Ausarbeitung zum Thema

Approximationsalgorithmen

im Rahmen des Fachseminars

24. Juli 2009

Robert Bahmann
robert.bahmann@gmail.com
FH Wiesbaden

Erstellt von:	Robert Bahmann
Zuletzt berarbeitet von:	Robert Bahmann
Email:	robert.bahmann@gmail.com
Datum:	24. Juli 2009
Version:	22

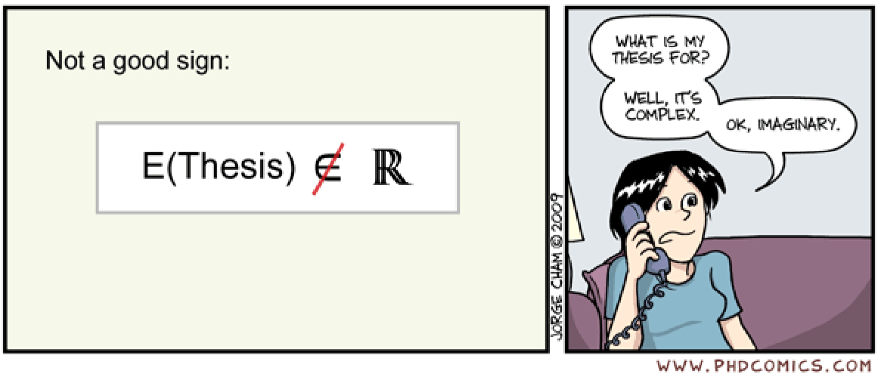

Abbildung 1: Piled High & Deeper [?]

Inhaltsverzeichnis

1 Einfhrung

Approximative Algorithmen sind Algorithmen, die mit schneller[1] Laufzeit, *nherungsweise* Ergebnisse fr Probleme liefern, fr die bis heute keine Algorithmen mit polynomieller Laufzeit bekannt sind oder existieren solange $\mathbb{P} \neq \mathbb{NP}$ gilt. Die Besonderheit der *Approximativen Algorithmen* besteht hier in der „Garantie“ der Gte. Whrend bei *Heuristischen Verfahren* die Qualittsaussagen experimentell u.a. mit Eingaben durch *Benchmark-Mengen* bestimmt werden. Oder den *Parametrisierten Algorithmen* bei denen zwar immer eine optimale Lsung gefunden wird, aber der nicht polynomielle Teil der Laufzeit vorher durch die Parameter eingeschrnkt werden muss.

1.1 Beispiel List Scheduling

Ein praktisch relevantes Beispiel stellt das Problem **JS!** (**JS!**) dar:

Beispiel 1 (Job Scheduling)
Auf p Prozessoren $(P_1, P_2, ..., P_n)$ sollen t Tasks $(T_1, T_2, ..., T_n)$ verteilt werden. Dabei hat jeder Task eine Laufzeit l_i, die der Prozessor bentigt, um diesen abzuarbeiten.

Zustzlich gilt:

- Angefangene Tasks knnen nicht abgebrochen werden
- Ein Prozessor kann zu jedem Zeitpunkt nur einen Task ausfhren
- Die Ausfhrungsgeschwindigkeit der Prozessoren ist identisch

Gesucht ist dann eine Belegung der Prozessoren, auch Schedule genannt, die alle Tasks mit der kleinsten Gesamtlaufzeit abarbeitet.

Beispiel 2 (Prozessorbelegung)
Auf einem Rechner mit einem Dual-Core mssen folgende Tasks abgearbeitet werden:

Task	Zeit
1	1
2	2
3	2
4	4
5	1

(1)

Die optimale Belegung hierzu wre $((0, P_1), (1, P_1), (3, P_1), (0, P_2), (4, P_2)$ und htte eine Dauer von 5 Zeiteinheiten.

[1] Schnell ist hier gleichbedeutend mit Polynomzeit

Die optimale Belegung lsst sich hier noch durch simples Probieren herausfinden, indem man alle Mglichkeiten durchprobiert und die jeweilige Laufzeit der Belegung notiert.

