP].

Christofides-Algorithmus $\frac{3}{2}$-Approximation. Siehe [ASP].

Minimum Vertex Cover Problem (Min-VCP)
Eingabe: $G = (V, E)$
Zulässige Lösungen: $C \subseteq V$ so dass jede Kante in E mind. einen Endpunkt in C hat.
Kosten: $cost(C) = |C|$
Ziel: min

Intuitiv: Alle Kanten mit mindestens einem Knoten abdecken.

2-Approximation für Min-VCP Siehe [ASP].

Set Cover Problem (SCP)
Eingabe: Grundmenge X und Mengensystem $\mathcal{F} \subseteq Pot(X)$ mit $X = \bigcup_{Q \in \mathcal{F}} Q$
Zulässige Lösungen: $C \subseteq \mathcal{F}$ so dass $X = \bigcup_{Q \in C} Q$
Kosten: $cost(C) = |C|$
Ziel: min

Intuitiv: alle Grundelemente mit mindestens einem Set abdecken. Verallgemeinerung von VCP auf Hypergraphen. [1]

[1] Aus VCP-Eingabe $G = (V, E)$ konstruiere SCP-Eingabe $(E, \{E_1, ..., E_n\})$ wobei $E_i \subseteq E$ alle Kanten enhält die adjazent zu $v_i \in V$ sind.

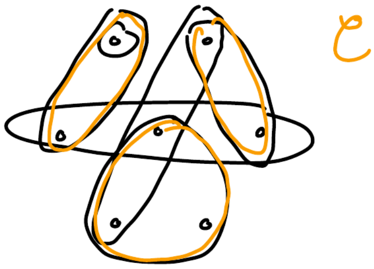


Figure 1: Beispiel SCP

Greedy-Algorithmus für SCP
Eingabe: $(X, \mathcal{F})$

1. Initialisierung: $C \leftarrow \emptyset, U \leftarrow X$ (U = alle noch nicht überdeckten Grundelemente)
2. while $U \neq \emptyset$ do:
   - Wähle $S \in \mathcal{F}$ so dass $|S \cap U|$ maximal
   - $U \leftarrow U \setminus S$
   - $C \leftarrow C \cup \{S\}$

Ausgabe: C

Theorem Greedy-SCP ist ein polynomieller $\ln(n)$-Approximationsalgorithmus für SCP.

Beweis (Laufzeit):

Eingabegrösse $n = |X| \cdot |\mathcal{F}|$. $\min\{|X|, |\mathcal{F}|\} \leq (|X| \cdot |\mathcal{F}|)^{1/2}$ Iterationen à $\mathcal{O}(|X| \cdot |\mathcal{F}|)$
$\implies \mathcal{O}((|X| \cdot |\mathcal{F}|)^{3/2}) = \mathcal{O}(n^{3/2})$

Beweis (Approximationsgüte):

Schritt 1: Verteile die Kosten von C auf die einzelnen Elemente.

Sei $C = \{S_1, ..., S_k\}$ die Ausgabe und S_i in Schritt i gewählt. Sei $D_{C,i} = S_i \setminus \bigcup_{j=1}^{i-1} S_j$ die Differenzmenge, d.h. die Menge der von S_i neu überdeckten Elemente.

Sei $\forall x \in D_{C,i}$ das $weight_C(x) = \frac{1}{|D_{C,i}|}$ und sei $\forall T \subseteq X$ das $weight_C(T) = \sum_{x \in T} weight_C(x)$. Beobachte dass (1) $weight_C(D_{C,i}) = 1$ und dass (2) jedes $x \in X$ in genau einem $D_{C,i}$ ist.

$\implies cost(C) = k = \sum_{i=1}^{k} weight_C(D_{C,i}) = \sum_{x \in X} weight_C(x)$

Schritt 2: Schätze das Gewicht eines beliebigen $S \in \mathcal{F}$ ab.

OBdA sei $S = \{x_1, ..., x_l\}$ so dass x_j bevor oder gemeinsam mit x_{j+1} überdeckt wird.
Ziel: $weight_C(x_j) \leq \frac{1}{l-j+1}$. Beweis per Widerspruch:

Nehme an es gelte für ein x_j:

$$weight_C(x_j) > \frac{1}{l-j+1} \iff |D_{C,i}| = \frac{1}{weight_C(x_j)} < l - j + 1$$

D.h. der Algorithmus überdeckt $< l - j + 1$ neue Elemente mit S_i (mit welchem er auch x_j neu überdeckt). Mit S hätte er aber $= l - j + 1$ neue Elemente überdecken können. Widerspruch zu Greedy!

Nun gilt $\forall S \in \mathcal{F}$:

$$weight_C(S) = \sum_{j=1}^{l} weight_C(x_j) \leq \sum_{j=1}^{l} \frac{1}{l-j+1} \leq \sum_{j=1}^{l} \frac{1}{j} = Har(l) \leq Har(\max\{|S| \mid S \in \mathcal{F}\})$$

wobei Har die Harmonische Zahl ist.

Schliesslich gilt:

$$\begin{aligned} cost(C) &= \sum_{x \in X} weight_C(x) \leq \sum_{S \in C_{Opt}} weight_C(S) \leq \sum_{S \in C_{Opt}} Har(\max\{|S| \mid S \in \mathcal{F}\}) \\ &\leq |C_{Opt}| \cdot Har(|X|) \leq |C_{Opt}| \cdot \ln(n) \end{aligned}$$

wobei C_{Opt} eine optimale Lösung ist.

$\Longrightarrow$ Min-SCP $\in$ LOGAPX. Greedy ist optimal für Min-SCP!

Weighted-VCP (WVCP)
Eingabe: (G, c) wobei $G = (V, E),\ c : V \mapsto \mathbb{N}^+$
Zulässige Lösungen: jedes vertex cover C von G
Kosten: $cost(C) = \sum_{v \in C} c(v)$
Ziel: min

Lineare Programmierung LP ist folgendes Optimierungsproblem:
Eingabe: Variablen $(x_1, ..., x_n)^T$, Konstanten $A^{n \times m} = (a_{ij})_{ij},\ b = (b_1, ..., b_m)^T,\ c = (c_1, ..., c_n)^T$
Ziel: $\min c^T x = \min \sum c_i \cdot x_i$ unter der Nebenbedingung dass $Ax = b$.

Theorem

- $x_i \in \mathbb{R}$ (LP) $\Longrightarrow$ in P
- $x_i \in \mathbb{Z}$ (Ganzzahl/Integer LP, ILP) $\Longrightarrow$ NP-schwer
- $x_i \in \{0, 1\}$ (0/1-LP) $\Longrightarrow$ NP-schwer

WVCP als LP Als 0/1-LP: minimiere $\sum c(v_i) \cdot x_i$ unter den Nebenbedingungen:

- $\forall \{r, s\} \in E :\ x_r + x_s \geq 1$
- $\forall j \in [1, n] :\quad x_j \in \{0, 1\}$

Intuitiv: $x_j = 1$ falls $v_i \in C$.

