Online Algorithmen

Christoph Jacob
chjacob@inf.fu-berlin.de

Andreas Rebenstorf
rebensto@inf.fu-berlin.de

13. Mai 2004

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.

Grundbegriffe

Offline Optimierungsproblem

Bei einem Optimierungsproblem wird zu einer gegebenen 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.

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.

Randomisierte Online-Algorithmen

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

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**.

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.

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 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 beschrieben. Ein komplexerer Algorithmus hat sogar einen Faktor von $\mathfrak{H}_k$. Es wurde von Fiat bewisen, dass dies auch die untere Schranke darstellt.

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.

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

Work Function Algorithmus

Der *Work Function Algorthmus* (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, 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.

Online-Algorithmen in der Praxis

Datenstrukturen

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, 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.

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. 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, der O($logn$)-kompetitv ist, aber der Beweis für O(1) konnte bisher nur unter bestimmten Bedingung bewiesen werden.

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 $log(|V| * m)$-kompetitiver Algorithmus bekannt, wobei $|V|$ die Anzahl der Knoten im Netzwerk und m das Verhältnis zwischen dem Minimum und dem Maximum der Wartezeit eines Pakets ist.

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.

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.

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.

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 $O(log^2 n)$.

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.

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 erste Richtung usw. Diese Strategie ist 9-kompetitiv.

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.

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 $\Theta(n)$ sein.

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
2. Verteilung von parallel arbeitenden Prozessen bei einem Multiprozessorsystems
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
4. Minimierung der Wartezeit von höher priorisierten Prozessen