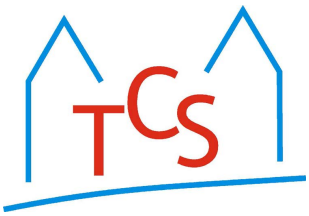
Algorithmendesign

Unterlagen zum Lehrmodul CS3000

Prof. Dr. R. Reischuk

Institut für Theoretische Informatik

Universität zu Lübeck



WS 2013/14

Hinweis: Dies ist keine vollständige Ausarbeitung der Themen dieser Veranstaltung.
Einige Themen werden in Lehrbüchern bereits ausführlich dargestellt (siehe Literaturhinweise).
In der Vorlesung und den Übungen werden weitere Thematiken und Details behandelt.

7 Online Algorithmen

Gegeben sei ein Optimierungsproblem Π, bei dem für eine Folge von Eingaben $X = x_1, \ldots, x_n$ eine Folge von Aktionen $Y = y_1, \ldots, y_n$ zu bestimmen ist. Falls die Eingabesequenz X von vorne herein vollständig bekannt ist, besteht die Aufgabe lediglich darin, eine optimale Lösung Y für X effizient zu finden. Oftmals stehen die Datenelemente in X jedoch nur sukzessive zur Verfügung. Ein **Online-Algorithmus** muß die Aktion y_i festlegen, nachdem er den Präfix $x_1, \ldots, x_i$ erhalten hat, aber bevor er x_{i+1} erfährt, und dies für $i = 1, \ldots, n$.

Scheduling-Probleme sind typische Beispiele für diese Online-Problematik. Bei gegebenem Maschinenpark $M_1, \ldots, M_m$ wird die Folge $X = x_1, x_2, \ldots$ der Jobs mit ihren Ausführungszeiten und Deadlines erst sukzessive bekannt. Bei Eintreffen von x_i sollte jedoch unverzüglich entschieden werden, auf welchem freien Prozessor dieser Job abgearbeitet wird. Bei vielen Varianten des Scheduling-Problems ist allerdings die Berechnung eines optimalen Produktionsplans auch bei vollständiger Information für einen offline-Algorithmus schwer. Man kann sich jedoch leicht an Beispielen verdeutlichen, daß durch die Unkenntnis über die noch eintreffenden Jobs eine gute Planung noch erheblich schwieriger wird.

7.1 Das Skifahrer-Problem: Leihen oder Kaufen

Zum Skifahren benötigt man eine spezielle und teure Ausrüstung (Ski, Schuhe, Stöcke, etc.). Diese kann man
– entweder kaufen, wofür einmalig K Euro aufzuwenden sind
(Verschleiß sei im folgenden außer acht gelassen);
– oder pro Saison für jeweils L Euro leihen.

Der Skifahrer muß sich zwischen diesen beiden Alternativen entscheiden, wobei es sein Ziel ist, die Gesamtkosten zu minimieren.Wäre vorab bekannt, für wieviele Winter die Ausrüstung benötigt wird, so ist die Entscheidung einfach: Kaufen, wenn die Summe der Leihkosten den Kaufpreis übersteigt, ansonsten leihen.Eine online-Problematik entsteht, wenn der Skifahrer nicht vorab weiß, wieviele Jahre lang er das Skifahren ausüben wird. Genauer nehmen wir an, daß solange er Ski fährt, nicht weiß, ob er dies auch die nächste Saison noch tun möchte. In diesem Fall kann es sogar Sinn machen, erst einmal zu leihen und dann nach einiger Zeit – bei dauerhafterem Gefallen an diesem Sport – zu kaufen. Der Skiläufer muß sich also zu Beginn einer jeden Saison zwischen Kaufen und Leihen entscheiden.

Die Eingabesequenz X ist beim Skifahrer-Problem eine binäre Folge, die mit einem Block von Einsen beginnt, – die Winter, wo man skifahren möchte – und dann irgendwann auf 0 wechselt. Auf Eingabe $x_i = 1$ muß man sich nun entscheiden, ob man die Aktion $y_i = \texttt{KAUF}$ durchführt, wenn dies in der Vergangenheit noch nicht geschehen ist, oder die Aktion $y_i = \texttt{LEIH}$.

Die möglichen **Strategien** S_r können wir durch den Parameter $r = 0, 1, \ldots$ beschreiben:
die ersten r Winter leihen;
falls man in Saison $r+1$ weiter skifahren möchte, kaufen.

Bezeichnet $t(X)$ die Anzahl der Einsen in X, so berechnen sich die Kosten von S_r als

$$\texttt{cost}(S_r) \;=\; \begin{cases} t(X) \cdot L & \text{falls } t(X) \le r, \\ r \cdot L + K & \text{andernfalls.} \end{cases}$$

Es stellt sich nun die Frage, um wieviel diese Kosten maximal höher ausfallen können im Vergleich zu einer optimalen Entscheidung, bzw. welche Strategie S_r man wählen sollte, um diese Mehrkosten möglichst gering zu halten, in dem Fall, daß $t(X)$ vorab nicht bekannt ist.