Relaxiere zu LP: Ersetze $x_j \in \{0, 1\}$ durch $x_j \geq 0$.

Algorithmus LP-VC für WVCP
Eingabe: $I = (G, c)$

1. Stelle I als $I_{0/1-LP}(I)$ dar und relaxiere zu $I_{LP}(I)$.
2. Löse $I_{LP}(I)$. Sei $x = (x_1, ..., x_n), x_i \in \mathbb{R}^+$ die gefundene optimale Lösung.
3. Setze $C = \{v_i \mid x_i \geq \frac{1}{2}\}$

Ausgabe: C

Theorem LP-VC ist eine 2-Approximation für WVCP.

Beweis:

Laufzeit: offensichtlich.

Korrektheit: $x_r + x_s \geq 1 \implies x_r \geq \frac{1}{2} \vee x_s \geq \frac{1}{2} \implies v_r \in C \vee v_s \in C \implies \{r, s\}$ abgedeckt

Approximationsgüte: Beachte dass: $Opt_{WVCP}(I) = Opt_{0/1-LP}(I_{0/1-LP}(I)) \geq Opt_{LP}(I_{LP}(I))$.
Es gilt:

$$cost(C) = \sum_{v \in C} c(v) = \sum_{x_i \geq \frac{1}{2}} c(v_i) \leq \sum_{x_i \geq \frac{1}{2}} \underbrace{2 \cdot x_i}_{\geq 1} \cdot c(v_i) \leq 2 \cdot \sum_{i=1}^{n} x_i \cdot c(v_i) = 2 \cdot Opt_{LP}(I_{LP}(I)) \leq 2 \cdot Opt_{WCP}(I)$$

2. Approximationsschemata

Konzepte

- PTAS, FPTAS
- SKP: 2-Approximation (Greedy), PTAS-SKP (Durchschnittsargument)
- KP: DPKP, FPTAS

Definition PTAS und FPTAS Eingabe (I, ε). PTAS: Laufzeit ist polynomiell in $|I|$ und beliebig in ε^{-1}. FPTAS: Laufzeit ist polynomiell in $|I|$ und in ε^{-1}. Approximationsgüte $(1+\varepsilon)$. Siehe [ASP].

Einfaches Rucksackproblem (Simple Knapsack Problem SKP) Gewichte = Kosten. NP-schwer. Siehe [ASP].

Greedy-SKP 2-Approximation. Absteigend sortieren, $\mathcal{O}(n \log n)$. Siehe [ASP].

PTAS-SKP $(1+\varepsilon)$-Approximation. $k \leftarrow \lceil \frac{1}{\varepsilon} \rceil$. Optimale Lösung für alle $\mathcal{O}(n^k)$ Teilmengen der Grösse k. Dann greedy erweitern in je $\mathcal{O}(n)$. Kosten mit *Durchschnittsargument*: $w_q \leq \frac{w_1+\ldots+w_k+w_q}{k+1} \leq \frac{cost(T_{Opt})}{k+1}$. Siehe [ASP].

Allgemeines Rucksackproblem (KP) Eingabe $I = (w_1, ..., w_n, c_1, ..., c_n, b)$. Siehe [ASP].

Exakter Algorithmus für KP (DPKP) Dynamische Programmierung. Siehe [ASP].

Sei I_i die Teilinstanz der ersten i Elemente. Berechne Tripel:

$$(k, W_{i,k}, T_{i,k}) \in (0, ..., \sum c_j) \times (0, ..., b) \times Pot(1, ..., n) = \text{Nutzen} \times \text{Gewicht} \times \text{Teilmenge}$$

wobei $W_{i,k}$ für Nutzen k minimal ist und $W_{i,k} \leq b$. Sei $TRIPLE_i$ die Menge alle Tripel für I_i.

Iteriere über alle i, und alle $TRIPLE_i$, und erweitere die Tripel um das i-te Element. Gebe das grösste k aus allen gefundenen Tripeln aus.

Laufzeit: $\mathcal{O}(|I| \cdot \sum c_j) = \mathcal{O}(n \cdot n \cdot \text{max-int}(I)) \implies$ pseudopolynomiell

Falls $b \ll \sum c_j$: speichere Tripel für jedes mögliche Gewicht den maximalen Nutzen (anstatt für jeden möglichen Nutzen das minimale Gewicht).

FPTAS-KP Siehe [ASP].

1. $t \leftarrow \frac{\varepsilon \cdot c_{max}}{(1+\varepsilon) \cdot n}$
2. Runde $c'_i \leftarrow \lfloor \frac{c_i}{t} \rfloor$
3. DPKP auf gerundete Instanz

Korrektheit: Lösung bleibt zulässig. Approximationsgüte: messy, siehe Buch/[ASP].
Laufzeit: $\mathcal{O}(n + n \cdot \sum c'_j) = \mathcal{O}(\frac{1}{\varepsilon} n^3) \implies \text{poly}(n, \varepsilon^{-1})$

3. Klassifizierung von Optimierungsproblemen

Siehe auch [ASP].

NPO $U = (L, M, cost, goal) \in NPO$ falls folgende Operationen polynomiell sind:

(1) Lesen der Eingabe/Ausgabe, (2) Verifizieren der Eingabe/Ausgabe auf Zulässigkeit, (3) Berechnen der Kosten einer Lösung.

PO Falls eine optimale Lösung in Polynomzeit berechenbar ist. $PO \subseteq NPO$.

Schwellwertsprache Sprache von Entscheidungsproblemen.

$$Lang_U = \{(I, k) \mid I \in L, k \in \mathbb{N}, Opt_U(I) \lesseqgtr k\}$$

NP-schwer (Optimierungsproblem)
Optimierungsproblem U heisst NP-schwer $\iff$ Entscheidungsproblem $Lang_U$ ist NP-schwer

NP-Vollständigkeit macht keinen Sinn für Optimierungsprobleme.

Approximationsklassen Beachte dass Approximationsalgorithmen in Polynomzeit laufen.

Klasse	Enthält Probleme falls...	Beispiele
PO		
FPTAS	$\exists$ ein FPTAS	KP
PTAS	$\exists$ ein PTAS	SKP, Euklidisches TSP, Makespan-Scheduling
APX	$\exists$ Approx.-Alg. mit konstanter Güte	Min-VCP, Max-CUT, Δ-TSP
LOGAPX	$\exists$ Approx.-Alg. mit logarithmischer Güte	Min-SCP
POLYAPX	$\exists$ Approx.-Alg. mit polynomieller Güte	Max-CLIQUE
NPO		TSP

4. Nichtapproximimierbarkeit

Konzepte

- Pseudopolynomielle Algorithmen, Stark NP-schwer
- AP-Reduktionen, Max-SAT $\leq_{AP}$ Max-CLIQUE
- GP-Reduktionen, Lückenproblem
- MAX-E3SAT $\leq_{GP}$ MAX-2SAT, Max-CLIQUE $\leq_{GP}$ Max-CLIQUE
- Probabilistische Verifizierer, $\mathrm{GAP}_{1-\varepsilon,1}$(E3SAT)
- Klasse PCP, PCP-Theorem, $\mathrm{GAP}_{\frac{15}{16},1}$(MAX-3SAT) NP-schwer

Nichtapproximimierbarkeit Motivation: zeige untere Schranken für die Polynomzeit-Approximierbarkeit von Problemen. Methoden:

- Reduktion auf NP-schwere Entscheidungsprobleme
- Approximationserhaltende Reduktionen (AP-Reduktionen)
- Anwendung PCP-Theorem

4.1. Reduktion auf NP-schwere Entscheidungsprobleme

Theorem Falls $P \neq NP$, so existiert kein polynomieller Approximationsalgorithmus für das TSP mit Approximationsgüte $p(n)$ für ein Polynom p. $\implies$ TSP $\notin$ POLYAPX.

