Amortisierte Analysen

Fernando Sánchez Villaamil, Michael Nett, Felix Reidl

1. Juni 2007

1 Watt, wer bist du denn?!

Es gibt viele Datenstrukturen, bei deren Komplexitätsanalyse das Problem auftaucht, dass die Ausführungen mancher Operationen Einfluss auf die Komplexität folgender Operationen haben (die selbe oder eine andere). So kann beispielsweise eine Einfüge-Operation in einem (a, b)-Baum sehr viele Split-Operationen verursachen. Dies hat den erwünschten Nebeneffekt, dass die folgenden Operationen sehr wahrscheinlich eine niedrigere Komplexität haben werden.
In einigen Fällen macht es Sinn nicht den Worstcase von einzelnen Operationen, sondern die Laufzeit von einer Folge von Operationen zu betrachten.
Ein Werkzeug für solche Fälle ist die amortisierte Analyse.

2 Vorgehensweisen

2.1 Idee

Die Idee der armortisierten Analyse ist es, zeitintensive Operationen gegen weniger zeitintensive Operationen aufzuwiegen. Unter Umständen kann dieses Aufwiegen dann in einer schärferen Laufzeitabschätzung resultieren.

2.2 Die Bankkontomethode

Die Idee hinter der *Bankkontomehode* ist sehr einfach: Bei den Operationen die nicht so viel Zeit[1] benötigen, stellen wir uns vor dass sie mehr Zeit beanspruchen würden. Die Zeit, welche wir nicht gebraucht haben, speichern wir als Rücklage auf einem *Konto*. Wenn also später eine Operation durchgeführt wird, die länger braucht, so „gönnen“ wir uns aufgesparte Zeit vom Konto.
Diese Methode ist intuitiv leicht zu fassen, nur ist sie in dieser Form mathematisch zu ungenau. Eine genauere Formulierung (und damit ein Werkzeug für den formalen Beweis!) kann mit Hilfe der Potentialfunktion geschehen.

2.3 Die Potenzialfunktionmethode

„Anscheinend waren manche Theoretiker der Meinung, dass die Bankkontomethode zu einfach ist. Vermutlich war es nicht formal genug. Vielleicht sind sie

[1] Zeit ist nicht ganz korrekt, aber unnötige Formalismen erschweren die Dinge.

auf irgendwelche Grenzen gestoßen und wollten darum ein formales Modell entwickeln. Unserer Meinung nach sollte man nichts der Intelligenz zuschreiben, was man auch der Bösartigkeit[2] *unter die Füße schieben kann."*

— Sánchez et al., 2007

Ziel dieser Methode ist es, eine sogenannte Potenzialfunktion $\Phi(i)$ zu finden, welche einer Datenstruktur zum Zeitpunkt i einen bestimmten Wert (ein Potenzial) zuordnet. Jede Operation $\{O_1, \ldots, O_k\}$ hat einen (evtl. vom Zeitpunkt abhängigen) Aufwand $\{T_1, \ldots, T_k\}$. Man versucht nun die Potenzialfunktion so zu wählen, dass man **für jede Operation** die rellen Kosten t_i, welche die Operation im i-ten Schritt verursacht, gegen eine amortisierte Schranke a_i abzuschätzen:

$$t_i + \Phi(i) - \Phi(i-1) \leq a_i \tag{1}$$

Dabei ist $\Phi(i) - \Phi(i-1)$ die *Änderung des Potenzials* der Datenstruktur.

Nachdem man für **alle möglichen** k Operationen nun amortisierte Kostenschranken $a_1, \ldots, a_k$ gefunden hat, kann man sich Gedanken über die Kosten von n gemischten Operationen machen:

Sei hierzu A die größte Schranke der a_i.

$$\sum_{i=1}^{n} (t_i + \Phi(i) - \Phi(i-1)) = \Phi(n) - \Phi(0) + \underbrace{\sum_{i=1}^{n} t_i}_{\text{Gesamtkosten}} \leq \sum_{i=1}^{n} a_i \leq n \cdot A \tag{2}$$