Die minimalen Kosten ergeben sich als

$$\texttt{cost}(\texttt{OPT}) \;=\; \begin{cases} t(X) \cdot L & \text{falls } t(X) \le \frac{K}{L}, \\ K & \text{andernfalls.} \end{cases}$$

In der folgenden Analyse betrachten wir zunächst den Fall, daß der Quotient K/L ganzzahlig ist. Als Verhältnis der Kosten von S_r zu den minimalen Kosten ergibt sich:

$$\frac{\texttt{cost}(S_r)}{\texttt{cost}(\texttt{OPT})} \;=\; \begin{cases} \dfrac{t(X) \cdot L}{t(X) \cdot L} \;=\; 1 & \text{falls } t(X) \le \min\{\frac{K}{L},\ r\}, \\ \dfrac{t(X) \cdot L}{K} & \text{falls } \frac{K}{L} < t(X) \le r, \\ \dfrac{r \cdot L + K}{t(X) \cdot L} & \text{falls } r < t(X) \le \frac{K}{L}, \\ \dfrac{r \cdot L + K}{K} & \text{falls } t(X) > \min\{\frac{K}{L},\ r\}. \end{cases}$$

Im zweiten Fall wird der Quotient maximal, wenn man $t(X)$ maximal wählt, d.h. $t(X) = r$, im dritten Fall für $t(X)$ minimal, d.h. $t(X) = r+1$. Er verbleibt daher die Aufgabe, den Parameter r so zu wählen, daß der Ausdruck

$$\max\{\frac{r \cdot L}{K},\ \frac{r \cdot L + K}{(r+1) \cdot L},\ \frac{r \cdot L + K}{K}\}$$

minimal wird unter den gegebenen Randbedingungen. Da der 1. Term durch den 3. dominiert wird, genügt es,

$$\frac{r \cdot L + K}{(r+1) \cdot L} \quad \text{und} \quad \frac{r \cdot L + K}{K}$$

zu betrachten. Der erste Term fällt monoton in r (da er äquivalent zu $1 + \frac{1}{r+1} \cdot \frac{K-L}{L}$ ist), der zweite wächst monoton in r. Das Minimum wird erreicht, wenn beide Terme

den gleichen Wert liefern. Da die Zähler identisch sind, müssen auch die beiden Nenner $(r+1) \cdot L$ und K gleich sein, d.h.

$$r \;=\; \frac{K}{L} - 1 \;.$$

Bei dieser Wahl ist der zweite Fall unkritisch. Als Quotient ergibt sich

$$\frac{(\frac{K}{L} - 1) \cdot L + K}{K} \;=\; 2 - \frac{L}{K} \;.$$

Ist K/L nicht ganzzahlig, so ist die Wahl $r = \lfloor K/L \rfloor - 1$ oder $r = \lfloor K/L \rfloor$ optimal.

7.2 Die kompetitive Rate

Beim Caching-Problem kann man eine optimale Cacheverwaltung durch eine einfache Greedy-Strategie erzielen, nämlich die Regel `EVICT-FURTHERST-IN-FUTURE`(`EFIF`).Dies ist ein Offline-Algorithmus, der er bei seiner Entscheidung Informationen über die Zugriffe in der Zukunft benötigt. Da man diese Information in der Praxis oftmals nicht besitzt, bietet sich die duale Greedy-Strategie `EVICT-LEAST-RECENTLY-USED` (`ELRU`) an, wo es genügt, die Zugriffe in der Vergangenheit zu kennen.Wir wollen nun der Frage nachgehen, ob dies eine gute Online-Strategie ist.

Definition 7.1
Es sei $\texttt{OPT}_k(X)$ die minimale Anzahl von Cache-Misses für eine Sequenz X bei Cache-Größe k. Für einen Algorithmus $\mathcal{A}$ bezeichne $\mathcal{A}_k(X)$ die Anzahl der Cache-Misses von $\mathcal{A}$ auf X.

Allgemein bezeichne für ein gegebenes Optimierungsproblem Q und eine Eingabe X sowie einen Algorithmus $\mathcal{A}$ für Q die Ausdrücke $\texttt{OPT}(X)$ bzw. $\mathcal{A}(X)$ den Wert einer optimalen Lösung bzw. den Wert, den $\mathcal{A}$ errechnet.