Beweis: Siehe [ASP]. Reduktion vom Hamiltonkreisproblem HCP (NP-schwer) auf die 2^n-Approximation von TSP. Polynomzeit-Transformation.

$$HCP = \{G = (V, E) \mid G \text{ enthält einen Hamiltonkreis}\}$$

Hamiltonkreis: jeder Knoten wird genau einmal besucht.

Zahlproblem (integer value problem IVP) Siehe [ASP].

Pseudopolynomieller Algorithmen $\mathrm{time}_A(x) \in \mathcal{O}\left(\mathrm{poly}(|I|, \text{max-int}(I))\right)$. Siehe [ASP].

Stark NP-schwer Zahlproblem U heisst *stark NP-schwer* falls das p-beschränkte Teilproblem[2] NP-schwer ist, für ein Polynom p. Siehe [ASP].

Theorem U stark NP-schwer $\implies$ $\nexists$ pseudopolynomieller Algorithmus für U
Siehe [ASP].

[2] D.h. $\text{max-int}(I) \leq p(|I|)$.

4.2. Approximationserhaltende Reduktionen (AP-Reduktionen)

Idee Gegeben ein Problem U_1 das bekanntermassen kein PTAS zulässt. Zeige dass ein anderes Problem U_2 auch kein PTAS zulässt durch Reduktion von U_1 auf U_2. Wichtig: die Reduktion muss die Approximationsgüte erhalten.

AP-reduzierbar Seien $U_1 = (L_1, M_1, cost_1, goal_1)$ und U_2 Optimierungsprobleme. U_1 ist *AP-reduzierbar* auf U_2, notiert $U_1 \leq_{AP} U_2$, falls Funktionen

- $F \ : \ L_1 \times \mathbb{Q}^+ \mapsto L_2$
- $H \ : \ L_1 \times \mathbb{Q}^+ \times \bigcup_{y \in L_2} M_2(y) \mapsto \bigcup_{x \in L_1} M_1(x) \quad ; \ x \geq 0$

und eine Konstante $\alpha > 0$ existieren so dass:

(i) $\forall x \in L_1$ mit $M_1(x) \neq \emptyset$, $\forall \varepsilon \in \mathbb{Q}^+ :\quad F(x, \varepsilon) \in L_2$ und $M_2(F(x, \varepsilon)) \neq \emptyset$

(ii) $\forall x \in L_1$ mit $M_1(x) \neq \emptyset$, $\forall \varepsilon \in \mathbb{Q}^+$, $\forall y \in M_2(F(x, \varepsilon)) :\quad H(x, \varepsilon, y) \in M_1(x)$

(iii) F, H in Zeit poly$(|x|, |y|)$ berechenbar für ein fixes ε

(iv) Zeitkomplexität von F, H wächst nicht mit ε für alle fixen $|x|, |y|$ [3]

(v) $\forall x \in L_1$, $\forall \varepsilon \in \mathbb{Q}^+$, $\forall y \in M_2(F(x, \varepsilon))$:

$$R_{U_2}(y, F(x, \varepsilon)) \leq 1 + \frac{\varepsilon}{\alpha} \quad \Longrightarrow \quad R_{U_1}(H(x, \varepsilon, y), x) \leq 1 + \varepsilon$$

D.h. die Approximationsgüte bleibt erhalten (α erlaubt eine leichte Veränderung).

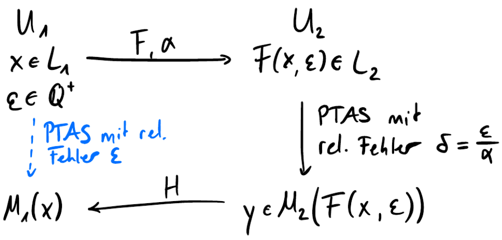


Figure 2: AP-Reduktionen

Lemma Seien U_1, U_2 Optimierungsprobleme wie oben. Falls $U_1 \leq_{AP} U_2$, so gilt:

$$\exists \text{ PTAS für } U_2 \implies \exists \text{ PTAS für } U_1$$

$$\nexists \text{ PTAS für } U_1 \implies \nexists \text{ PTAS für } U_2$$

Beweis: offensichtlich. Siehe Buch S.320.

APX-vollständig Ein Problem $U \in NPO$ heisst *APX-vollständig (APX-complete)* falls gilt:

1. $U \in APX$
2. $\forall W \in APX \ : \ W \leq_{AP} U$

[3] Nicht notwendig, aber wünschenswert.

Max-SAT Eingabe: Formel $C = C_1 \wedge ... \wedge C_m$ in KNF über Variablen $x_1, ..., x_n$. Ziel: Belegung die die Anzahl erfüllter Klauseln maximiert.

Max-CLIQUE (Max-CL) Eingabe: ungerichteter Graph G. Ziel: Clique in G mit maximaler Knotenanzahl.

Lemma Max-SAT $\leq_{AP}$ Max-CLIQUE

Beweis: Konstruktion (sei $C_i = l_{i1} \vee ... \vee l_{ij_i}$):

- $\alpha = 1$
- $F(C, \varepsilon) = F(C) = G_C = (V, E)$ mit $V = \{(i,k) \mid 1 \leq i \leq m \ , \ 1 \leq k \leq j_i\}$ (jede occurence eines Literals hat einen Knoten) und $E = \{\{(r,s),(p,q)\} \mid r \neq p \ , \ l_{rs} \not\equiv \bar{l}_{pq}\}$ (keine Kanten innerhalb einer Klausel, und keine Widersprüche) [4]
- H wählt eine Belegung γ wie folgt: für die Literale in der gefundenen Max-Clique, wähle $x_i = 1$ und $\overline{x}_i = 0$. Wähle alle anderen Literale beliebig.

Dies erfüllt unsere Bedingungen: (i) Eingabe wird gemapped. (ii) Ausgabe wird gemapped. (iii) F, H in Polynomzeit berechenbar. (iv) F, H unabhängig von ε. (v) Jede Clique Q der Grösse q erfüllt $\geq q$ Klauseln. Jede Belegung die r Klauseln erfüllt, bestimmt eine Clique der Grösse genau r.
$\implies cost_{Max-SAT}(H(G_C)) \geq cost_{Max-CLIQUE}(Q)$.