Eine einfache Implementierung wre der Algorithmus Job Scheduling nach Graham ([?]). Dieser verteilt die Tasks auf den nchsten jeweils gerade frei gewordenen Prozessor. Dies ist zwar am einfachsten zu implementieren, hat aber den Nachteil, dass der Algorithmus die Eingabe als Gesamtes nicht betrachtet.

Algorithmus 1 : Job Schedule

Input : p Prozessoren,t Tasks und die Laufzeiten l_i fr jeden Task
Output : Eine Prozessorbelegung S mit minimalen Makespan
S = null ;
for $i = 1$ *to* p **do**
    a_i = null;
for $k = 1$ *to* t **do**
    Berechne das kleinste i mit $a_i = min\{a_t; t \in \{1..., p\}\}$;
    $s_k = a_i$;
    $a_i = a_i + l_k$;
    $S = S \cup \{(S_k, P_i)\}$
return S;

Der Makespan des Algorithmus zu Beispiel **??** wrde dann wie folgt aussehen:

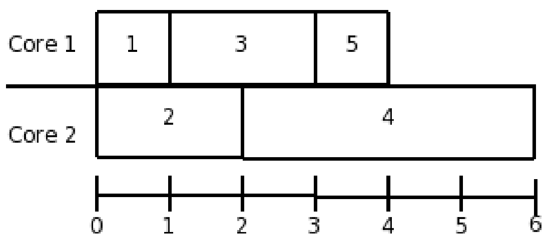

Abbildung 2: Nicht ganz so optimale Belegung

2 Definitionen

Wenn Σ Alphabet ist, dann ist Σ^* die Menge aller endlichen Folgen von Elementen aus Σ. Man spricht hier auch von *Wrtern*. In dieser Definition ist auch die leere Folge, auch leeres Wort genannt, mit eingeschlossen, welche mit ϵ bezeichnet wird. Siehe auch [?, S.11]

Beispiel 3 (Binres Alphabet)
Sei ein Alphabet $\Sigma = \{1, 0\}$ gegeben.
So ist $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, ...\}$.

Definition 2.1 (Charakteristische Funktion)
Die charakteristische Funktion einer Menge $P \subseteq \Sigma^*$ ist $f_P : \Sigma^* \to \{0, 1\}$. Wobei fr alle $w \in \Sigma^*$ gilt:

$$f_P(w) = \begin{cases} 1, w \in P \\ 0, w \notin P \end{cases} \tag{2}$$

Die charakteristische Funktion einer Menge gibt also Auskunft darber, ob ein Wort w in der Menge Σ^* enthalten ist. Wie in der Informatik gebruchlich entspricht $1 \triangleq$ ***Wahr*** und $0 \triangleq$ ***Falsch***.

2.1 Probleme

Generell kann man zwischen 4 Problemarten unterscheiden:

Entscheidungsproblem Gibt es m?

Suche Finde den Weg von l nach m

Optimierung Finde den *krzesten* Weg von l nach m

Zhlen Zhle die Wege von l nach m

Definition 2.2 (Entscheidungsproblem)
Ein (Entscheidungs-)Problem ist eine Menge P $\subseteq \Sigma^*$ fr die, die charakteristische Funktion C halb oder ganz berechenbar ist.

Siehe auch [**?**, S. 122].

Definition 2.3 (Entscheidungsalgorithmus)
Ein Algorithmus A lst ein Entscheidungsproblem E gdw. er fr alle $x \in E_p$ hlt und wenn $x \in Y_p$ gilt der Algorithmus 1 ausgibt.

I_E ist die Menge aller mglichen Eingaben. Und jede mgliche Eingabe I_E fhrt zu einer Antwort 1 oder 0 (Wobei 1 = Ja und 0 = Nein). Dies ist deshalb wichtig, da aus einem Optimierungsproblem durch eine zustzliche Schranke ein Entscheidungsproblem gemacht werden kann.