Diese Abschätzung gilt natürlich nur unter folgender Voraussetzung:

$$\Phi(n) - \Phi(0) \geq 0 \qquad \text{und} \qquad \Phi(0) = 0 \tag{3}$$

Denn nur dann können wir sicher sein, dass wir in der obigen Ungleichung zu den Gesamtkosten etwas positives addieren (vergleichbar dazu ist die Tatsache, dass bei der Bankkontomethode niemals ein negativer Kontostand vorkommen darf).
Es sollte jetzt klar sein, warum es mathematisch korrekt ist eine solche Abschätzung zu verwenden. Wir haben allerdings noch kein Wort darüber verloren, warum man sich soetwas überhaupt antun sollte. Was hier geschieht, ist, dass der Aufwand von komplexen Operationen auf einfachere Operationen abgewälzt wird.
Das Wichtigste ist hier, dass der je nach Operation stark schwankende Aufwand durch die amortisierten Kosten a_i abgeschätzt wird, wobei die a_i ihrerseits dann „glatter" sind.
Zentrales Problem ist natürlich die Potenzialfunktion – der Rest ist straightforward. Nur wie findet man sie? Das hängt von Fantasie, Glück und Geduld des Anwenders ab. Wir werden aber versuchen euch ein paar Tipps zu geben.

[2] Vor allem gegen Studenten

- Die Potenzialfunktion ist abhängig von i. Das bedeutet aber **nicht**, dass die Zahl i in der Funktion vorkommen muss. Das Argument i symbolisiert lediglich die wievielte Operation gerade betrachtet wird. Als Grundlage der Potenzialfunktion dienen alle Werte die sich aus dem Zeitpunkt i ableiten oder schätzen lassen, so wie z.B.: Wie viele Schritte schon gemacht wurden, wie viele Elemente eingefügt wurden, die Höhe eines Baumes, Anzahl der Rotationen seit dem letzten Einfügen, etc.

- Es ist oft nützlich, dass in der Definition der Potenzialfunktion das Wort „seit“ auftaucht, vor allem in Datenstrukturen, in denen sich bestimmte Vorgänge wiederholen.

- Es ist also prinzipiell eine gute Idee zu beobachten, wann in der Datenstruktur etwas besonderes geschieht und dieses dann durch die Potenzialfunktion mit der Anzahl der Operationen oder dem Zustand der Datenstruktur an einem bestimmten Zeitpunkt in Verbindung zu bringen.

- Die Potenzialfunktion addiert normalerweise ein Bisschen zu „schnellen“ Operationen und subtrahiert viel von „langsamen“. Wobei hier „viel“ und „Bisschen“ zu definieren ist.

- Ein Indiz für eine richtige Potenzialfunktion ist, dass die Potentialdifferenz für die aufwendigen Operationen negativ, für die günstigen positiv (oder null) wird.

3 Beispiele

3.1 T11 / Kombination Baum-Liste mit „lazy-insert“

(http://www-lti.cs.rwth-aachen.de/lehre/DA/SS2007/uebung/ueb05.pdf)

Wir schauen uns zuerst die Aufgabe T11 an. Bitte jetzt die Aufgabenstellung lesen. Versuchen wir jetzt einmal zu vergessen, dass für die Aufgabe eine amortisierte Analyse nicht wirklich nötig ist, und widmen wir uns der amortisierten Analyse[3].
Suchen wir zunächst die Potenzialfunktion. Wenn man die Datenstruktur näher betrachtet, sieht man, dass nach einer langsamen Operation (Suchen oder Löschen) die Liste wieder die Länge 0 hat. Wir sollten versuchen, dass bei diesen langsamen Operationen durch die Addition von $\Phi(i) - \Phi(i-1)$ viel abgezogen wird, um somit die amortisierten Kosten der langsamen Operationen nach unten zu drücken. Man sieht, dass die Länge der Liste zum Zeitpunkt $i-1$ (vorher) groß und zum Zeitpunkt i (nachher) wieder 0 ist, da eine solche Operation die Liste leert. Wir folgern: Die Listengröße sollte in die Potenzialfunktion mit einfließen. Versuchen wir einmal als Ansatz folgende Potenzialfunktion:

$$\Phi(i) = c \cdot l_i \cdot \log n_i \tag{4}$$

Hier ist l_i die Anzahl der Elemente in der Liste, n_i die Anzahl der Elemente in der Datenstruktur und c eine geeignete Konstante (dazu später mehr). Jetzt können

[3] Merke: ein gutes Argument spart mühsame Analysen

wir die amortisierten Kosten für das Suchen und Löschen in der Datenstruktur abschätzen:

$$\underbrace{\overbrace{\mathcal{O}(\log n)}^{\text{Suchen / Löschen}} + \overbrace{\mathcal{O}(l \cdot \log n)}^{\text{Liste} \to \text{Baum}}}_{\text{reelle Kosten}} + \underbrace{c \cdot 0 \cdot \log n}_{\Phi(i)=0} - \underbrace{c \cdot l \cdot \log(n-1)}_{\Phi(i-1)} \in \mathcal{O}(\log n) \tag{5}$$