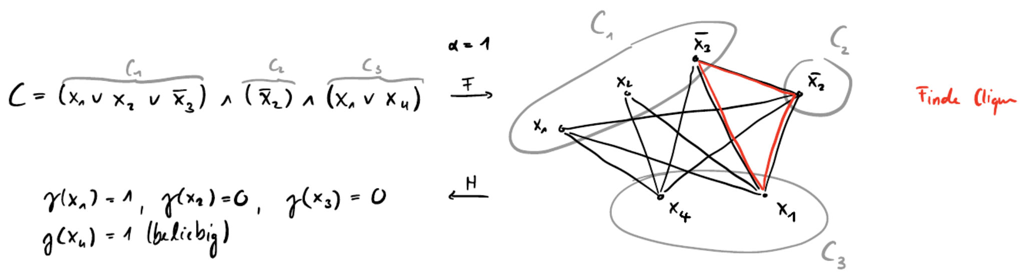


Figure 3: Beispiel AP-Reduktion Max-SAT $\leq_{AP}$ Max-CLIQUE

Fazit AP-Reduktionen sind intuitiv, aber in der Praxis schwer zu finden.

---

[4] D.h. F ist unabhängig von ε.

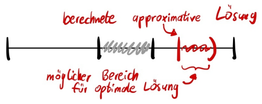
4.3. Lückenerhaltende Reduktionen (GP-Reduktionen)

Schwellwertproblem Liegt die optimale Lösung oberhalb oder unterhalb des Schwellwerts t?

Lückenproblem Liegt die optimale Lösung oberhalb oder unterhalb der Lücke $[s, c)$? D.h. ist $Opt < s \vee Opt \geq c$? Falls Lösung in der Lücke: keine Aussage.

Siehe *Promise-Problem*: Es ist garantiert dass die Eingabe nicht $Opt \in [s, c)$ hat.

Fallunterscheidung: Sei y die berechenbar Lösung.

- Fall 1: $y \geq c$ (oberhalb): 

$\implies$ *Opt* muss oberhalb liegen.
- Fall 2: $y < s$ (unterhalb):  $\implies$ *Opt* muss unterhalb liegen.
Annahme: Approximation ist gut genug, so dass sie nicht über die Lücke geht.
- Fall 3: $y \geq s$ (innerhalb):  $\implies$ *Opt* muss oberhalb liegen (kann nicht in der Lücke liegen).

In allen Fällen ist das Lückenproblem entscheidbar.

$$\exists \text{ gute Approximation (kleine Lücke möglich)} \implies \text{Lückenproblem entscheidbar}$$

$$\text{Lückenproblem schwer} \implies \nexists \text{ gute Approximation}$$

Beispiel: Max-SAT ist selbst bei Lücke $[s, c) = \left[\frac{7}{8}, 1\right)$ noch schwer.

Figure 4: Schwellwertproblem und Lückenproblem

Definition Lückenproblem GAP Sei $s, c \in \mathbb{R}^+, 0 \leq s \leq c \leq 1.$[5] Sei $U = (L, M, cost, goal) \in NPO$. Das Lückenproblem $GAP_{s,c}(U)$ ist folgendes Entscheidungsproblem:

Eingabe: $x \in L$ so dass $\frac{Opt_U(x)}{|x|} < s$ oder $\frac{Opt_U(x)}{|x|} \geq c$.
Ausgabe: JA, falls $\frac{Opt_U(x)}{|x|} \geq c$, sonst NEIN.

[5]Wir normalisieren auf $[0, 1]$, siehe auch $|x|$ unten.

Lemma $GAP_{s,c}(U)$ NP-schwer ist $\implies \nexists$ polyzeit $\frac{c}{s}$-Approximation für U (falls P $\neq$ NP).

Beweis: Per Widerspruch. Nehme an es gäbe einen polyzeit $\frac{c}{s}$-Approximationsalgorithmus A. OBdA sei U ein Maximierungsproblem. Zeige dass gilt:

$$cost(A(x)) < s \cdot |x| \iff Opt_U(x) < s \cdot |x|$$

"$\Leftarrow$" : offensichtlich da $cost(A(x)) \leq Opt_U(x)$.

"$\Rightarrow$" : Nehme (für Widerspruch) an dass $Opt_U(x) \geq s \cdot |x|$. Da die Lücke leer sein muss, gilt also $Opt_U(x) \geq c \cdot |x|$. Ausserdem gilt per Definition von $\frac{c}{s}$-Approximation $\frac{Opt_U(x)}{cost(A(x))} \leq \frac{c}{s}$. Nun folgt ein Widerspruch:

$$cost(A(x)) \geq \frac{s}{c} \cdot Opt_U(x) \geq \frac{s}{c} \cdot c \cdot |x| \geq s \cdot |x|$$

$\implies$ mit A lässt sich $GAP_{s,c}(U)$ entscheiden. Widerspruch zu NP-schwer!

GP-Reduktion Seien U_1, U_2 Maximierungsprobleme. Eine *Lückenerhaltende Reduktion (gap-preserving reduction, GP-Reduktion)* von U_1 zu U_2 mit Parametern (s, c) und (s', c') ist ein polyzeit Algorithmus A mit:

(i) $\forall x \in L_1 \; : \; A(x) \in L_2$

(ii) $\frac{Opt_{U_1}(x)}{|x|} \geq c \implies \frac{Opt_{U_2}(A(x))}{|A(x)|} \geq c'$

(iii) $\frac{Opt_{U_1}(x)}{|x|} < s \implies \frac{Opt_{U_2}(A(x))}{|A(x)|} < s'$

Bemerkung: Falls $\exists$ GP-Reduktion und $GAP_{s,c}(U_1)$ NP-schwer ist $\implies GAP_{s',c'}(U_2)$ NP-schwer

$\implies$ untere Schranke $\frac{c}{s}$ für Approximation von $U_1 \implies$ untere Schranke $\frac{c'}{s'}$ für Approx. von U_2

Motivation: GP-Reduktion ist simpler als AP-Reduktion (nur ein Algorithmus).

Beispiel (MAX-E3SAT, MAX-2SAT) Eingabe: KNF-Formel. Ziel: Belegung finden die alle/möglichst viele Klauseln erfüllt.

- **E3SAT**: exakt 3 verschiedene Literale pro Klausel
- **2SAT**: ≤ 2 Literale pro Klausel, in P
- **MAX-2SAT**: NP-schwer

Lemma Für alle $a, b,\ a \neq 0, a \leq b$ existiert eine GP-Reduktion von MAX-E3SAT auf MAX-2SAT mit Parametern (a, b) und $\left(\frac{3}{5} + \frac{a}{10}, \frac{3}{5} + \frac{b}{10}\right)$. [6]

Beweis: Sei $C = C_1 \wedge ... \wedge C_m$ eine MAX-E3SAT-Instanz mit Klauseln $C_i = l_{i1} \vee l_{i2} \vee l_{i3}$ und Variablen $x_1, ..., x_n$.

