Online Algorithmen

Christoph Jacob
chjacob@inf.fu-berlin.de

Andreas Rebenstorf
rebensto@inf.fu-berlin.de

13. Mai 2004

Zusammenfassung: In Kapitel 1 beschreiben wir den Begriff des Online-Algorithmus zunächst intuitiv. In Kapitel 2 werden die Grundbegriffe formalisert und ein mathematisches Modell für Online-Problme vorgestellt. Als klassisches Online-Problem untersuchen wir die Verwaltung von Speicherseiten in einem zweistufigen Speichersystem (Paging) in Kapitel 3. Bevor wir uns in Kapitel 6 mit weiteren in der Praxis verwendeten Algorithmen beschäftigen, gehen wir in Kapitel 4 auf allgemeine Modelle für Online-Probleme ein.

1 Was sind Online-Algorithmen?

Online-Algorithmen werden bei Optimierungsproblemen verwendet, bei denen die Eingabe nicht vollständig gegeben ist. Es müssen Entscheidungen ohne Wissen zukünftiger Ereignisse getroffen werden.

2 Grundbegriffe

2.1 Offline Optimierungsproblem

Ein **Optimierungsproblem** (über einem Alphabet Σ) ist ein Quadrupel $(\mathfrak{I}, F, C, M)$ wobei:

- $\mathfrak{I} \subseteq \Sigma^*$ die Menge der Eingabeinstanzen (Input)
- $F(I) \subseteq \Sigma^*$ die Menge der zulässigen Lösungen (Output) für die Instanz $I \in \mathfrak{I}$
- $C : \mathfrak{I} \times \Sigma^* \to \mathbb{R}_{\geq 0}$ bezeichnet die Kosten der Lösung O bei Eingabe von I. Der Wert $C(I,O)$ ist nur definiert wenn $O \in F(I)$.
- $M \in \{min, max\}$

Es wird nun für eine gegebene Eingabeinstanz I eine Lösung O gesucht, so dass $C(I,O)$ minimal bzw. maximal ist. Die Lösung dieses Offline-Problems wird als **optimale Lösung** und die Kosten dieser Lösung als **optimale Kosten** $OPT(I)$ bezeichnet. Im folgenden betrachten wir nur Minimierungsprobleme.

2.2 Online-Problem und Online-Algorithmus

Ein **Online-Problem** ist ein Optimierungsproblem, bei dem jede Instanz als Folge $\sigma = r_1, ..., r_n$ von Anfragen (Requests) gegeben ist. Ein **Online-Algorithmus** A, der die Folge σ bearbeitet, generiert nach jeder Anfrage eine Antwort. Die somit generierte Lösung ist eine Folge $A(\sigma) = a_1, a_2, ..., a_n$. Die Antwort a_i darf nur von den Anfragen $r_1, ..., r_i$ abhängen.

2.3 Randomisierte Online-Algorithmen

Ein randomisierter Online-Algorithmus ist ein Algorithmus, welcher Zufallsentscheidungen benutzt, um auf Anfragen zu reagieren.

2.4 kompetitive Analyse

Wie kann man nun die Qualität eines Online-Algorithmus messen? Die klassischen Analysen wie *Worst-Case* oder *Average-Case* machen hier meistens keinen Sinn, denn z.B. beim Paging bei einer Anfrage produziert jeder Algorithmus im schlechtesten Fall einen Seitenfehler. Das Problem bei einer Average-Case Analyse ist, daß man ein statisches Modell für die Eingabe benötigt.

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**.

2.4.1 c-kompetitiver Algorithmus

Sei $\mathfrak{I} = \{I_1, I_2\}$ die Menge der Eingabeinstanzen eines Online-Algorithmus, wobei jede Instanz eine endliche Folge von Anfragen ist. $A(I)$ bezeichne die Kosten des Algorithmus für die Instanz $I \in \mathfrak{I}$ und $OPT(I)$ seien die Kosten einer optimalen Lösung.

A nennt man $c_{\mathfrak{I}}$-kompetitiv, wenn gilt:

$$\forall I \in \mathfrak{I}: \quad A(I) \leq c_{\mathfrak{I}} OPT(I) + b_{\mathfrak{I}}$$

wobei $c_{\mathfrak{I}}$ und $b_{\mathfrak{I}}$ Konstanten sind. Kann $b_{\mathfrak{I}} = 0$ gewählt werden, so nennt man den Algorithmus **strikt c-Kompetiv**.