Für eine reelle Zahl $\rho \geq 1$ heißt ein Algorithmus für ein Minimierungsproblem $\mathcal{A}$ **stark ρ-kompetitiv**, falls für jede Eingabe X gilt: $\mathcal{A}(X) \leq \rho \cdot \texttt{OPT}(X)$.
$\mathcal{A}$ heißt **ρ-kompetitiv**, falls sich das Verhältnis in der Form $\mathcal{A}(X) \leq \rho \cdot \texttt{OPT}(X) + b$ abschätzen läßt, wobei $b \in \mathbb{N}$ eine feste Konstante ist.Bei Maximierungsproblemen verlangen wir $\mathcal{A}(X) \geq \rho^{-1} \cdot \texttt{OPT}(X)$ bzw. $\mathcal{A}(X) \geq \rho^{-1} \cdot \texttt{OPT}(X) - b$.

Die **kompetitive Rate** $\texttt{CR}(\mathcal{A})$ von $\mathcal{A}$ ist das Infimum über alle derartigen Werte ρ.

Offensichtlich gilt beim Caching Problem für alle k und X:
$\texttt{OPT}_{k+1}(X) \leq \texttt{OPT}_k(X) \leq \mathcal{A}_k(X)$.

Betrachten wir den Algorithmus `ELRU`, so kann man bei Cache-Größe k zeigen:
$\texttt{ELRU}_k(X) \;\leq\; k \cdot \texttt{OPT}_k(X)$
für alle Requestsequenzen X. Andererseits ergibt sich für die Sequenz

$$X \;=\; a_1 a_2 \ldots a_k a_{k+1} a_1 a_2 \ldots a_k a_{k+1} \ldots$$

daß ELRU bei jedem Request einen Cache-Miss besitzt, während dies dem optimalen offline-Algorithmus EFIF nur alle k Schritte passiert. Somit ergibt sich

$$\mathtt{CR}(\mathtt{ELRU}_k) \;=\; k\;.$$

Man kann zeigen, daß kein online-Algorithmus $\mathcal{A}$ eine bessere Rate erreichen kann: Eine **Adversary**, ein Gegenspielder für $\mathcal{A}$, generiert eine Request-Sequenz über dem Universum $U = \{a_1, a_2, \ldots, a_{k+1}\}$, indem er als nächstes immer gerade das Element auswählt, welches $\mathcal{A}$ gerade nicht in seinem Cache hat. Damit macht $\mathcal{A}$ bei jedem Request einen Cache-Fehler, während EFIF höchstens alle k Schritte ein neues Element einlagern muß.

Man kann andererseits zeigen, daß jede Online-Caching-Strategie, die keine unnötigen Evicts ausführt, auch keine schlechtere Rate als k erzielt.

7.3 Das Listenzugriffsproblem LZP

Gegeben sei eine lineare Liste $L = a_1, a_2, \ldots, a_\ell$ der Länge ℓ, in der eine Teilmenge A eines Universums U in beliebiger Reihenfolge abgespeichert ist, sowie eine Folge $X = x_1, x_2, \ldots$ von Requests auf Elemente aus U. Befindet sich ein Element x an der k-ten Stelle in L, d.h. $x = a_k$ so verursacht dieser Request Kosten k. Wir beschränken uns im folgenden auf die statische Version dieses Problems, bei der sich die Menge A der Elemente in der Liste nicht ändert und das gesuchte Element auch immer in der Liste vorkommt. Im dynamischen Fall sind zusätzlich Insert- und Delete-Operationen möglich, womit neue Elemente in L eingefügt oder vorhandene entfernt werden können.

Die Liste kann durch eine Folge von Vertauschungen jeweils zweier benachbarter Elemente, sogenannte Transpositionen, umgeordnet werden, um spätere Zugriffe auf Elemente aus A kostengünstiger zu gestalten. Dabei unterscheiden wir zwischen **freien Transpositionen**, die nach einem Request x_i dies Element mit Vorgängern vertauschen – diese verursachen keine weiteren Kosten – und **kostenpflichtigen Transpositionen** mit jeweils Kosten 1 für alle anderen Paare von Elementen.

Selbst wenn man die vollständige Request-Sequenz X von Anfang an kennt, ist nicht so einfach wie beim Caching–Problem klar, wie eine optimale Strategie aussieht, die die Gesamtkosten minimiert. Im online-Fall sind verschiedene Strategien denkbar, wie man die Kosten auch ohne Kenntnis der zukünftigen Zugriff gering halten könnte.

1. MTF *(move-to-front):* das nachgefragte Element $x_i = a_k$ wird an den Anfang der Liste bewegt;

2. TRANS *(transpose):* vertausche a_k mit seinem direkten Vorgänger a_{k-1};

3. FC *(frequency count):* die Elemente werden gemäß einer Statistik über die bisherigen Zugriffe nach fallender Häufigkeit in L angeordnet.

