# k-Server-Algorithmen

Alexander Leider

4. Februar 2007

# Inhaltsverzeichnis

# 1 Einleitung

In diesem Kapitel werden einige Begriffe erklärt, die zu den weiteren Überlegungen hilfreich sein werden.

## 1.1 Online-Algorithmen

Online-Algorithmen werden bei Optimierungsproblemen verwendet, bei denen die Eingabe nicht vollständig bekannt ist, sondern nach und nach gegeben wird. Es müssen also Entscheidungen ohne Wissen zukünftiger Ereignisse getroffen werden. Algorithmen mit vorgegebener Eingabe werden dagegen Offline-Algorithmen genannt.

## 1.2 Kompetitive Algorithmen

Bei der kompetitiven Analyse vergleicht man die Kosten des Online-Algorithmus im schlechtesten Fall mit den optimalen Kosten des Offline-Algorithmus. Dieses Verhältnis nennt man den *kompetitiven Faktor*.

### 1.2.1 c-kompetitiver Algorithmus

Ein Online-Algorithmus A ist c-kompetitiv, falls eine Konstante $\alpha$ existiert, so dass für alle endliche Eingabesequenzen I gilt:

$$\mathrm{A(I)} \leq \mathrm{c} \cdot \mathrm{OPT(I)} + \alpha$$

wobei OPT(I) die Kosten einer optimalen Lösung seien.

### 1.2.2 Kompetitivität als Spiel

Man kann sich die kompetitive Analyse auch als Spiel zwischen einem Online-Spieler und einem böswilligen Gegner vorstellen. Der Online-Spieler arbeitet mit dem Online-Algorithmus. Der Gegner kennt die Funktionsweise des Algorithmus und darf die Eingabe vorgeben. Das Ziel des Gegners ist es, den Quotienten aus Online- und Offlinekosten zu maximieren, wogegen der Online-Spieler versucht, diesen zu minimieren. Den Gegner zusammen mit dem optimalen Offline-Algorithmus bezeichnet man auch als Offline-Spieler. Bei randomisierten Algorithmen wird unterschieden, wieviel Informationen der Gegner über den Online-Algorithmus hat.

- **Blinder Gegner** (oblivious Adversary) Der blinde Gegner hat kein Wissen über den Ausgang der Zufallsentscheidungen. Er kennt allerdings den Online-Algorithmus und die benutzte Wahrscheinlichkeitsverteilung. Er muss die komplette Eingabefolge im Voraus wählen.

- **Adaptiver Gegner** (adaptive Adversary) Der adaptive Gegner kann jede Anfrage mit dem kompletten Wissen über die bisherigen Aktionen und den Ausgang aller Zufallsentscheidungen treffen.

## 1.3 Das k-Server-Problem

Betrachten wir eine Taxizentrale mit $k$ fahrenden Taxis. Wenn eine Taxibestellung bei der Zentrale ankommt, muss die Taxizentrale entscheiden welches Taxi die Bestellung bedient. Nach Beenden der Fahrt muss auch entschieden werden, ob das Taxi zurückfährt oder sich zu einer anderen Position bewegt, damit spätere Bestellungen am besten bedient werden können. Die Taxizentrale will ihre Entscheidungen so treffen, dass bestimmte Kosten minimiert werden, z.B. Summe der zurückgelegten Strecken oder die Wartezeiten der Kunden.

## 1.4 Formalisierung des k-Server-Problems

Sei R = (M, dist) ein metrischer Raum. M ist die Menge von Punkten im Raum und wir haben $1 \leq k < |M|$ Server zur Verfügung. Eine Anfrage besteht aus einem Punkt $p \in$ M. Die Anfrage wird bedient, indem Server zu den entsprechenden Punkten geschickt werden, sofern dort noch kein Server anwesend ist. Sei dist(i, j) die Entfernung bzw. die Kosten, um einen Server von i nach j zu schicken. Zu minimieren ist die Summe der von allen k Servern zurückgelegten Distanz.