Ist A ein randomisierter Algorithmus, so nennt man A $c_{\mathfrak{I}}$-kompetitiv falls:

$$\forall I \in \mathfrak{I}: \quad E[A(I)] \leq c_{\mathfrak{I}} OPT(I) + b_{\mathfrak{I}}$$

wobei $c_{\mathfrak{I}}$ und $b_{\mathfrak{I}}$ wieder konstant sind.

2.4.2 Spiel zwischen Online-Spieler und Gegener

Man kan die kompetitive Analyse auch als *Spiel* zwischen einem *Online-Spieler* und einem böswilligen *Gegner* auffasen. 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-Spielers 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.

2.4.3 Beispielanalyse am Ski-Rental Problem

Eine Sportlerin geht das erste Mal in ihrem Leben zum Ski fahren. Seien x die Kosten für das Mieten der Skier für einen Tag und y der Kaufpreis. Es gelte $y = cx$ mit einem ganzzahligen $c > 1$. Die Anzahl t der Tage, die man Ski fährt sind bei dem Online-Problem natürlich vorher nicht bekannt. Vor jedem Ski-Trip entscheidet man sich, ob man die Skier für einen weiteren Tag mietet oder ob man die Skier kauft. Die Strategie, die verwendet wird, sei k, d.h. es wird k-mal gemietet bevor die Skier gekauft werden.

Die Online-Kosten dieser Strategie betragen tx für $t \leq k$ und $kx+y$ für $t > k$. Die optimalen Kosten betragen tx für $t \leq c$ und y für $t > c$.

Der kompetitive Faktor beträgt damit

$$max\left\{\frac{kx+y}{tx}, \frac{kx+y}{y}\right\}.$$

Beobachtung 1 *Jede Strategie k ist für den Fall t=k+1 nicht optimal.*

Beobachtung 2 *Die Strategie $k = y/x$ ist strikt 2-kompetitiv.*

Gibt es eine bessere Strategie? Man kann zeigen, dass dies nicht der Fall ist:

Satz 1 *Keine Strategie hat einen niedrigeren kompetitiven Faktor als 2.*

Gibt es vielleicht einen randomisierten Online-Algorithmus, der einen besseren kompetitiven Faktor hat?

Sei A_i die Strategie, $i-1$ mal zu mieten und beim i-ten mal zu kaufen. Eine randomisierte Strategie ist durch eine Wahrscheinlichkeitsverteilung gegeben, wobei π_i die Wahrscheinlichkeit bezeichnet, dass die Strategie A_i verwendet wird. Wir gehen davon aus, dass der erwartete kompetitive Faktor höchstens α beträgt.

Die Wahrscheinlichkeitsverteilung muss dann folgende Bedingungen erfüllen:

$$E[\text{online cost}] \leq \alpha tx \qquad \text{für } t \leq c \tag{1}$$

$$E[\text{online cost}] \leq \alpha y \qquad \text{für } t > c \tag{2}$$

Wenn man aus den Ungleichungen Gleichungen konstruiert und die so entstehenden Gleichungen löst erhält man:

$$\pi_i = \begin{cases} \frac{\alpha-1}{c}(\frac{c}{c-1})^i & \text{für } i = 1...c \\ 0 & \text{ansonsten} \end{cases}$$

Mit $\sum \pi_i = 1$ erhält man

$$\alpha = \frac{1}{(1+\frac{1}{c-1})^c - \frac{c-1}{c}} + 1$$

Es kann gezeigt werden, dass kein besserer Faktor möglich ist. Für $c \to \infty$, also wenn der Mietpreis beliebig klein im Vergleich zum Kaufpreis wird, erhält man einen Faktor von

$$\frac{e}{e-1} \approx 1,58.$$

3 Paging als klassischer Online-Algorithmus

Gegeben sei eine zweistufige Speicherhierachie mit einem kleinen schnellen Hauptspeicher mit Platz für k Seiten und einem großem aber dafür langsamen Hintergrundspeicher mit Platz für $n \gg k$ Seiten. Die Eingabe besteht aus einer Sequenz von Speicheranforderungen. Fordert ein Programm ein Datum im Speicher an, das sich in einer Seite befindet, die nicht im Hauptspeicher vorhanden ist, so blockiert das Programm solange die entsrechende Seite aus dem Hintergrundspeicher in den Hauptspeicher geladen wurde. Bei einem solchen sog. Seitenfehler wird evtl. eine andere Seite im Hauptspeicher verdrängt, um Platz für die angeforderte Seite zu schaffen. Das Ziel einer Verdrängungsstrategie (**Paging Algorithmus**) ist es, die Anzahl der Seitenfehler zu minimieren.