Konstruiere MAX-2SAT-Instanz: $\Phi_C = \bigwedge_{i=1}^m \Phi(C_i)$ mit zusätzlichen Variablen $y_1, ..., y_m$ und mit

$$\begin{aligned}\Phi(C_i) = &(l_{i1}) \wedge (l_{i2}) \wedge (l_{i3})\\ &\wedge (\overline{l_{i1}} \vee \overline{l_{i2}}) \wedge (\overline{l_{i1}} \vee \overline{l_{i3}}) \wedge (\overline{l_{i2}} \vee \overline{l_{i3}})\\ &\wedge (l_{i1} \vee \overline{y_i}) \wedge (l_{i2} \vee \overline{y_i}) \wedge (l_{i3} \vee \overline{y_i})\\ &\wedge (y_i)\end{aligned}$$

[6] D.h. die Lücke/Approximation wird etwas schlechter.

Restlicher Beweis siehe Buch S.327. Idee: für jede Belegung α für C kann man y_i so wählen dass ≤ 7 der 10 Klauseln erfüllt werden.

Lemma (Max-CLIQUE) Max-CLIQUE kann GP-reduziert werden auf Max-CLIQUE mit Parametern $(\alpha, 1-\varepsilon)$, $(\alpha^2, (1-\varepsilon)^2)$ für alle $\alpha \in (0, 1-\varepsilon)$, $\varepsilon \in (0, \frac{1}{2})$.

Beweis: Sei $G = (V, E)$, $|G| = |V|$. Konstruiere $G \times G = (V_{G\times G}, E_{G\times G})$ mit

- $V_{G\times G} = V \times V$
- $E_{G\times G} = \{\{(v,u),(r,s)\} \mid v,u,r,s \in V \text{ und } ((v = r, \{u,s\} \in E) \text{ oder } (\{v,r\} \in E))\}$

$Opt_{Max-CLIQUE}(G) =$ Clique der Grösse k in $G \implies$ Clique der Grösse k^2 in $G \times G$.

Fallunterscheidung:

- $Opt_{Max-CLIQUE}(G) < a \cdot |G| \implies Opt_{Max-CLIQUE}(G \times G) < a^2 \cdot |G|^2 = a^2 \cdot |G \times G|$
- $Opt_{Max-CLIQUE}(G) \geq (1-\varepsilon) \cdot |G| \implies Opt_{Max-CLIQUE}(G \times G) \geq (1-\varepsilon)^2 \cdot |G|^2 = (1-\varepsilon)^2 \cdot |G \times G|$

Intution: Durch das Quadrieren werden beide Werte kleiner und gehen gegen 0. $\implies$ Die Lücke wird kleiner. $\implies$ Die Approximation wird besser.

Implikation:

Falls es eine konstante c-Approximation gibt, dann lässt sie sich iterativ verbessern (von c auf $\sqrt{c}$, für $c > 1$), wodurch ein PTAS entsteht.
$\implies$ *entweder* Max-CLIQUE $\in$ PTAS *oder* Max-CLIQUE $\notin$ APX (= nicht konstant approximierbar).

Max-SAT $\leq_{AP}$ Max-CLIQUE (siehe subsection 4.2) und Max-SAT ist APX-vollständig [7]
$\implies$ Max-CLIQUE $\notin$ APX

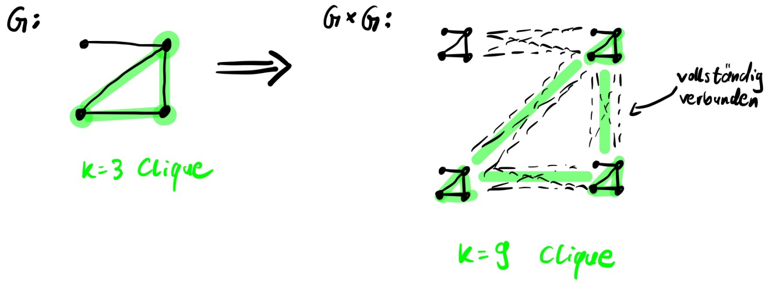


Figure 5: GP-Reduktion Max-CLIQUE

[7] Nicht in Vorlesung bewiesen.

4.4. PCP-Theorem

Motivation Klasse NP auf PCP-Theorem aufbauen bzw. charakterisieren (anstatt mit Satz von Cook und SAT). Reduktion von jedem Entscheidungsproblem in NP auf $GAP_{a,1}$(Max-SAT) für $a > 1$.

Probabilistische Verifizierer Ein probabilistischer Verifizierer V ist eine Polynomzeit-Turing-Maschine mit vier Bändern:

- Eingabeband (endlich lang, lesen, Eingabe x)
- Arbeitsband (unendlich lang, lesen/schreiben), "interner Arbeitsspeicher"
- Zufallsband (unendlich lang, lesen, Zufallsbits $\tau \in \{0,1\}^*$)
- Beweisband (endlich (aber sehr lang), Beweiskandidat $\pi \in \{0,1\}^*$ für $x \in L$)

Arbeitsweise:

1. Lese x und τ und berechne Indizes $i_1, ..., i_c$.
2. Lese $\pi_{i_1}, ..., \pi_{i_c}$.
3. Entscheide ob x akzeptiert oder verworfen wird, abhängig von $\pi_{i_1}, ..., \pi_{i_c}, x, \tau$.

V schaut sich nur einen Teil des Beweiskandidats an! Für ein fixes τ arbeitet V deterministisch. Verallgemeinerung von bekannten deterministischen Verifizierern, die kein Zufallsband haben.

Laufzeit: poly($|x|$) $\implies c, |\tau| \in$ poly($|x|$) (aber π unbeschränkt)

$V(x, \tau, \pi) \in \{$accept, reject$\}$

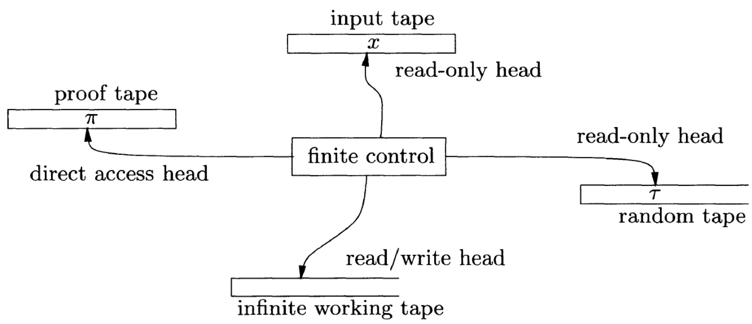


Figure 6: Probabilistischer Verifizierer als 4-Band-TM

Beschränkte Prob. Verifizierer Seien $r, q \in \mathbb{N} \mapsto \mathbb{N}$. Ein $(r(n), q(n))$-*beschränkter probabilistischer Verifizierer* ist ein Verifizierer, der für jede Eingabe x nur $\leq r(n)$ Zufallsbits und $\leq q(n)$ Beweisbits liest, für $|x| = n$.