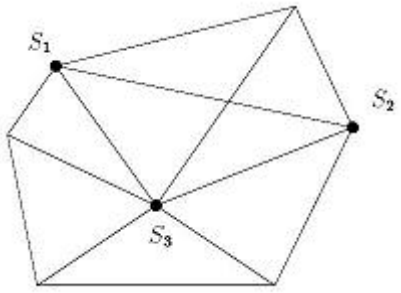


Abbildung 1: Das k-Server-Problem

Für dieses Problem existiert ein (2k-1)-kompetitiver deterministischer Online-Algorithmus. Die grösste bekannte untere Schranke ist k. Für randomisierte Online-Algorithmen ist eine untere Schranke von

$$\Omega(\sqrt{(\log k)/(\log \log k)}$$

[**?**] bekannt.

Das k-Server-Model ist eine Verallgemeinerung folgender interessanter Probleme:

- **Paging** ist ein Spezialfall mit dist(i, j)=1 für alle i≠j, wobei k Server die k Speicherslots im Cache repräsentieren und N=|M| die gesamte Anzahl der Pages im Speicher sind.
- **Gewichtetes Paging** Im Gegensatz zum Paging sind hier die Kosten zur Verschiebung eines Servers vom Punkt i nach j ungleich den Kosten zur Verschiebung von j nach i. D.h. dist(i,j)≠dist(j,i). Dieses Problem taucht in seiner natürlichen Form in Rechnersystemen auf, die z.B. verteilte Dateisysteme verwenden.
- **k-headed Disk** Dieses Problem findet statt falls z.B. k lesende / schreibende Festplattenarme auf eine Festplattendisk zugreifen. Dabei kann jeder der k Festplattenarme jede beliebige Position auf der Festplatte erreichen. Ein Mass der Performance ist hier die gesamte zurückgelegte Distanz der k Festplattenarme.

# 2 Deterministische Ansätze

In diesem Abschnitt betrachten wir einige naheliegende deterministische Ansätze, die zur Lösung des k-Server Problems eingesetzt werden können.

## 2.1 Paging-Problem

**Algorithmus MIN :** Entferne stets diejenige Seite aus dem Cache, deren nächste Anfrage am weitesten in der Zukunft liegt.

**Satz: 1.** *Algorithmus MIN ist ein optimaler Offline-Algorithmus, d.h. er erzeugt für jede Anfragesequenz $\sigma$ stets die minimale Anzahl an Seitenfehlern.*

*Beweis.* Sei $\sigma$ eine beliebige Anfragesequenz und $A$ ein Algorithmus, der auf $\sigma$ eine minimale Anzahl von Seitenersetzungen erzeugt. Dazu bauen wir $A$ so um, dass er schliesslich wie MIN arbeitet. Bei diesem Umbau wird die Anzahl der Seitenersetzungen nicht erhöht. □

## 2.2 Der Greedy-Ansatz

**Algorithmus Greedy:** Bewege den nächstliegenden Server.

Obwohl diese Strategie sinnvoll zu sein scheint, verhält sich diese Strategie katastrophal für die Menge X = $\{a, b, c, d\}$ mit Distanzen $\alpha$ und $\beta$, zwei Server, die sich zum Anfang auf a und b befinden, und eine Anfrage $\sigma$=$\{c, d, c, d, ...\}$. Dabei bewegt sich nur einer der beiden Server ständig und der Andere bleibt nutzlos. Da es in diesem Fall genügt, die beiden Server einmalig auf c und d zu setzen, was optimal wäre, existiert keine Konstante, so dass der Greedy-Ansatz k-kompetitiv wird.

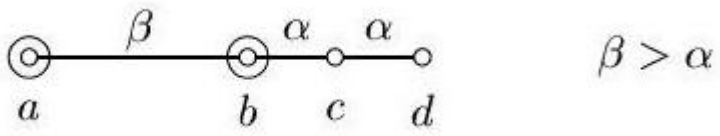


Abbildung 2: Algorithmus Greedy

## 2.3 Algorithmus auf der Datenstruktur Linie

Jetzt wollen wir den Spezialfall betrachten, dass der metrische Raum X die reelle Gerade $\mathbb{R}$ ist.

**Algorithmus Double-Coverage (DC):** Auf einer Linie sind k Server positioniert. Falls eine Position P bedient werden muss, die sich zwischen zwei Server befindet, bewege diese beide Server mit gleicher Geschwindigkeit zu P. Ansonsten befindet sich P ausserhalb der konvexen Hülle aller Server. In diesem Fall bewege einen Server, der am Nähesten zu P liegt.

