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

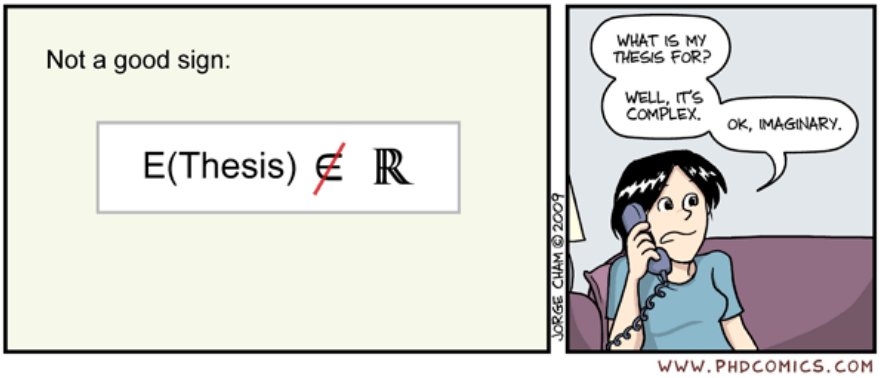

Abbildung 1: Piled High & Deeper [Cha09]

Inhaltsverzeichnis

1 Einführung

Approximative Algorithmen sind Algorithmen, die mit schneller[1] Laufzeit, *näherungsweise* Ergebnisse für Probleme liefern, für 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 Güte. Während bei *Heuristischen Verfahren* die Qualitätsaussagen experimentell u.a. mit Eingaben durch *Benchmark-Mengen* bestimmt werden. Oder den *Parametrisierten Algorithmen* bei denen zwar immer eine optimale Lösung gefunden wird, aber der nicht polynomielle Teil der Laufzeit vorher durch die Parameter eingeschränkt werden muss.

1.1 Beispiel List Scheduling

Ein praktisch relevantes Beispiel stellt das Problem **Job Scheduling** (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 benötigt, um diesen abzuarbeiten.

Zusätzlich gilt:

- Angefangene Tasks können nicht abgebrochen werden
- Ein Prozessor kann zu jedem Zeitpunkt nur einen Task ausführen
- Die Ausführungsgeschwindigkeit 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 müssen folgende Tasks abgearbeitet werden:

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

(1)

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

[1] Schnell ist hier gleichbedeutend mit Polynomzeit

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

Eine einfache Implementierung wäre der Algorithmus JOB SCHEDULING nach Graham ([Gra71]). Dieser verteilt die Tasks auf den nächsten 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 für 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 2 würde dann wie folgt aussehen:

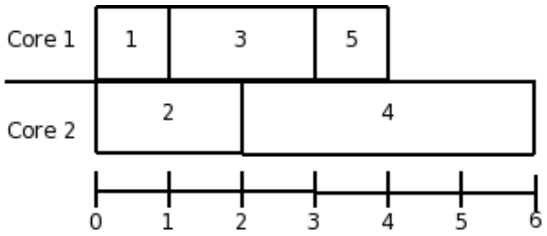

Abbildung 2: Nicht ganz so optimale Belegung

2 Definitionen

Wenn Σ Alphabet ist, dann ist Σ^* die Menge aller endlichen Folgen von Elementen aus Σ. Man spricht hier auch von *Wörtern*. In dieser Definition ist auch die leere Folge, auch leeres Wort genannt, mit eingeschlossen, welche mit ϵ bezeichnet wird. Siehe auch [Sch92, S.11]

Beispiel 3 (Binäres 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 für 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 darüber, ob ein Wort w in der Menge Σ^* enthalten ist. Wie in der Informatik gebräuchlich 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 *kürzesten* Weg von l nach m

Zählen Zähle die Wege von l nach m

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

Siehe auch [Sch92, S. 122].

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

I_E ist die Menge aller möglichen Eingaben. Und jede mögliche Eingabe I_E führt zu einer Antwort 1 oder 0 (Wobei 1 = Ja und 0 = Nein). Dies ist deshalb wichtig, da aus einem Optimierungsproblem durch eine zusätzliche 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 $F(i) =_{\text{def}}$ {Menge aller zulässigen Lösungen}

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

Bei Optimierungsproblemen existieren meist zu einer Eingabe mehrere Lösungen. Hier soll dann abhängig von einem Bewertungskriterium eine Lösung mit minimalen Kosten und maximalen Nutzen gefunden werden. Bei dem Beispiel des JS wären 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 für ein Optimierungsproblem O ist ein Algorithmus der zur Eingabe I

- polynomielle Laufzeit hat
- eine zulässige Lösung $L_O \in F(i)$ berechnet

2.2 Komplexität

Um Algorithmen bewerten zu können werden einige Aussagen über die Laufzeit und Komplexität benötigt. Der Einfachheit halber messen wir die Laufzeit eines Algorithmus in Instruktionen die abgearbeitet werden müsssen bis der Algorithmus terminiert. Dabei entspricht eine Anweisung im Code einer Instruktion im Prozessor.