V *akzeptiert* Sprache L falls gilt:

(i) Completeness: $\forall x \in L \ : \ \exists \pi \ \forall \tau \ : \ V(x, \tau, \pi) =$ accept

(ii) Soundness: $\forall x \notin L \ : \ \forall \pi \ : \ \Pr[V(x, \tau, \pi) = \text{accept}] \leq \frac{1}{2}$

wobei $\Pr[V(x, \tau, \pi) = \text{accept}] = \sum_{\tau, V(x,\tau,\pi)=\text{accept}} \Pr[\tau]$.
Annahme: Zufallsbits uniform verteilt, d.h. $\Pr[\tau] = \frac{1}{2^{|\tau|}}$.

Beispiel (SAT) $(0, n)$-beschränkter Verifizierer für SAT. Liest $m \leq n = |\Phi|$ Beweisbits und interpretiert sie als Belegung für $x_1, ..., x_m$. Akzeptiert x wenn die Belegung Φ erfüllt. Deterministisch.

$\implies$ für alle Probleme in NP existiert ein deterministischer Polynomzeit-Verifizierer (da SAT NP-vollständig ist).

Beispiel (GAP$_{1-\varepsilon,1}$(E3SAT)) für $\varepsilon \in (0,1)$. Eingabe Φ ist entweder erfüllbar, oder ein Teil der m Klauseln sind erfüllbar (d.h. $Opt_{Max-E3SAT}(\Phi) < (1-\varepsilon) \cdot m$).

Ziel: $\left(\log_{1-\varepsilon} \frac{1}{2} \cdot \log n, 3 \cdot \log_{1-\varepsilon} \frac{1}{2}\right)$-beschränkter prob. Verifizierer. Logarithmisch viele Zufallsbits, konstant viele Beweisbits.

Sei $\Phi = F_1 \wedge ... \wedge F_m$ eine E3-KNF-Formel über $x_1, ..., x_d$, $d \leq 3m$. Sei $n = |\Phi| \geq 3m$, sei $k = \lceil \log_{1-\varepsilon} \frac{1}{2} \rceil$. V wählt zufällig k Klausel-Indizes (= liest $k \cdot \lceil \log_2 m \rceil$ Zufallsbits). V interpretiert $\pi_1, ..., \pi_d$ als Belegung α für $x_1, ..., x_d$ (wobei es nur die $\leq 3k$ Belegungen der Variablen aus $F_{i1} \wedge ... \wedge F_{ik}$ lesen muss). V prüft ob α diese k zufälligen Klauseln erfüllt. Wenn ja akzeptiere, sonst verwerfe.

ad (i): Φ erfüllbar $\implies \exists$ erfüllende Belegung/Beweis π für die V immer akzeptiert.

ad (ii): Φ nicht erfüllbar $\implies \forall$ Belegung erfüllt $< (1-\varepsilon) \cdot m$ Klauseln, d.h. $\geq \varepsilon \cdot m$ sind nicht erfüllt.

$$
\begin{aligned}
& \Pr[\text{eine von } \alpha \text{ erfüllte Klausel zu wählen }] \geq \frac{\varepsilon \cdot m}{m} = \varepsilon \\
\implies & \Pr[\alpha \text{ erfüllt alle } F_{i1}, ..., F_{ik}] \leq (1-\varepsilon)^k \\
\implies & \Pr[V \text{ reject}] = \Pr[\text{mind. eine Klausel von } F_{i1}, ..., F_{ik} \text{ nicht erfüllt}] \\
& \geq 1 - (1-\varepsilon)^k = 1 - (1-\varepsilon)^{\lceil \log_{1-\varepsilon} \frac{1}{2} \rceil} \geq \frac{1}{2} \\
\implies & \Pr[V \text{ accept}] \leq \frac{1}{2}
\end{aligned}
$$

$\implies$ Mit Randomisierung können wir die Anzahl notwendiger/gelesener Beweisbits reduzieren, selbst für schwere Probleme!

Klasse PCP *Probabilistically Checkable Proofs.* Für alle $r, q : \mathbb{N} \mapsto \mathbb{N}$ ist

$$PCP(r, q) = \{L \mid \text{ L is accepted by an } (r(n), q(n))\text{-restricted prob. verifier}\}$$

Falls $\mathcal{W}, \mathcal{V}$ Klassen nicht-fallender Funktionen sind, so gilt:

$$PCP(\mathcal{W}, \mathcal{V}) = \bigcup_{r \in \mathcal{W}, q \in \mathcal{V}} PCP(r, q)$$

<u>Beobachtung:</u>

- GAP$_{1-\varepsilon,1}$(E3SAT) $\in PCP(\mathcal{O}(\log_2 n), \mathcal{O}(1))$
- P = PCP(0,0) $\longrightarrow$ deterministisch, Beweis wird selber berechnet
- NP = PCP(0, poly(n)) $\longrightarrow$ vgl. Verifizierer-basierte Definition von NP

PCP-Theorem

$$NP = PCP(\mathcal{O}(\log_2 n), \mathcal{O}(1))\ ^{8}$$

$\implies$ Jedes Problem in NP hat einen Beweis der sich mit konstant vielen Beweisbits randomisiert verifizieren lässt.

Theorem $\text{GAP}_{\frac{15}{16},1}$(MAX-3SAT) ist NP-schwer.

Beweis: Zeige dass jedes $L \subseteq \Sigma^*$ in NP auf $\text{GAP}_{\frac{15}{16},1}$(MAX-3SAT) reduzierbar ist. Aus dem PCP-Theorem folgt, dass es einen $(c \cdot \log_2 n, 3)$-beschränkter prob. Verifizierer V geben muss, das L akzeptiert.

Idee: Konstruiere für jedes $x \in \Sigma^*$ eine 3KNF-Formel Φ_x so dass gilt:

(i) $x \in L \implies \Phi_x$ erfüllbar

(ii) $x \notin L \implies \forall$ Belegungen von Φ_x bleiben $\geq \frac{1}{16}$ der Klauseln unerfüllt

(iii) Φ_x in Polynomzeit berechenbar

Dies reduziert das Akzeptieren der Sprache L auf $\text{GAP}_{\frac{15}{16},1}$(MAX-3SAT).

Konstruktion 3SAT-Instanz Φ_x (gegeben Eingabe x und Verifizierer V):
Sei $\pi = \pi_1\pi_2...$ das Beweisband von V für Eingabe x. Sei y_i die Variable für Beweisbit π_i. Konstruiere nun $\Phi_x = \bigwedge_\tau \Phi_{x,\tau}$ wobei τ über alle n^c möglichen Zufallsstrings. [9]

Für fixe x, τ gilt: V liest (laut PCP-Theorem) drei Beweisbits $\pi_{i_1}, \pi_{i_2}, \pi_{i_3}$ und berechnet die Boolsche Funktion $f_{x,\tau}$:

$$f_{x,\tau} \;:\; \{0,1\}^3 \mapsto \underbrace{\{0,1\}}_{\text{reject, accept}} \qquad \text{über } Y_\tau = \{y_{i_1}, y_{i_2}, y_{i_3}\}$$

$$f_{x,\tau}(a_{i_1}, a_{i_2, i_3}) = 1 \iff V(x, \tau, \pi_1\pi_2...a_{i_1}...a_{i_2}...a_{i_3}...) = \text{ accept}$$

$f_{x,\tau}$ kann eindeutig durch eine 3KNF-Formel $\Phi_{x,\tau}$ mit $\leq 2^3 = 8$ Klauseln dargestellt werden (siehe truth table).