Hier zeigt sich der Grund für die Wahl des Logarithmus in der Potenzialfunktion: Sie kann mit den reellen Kosten (in denen eben ein Logarithmus zu finden ist) verrechnet werden. Der $\mathcal{O}$-Term in der Ungleichung lässt sich mit geeigneter Wahl für c nun ebenfalls abschätzen (Das wählbare c ist eigentlich auch in der $\mathcal{O}$-Klasse versteckt, allerdings schreibt man es hier bekanntlich nicht mit hin). Das ist bisher ein sehr gutes Ergebnis. Wir haben die Operationen mit der höchsten Komplexität und einem genügend großem c durch $\mathcal{O}(\log n)$ abgeschätzt. Bleibt zu prüfen, was beim Einfügen passiert:

$$\mathcal{O}(1) + c \cdot l \cdot \log n - c \cdot (l-1) \cdot \log(n-1) \in \mathcal{O}(\log n) \tag{6}$$

Man kann also **jeden** Term mit $\mathcal{O}(\log n)$ abschätzen, wenn das c entsprechend gewählt wird. Für die Kosten von n gemischten Operationen folgt dann:

$$\underbrace{\sum_{i=1}^{n} t_i}_{\text{Gesamtkosten}} \leq \Phi(n) - \Phi(0) + \sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i \in n \cdot \mathcal{O}(\log n) \tag{7}$$

Damit ist gezeigt, dass bei der Datenstruktur n gemischte Operationen eine Komplexität von $n \cdot \mathcal{O}(\log n)$ haben und eine Operation im amortisierten Mittel eine Komplexität von $\mathcal{O}(\log n)$ besitzt.

Überlegen wir kurz, was passiert wäre, wenn wir eine andere (größere) Potenzialfunktion gewählt hätten, etwa

$$\Phi'(i) = c \cdot l_i \cdot n_i \tag{8}$$

Offentsichtlich gilt dann für das Suchen und Löschen:

$$\underbrace{\overbrace{\mathcal{O}(\log n)}^{\text{Suchen / Löschen}} + \overbrace{\mathcal{O}(l \cdot \log n)}^{\text{Liste} \to \text{Baum}}}_{\text{reelle Kosten}} + \underbrace{c \cdot 0 \cdot n}_{\Phi'(i)=0} - \underbrace{c \cdot l \cdot (n-1)}_{\Phi'(i-1)} \in \mathcal{O}(\log n) \tag{9}$$

Allerdings funktioniert diese Funktion nicht für das Einfügen:

$$\mathcal{O}(1) + c \cdot l \cdot n - c \cdot (l-1) \cdot (n-1) \in \Omega(n) \tag{10}$$

Also war diese zweite Potenzialfunktion wirklich „zu groß“

3.2 Stacks mit Multipop

Jetzt betrachten wir einen Stack mit den Operationen `Push` und `Pop`. Erweitert wird die bekannte Datenstruktur mit der Funktion `Multipop(k)`, welche die obersten k Elemente vom Stack entfernt, also prinzipiell so wirkt wie ein k-maliger Aufruf von `Pop`.

In der amortisierten Analyse werden wir die $\mathcal{O}$-Notation nicht benutzen und Annehmen, dass „in konstanter Zeit" gleichbedeutend mit Kosten von 1 ist (Das erspart es uns, wie im ersten Beispiel eine Konstante c einzuführen). Wobei wir annehmen, dass der Stack zu Beginn leer ist.
Die langsame (und somit teure) Operation ist in dieser Datenstruktur das `Multipop`, also wollen wir überlegen, wie wir die amortisierten Kosten dafür reduzieren können. Wir sehen direkt, dass sich die Anzahl der Elemente bei einem `Multipop` im Vergleich zu den anderen Operationen stark verringert. Demzufolge sollten wir als Ansatz für die Potenzialfunktion die Anzahl der Elemente auf dem Stack wählen, also[4]:

$$\Phi(i) = k, \text{wobei } k \text{ die Anzahl der Elemente auf dem Stack ist.} \tag{11}$$

Probieren wir mal eine amortisierte Schranke zu finden, falls `Push` die Operation im i-ten Schritt ist:

$$\underbrace{t_{\texttt{push}}}_{=1} + \underbrace{\Phi(i)}_{=k} - \underbrace{\Phi(i-1)}_{=k-1} = 1 + k - (k-1) = 2 \leq 2 \tag{12}$$