Man beachte, daß diese Strategien nur freie Transpositionen verwenden. Welche dieser Strategien ist nun die beste? Die Vermutung liegt nahe, daß FC die Elemente am geschicktesten in der Liste plaziert, da die mit den häufigsten Zugriffen an den Anfang gebracht werden. Betrachten wir jedoch folgendes Szenario. Die anfängliche Liste sei $a_1, a_2, \ldots, a_\ell$. Die Requestsequenz X besteht aus m Requests für a_1, gefolgt von $m-1$ Requests für a_2, dann $m-2$ Requests für a_3 bis zu $m-\ell+1$ Request für a_ℓ. Da für jedes k zu jedem Zeitpunkt das Element a_k mindestens soviele Zugriffe erfahren hat wie a_{k+1}, wird FC die Liste niemals umordnen. Es ergeben sich die Kosten

$$\texttt{FC}(X) \;=\; \sum_{i=1}^{\ell} i \cdot (m+1-i) \;=\; \frac{\ell \cdot (\ell+1)}{2} \cdot m - \frac{(\ell^2-1)\ell}{3} \;.$$

Für dieses X wäre jedoch MTF wesentlich besser und – was nicht schwer einzusehen ist – sogar optimal mit Kosten

$$\texttt{MTF}(X) \;\leq\; \sum_{i=1}^{\ell} i + (m-i) \;=\; \ell\, m \;.$$

Dies impliziert

$$\texttt{CR}(\texttt{FC}) \;\geq\; \frac{\frac{\ell(\ell+1)}{2} \cdot m - \frac{(\ell^2-1)\cdot\ell}{3}}{\ell\, m} \;=\; \frac{\ell+1}{2} - o(1) \;.$$

Die kompetitive Rate von FC wächst damit mit der Länge der Liste und die untere Schranke $(\ell+1)/2$ ist bereits nicht wesentlich besser als die triviale obere Schranke $\texttt{CR}(\mathcal{A}) \leq \ell$, die bereits der Algorithmus erreicht, der die Liste niemals umsortiert.

Nun kann man aber auch leicht Beispiele von Sequenzen X finden, für sich MTF nicht optimal verhält. Erstaunlicherweise kann man jedoch zeigen, daß diese Strategie niemals mehr als doppelt so hohe Kosten produziert.

Für die folgende Analyse bezeichne für einen Algorithmus $\mathcal{A}$ für LZP
$\mathcal{A}_f(X)$ die Anzahl freier Transpositionen von $\mathcal{A}$ auf X,
$\mathcal{A}_p(X)$ die Anzahl kostenpflichtiger Transpositionen von $\mathcal{A}$ auf X,
$\mathcal{A}_C(X)$ die Summe aller Kosten von $\mathcal{A}$ ohne die kostenpflichtigen Transpositionen,
sowie wie üblich
$\mathcal{A}(X)$ die Summe aller Kosten.

Theorem 7.1 $\texttt{MTF}(X) \;\leq\; 2 \cdot \texttt{OPT}_C(X) + \texttt{OPT}_p(X) - \texttt{OPT}_f(X) - |X|$,
wobei angenommen wird, daß MTF und OPT mit der gleichen Anfangsliste starten.

Beweis: Es bezeichne k_i die Kosten von MTF im Schritt i bei Request von x_i, d.h. x_i befindet sich in der MTF-Liste vor dem i-ten Request an Position k_i. Für 2 Listen L und L' der Elemente aus A heißt ein Paar $\{a, b\}$ von Elementen eine **Inversion**, wenn a in der einen Liste vor b steht und in der anderen Liste dahinter. Φ_i sei die Anzahl der

Inversionen der Listen von MTF und OPT nach Schritt i. Da die beiden Listen zu Anfang gleich sind, gilt $\Phi_0 = 0$. Wir definieren als **amortisierte Kosten** von MTF im Schritt i den Wert

$$a_i \;:=\; k_i + \Phi_i - \Phi_{i-1} \;.$$

Dann gilt mit $n = |X|$

$$\mathtt{MTF}(X) \;=\; \mathtt{MTF}_C(X) \;=\; \sum_{i=1}^{n} k_i \;=\; \sum_{i=1}^{n} a_i + \Phi_0 - \Phi_n \;.$$

Lemma 7.1 Das im Schritt i zugegriffene Element x_i an Position k_i befinde sich in der Liste von OPT an Position q_i. Dann gilt

$$a_i \le (2q_i - 1) + p_i - f_i \;,$$

wobei p_i und f_i die Anzahl bezahlter bzw. freier Transpositonen bezeichnet, die OPT in Schritt i ausführt.

Beweis: Es sei μ_i die Anzahl der Inversionen der Form $\{a, x_i\}$, wobei a in der MTF-Liste vor x_i steht und in der OPT-Liste hinter x_i. Dann befinden sich die übrigen $k_i - 1 - \mu_i$ Elemente b in der MTF-Liste vor x_i auch in der OPT-Liste vor x_i. Dies impliziert $q_i - 1 \ge k_i - 1 - \mu_i$, d.h. $k_i - \mu_i \le q_i$.