Bestimmung von $f_{x,\tau}$: Simuliere $V(x, \tau, \tilde{\pi})$ für alle 8 Belegungen von $\{y_{i_1}, y_{i_2}, y_{i_3}\}$ (in Polynomzeit, da V in Polynomzeit).

Bedingungen:
ad (iii): n^c Teilformeln $\Phi_{x,\tau} \implies \Phi_x$ in Polynomzeit konstruierbar

ad (i): Falls $x \in L$: $\exists \pi$ s.t. $V(x, \tau, \pi) =$ accept (per PCP-Definition) $\implies \Phi_x$ erfüllbar

ad (ii): Falls $x \notin L$: V akzeptiert $\forall \pi$ nur $\leq \frac{1}{2}$ der τ (laut PCP).
$\implies \forall$ Belegung von Φ_x ist $\geq \frac{1}{2}$ der $\Phi_{x,\tau}$ unerfüllt.
$\implies \geq \frac{n^c}{2}$ viele $\Phi_{x,\tau}$ unerfüllt (von $\leq 8n^c$ Klauseln insgesamt in Φ_x).
$\implies \geq \frac{1}{16}$ der Klauseln unerfüllt.

$\implies \text{GAP}_{\frac{15}{16},1}$(MAX-3SAT) ist NP-schwer
$\implies \nexists \frac{16}{15}$-Approximationsalgorithmus für MAX-E3SAT (falls P $\neq$ NP).

[8] Je nach Formulierung sind es konstant 3 Beweisbits.
[9] Länge $c \cdot \log n \implies 2^{c \cdot \log n} = (2^{\log n})^c = n^c$ viele möglichen Strings.

Part II.
Online-Algorithmen

5. 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, an Tag k kaufen	Ab Tag $k+1$ schlechtes Wetter	$\frac{2k-1}{k} = 2 - \frac{1}{k}$

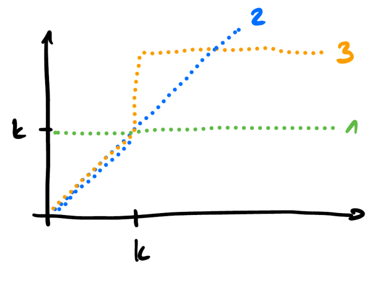


Figure 7: Skirental Szenarios

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). [10]

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 $Opt(I_i) \overset{i \to \infty}{\longrightarrow} \infty$.

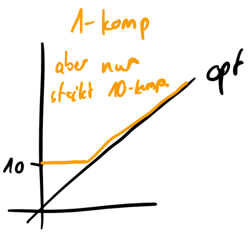


Figure 8: *Opt* in schwarz. $\mathcal{A}$ in orange, 1-kompetitiv und strikt-10-kompetitiv.

5.1. Das Paging-Problem

Paging

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

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

- Cache-Speicher mit k Seiten: $B = (s_{j_1}, ..., s_{j_k})$, initialisiert mit $(s_1, ..., s_k)$ [11]
- 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}. [12]
- $cost(\mathcal{A}(I)) := |\{i \mid y_i > 0\}|$
- 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).
$\implies$ Zu Beginn von Phase P ist s im Cache von *Opt* und von $Fifo$.
$\implies$ 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.
$\implies$ Während P muss *Opt* mindestens einen Seitenfehler machen.
$\implies$ $Fifo$ 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. [13] 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 $\implies$ k-kompetitiv. [14]

[11] Der Vorsprung eines selbstgewählten Startinhalts kann in α versteckt werden.
[12] Zusätzliches, proaktives Entfernen bringt keinen Vorteil.
[13] $k+1$ macht die Aussage nur stärker. Warum?
[14] 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.

5.2. 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 $2H_k$. [15] 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 $((P_1, P_2), (P_3, P_4), \ldots$ vs $P_1, (P_2, P_3), (P_4, P_5), \ldots)$ 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.).

5.3. 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 $\mathcal{I} = \{I_1, \ldots, I_m\}$ und Algorithmen $strat(Rand) = \{A_1, \ldots, A_l\}$ (d.h. Eingabelänge n begrenzt).

[15] 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.

<u>Beweis:</u> 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:

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

6. k-Server-Problem

Konzepte

- k-Server-Problem
- Potentialfunktionen, amortisierte Kosten (Teleskopsumme)
- Greedy, Double Coverage (Alternating Moves)

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

"Heiliger Gral" der Online-Algorithmen, wie TSP für Approximations-Algorithmen oder SAT für NP.

Metrischer Raum Sei S eine Menge von Punkten, sei $\text{dist} : S \times S \mapsto \mathbb{R}$ 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* [16] 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 (lazy)* 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?

[16] Achtung Verwechslungsgefahr!

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.

D.h. die untere Schranke ist erreichbar. Wenn dies wahr ist, dann ist k-Server genauso schwer wie Paging! Aktueller Stand: $2k-1$.

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

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

Beweis: Siehe Instanz in Figure 9. Hier gilt $\frac{cost(Greedy(I))}{cost(I)} = \frac{n}{2}$, d.h. es gibt keine Konstante c so dass *Greedy* c-kompetitiv wäre.

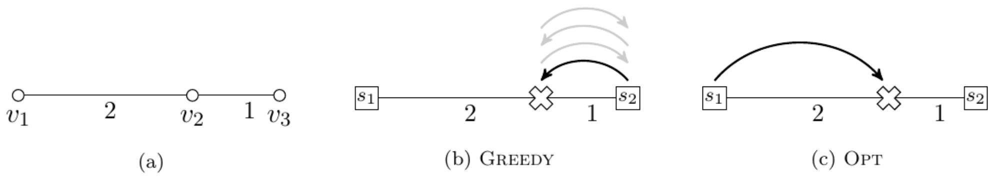


Figure 9: k-Server: *Greedy* versus *Opt*

6.1. 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*. [17]

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:

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

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

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 (*Teleskopsumme*). Dann $\alpha := \beta$ setzen.

6.2. 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!