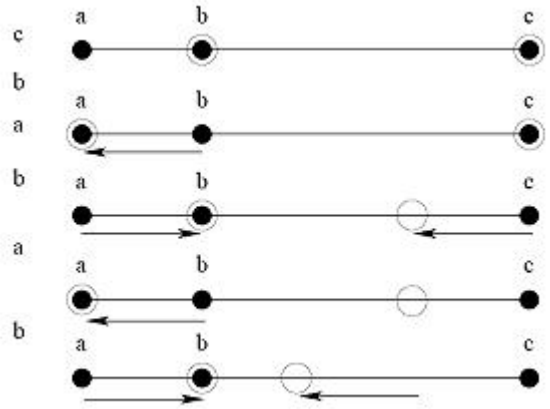


Abbildung 3: Algorithmus Double-Coverage

DC ist k-kompetitiv.[**?**]

Auf einen Beweis verzichten hier, da dieser den Rahmen dieser Arbeit sprengen würde.

## 2.4 Untere Schranke für deterministische Algorithmen

Für jede Instanz des k-Server-Problems gibt es einen deterministischen kompetitiven Algorithmus. Dies kann mit Hilfe des im Folgenden vorgestellten Work Function Algorithmus bewiesen werden.

### 2.5 Der k-Server Work Function Algorithmus

Angenommen, der Algorithmus hat die Konfiguration $A_{t-1}$ erreicht und erhält eine neue Anforderung $\sigma_t$. Dann wird ein Server auf Element x die Anforderung $\sigma_t$ erfüllen, d.h. zur Position $\sigma_t$ bewegt. Somit erhalten wir eine neue Konfiguration

$$A_t = A_{t-1} \backslash \{x\} \cup \{\sigma_t\}.$$

Welcher der k Server soll gewählt werden? Hier wollen wir versuchen eine Konfiguration $A_t$ zu erreichen, die vom Standpunkt eines optimalen Offline-Algorithmus unter den möglichen Nachfolgekonfigurationen von $A_{t-1}$ die geringsten Kosten hat. Allerdings darf diese Nachfolgekonfiguration für den Online-Algorithmus nicht zu teuer sein: Das Einfügen der Online-Kosten dist(x,$\sigma_t$) sorgt für einen Kompromiss.
Der Work Function Algorthmus (WFA) versucht sich also an den optimalen Kosten zu orientieren und berechnet den nächsten Schritt nicht nur an den aktuell minimalen Kosten, sondern auch an optimalen Kosten aller bisherigen Schritten. Der WFA benutzt dafür dynamisches Programmieren. Dieser Algorithmus ist mindestens (2k-1)-kompetitiv [**?**], aber auf einen Nachweis der k-Kompetitivität wollen wir hier verzichten. Hierbei sei erwähnt, dass der Algorithmus $\Theta(nk)$ Speicher benötigt, was für große k nicht praktikabel ist.

## 3 Randomisierte Ansätze

Hier betrachten wir randomisierte Algorithmen zur Lösung des k-Server-Problems. Ein randomisierter Algorithmus ist ein Algorithmus, welcher Zufallsentscheidungen benutzt, um auf Anfragen zu reagieren.

### 3.1 Der randomisierte Greedy-Ansatz

Der randomisierte Greedy-Ansatz schneidet im Vergleich zum einfachen Greedy-Ansatz wesentlich besser ab.

**Algorithmus Randomized Greedy:** In diesem Fall wird ein Server zufällig gewählt, wobei die Wahrscheinlichkeit $p_i$, dass der Server $i$ mit Standort $x_i$ bei Anforderung $x$ gewählt wird, invers proportional zur Distanz dist($x$, $x_i$) ist, also

$$p_i = \frac{1}{dist(x, x_i)} \cdot \frac{1}{\sum_{j=1}^{k} \frac{1}{dist(x, x_j)}}$$

Dieser Ansatz besitzt den kompetitiven Faktor $\frac{5}{4}$k$\cdot 2^k$-$2k$, [**?**]. Diese Lösung ist nicht die effizienteste, doch sie besitzt kompetitiven Faktor und ist damit viel besser als die nicht randomisierte.

## 3.2 Randomisiertes Paging

Als nächstes untersuchen wir die randomisierte Strategie RANDOM Paging für das Paging-Problem.