MTF bewegt das Element x_i an den Anfang seiner Liste. Dadurch entstehen $k_i - 1 - \mu_i$ zusätzliche Inversionen vom Typ $\{x_i, b\}$ zwischen der neuen MTF-Liste und der alten OPT-Liste, dagegen werden die μ_i vielen vom Typ $\{a, x_i\}$ beseitigt. Führt OPT keine Transpositionen im Schritt i aus, d.h. $p_i = f_i = 0$,so ergibt sich

$$a_i \;=\; k_i + \Phi_i - \Phi_{i-1} \;=\; k_i + (k_i - 1 - \mu_i) - \mu_i \;=\; 2(k_i - \mu_i) - 1 \;\le\; 2q_i - 1 \;.$$

Jede kostenpflichtige Transposition von OPT ändert die Anzahl der Inversionen um 1, also kann die Differenz $\Phi_i - \Phi_{i-1}$ um höchstens 1 wachsen. Jede freie Transposition reduziert die Anzahl der Inversionen um 1. Dies impliziert $a_i \;\le\; 2q_i - 1 + p_i - f_i$. ∎

Somit können wir schließen

$$\begin{aligned} \mathtt{MTF}(X) \;&=\; \sum_{i=1}^{n} a_i + \Phi_0 - \Phi_n \;\le\; \sum_{i=1}^{n} (2q_i + p_i - f_i - 1) + \Phi_0 - \Phi_n \\ &\le\; 2\,\mathtt{OPT}_C(X) + \mathtt{OPT}_p(X) - \mathtt{OPT}_f(X) - |X| \;. \end{aligned}$$

Korollar 7.1 $\qquad \mathtt{CR}(\mathtt{MTF}) \;\le\; 2 - \dfrac{1}{\ell}$.

Beweis: Das Theorem impliziert $\texttt{MTF}(X) \leq 2\,\texttt{OPT}(X) - |X|$; außerdem gilt trivialerweise $\texttt{OPT}(X) \leq \ell \cdot |X|$, d.h.

$$\texttt{MTF}(X) \;\leq\; 2\,\texttt{OPT}(X) - \frac{\texttt{OPT}(X)}{\ell} \;.$$

Eine Modifikation $\texttt{MTF}^\star$ bewegt ein Element nur bei jedem 2. Zugriff zum Listenanfang. Man kann zeigen, daß auch diese Strategie stark 2-kompetitiv ist. Gilt dies auch für TRANS? Wählt der Adversary als Request jeweils das letzte Element in der Liste, die TRANS unterhält, so wechseln sich die beiden letzten Elemente der Anfangsliste ab. Für diese Sequenz erhalten wir als Kosten $\texttt{TRANS}(X) = |X| \cdot \ell$. Optimal dagegen wäre es, diese beiden Elemente nach vorne zu bringen und dann die Liste nicht mehr zu verändern. Weitere Requests kosten dann nur noch 1 bzw. 2. Das bedeutet für $|X| = n$

$$\texttt{CR(TRANS)} \;\geq\; \frac{n \cdot \ell}{2\ell + n \cdot 3/2} \;\geq\; \frac{2}{3}\ell - o(1) \;.$$

7.4 Potentialfunktionen

Betrachten wir die obige Beweismethodik etwas allgemeiner. Gegeben ist eine Sequenz X von Aufgaben für ein gegebenes System – im obigen Fall eine Liste von Datenelementen. Ein Algorithmus zur Lösung dieser Aufgaben kann das System modifizieren – bei LZP die Reihenfolge der Elemente vertauschen, dies gilt sowohl für Online- als auch Offline-Algorithmen. Von außen ist jeweils nur die aktuelle Konfiguration des Systems sichtbar, andere Dinge wie etwa das Verhalten des Algorithmus in der Vergangenheit oder Zugriffshäufigkeiten beispielsweise zunächst nicht.

Für kompetitive Analyse kombinieren werden die Konfigurationsfolgen der beiden Algorithmen zu einer **Ereignissequenz** kombiniert. Es seien $S_{\mathcal{A}}$ und $S_{\texttt{OPT}}$ die Menge der möglichen Konfigurationen, die $\mathcal{A}$, bzw. OPT erzeugen kann. Eine Potentialfunktion ist eine Abbildung

$$\Phi \;:\; S_{\mathcal{A}} \times S_{\texttt{OPT}} \to \mathbb{R} \;.$$

$E = e_1, e_2, \ldots, e_n$ bezeichne die Ereignissequenz, die das Verhalten der Algorithmen auf die Eingabe X beschreibt. e_i bewirkt eine Transition auf $S_{\mathcal{A}} \times S_{\texttt{OPT}}$. Mit Φ_i bezeichnen wir das Potential nach Ereignis e_i, dabei sei Φ_0 das Anfangspotential.