Als einer der ersten Paging Algorithmen wurde der LRU-Algorithmus (Least Recently Used) von Sleator and Tarjan [1] kompetitiv analysiert. Sie bewiesen (in verallgemeinerter Form) die folgenden Theoreme:

Theorem 1 *LRU hat einen kompetitiven Faktor von k.*

Theorem 2 *Kein deterministischer Algorithmus hat einen kompetitiven Faktor kleiner als k.*

Dieses Ergebnis wird der Realität allerdings wenig gerecht. Der in der Praxis deutlich schlechtere FIFO-Algorithmus hat ebenfalls einen Faktor von *k*. Empirische Untersuchungen von Programmabläufe haben gezeigt, dass LRU einen Faktor besitzt, der deutlich unter *k* liegt. Er liegt in der Praxis sogar unter $logk$, meistens so bei 2.

Ein besserer kompetitiver Faktor kann wieder durch Randomisierung erreicht werden. Ein simpler randomisierter Algorithmus, der einen kompetitiven Faktor von $2\mathfrak{H}_k$ besitzt, wobei $\mathfrak{H}_k = \sum_{1 \leq i \leq k} 1/k$ die k'te Harmonische Zahlt ist, wird von Fiat und anderen [2] beschrieben. Ein komplexerer Algorithmus hat sogar einen Faktor von $\mathfrak{H}_k$ [3]. Es wurde von Fiat bewisen, dass dies auch die untere Schranke darstellt.

4 k-Server Modell

Das k-Server Modell kann man als eine Verallgemeinerung des Paging-Problems betrachten. Sei M ein *metrischer Raum* und S die Positionen der k Server in M (Konfiguration), dann werden bei jeder Anfrage bis zu k Positionen der Server verändert, wobei dabei der Abstand $s_1,\ldots,s_k$ (Kosten) zurückgelegt wird.

k-server Vermutung: Für jede Instanz des k-server Problems gibt es einen deterministischen k-kompetitiven Algorithmus. [4]

Diese Vermutung konnte bisher nicht bewiesen werden, aber ein guter Kandiatat könnte der *Work Function Algorithmus* sein.

4.1 Work Function Algorithmus

Der *Work Function Algorithmus* (WFA) versucht sich im nächsten Schritt an den optimalen (off line) Kosten zu orientieren und berechtet den nächsten Schritt nicht nur an den aktuell minimalen Kosten, sondern auch an optimalen Kosten aller bisherigen Schritten. Der WFA benutzt hierfür dynamisches Programmieren.

Die optimalen Kosten $g_i(X)$ lassen sich wie folgt berechnen:

$$g_i(X) = \overset{min}{X'} \{g_{i-1}(X') + \delta(X, X')\}$$

Algorithmus: Sei r die u-te Anfrage, $X_u - 1$ die aktuelle Konfiguration der Server und X_u die Konfiguration, welche die folgende Funktion minimiert: g_t(X) + $\delta(X_{u-1}$,X)

Dieser Algorithmus ist mindestens 2*k*-1 kompetitiv [5], aber der Nachweis der k-Kompetitivität ist noch nicht erbracht. Hierbei sei erwähnt, dass der Algorithmus $\Theta(n_k)$ Speicher benötigt, was für großes k nicht praktikabel ist.

5 Metrische Task-Systeme

5.1 Modell

Ein metrisches Task-System ist ein Paar (M,R) wobei $M=(M,d)$ ein metrischer Raum und R eine Menge von Aufgaben r mit $r : M \to R_{\geq 0} \cup \{\infty\}$.

Für $s \in M$: sind $r(s)$ die Kosten der Erledigung von Aufgabe r im Zustand s.