Definition 2.4 (Optimierungsproblem)
Ein **Optimierungsproblem** O besteht aus einem Quadruppel $O(I_o, F, w)$:

- Eine Menge I_o bestehend aus einer Menge von Instanzen
- Zu jeder Instanz i aus I_O gibt es eine $F(i) =_{\text{def}} \{\text{Menge aller zulssigen Lsungen}\}$

- Zu jeder Lsung L_O aus $F(i)$ gibt es einen Funktion $w(L_O)$ die, die Lsung bewertet
- Ziel $w \in \{\min,\max\}$

Bei Optimierungsproblemen existieren meist zu einer Eingabe mehrere Lsungen. Hier soll dann abhngig von einem Bewertungskriterium eine Lsung mit minimalen Kosten und maximalen Nutzen gefunden werden. Bei dem Beispiel des **JS!** wren die Instanzen I die Anzahl der Prozessoren (oder Kerne), sowie die abzuarbeitenden Tasks und deren Bearbeitungszeit. $F(i)$ sind alle Schedules mit allen abzuarbeitenden Prozessen. w entspricht hier der Zeit der Bearbeitung aller Jobs und ist zu minimieren (Minimierungsproblem).

Definition 2.5 (Approximationsalgorithmus)
Ein Approximationsalgorithmus fr ein Optimierungsproblem O ist ein Algorithmus der zur Eingabe I

- polynomielle Laufzeit hat
- eine zulssige Lsung $L_O \in F(i)$ berechnet

2.2 Komplexitt

Um Algorithmen bewerten zu knnen werden einige Aussagen ber die Laufzeit und Komplexitt bentigt. Der Einfachheit halber messen wir die Laufzeit eines Algorithmus in Instruktionen die abgearbeitet werden msssen bis der Algorithmus terminiert. Dabei entspricht eine Anweisung im Code einer Instruktion im Prozessor.

Definition 2.6 (Komplexittsklasse $\mathbb{P}$)
Probleme gehren zur Komplexittsklasse $\mathbb{P}$ gdw. fr sie ein Algorithmus existiert, der sie unabhngig von der Eingabelnge der Instanzen in polynomieller Laufzeit lst.

Auch wenn die Klasse $\mathbb{P}$ in der Algorithmik als *schnell* gilt. So ist zu beachten, dass ein Algorithmus mit einer Laufzeit von $n^{(2000)}$ sich zwar in $\mathbb{P}$ befindet aber alles andere als eine kurze Laufzeit hat.

Definition 2.7 (Komplexittsklasse $\mathbb{NP}$)
Eine Sprache $L \subseteq \Sigma^*$ befindet sich in $\mathbb{NP}$, wenn fr sie ein polynomiell beschrnkter Algorithmus A und ein Polynom p existieren, so dass fr alle $x \in \Sigma^*$ gilt:
$x \in L$ genau dann wenn $\exists w$ mit $|w| \leq p(|x|)$so, dass $A(x, w) =$ wahr

Die Klasse $\mathbb{NP}$ besteht also aus Problemen, fr deren Lsung kein effizienter Algorithmus bekannt ist, die Lsung selber aber in polynomialzeit berprfbar ist.[2]

[2] Eine Liste mit NP-Vollstndingen Problemen lsst sich hier finden: `http://www.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html`

Einen Schritt weiter geht die Klasse $\mathbb{NPO}$, die die Komplexittsklasse der NP der Optimierungsprobleme darstellt.

Definition 2.8 (Komplexittsklasse $\mathbb{NPO}$)
Ein Optimierungsproblem $O = (I_o, F, w)$ liegt in der Klasse $\mathbb{NPO}$ gdw.:

- In polynomieller Zeit getestet werden kann ob eine Eingabe auch eine Instanz kodiert
- Die Bewertungsfunktion ist in polynomieller Zeit berechenbar
- Es exisitieren Polynome p und q so dass fr alle Instanzen $i \in I$ gilt:

  Fr jede zulssige Lsung $x \in F(i)$ gilt $|x| \leq p(|i|)$

  Fr jeden String y mit $|y| \leq p(|i|)$ kann in Zeit $q(|i|)$ getestet werden, ob $y \in F(i)$

Definition 2.9 (Komplexittsklasse $\mathbb{PO}$)
Ein Optimierungsproblem O liegt in der Klasse $\mathbb{PO}$, wenn $O \in \mathbb{NPO}$ liegt und zu jeder Instanz in polynomieller Zeit eine optimale Lsung berechnet werden kann.