Es seien $\mathcal{A}_i$ und $\texttt{OPT}_i$ die **tatsächlichen Kosten** der beiden Algorithmen für das Ereignis e_i. Als **amortisierte Kosten** von $\mathcal{A}$ bzgl. OPT definieren wir den Ausdruck

$$a_i \;=\; \mathcal{A}_i + \Phi_i - \Phi_{i-1} \;.$$

Sinn der Potentialfunktion ist es einen Ausgleich zu schaffen bei Ereignissen, wo ein Offline-Algorithmus sehr geringe Kosten hat, weil er dies Ereignis vorhersehen konnte, während ein Online-Algorithmus in Unkenntnis der zukünftigen Ereignisse erheblich mehr bezahlen muß. Passiert so etwas nur selten, so kann $\mathcal{A}$ über die Potentialfunktion *Geld für schlechte Zeiten ansparen.*

Eine analoge Abschätzung wie im Beweis von Theorem 7.1 ergibt:

Lemma 7.2 Eim Online-Algorithmus $\mathcal{A}$ ist ρ-kompetitiv, falls eine Potentialfunktion Φ und eine Zahl $b \in \mathbb{Z}$ existiert, die für alle Ereignisfolgen $E = e_1, e_2, \ldots$ die beiden folgenden Bedingungen erfüllt :

1.) $\forall\, i \quad a_i \leq \rho \cdot \mathtt{OPT}_i$,

2.) $\forall\, i \quad \Phi_i \geq b$, d.h. die Potentialfunkton ist nach unten beschränkt.

($\mathcal{A}$ kann nicht beliebig hohe Schulden machen).

Die erste Bedingung kann noch weiter verfeinert werden in Abhängigkeit vom Typ der Ereignisse. Es sei $\Delta_i := \Phi_i - \Phi_{i-1}$ die Veränderung des Potentials durch e_i :

1.1) falls in e_i $\mathcal{A}$ keine Kosten hat und $\mathtt{OPT}$ die Kosten $\mathtt{OPT}_i$: $\Delta_i \leq \rho \cdot \mathtt{OPT}_i$;

1.2) falls in e_i nur $\mathcal{A}$ Kosten $\mathcal{A}_i$ verursacht: $\Delta_i \leq -\mathcal{A}_i$.

7.5 Untere Schranken

Theorem 7.2 Für jeden Online-Algorithmus $\mathcal{A}$ für LZP gilt bei Listenlänge ℓ :

$$\mathtt{CR}(\mathcal{A}) \;\geq\; 2 - \frac{2}{\ell + 1} \;.$$

Beweis: Der Adversary greift jeweils auf das letzte Element in der Liste von $\mathcal{A}$ zu. Dies definiert eine Sequenz X der Länge n mit Kosten $\mathcal{A}(X) = \ell \cdot |X|$.

Wie sieht nun ein optimaler Offline-Algorithmus aus? Wir konstruieren ihn nicht explizit, sondern verwenden ein Durchschnittsargument. Für jede Permutation $\pi \in \Pi_\ell$ definieren wir den Algorithmus B_π folgendermaßen: B_π sortiert die Liste zunächst gemäß π – dies kostet höchstens $b \leq \binom{\ell}{2}$ Transpositionen – und arbeitet dann die Zugriffssequenz X ohne weitere Reorganisationen der Liste ab.

Für einen Zugriff auf eine Element x der Liste berechnet sich die Summe der Kosten über alle Algorithmen B_π als

$$\sum_{i=1}^{\ell} i(\ell - 1)! \;=\; (\ell - 1)!\; \frac{\ell\,(\ell + 1)}{2} \;.$$

Summiert über alle Element der Zugriffssequenz X ergibt Gesamtkosten der Höhe maximal

$$|X|\;(\ell - 1)!\; \frac{\ell\,(l + 1)}{2} + \ell! \cdot b \;.$$

Es muß eine Permutation π geben, so daß der Algorithmus B_π nicht mehr als die durchschnittlichen Kosten über alle Permutationen, nämlich $\frac{1}{2}|X|(\ell + 1) + b$ generiert. Dies bedeutet für die kompetitive Rate

$$\mathtt{CR}(\mathcal{A}) \;\geq\; \frac{\ell \cdot n}{\frac{1}{2}n(\ell + 1) + b} \;\geq\; 2\frac{\ell}{\ell + 1} - o(1) \;,$$

und damit $\texttt{CR}(\mathcal{A}) \geq 2 - \frac{2}{\ell+1}$. ∎

Eine alternative Abschätzung für die maximalen Kosten eines Offline-Algorithmus bei einer beliebigen Sequenz X der Länge n kann wie folgt gegeben werden: OPT ermittelt für jedes Element der Liste die Häufigkeit des Zugriffes in der Folge X und ordnet danach zunächst die Liste absteigend. Dann gibt es in X mindestens n/ℓ Zugriffe auf das erste Element, d.h. mit Kosten höchstens 1. Mindestens n/ℓ weitere Zugriffe erfolgen auf das erste oder zweite Element und verursachen damit Kosten höchstens 2, usw. Damit lassen sich die Gesamtkosten abschätzen durch

$$\binom{\ell}{2} + \sum_{i=1}^{\ell} \frac{n}{\ell} \cdot i \;=\; \binom{\ell}{2} + n \cdot \frac{\ell+1}{2} \;.$$