Die amortisierten Kosten für ein `Push` sind also 2. Jetzt widmen wir uns dem Fall, das `Pop` die i-te Operation ist:

$$\underbrace{t_{\texttt{pop}}}_{=1} + \underbrace{\Phi(i)}_{=k-1} - \underbrace{\Phi(i-1)}_{=k} = 1 + k - 1 - k = 0 \leq 0 \tag{13}$$

Wichtig ist zu beobachten, dass sich die Anzahl der Elemente k auf dem Stack mit einem `Push` vergrößert und mit einem `Pop` verkleinert!
Die letzte Operation die wir uns anschauen müssen ist das `Multipop(l)`. Hier werden l Elemente vom Stack entfernt. Falls sich weniger Elemente auf dem Stack befinden ($k \leq l$) als gelöscht werden sollen, terminiert die Funktion wenn der Stack leer ist, also nach k Schritten. Da im Zweifelsfall also der Aufwand für `Multipop(l)` geringer wird, können wir davon ausgehen das genug Elemente auf dem Stack sind (wir suchen ja schließlich nach einem Worstcase). Überprüfen wir also die amortisierte Schranke für `Multipop(l)`:

$$\underbrace{t_{\texttt{multipop(l)}}}_{=l} + \underbrace{\Phi(i)}_{=k-l} - \underbrace{\Phi(i-1)}_{=k} = l + k - l - k = 0 \leq 0 \tag{14}$$

Jetzt wissen wir, dass wir **alle möglichen Operationen** zu **jedem möglichen Zeitpunkt** durch die amortisierte obere Schranke $a_i \leq 2$ abschätzen können. Also gilt für eine Folge von n gemischten Operationen:

[4]Klar ist, dass wir k im i-ten Schritt durch i abschätzen könnten. Aber wir tun einfach erstmal so, als würden wir wissen wieviele Elemente auf dem Stack sind und schauen wie weit wir damit kommen.

$$\underbrace{\sum_{i=1}^{n} t_i}_{\text{Gesamtkosten}} \leq \underbrace{\Phi(n) - \Phi(0)}_{\geq 0} + \sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i \leq \sum_{i=1}^{n} 2 = 2n \tag{15}$$

Wichtig ist wie immer, dass $\Phi(0) = 0$ ist[5] und zusätzlich für alle $n \in \mathbb{N}$ gilt $\Phi(n) \geq \Phi(0)$, denn sonst würde die letzte Abschätzung nicht funktionieren. Wir haben also mit Hilfe einer amortisierten Analyse gezeigt, dass eine Operation auf einem solchen Stack durch die mittleren Kosten in Höhe von 2 beschränkt ist.

3.3 Sporadisch balancierter Baum

Jetzt betrachten einen regulären binären Suchbaum. Dieser Suchbaum unterstützt die Operationen `Search`, `Insert` und `Remove`. Im Gegensatz zu einem normalen Suchbaum, wird dieser Baum balanciert, falls er zu stark unausgeglichen ist (dazu gleich mehr). Die Balancierung wird durchgeführt, indem alle Elemente im Baum in ein Array geschrieben werden. Uns ist klar, das diese Aktion in $\mathcal{O}(n)$ Schritten abläuft, wobei n die Anzahl der Elemente im Baum ist. Im Anschluss wird das (bereits sortierte) Array wieder in einen Suchbaum verteilt[6]. Diese Aktion benötigt ebenfalls $\mathcal{O}(n)$ Schritte und endet in einem Suchbaum der optimal bezüglich der Höhe ist. Wir sehen, dass die gesamte Rebalancierung ebenfalls in $\mathcal{O}(n)$ Schritten beendet ist.
Wir wissen schon, dass der Baum rebalanciert wird, falls er unzureichend balanciert ist. Aber wie sieht das Kriterium dafür genau aus? Wir definieren:

$$\alpha := \frac{\lceil \log n \rceil}{h_T}, \alpha \in (0, 1] \tag{16}$$

Eine erste Betrachtung dieses Balancefaktors α zeigt uns, dass ein höhenoptimaler Baum $\alpha = 1$ hat. Je mehr die Höhe des Baumes degeneriert, desto geringer wird der Balancefaktor. An einem Beispiel für $n = 2$ Elemente sehen wir auch, dass eine Rebalancierung nicht $\alpha = 1$ sondern nur $\alpha \geq \frac{1}{2}$ garantiert.
Es ist also gut, dass die Datenstruktur als Grenze für die Balancierung genau $\alpha = \frac{1}{2}$ hat. Genauer wird nach dem Löschen oder Einfügen das α bestimmt[7]. Sollte es unter $\frac{1}{2}$ liegen, so wird der Baum automatisch nach obiger Methode neu aufgebaut und balanciert.
Beginnen wir also die Analyse! Es ist klar, dass zu jedem Zeitpunkt[8] gilt $\alpha \geq \frac{1}{2}$. Betrachten wir zuerst die Operation `Search`. Wie lange eine Suche dauert hängt davon ab, wie hoch der Baum ist. Der garantierte Balancefaktor lässt uns aber jetzt die Höhe abschätzen:

$$\alpha = \frac{\lceil \log n \rceil}{h} \geq \frac{1}{2} \Rightarrow h \leq 2 \cdot \lceil \log n \rceil \in \mathcal{O}(\log n) \tag{17}$$