Gegeben ist nun eine Aufgabenfolge $\sigma = r_1, r_2, ..., r_n$ und ein Anfangszustand $s_0 \in M$.

Der Algorithmus muss nun die Aufgaben in der gegebenen Reihenfolge bearbeiten, kann aber die Zustände wählen, in der er diese bearbeitet.

Die Kosten des Algorithmus sind dann

$$ALG(\sigma) = \sum_{i=1}^{n} d(s_{i-1}, s_i) + \sum_{i=1}^{n} r_i(s_i)$$

wobei die erste Summe die Übergangskosten und die zweite Summe die Bearbeitungskosten sind.

5.2 Beispiel: Paging

Cachegröße ist k und N die Anzahl aller Datenseiten. M ist die Menge der möglichen Cache-Inhalte, die Kardinalität von M ist $\binom{N}{k}$. Die Metrik d ist wie folgt gegeben: $d(s_1,s_2) =$ Anzahl der anzufordernden Seiten, um von s_1 nach s_2 zu gelangen $= k - \mid s_1 \cap s_2 \mid$. Die Aufgaben sind die Seitenanforderungen. Kosten der Anforderung r im Zustand s ist $r(s) = \begin{cases} 0 \text{ für } r \in s \\ \infty \text{ sonst} \end{cases}$.

5.3 Borodin-Linial-Saks Algorithmus

Dieser Algorithmus ist eine Erweiterung vom WFA-Algorithmus im k-Server-Modell, obwohl er zuerst erschienen ist. Die Funktion $g_i(s)$ für die optimalen Kosten wird wie gehabt berechnet und bezeichnet die minimalen Kosten, um die Aufgaben 1 bis i zu bearbeiten und danach in den Zustand s zu gelangen. Wenn r die t-te Anfrage ist und s_{t-1} den aktuellen Zustand, wenn die Anfrage r ankommt. Dann bewegt man sich in den Zustand s_t der $g_t(s_t) + d(s_t, s_{t-1}) + r(s_t)$ minimiert.

Borodin, Linial und Saks haben bewiesen, dass der Algorithmus $2n-1$-kompetitiv für jedes metrische Task-System ist. Sie haben ebenfalls bewiesen, dass kein deterministischer Algorithmus einen besseren Faktor besitzen kann.

6 Online-Algorithmen in der Praxis

6.1 Datenstrukturen

6.1.1 Sequentielle Suche bzw. list-update Problem

Beim *sequential search* bzw. *list-update* Problem sollen Werte aus einer verketteten Liste möglichst effizient heraus geholt werden. Nachdem der richtige Wert gefunden wurde kann der sequentielle Suchvorgang

abgebrochen und die Liste umgeordnet werden, indem die adjazenten Paare vertauscht werden. Die Umordnung sollte so erfolgen, dass die meist benutzen Elemente an den Anfang geschoben werden. Eine mögliche Optimierung ist die *move-to-front* Regel, die ein gefundenes Element immer an den Anfang der Liste verschiebt.

Die move-to-front Optimierung ist 2-kompetitiv [6], wobei es auch keinen besseren deterministischen Algorithmus geben kann. Entscheidet man die Verschiebung per Zufall, dann existiert ein 1,6-kompetitiven Algorithmus gegen einen blinden Gegner, was in Bezug auf die untere Schranke von 1,5 für alle randomisierten Algorithmen recht zufriedenstellend ist [7].

6.1.2 Dynamische binäre Bäume

Eine andere Datenstruktur für das Lesen und Speichern von Werten sind dynamische binäre Bäume, wie z.B. AVL-Bäume oder rot/schwarz-Bäume. Die Umordnung erfolgt hierbei durch die Rotationen der Bäume [8]. Häufig benutze Elemente werden dabei näher an die Wurzel verschoben, um die Zeit für das Suchen zu minimieren.

Balancierte Bäume sind O($logn$)-kompetitiv, obwohl ein O(1)-kompetitiver Algorithmus vielleicht möglich wäre, konnte dieser bislang nicht gefunden werden. Ein möglicher Kandidat ist der *splay tree* von Sleator und Tarjan [9], der O($logn$)-kompetitv ist, aber der Beweis für O(1) konnte bisher nur unter bestimmten Bedingung bewiesen werden.