**Algorithmus RANDOM Paging:** Entferne bei jedem Seitenfehler eine zufällig gemäss Gleichverteilung gewählte Seite aus dem Cache.

**Satz: 2.** *RANDOM Paging ist nicht besser als k-kompetitiv gegen den blinden Gegner.*

*Beweis.* Wähle für $N \gg k$ folgende Eingabesequenz:

$$\sigma = \underbrace{(b_0 a_1 ... a_{k-1})}_{P_0} \underbrace{(b_1 a_1 a_{k-1})^N}_{P_1} \underbrace{(b_2 a_1 a_{k-1})^N}_{P_2} ...$$

Da in jeder Phase $P_i$ die Wahrscheinlichkeit für das Auslagern der Seite $b_i$ bei jedem Zugriff $k^{-1}$ ist, macht RANDOM Paging in jeder Phase im Mittel (Erwartungswert) $k$ Fehler, bis er sich von $b_i$ trennt.

□

## 3.3 Algorithmus CIRC

Als nächstes betrachten wir einen randomisierten Algorithmus für das k-Server-Problem auf der Datenstruktur Kreis.

**Algorithmus CIRC:** Gegeben ist ein Kreis vom Umfang C. C bezeichne die Anzahl der Punkte, die sich auf dem Kreis befinden und von k Server bedient werden können. Der Online-Spieler wählt zufällig einen Punkt P aus dem Kreis. Sei P dem blinden Gegner nicht bekannt. P zerschneidet den gegebenen Kreis in eine Linie. Auf diese Weise wird es möglich, den Algorithmus DC aus Abschnitt 2.3 anzuwenden.

**Satz: 3.** *CIRC ist 2k-kompetitiv gegen den blinden Gegner.*

*Beweis.* Seien zwei optimale Offline-Algorithmen OPTLINE, der eine Anfrage auf der Datenstruktur Linie bedient, und OPT auf der Datenstruktur Kreis. Weiter für jede Anfragensequenz $\sigma$, gilt:

$$CIRC(\sigma) \leq k \cdot OPTLINE(\sigma) \tag{1}$$

Wir begrenzen jetzt OPTLINE($\sigma$) hinsichtlich OPT($\sigma$). Betrachte ein Algorithmus OPT2, der sich wie OPT verhält, falls aber OPT den Punkt P hinübergeht, macht OPT2 einen Umweg d.h. bewegt sich zurück entlang des Kreises. Da OPT2 ein Algorithmus für die Datenstruktur Linie ist, und vom Algorithmus für die Datenstruktur Kreis abgeleitet wurde, gilt:

$$OPTLINE(\sigma) \leq OPT2(\sigma)$$

Jede Distanz $d_i$, die vom OPT2 zusätzlich zurückgelegt wird, kommt mit der grösstmöglichen Wahrscheinlichkeit $\frac{d_i}{C}$ vor. Somit sind die erwarteten Kosten für alle Umwege nicht grösser als

$$D = \sum_i \frac{d_i}{C} \cdot C = OPT(\sigma)$$

Zusammenfassend ergibt sich:

$$OPTLINE(\sigma) \leq E[OPT2(\sigma)] \leq OPT(\sigma) + D = 2 \cdot OPT(\sigma) \qquad (2)$$

Aus (1) und (2) folgt:

$$E[CIRC(\sigma)] \leq 2k \cdot OPT(\sigma)$$

□

## 3.4 Kompetitivität von randomisierten Algorithmen

Abschliessend betrachten wir Kompetitivität der randomisierten Algorithmen.

**Satz: 4.** *R sei ein randomisierter Algorithmus, welcher k Server verwaltet. Dann gilt: R ist k-kompetitiv gegenüber adaptiven Online-Gegnern.*