[5] Wir gehen davon aus, dass wir mit einem leeren Stack beginnen.
[6] Hierzu greift man sich das mittlere Arrayelement heraus und fügt es als neue Wurzel ein. Der linke Rest L und rechte Rest R des Arrays wird benutzt um auf die gleiche Art den linken und rechten Unterbaum der Wurzel zu konstruieren.
[7] Die Höhe wird beim Einfügen aufgeschrieben, da wir ja nur an Blättern einfügen und die Anzahl der Knoten n wird generell bei jedem Einfügen oder Löschen aktualisiert!
[8] Wäre das nicht der Fall, so hätte eine Rebalancierung den gewünschten Zustand automatisch wieder hergestellt.

Nächster Punkt sind die Operationen `Insert` und `Remove`. Im schlimsten Fall, wird eine solche Aktion von einer Rebalancierung begleitet, was für uns linearen Aufwand mit $\mathcal{O}(n)$ bedeutet. Irgendwie müssen wir also eine Potenzialfunktion finden, mit der wir hier kräftig etwas abziehen können. Einfach nur die Anzahl der Knoten n zu verwenden wäre unklug, da sich diese bei den Aktionen vorher und nachher jeweils nur um 1 unterscheidet, und dass reicht uns nicht[9]. Was sich allerdings sehr stark ändert, ist der Balancefaktor α[10]. Da wir eine gute amortisierte Schranke a_i der Form

$$\mathcal{O}(n) + \Phi(i) - \Phi(i-1) \leq a_i \tag{18}$$

suchen, könnten wir probieren die Potenzialfunktion von n, α und einer gewählten Konstanten c (damit wir mit den $\mathcal{O}$-Klassen zurecht kommen) abhängig machen. Unsere wahl fällt auf:

$$\Phi(i) := \frac{c \cdot n}{\alpha_i} \tag{19}$$

Wobei wir die Anzahl der Elemente mit n abschätzen, da wir ja in den betrachteten Operationen maximal n Elemente in den Baum stopfen können. Mit dieser Funktion suchen wir eine amortisierte Schranke für das Einfügen und Löschen aus der Datenstruktur:

$$\mathcal{O}(n) + \underbrace{\alpha_i \cdot c \cdot n}_{\alpha_i = 1} - \underbrace{\alpha_{i-1} \cdot c \cdot (n-1)}_{\alpha_{i-1} < \frac{1}{2}} \leq \mathcal{O}(n) + c \cdot n - 2 \cdot c \cdot (n-1) \tag{20}$$

$$= \mathcal{O}(n) + 2 \cdot c - c \cdot n \leq 0 \overset{(c>0)}{\Rightarrow} \mathcal{O}(n) - c \cdot n \leq 0 \tag{21}$$

Da eine Suche die Datensturktur nicht verändert, ist uns klar, dass eine Suchoperation im i-ten Schritt eine amortisierte Schranke von

$$\mathcal{O}(2 \log n) + \underbrace{\Phi(i) - \Phi(i-1)}_{=0} \leq \mathcal{O}(2 \log n) \tag{22}$$

hat. Die Schranke der Suchoperation stellt für ein geeignet gewähltes c also eine Schranke für jede Operation dar. Demzufolge ergibt sich für n gemischte Operationen auf eine anfangs leere Datenstruktur:

$$\Phi(n) - \Phi(i) + \sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} \mathcal{O}(2 \log n) = n \cdot \mathcal{O}(\log n) \tag{23}$$

Was haben wir nun gezeigt? Im amortisierten Mittel benötigt eine Operation auf dieser Datenstruktur nur $\mathcal{O}(\log n)$ Schritte. Feine Sache, oder?

[9] Die Differenz von $\Phi(i) - \Phi(i-1)$ würde hier relativ kleine Werte liefern.

[10] „Stark“ klingt vielleicht komisch, aber man muss sich im klaren darüber sein, das der Wert von α nicht viel Spiel hat, nämlich nur in $(0, 1]$ liegen kann!