6.2 Durchsatzmaximierung

In einen Netzwerk, wie z.B. dem Internet, ist ein Ziel die Durchsatzmaximierung der Leitung, wobei jede Leitung durch eine bestimmte Kapazität begrenzt ist. Es gilt nun zu entscheiden, welche Pakete angenommen und weggeworfen werden sollen.

Es ist ein $log(|E| * m)$-kompetitiver Algorithmus bekannt, wobei $|E|$ die Anzahl der Knoten im Netzwerk und m das Verhältnis zwischen dem Minimum und dem Maximum der Wartezeit eines Pakets ist. [10] [11] [12].

6.3 verteilte Datenspeicherung

Unter verteilter Datenspeicherung versteht man das Speichern von Dateien in einem Netzwerk, wobei die Dateien gelesen und geschrieben werden können und dabei entsprechend abgeglichen werden müssen.

Es gibt mögliche Einschränkungen für die verteilte Datenspeicherung:
1. es dürfen nur Lesezugriffe durchgeführt werden (*Replikation*)
2. es darf nur eine Datei im Netzwerk vorhanden sein (*Migration*)
Zusammengenommen kann man das Problem auch unter dem allgemeinen Problem der *Dateilokalisierung* beschreiben.

6.3.1 Replikation

Wenn wir annehmen, dass nur zwei Server existieren, dann lässt sich das Replikationsproblem auf das Ski-Rental Problem abbilden, womit es einen 2-kompetitiven Algorithmus für deterministische Algorithmen und einen $e/(e-1)$-kompetitiven Algorithmus für randomisierte Algorithmen gibt. Der allgemeine Fall ist komplexer, wofür ein $\Theta(logn)$-kompetitiven Algorithmus existiert. [13]

6.3.2 Migration

Die Migration ist sehr verbreitet im Zusammenhang mit *virtual shared memory* und es sind Algorithmen bekannt mit einem kompetitiven Faktor zwischen 2 und 5.

6.3.3 Dateilokalisierung

Hierbei sind deterministische und randomisierte Algorithmen mit kompetitiven Faktor von O($logn$) bekannt. Handelt es sich bei dem Netzwerk um einen Bus oder einen Baum, dann gibt es sogar Algorithmen mit einem konstanten Faktor 2-3. Normalerweise gibt es aber in Netzwerken auch das einfache Problem herauszufinden, welches die nächstegelegene Datei ist. Rechnet man diesen Faktor mit ein, dann ergibt sich der Faktor zu O($log^2 n$). [14]

6.4 Roboternavigation

Bei der Roboternavigation wird ein Roboter in eine unbekannte Umgebung mit Hindernissen gestellt und diser muss selbstständig zum Ziel finden. Dabei werden ihm die möglichen Wege der nächsten Position erst bekannt, wenn er diese erreicht hat.

6.4.1 1-dimensionale Suche

Betrachtet man das Problem zunächst eindimensional, dann ist eine mögliche Strategie zunächst ein Schritt ($n = 1$) in die eine Richtung zu gehen, und bei einem Misserfolg zum Ausgang zurückzukehren und $n+2$-Schritte in die andere Richtung zu gehen, dann wieder zum Ausgang und $n+4$-Schritte in die andere Richtung usw. Diese Strategie ist 9-kompetitiv.

6.4.2 2-dimensionale Suche

Ein Roboter wird in eine unbekannte Umgebung gestellt und soll durch einen On-line Algorithmus möglichst schnell zum Ziel finden. Dieses Problem läßt sich auch auf einen gerichteten Graphen abbilden, indem der Startpunkt ein beliebiger Knoten des Graphen ist und jeder Knoten durch den Roboter eindeutig identifiziert werden kann. Der Roboter kann entsprechend nur die inzidenten Knoten sehen, weiß aber nicht wohin diese führen, bevor er diese erreicht hat.

Eine typische Ausprägung dieses Problems ist das Finden eines Ziels in einem Labyrinth, wobei der Roboter jederzeit seine Position bestimmen kann und ihm die Koordinaten des Ziels bekannt sind [15].