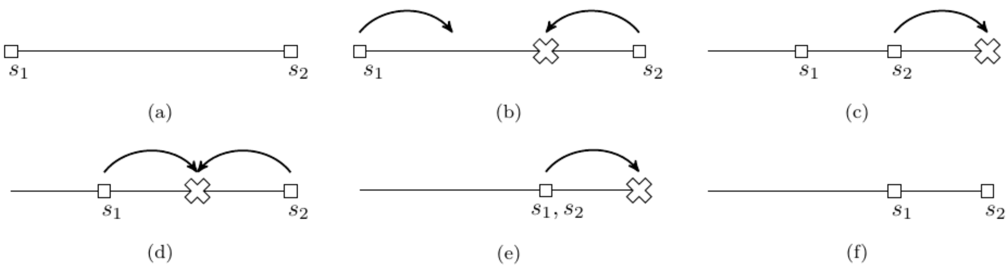


Figure 10: k-Server: *DoubleCoverage* anhand von *Greedys* worst-case Beispiel

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 $K_{DC} = \{p_1^{DC}, \dots, p_k^{DC}\}$ 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): [18]

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

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 (*Alternating Moves*).

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 $k \cdot cost(DC(x_i))$.
$\Longrightarrow$ 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.
$\Longrightarrow$ insgesamt gilt $(\star)$

$\Longrightarrow$ Bedingung 1 + 2 vom Satz erfüllt $\Longrightarrow$ *DC* ist k-kompetitiv.

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

7. 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.

<u>Beweis:</u> 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.
$\Longrightarrow$ 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. $\Longrightarrow$ $n + k$

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.

<u>Beweis:</u> Konstruiere einen vollständigen Graph wie in Figure 11. 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. [19]

$\implies$ 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.

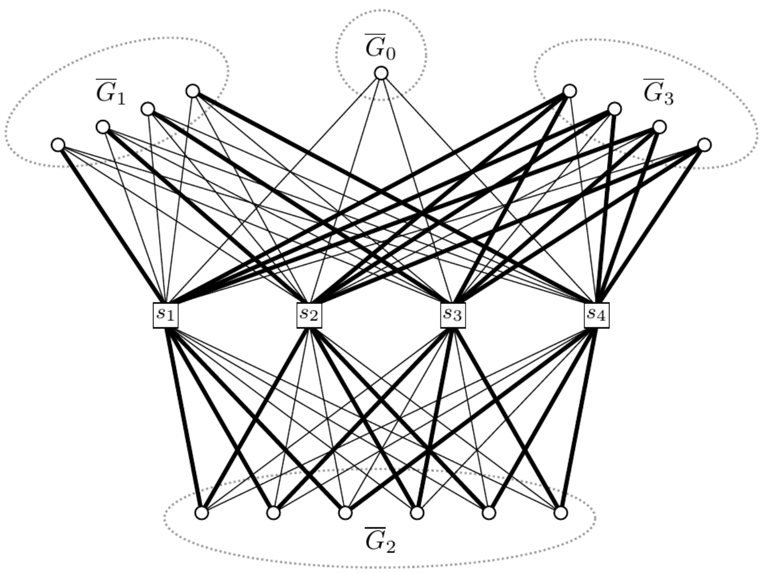


Figure 11: Konstruktion Graph für Beweis Advice-Komplexität von k-Server

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.

[19] 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. [20] 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. [21] 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.

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

8. Online-Rucksackproblem

Konzepte

- Online-Rucksackproblem
- Ansätze: deterministisch, mit Advice, randomisiert
- Untere Schranken für den (erwarteten) kompetitiven Faktor und die Advice-Komplexität

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.

8.1. Deterministisch

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

<u>Beweis:</u> 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 $> 1-\beta$.

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

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

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

8.2. Mit Advice

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

<u>Beweis</u>: 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.

<u>Beweis:</u> 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.

<u>Beweis:</u>
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:

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

Fazit Mit $n-1$ Advice-Bits sind wir optimal, mit 1 Advice-Bit bereits 2-kompetitiv. Konstant mehr Bits helfen nicht.

Untere Schranke: kein OA mit $< \log_2(n-1)$ Advice-Bits kann besser als $(2-\varepsilon)$-kompetitiv sein. [22]
Obere Schranke: mit $\mathcal{O}(\log n)$ Advice-Bits kommen wir beliebig nah an die optimale Lösung heran.

8.3. Randomisiert

Motivation Ein Advice-Bit ist mächtig. Wie mächtig ist ein Zufallsbit?

Algorithmus: RKPone' Wie KPone, aber statt Advice-Bit nimmt er ein Zufallsbit.

Satz Der Barely-Random-Algorithmus $RKPone'$ ist strikt 4-kompetitiv im Erwartungswert. Diese Schranke ist dicht (d.h. er kann nicht besser sein).

[22] Beweis ähnlich wie oben, via Konstruktion einer Klasse von Instanzen und Schubfachprinzip.

Algorithmus: RKPone Wähle u.a.r. einen deterministischen OA aus $strat(RKPone) = \{Greedy_1, Greedy_2\}$. $Greedy_1$ ist eine normale Greedy-Strategie.
$Greedy_2$ simuliert $Greedy_1$, akzeptiert aber nichts. Sobald ein Objekt nicht mehr in $Greedy_1$s Rucksack passt, akzeptiert $Greedy_2$ dieses und folgt von dann an einer Greedy-Strategie.

Satz Der Barely-Random-Algorithmus $RKPone$ ist strikt 2-kompetitiv im Erwartungswert.

Beweis:

Fall 1: $\sum w_i \leq 1 \implies gain(Greedy_1(I)) = gain(Opt(I)) \leq 1$ und $gain(Greedy_2(I)) = 0$
$\implies$ erwartete Gewinn $= \frac{1}{2} \cdot gain(Opt(I))$

Fall 2: $\sum w_i \geq 1$. Dann ist der erwartete Gewinn:

$$\frac{1}{2} gain(Greedy_1(I)) + \frac{1}{2} gain(Greedy_2(I)) = \frac{1}{2}\left(gain(Greedy_1(I)) + gain(Greedy_2(I))\right) \geq \frac{1}{2}$$

wobei wir verwenden dass $gain(Greedy_1(I)) + gain(Greedy_2(I)) \geq 1$ (da $Greedy_2$ mindestens das Objekt akzeptiert mit dem $Greedy_1$ die Rucksackkapazität überschritten hätte).

Satz (Untere Schranke) Kein randomisierter OA für das Rucksackproblem ist besser als strikt 2-kompetitiv im Erwartungswert.

Intuitiv: ein Advice-Bit ist genauso mächtig wie beliebig viel Randomisierung.

Beweis: Betrachte die Klasse von Instanzen $\mathcal{I} = \{I_1 = (\varepsilon), I_2 = (\varepsilon, 1)\}$. Sei $Rand$ ein randomisierter OA der x_1 mit Wahrscheinlichkeit p akzeptiert. [23]

Für den kompetitive Faktor gilt:

$$KF(I_1) = \frac{\varepsilon}{p \cdot \varepsilon + (1-p) \cdot 0} = \frac{1}{p} \qquad ; \qquad KF(I_2) = \frac{1}{p \cdot \varepsilon + (1-p) \cdot 1}$$

Durch Gleichsetzen folgt $p = \frac{1}{2-\varepsilon}$, und durch Rückeinsetzen in KF folgt dass der kompetitive Faktor $\geq 2 - \varepsilon$ ist.

[23] $p > 0$ muss gelten da $Rand$ sonst auf I_1 einen erwarteten Gewinn von 0 hat.