2.3 Gte

Damit konkrete Aussagen ber die Qualitt der errechneten Ergebnisse gemacht werden knnen, mssen diese mit der optimalen Lsung verglichen werden. Der einfachste Weg ist die Differenz. Je kleiner sie ist desto nher liegt der errechnete Wert am Optimum. Stellt man Aussagen ber diese Art des Vergleichs an, so spricht man von einer absoluten Gte, da das Ergebnis im absoluten Zusammenhang mit dem Optimalen Wert steht.

Definition 2.10 (Absolute Gte)
Sei Π ein Optimierungsproblem und A ein Approximationsalgorithmus fr Π.

- $A(I)$ hat eine absolute Gte von $\mathcal{AG}_A(\mathcal{I}) = |A(\mathcal{I}) - \mathrm{OPT}(\mathcal{I})|$
- $\mathcal{AG}_A^{WC}(n) = \max\{\mathcal{AG}_A(\mathcal{I})| \in \mathcal{D}, |\mathcal{I}| \leq \mathrm{n}\}$ ist die *absolute*-worst-case-Gte

Gilt fr alle n: $\mathcal{AG}_A^{WC}(\mathrm{n}) \leq \mathcal{AG}_A(n)$ so **garantiert** A eine absolute Gte von $\mathcal{AG}_A(n)$. Man spricht dann von einer *absoluten Gtegarantie*. Solch eine absolute Garantie ist dann sehr praktisch, wenn die Lsungen gro sind. So ist eine absolute Abweichung von 5 bei einem optimalen Wert von 6 mehr oder weniger katastrophal, bei einem optimalen Wert von 100.000 jedoch praktisch zu vernachlssigen. Ein weitere Mglichkeit ist es die Gte in Relation zum optimalen Wert anzugeben. Man spricht dann von einer relativen Gtegarantie.

Definition 2.11 (Relative Gte)
Sei A ein Approximationsalgorithmus fr das Optimierungsproblem O

- A hat eine *relative* Gte von

$$\mathcal{RG}_A(\mathcal{I}) = \max\left\{\frac{A(\mathcal{I})}{OPT(\mathcal{I})}, \frac{OPT(\mathcal{I})}{A(\mathcal{I})}\right\} \tag{3}$$

- $\mathcal{RG}_A^{WC}(n) = \max\{\mathcal{RG}_A(\mathcal{I}) | \in \mathcal{D}, |\mathcal{I}| \leq \mathrm{n}\}$ ist die *relative*-worst-case-Gte

Gilt fr alle n $\mathcal{RG}_A^{WC}(n) \leq \mathcal{RG}_A(N)$ dann **garantiert** A eine relative Gte von $\mathcal{RG}_A^{WC}(n)$. Die Gte kann demnach in Dezimalzahlen von 0 bis ∞ gemessen werden. Wobei 0 dem Optimum entspricht.

Spricht man von einer relativen Gte muss auch der relative Fehler definiert werden:

Definition 2.12 (Relativer Fehler)
Ein Approximationsalgorithmus A hat einen relativen Fehler $\mathcal{RF}$ von

$$\mathcal{RF}_A(\mathcal{I}) = \frac{|A(\mathcal{I}) - OPT(\mathcal{I})|}{OPT(\mathcal{I})} = \left|\frac{A(\mathcal{I})}{OPT(\mathcal{I})} - 1\right| \tag{4}$$

3 Approximationsklassen

Bisher wurde in den Definitionen davon ausgegangen, dass die Abweichung vom optimalen Wert sich nicht ndert. Allerdings gibt es auch Algorithmen bei denen man sich auf Kosten der Laufzeit eine kleinere Abweichung „erkauft“. Man bergibt dem Algorithmus einen maximal erlaubten Fehler und macht die Laufzeit von jenem abhngig.