Wenn die Hindernisse nur aus Rechtecken bestehen, die parallel zu den Achsen verlaufen, dann sind Navigationsalgorithmen mit einen kompetitiven Faktor von O($\sqrt{n}$) bekannt. Sind die Hindernisse konkav und die Seite parallel zu den Achsen, dann kann kein Algorithmus besser als Θ(n) sein.

6.5 Task-Scheduling / Load-Balancing

Das Problem Task-Scheduling bzw. Load-Balancing hat in der Praxis verschiedenste Ausprägungen, die im folgenden exemplarisch genannt werden:

1. Minimierung der Rechenzeit für sequenziell eintreffende Prozesse auf einem Multiprozessorsystem [16]
2. Verteilung von parallel arbeitenden Prozessen bei einem Multiprozessorsystems [17]

3. Minimierung der Auslastung bei Servern und damit die Verteilung der eintreffenden Anfragen, wobei jeder Prozess nur von einem Teil der Server beantwortet werden kann [18]

4. Minimierung der Wartezeit von höher priorisierten Prozessen [19]

Literatur

[1] Sleator,D. and Tarjan,R.E., Amortized efficiency of list update and paging rules. Communications of the ACM,28,202-208, 1985a.

[2] Fiat, A.,Karp, R., Luby, M., McGeoch, L.A, Sleator, D. and Young, N., Competitive paging algorithms. Journal of Algorithms, 12,685-699, 1991.

[3] McGeoch, L. and Sleator, D., A strongly competitive randomized paging algorithm. Journal of Algorithms, 6,816-825, 1991.

[4] Manasse, M. McGeoch, L.A., und Sleator, D. Competitive algorithmus for online problems. In Proc. 20th Annual ACM Symposium on Theory of Computing, 322-333, 1988.

[5] Koutsopias, E. und Papdimtriou, C., On the k-server conjecture. In Proc. 25th Symposium on Theory of Computing, 507-511, 1994a.

[6] Sleator, D. and Trajan, R.E., Amortized efficiency of list update and paging rules. Communications of the ACM, 28, 202-208, 1985a.

[7] Teia, B., A lower bound for randomized list update algorithmus. Information Processing Letters, 48,5-9, 1993.

[8] T.H. Corman, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithmus. Second Edition Kapitel 13, Seite 273f, 2001.

[9] Sleator, D.D. an Tarjan, R.E., Self-adjusting binary search trees. J. Assoc. Comput. Mach., 32, 652-686, 1985b.

[10] Garay, J. and Gopal, I. Call preemption in communication networks. In Proc. Infocom, 1992.

[11] Garay, J. Gopal, I., Kutten, S., Mansour, Y., and Yung, M., Efficient online call control algorithmus. In Proc. 2nd Israel Symposium on Theory of Computing and Systems, 285-293, 1993.

[12] Awerbuch, B. Azar, Y. and Plotkin, S., Throughput-competitive online routing. In 34th IEEE Symposium on Foundations of Computer Science. 32-40, 1993.

[13] Imase, M. und Waxman, B.M., Dynamic Steiner tree problem. SIAM J. Discrete Math., 4, 369-384, 1991.

[14] Motwani, R., Phillips, S. and Torng, E., Non-clairvoyant scheduling. In Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithmus, 422-431, 1993.

[15] Blum, A., Raghavan, P. and Schieber, B., Navigating in unfamiliar geometric terrain. In Proc. 23rd STOC, 494-504, 1991.

[16] Shmoys, D.B., Wein, J. und Williamson, D.P., Scheduling parallel machines on-line. In McGeoch, L.A. und Sleator, D.D., Eds., On-Line Algorithmus, volume 7 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 163-166. AMS/ACM, 1991.

[17] Feldmann, A., Kao, M.Y. Sgall, J. und Teng, S.H., Optimal online scheduling of parallel jobs with dependencies. In Proc. 25.th ACM Symposium on Theroy of Computing, 642-651. ACM, 1993.

[18] Azar, Y. Broder, A. und Plotkin, S., Throughput-competitive online routing. In Proc. 33rd IEEE Symposium on Foundations of Computer Science, 218-225. To appear in Theoretical Computer Science, 1992.

[19] Hall, L., Shmoys, D. and Wein, J., Scheduling to minimize avarage completion time: Off-line an online algorithmus. In Proc. of 7th ACM-SIAM Symposium on Discrete Algorithmus, 142-151, 1996.