*Beweis.* Zuerst wird gezeigt, welche Kosten einem (beliebigen) Algorithmus R aufgezwungen werden. Hierfür definiert man die Menge H, welche alle Punkte enthält, die zu Beginn von R abgedeckt werden, plus einem weiteren beliebigen Punkt, also |H|=k+1. Der adaptive Gegner sorgt dafür, dass bei jedem Schritt der Anfragesequenz $\rho_i$ der Punkt aus H verlangt wird, welcher nicht durch R abgedeckt ist. Das bedeutet, dass im i-ten Schritt für das abgefragte $\rho_i$ ein Server bewegt werden muss. Der freigewordene Punkt wird dann im nächsten Schritt zu $\rho_{i+1}$. Dadurch entstehen bei jedem Schritt i Kosten für R in Höhe von $c_{\rho_{i+1}\rho_i}$. Die sicheren Gesamtkosten für R lassen sich dann ausdrücken durch

$$M_R(\rho_1, \rho_2, ..., \rho_N) = \sum_{i=1}^{N} c_{\rho_{i+1}\rho_i} = \sum_{i=1}^{N} c_{\rho_i\rho_{i+1}}$$

Um die Behauptung des Theorems zu zeigen, konstruiert man k verschiedene Online-Gegner $B_j$, deren summierte Kosten in der Hohe von $M_R$ liegen. Die $B_j$ bedecken in der Ausgangslage dieselben Punkte wie R, mit einer Ausnahme: Jeder $B_j$ lässt genau einen Punkt aus der Startkonfiguration von R aus, und jeder $B_j$ einen anderen. Den dadurch freigewordenen Server

setzt jeder $B_j$ auf den Punkt aus der Menge H, der nicht zur Konfiguration von R zählt. Dieser Punkt wird auch das erste Element $\rho_1$ der Anfrage sein. Diese erste Anfrage bereitet noch keinem der k Online-Gegner Kosten (im Gegensatz zu R). Im folgenden aber bewegt ein $B_j$, falls er im i-ten Schritt einen Fehler begeht, den Server von $\rho_{i-1}$ auf $\rho_i$. Durch diese Vorschrift wird sichergestellt, dass sich alle zu jedem Zeitpunkt voneinander unterscheiden: Zu Beginn ist das per Konstruktion so. Gilt dies für alle Konfigurationen vor dem i-ten Schritt und ein $B_j$ begeht einen Fehler, so ist der Punkt $\rho_{i-1}$ nach dem i-ten Schritt nicht mehr in der Konfiguration des $B_j$. Da aber alle anderen Online-Gegner keinen Fehler begangen haben, decken sie noch alle mit einem Server den Punkt $\rho_{i-1}$ ab. Also unterscheidet sich $B_j$ auch nach diesem Schritt weiterhin von allen übrigen Gegnern. Und weil sich die übrigen k-1 Gegner vor Schritt i untereinander unterschieden und keiner von ihnen handeln musste, unterscheiden sie sich auch nach Schritt i noch.
Dies führt zu der Aussage, dass pro Schritt i (mit Ausnahme des ersten Schrittes, der einen Sonderfall darstellt) immer genau ein $B_j$ einen Fehler begeht, denn nur genau einem kann der angefragte Punkt $\rho_i$ fehlen. Zu jedem Fehler von R gibt es also einen zugehörigen Fehler eines der $B_j$, daher gilt

$$\sum_{j=1}^{k} M_{B_j}(\rho_1, \rho_2, ..., \rho_N) \leq M_R(\rho_1, \rho_2, ..., \rho_N).$$

Man kann nicht genau sagen, welche exakten Kosten für einen der $B_j$ entstehen, aber es lässt sich auf jeden Fall mindestens ein Online-Gegner finden, dessen Kosten mindestens um den Faktor k geringer ausfallen als die Kosten von R. □

# 4 Zusammenfassung

In diesem Paper wurde auf deterministische und randomisierte Algorithmen für das k-Server-Problem eingegangen. Dabei ist gezeigt worden, dass die Randomisierung ein Werkzeug ist, mit Hilfe dessen Algorithmen überhaupt kompetitiv werden können. Es sind keine einfache deterministische Algorithmen für die Datenstruktur Kreis bekannt, hier schafft die Randomisierung wieder eine grosse Abhilfe.
Doch es lässt sich auch nicht sagen, dass die Randomisierung immer zur Verbesserung der Kompetitivität oder Komplexität eingesetzt werden kann. Meistens sind bestimmte Voraussetzungen oder Eingrenzungen dafür nötig.

## Literatur

[OCCA, 1998] "Online Computation and Competitive Analysis"; Allan Borodin, Ran El-Yaniv.

[RA, 2000] "Randomized Algorithms"; Rajeev Motwani, Prabhakar Raghavan.