Es ergibt sich derselbe Wert wie im obigen Beweis, d.h. wir können festhalten:

Lemma 7.3 Für jede Zugriffssequenz X gilt: $\texttt{OPT}(X) \leq |X| \cdot (\frac{\ell+1}{2} + o(1))$.

Diese bessere Abschätzung, bzw. deren Umformung $|X| \geq \texttt{OPT}(X)/(\frac{\ell+1}{2} + o(1))$ angewandt in in Korollar 7.1 zeigt, daß die Rate $2 - \frac{2}{\ell+1}$ durch MTF auch erreicht werden kann. Somit gilt

$$\texttt{CR}(\texttt{MTF}) \;=\; 2 - \frac{2}{\ell+1} \;,$$

d.h. MTF erzielt die bestmögliche kompetitive Rate für das Problem LZP. Legen wir als Gütekriterium für einen Online-Algorithmus die (stark) kompetitive Rate zu Grunde, so ist MTF ein bestmöglicher Algorithmus für das Listenzugriffsproblem.

7.6 Das k-Server-Problem

Viele Online-Probleme kann man auf folgende Weise modellieren. Gegeben seien $k > 1$ mobile Server, die sich in einem metrischen Raum $\mathcal{M} = (M, d)$ bewegen können. M ist dabei eine Menge von Punkten und d eine Abstandsfunktion zwischen Punkten. Die Bewegung eines Servers von Punkt u zum Punkt v verursacht Kosten $d(u, v)$. Gegeben eine Requestsequenz $X = x_1, x_2, \ldots \in M^*$, d.h. eine Folge von Punkten aus M, so ist es Aufgabe eines Server-Algorithmus $\mathcal{A}$, die Requests x_i der Reihe nach zu bedienen, wozu die Anwesenheit eines Servers im Punkt x_i erforderlich ist. Falls sich dort noch kein Server befindet, muß $\mathcal{A}$ einen Server von einem anderen Punkt nach x_i schicken (Beispiele: Taxizentrale, Feuerwehrleitstelle, die die Fahrzeuge zu den Kunden bzw. Brandstellen dirigiert). Ziel ist es, die Gesamtkosten der Bewegungen von Servern möglichst klein zu halten. Es stellt sich die Frage, welche kompetitive Rate ein Online-Algorithmus in Abhängigkeit von $\mathcal{M}$ und k erreichen kann.

Falls $\mathcal{M}$ endlich ist, können wir die m Punkte als Knoten eines vollständigen Graphen darstellen und die Metrik als eine Kostenfunktion auf den Kanten. Der Konfigurationsraum $\mathcal{S}$ dieses Problems hat die Größe $\sigma = \binom{m}{k}$ bzw. m^k oder $m^k/k!$ – die möglichen

Verteilungen der k Server auf M, je nachdem ob auch mehrere Server sich auf einem Punkt befinden können und ob jeder Server eine eigene Identität besitzt.

Modelliert man das Caching-Problem als ein Server-Problem, so entspricht die Anzahl der Server der Größe des Caches, der Raum M ist die Menge aller Datenelemente oder Pages. Befindet sich ein Server auf einem Datenelement v, so bedeutet dies, daß v in einem der k Cacheplätze eingelagert ist. Die Abstandsfunktion ist in diesem Fall uniform: $d(u,v) = 1$ für alle $u \neq v$.

Wie berechnet man offline eine optimale Lösung für dies Problem? Zunächst überlegt man sich, daß es bei jedem neuen Request eines Punktes ohne Server genügt, genau einen Server dorthin zu bewegen. Um eine Requestsequenz X der Länge n abzuarbeiten, muß somit eine Folge von Konfigurationen $\mathcal{C} = C_0, C_1, \ldots, C_n$ gefunden werden, wobei C_0 die Ausgangspositionen der Server beschreibt und für C_i für $i \geq 1$ die Konfiguration unmittelbar nach Erfüllung von x_i ist. In C_i befindet sich daher ein Server auf dem Punkt x_i.