Definition 2.6 (Komplexitätsklasse $\mathbb{P}$)
Probleme gehören zur Komplexitätsklasse $\mathbb{P}$ gdw. für sie ein Algorithmus existiert, der sie unabhängig von der Eingabelänge der Instanzen in polynomieller Laufzeit löst.

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 (Komplexitätsklasse $\mathbb{NP}$)
Eine Sprache $L \subseteq \Sigma^*$ befindet sich in $\mathbb{NP}$, wenn für sie ein polynomiell beschränkter Algorithmus A und ein Polynom p existieren, so dass für 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, für deren Lösung kein effizienter Algorithmus bekannt ist, die Lösung selber aber in polynomialzeit überprüfbar

ist.[2] Einen Schritt weiter geht die Klasse $\mathbb{NPO}$, die die Komplexitätsklasse der NP der Optimierungsprobleme darstellt.

Definition 2.8 (Komplexitätsklasse $\mathbb{NPO}$)
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 für alle Instanzen $i \in I$ gilt:

  Für jede zulässige Lösung $x \in F(i)$ gilt $|x| \leq p(|i|)$

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

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

2.3 Güte

Damit konkrete Aussagen über die Qualität der errechneten Ergebnisse gemacht werden können, müssen diese mit der optimalen Lösung verglichen werden. Der einfachste Weg ist die Differenz. Je kleiner sie ist desto näher liegt der errechnete Wert am Optimum. Stellt man Aussagen über diese Art des Vergleichs an, so spricht man von einer absoluten Güte, da das Ergebnis im absoluten Zusammenhang mit dem Optimalen Wert steht.

Definition 2.10 (Absolute Güte)
Sei Π ein Optimierungsproblem und A ein Approximationsalgorithmus für Π.

- $A(I)$ hat eine absolute Güte 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-Güte

Gilt für alle n: $\mathcal{AG}_A^{WC}(\mathrm{n}) \leq \mathcal{AG}_A(n)$ so **garantiert** A eine absolute Güte von $\mathcal{AG}_A(n)$. Man spricht dann von einer *absoluten Gütegarantie*. Solch eine absolute Garantie ist dann sehr praktisch, wenn die Lösungen groß 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 vernachlässigen. Ein weitere Möglichkeit ist es die Güte in Relation zum optimalen Wert anzugeben. Man spricht dann von einer relativen Gütegarantie.

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

Definition 2.11 (Relative Güte)
Sei A ein Approximationsalgorithmus für das Optimierungsproblem O

- A hat eine *relative* Güte von

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

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

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

Spricht man von einer relativen Güte 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 ändert. Allerdings gibt es auch Algorithmen bei denen man sich auf Kosten der Laufzeit eine kleinere Abweichung „erkauft“. Man übergibt dem Algorithmus einen maximal erlaubten Fehler und macht die Laufzeit von jenem abhängig.

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

PTAS steht **P**olynomial-**T**ime **A**pproximation **S**cheme und enthält alle Approximationsalgorithmen die sich lediglich zur Güte polynomiell verhält..

APX Alle approximierbaren Algorithmen.

4 Beispielanalyse

Anfangs wurde das Problem JS und folgender Algorithmus dafür vorgestellt:

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

Algorithmus 2 : JOB SCHEDULE

Input : p Prozessoren,t Taks und die Laufzeiten l_i für 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;

Z 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 beträgt

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

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

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

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

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

So folgt daraus:

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

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

Literatur

[Aus99] Ausiello, Giorgio: *Complexity and approximation.* Springer, 1999.

[Cha09] Cham, Jorge: *PHD Comic: Not a good sign either.* `http://www.phdcomics.com/comics/archive.php?comicid=1160`, 4 2009.

[Gra71] Graham, RL: *Bounds on multiprocessing anomalies and related packing algorithms.* In: *Proceedings of the November 16-18, 1971, fall joint computer conference*, Seiten 205–217. ACM New York, NY, USA, 1971.

[Sch92] Schöning, Uwe: *Theoretische Informatik kurz gefasst.* B.I. Wiss. Verl., 1992.

[THC07] Thomas H Cormen, Charles E. Leiserson: *Algorithmen - Eine Einführung.* Oldenbourg Wissenschaftsverlag, 2007.

[Wan06] Wanka, R.: *Approximationsalgorithmen eine Einführung.* Teubner, 2006.