FPTAS steht fr **F**ully **P**olynomial-**T**ime **A**pproximation **S**cheme und enthlt alle Approximationsalgorithmen deren Laufzeit polynomiell zur Eingabe und zur Gte ist.

PTAS steht **P**olynomial-**T**ime **A**pproximation **S**cheme und enthlt alle Approximationsalgorithmen die sich lediglich zur Gte polynomiell verhlt..

APX Alle approximierbaren Algorithmen.

4 Beispielanalyse

Anfangs wurde das Problem **JS!** und folgender Algorithmus dafr vorgestellt:

Nun soll die Gte des List Schedule analysiert werden. Seien also p Prozessoren gegeben und t Tasks die abgearbeitet werden mssen. Dabei seien wie im Algorithmus s_k der Startzeitpunkt, der dem Job vom Algorithmus zugewiesen wurde. Z

Algorithmus 2 : Job Schedule

Input : p Prozessoren,t Taks und die Laufzeiten l_i fr jeden Task
Output : Eine Prozessorbelegung S mit minimalen Makespan
S = null ;
for i = *1 to* p **do**
  a_i = null;
for k = *1 to* t **do**
  Berechne das kleinste i mit $a_i = min\{a_t; t \in \{1..., p\}\}$;
  $s_k = a_i$;
  $a_i = a_i + l_k$;
  $S = S \cup \{(S_k, P_i)\}$
return S;

sei die Laufzeit des gesamten vom Algorithmus erzeugten Schedule. $Z_k = s_k + l_k$ sei der Zeitpunkt an dem der k-te Task endet. Des weiteren sei T_{last} der Task der als letztes beendet wird. Daher gilt: $Z_{last} = Z$. Die durchschnittliche Laufzeit des gesamten Schedule betrgt

$$\frac{1}{p}\sum_{k=1}^{t} l_k \tag{5}$$

und die durchnittliche Laufzeit jeder Maschine bis zum Startzeitpunkt des letzten Tasks

$$\frac{1}{p}\sum_{k\in\{1,...,t\}\setminus\{last\}} l_k \tag{6}$$

Es gibt also mindestens eine Maschine, die bis zu diesem Zeitpunkt voll ausgelastet war. Daraus lsst sich schlieen, dass der Startzeitpunkt des letzten Tasks entweder genau auf diesem Punkt liegt oder frher (auf einer anderen Maschine).

$$s_{last} \leq \frac{1}{p}\sum_{k\in\{1,...,t\}\setminus\{last\}} l_k \tag{7}$$

So folgt daraus:

$$\begin{aligned}
Z &= Z_{last} && (8)\\
&= s_{last} + l_{last} && (9)\\
&\overset{(8)}{\leq} \frac{1}{p}\sum_{k\in\{1,...,t\}\setminus\{last\}} l_k + l_{last} && (10)\\
&= \frac{1}{p}\sum_{k=1}^{t} l_k + \left(1 - \frac{1}{p}\right) l_{last} && (11)\\
& && (12)
\end{aligned}$$

Sei OPT ein Optimaler Schedule so ist $OPT \geq l_{last}$. Des Weiteren gilt: $OPT \geq \frac{1}{p}\sum_{k=1}^{t} l_k$.

$$\leq OPT + \left(1 - \frac{1}{p}\right) OPT \tag{13}$$

$$\leq \left(2 - \frac{1}{p}\right) OPT \tag{14}$$

Damit kann also gezeigt werden, dass der Algorithmus eine *garantierte* Gte von $\left(2 - \frac{1}{p}\right)$ hat, wobei p die Anzahl der Prozessoren ist. Die Schwachstelle dieses Algorithmus ist, dass er durch groe Tasks am Ende eine stark unausgeglichene Verteilung erzeugt. Dies knnte durch vorheriges sortieren der Tasks vermieden werden.