Ist m sehr groß im Vergleich zu n (insbesondere für unendliche Räume), genügt es, sich auf den Teilraum der Punkte zu beschränken, die in X vorkommen, d.h. wir können $\sigma \leq n$ annehmen. Ist die Konfigurationsfolge $\mathcal{C}$ bekannt, dann genügt es, jeweils die *Übergangskosten* von C_{i-1} nach C_i zu minimieren. Die minimalen Kosten $\delta(C_{i-1}, C_i)$ für solch einen Übergang entsprechen einem kostenminimalen Matching zwischen C_{i-1} und C_i bezüglich der Abstandsfunktion d zwischen den Punkten. Wenn pro Request maximal ein Server bewegt wird, unterscheiden sich diese beiden Konfigurationen um höchstens eine Serverposition.

Die Schwierigkeit ist somit, geeignete Konfigurationen C_i zu finden, die jeweils den Request x_i enthalten und in der Summe möglichst geringe Übergangskosten generieren. Eine optimale Folge $\mathcal{C}$ kann durch dynamische Programmierung gefunden werden, indem man suksessive für $i = 1, \ldots, n$ und für jede mögliche Konfiguration C eine kostenminimale Konfigurationsfolge $\mathcal{C}(i, C) = C_0, C_1, \ldots, C_i, C$, in der durch die C_i zunächst die Requests $x_1, \ldots x_i$ erfüllt werden und die Server dann in die Endkonfiguration C wechseln. Die entsprechenden Kosten lassen sich dann berechnen durch die Funktion

$$\begin{aligned} &\texttt{mincost}(\mathcal{C}(i,C)) \quad = \\ &\min_{C' \in \mathcal{S};\, y \in C'} \texttt{mincost}\ (\mathcal{C}(i-1, C')) + \delta(C', C' + x_i - y) + \delta(C' + x_i - y, C)\ . \end{aligned}$$

Hierbei ist $C' + x_i - y$ die Konfiguration, die sich aus C' durch Bewegen eines Servers auf dem Punkt $y \in C'$ auf den Punkt x_i ergibt, somit $\delta(C', C' + x_i - y) = d(y, x_i)$. Der Zeitaufwand für diese dynamischen Programmierung läßt sich abschätzen durch

$$O(n \cdot k \cdot \sigma) \ \leq\ O(n \cdot k \cdot m^k)\ .$$

Wie kann online eine gute Lösung gefunden werden? Auf Grund der Ergebnisse für das Caching-Problem folgt, daß die kompetitive Rate nicht besser als k sein kann. Der naheliegende Online-Greedy-Agorithmus, jeweils einen der am nächsten am Request vorhandenen

Server dorthin zu bewegen, ist überhaupt nicht kompetitiv, wie man an dem einfachen Beispiel einer Menge von 3 Punkten auf einer Linie und 2 Servern sehen kann.

Die berühmte, aber bis heute nicht bewiesene ***k*-Server-Vermutung** behauptet, daß es für jeden metrischen Raum einen k-kompetitiven deterministischen Online-Algorithmus gibt. Die beste bislang bekannte allgemeine obere Schranke ist $2k - 1$. Ansonsten konnte diese Vermutung für einige Speziallfälle von metrischen Räumen bewiesen werden, beispielsweise im eindimensionalen Fall (Punkte auf einer Linie) mit der Euklidschen Abstandsfunktion.

Der **Double-Coverage-Algorithmus** `DOUBCOVER` bewegt jeweils den nächstliegenden Server zur Linken des Requests und den zur Rechten – soweit existent, d.h. der Request liegt in der konvexen Hülle des von den Servern abgedeckten Gebietes – mit gleicher Geschwindigkeit, bis einer der beiden den Requestpunkt erreicht. Der andere bleibt dann dort stehen, wo er sich gerade befindet. Diese Strategie bewegt offensichtlich mehr Server als notwendig und erfüllt damit nicht eine Eigenschaft, die man in der Fachsprache als *lazy* bezeichnet. `DOUBCOVER` verursacht somit bei einem einzelnen Request höhere Kosten als notwendig (genau doppelt so hohe). Man kann jedoch beweisen:

Theorem 7.3 Im eindimensionalen Euklidschen Raum ist `DOUBCOVER` k-kompetitiv. Dies gilt auch, wenn man den Raum von einer Geraden zu einem Baum erweitert (eine spezielle Teilmenge der zweidimensionalen Euklidschen Ebene).

Eine Verallgemeinerung ist das **(h, k)-Server-Problem**. Online-Algorithmen, die über k Server verfügen, werden dabei verglichen mit Offline-Algorithmen, denen nur $h \leq k$ Server zur Verfügung stehen. Die fehlende Information über die zukünftigen Requests soll also partiell dadurch kompensiert werden, daß online mehr Server zur Verfügung stehen. Es stellt sich die offensichtliche Frage, wie weit dies einen Ausgleich schaffen kann. Das simple Caching-Problem zeigt bereits, daß die kompetitive Rate auch hier nicht besser als $k/(k - h + 1)$ sein kann.