Amortisierte Analysen

26. Mai 2016

1 Einleitung

Es gibt viele Datenstrukturen, bei deren Komplexitätsanalyse das Problem auftaucht, dass die Ausführung mancher Operationen Einfluss auf die Komplexität der folgenden Operationen hat (die gleiche oder eine andere). Eine Einfüge-Operation in einem (a, b)-Baum kann zum Beispiel sehr viele Split-Operationen verursachen, dies hat den erwünschten Nebeneffekt, dass die folgenden Operationen sehr wahrscheinlich eine niedrigere Komplexität haben werden.

In einigen Fällen ist es Sinnvoll 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 Amortisierte Analyse

Die Idee der amortisierten Analyse ist es, zeitintensive Operationen gegen weniger zeitintensive Operationen aufzuwiegen. Unter Umständen kann dieses Aufwiegen dann in eine schärfere Laufzeitabschätzung resultieren.

2.1 Die Bankkontomethode

Die Idee hinter der *Bankkontomehode* ist sehr einfach: Bei den Operationen die nicht so viel Zeit 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. Dies ist Zeit die wir für die Komplexität berechnet aber nicht wirklich gebraucht haben.

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.

2.2 Die Potenzialfunktionmethode

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 reellen Kosten t_i, welche die

Operation im i-ten Schritt verursacht, gegen eine amortisierte Schranke a_i abzuschätzen:

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

Dabei ist $\Phi(i) - \Phi(i-1)$ die *Änderung des Potenzials* der Datenstruktur: $\Phi(i-1)$ ist der Kontostand vor und $\Phi(i)$ nach dem jetzigen Schritt. Wenn also $\Phi(i) - \Phi(i-1)$ negativ ist, wurde etwas aus dem Konto genommen, sonst wurde zum Konto hizugefügt.

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 das maximum von $a_1, \ldots, a_n$.

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

Diese Abschätzung gilt natürlich nur wenn $\Phi(0) = 0$ und $\Phi(n) - \Phi(0) \geq 0$: Am Anfang muss das Konto leer sein da wir nur was auf dem Konto hinzufügen dürfen wenn dadurch eine Operation teurer wird. Am Ende darf der Kontostand nicht negativ sein, da wir nicht mehr als wir hinzugefügt haben abziehen dürfen.

Was hier geschieht ist, dass der Aufwand von komplexen Operationen auf einfachere Operationen abgewälzt wird. Wir addieren also zu den schnellen Operationen etwas hinzu während wir von den langsamen etwas subtrahieren mit dem Ziel zu einer glatteren Schranke zu kommen. Mit anderen Worten wird der je nach Operation stark schwankende Aufwand durch die amortisierten Kosten a_i abgeschätzt.

Zentrales Problem ist natürlich die Potenzialfunktion. Nur wie findet man sie? Ein paar Tipps:

- 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 Beschreibung 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 Binär Baum mit lazy-insert

Wir betrachten die folgende Datenstruktur: Einen binären Baum und eine Queue. Anstatt die Elemente in den Baum direkt einzufügen, fügen wir sie (in konstanter Zeit) in die Queue ein. Suchen und löschen führen wir auf dem Baum aus, dabei müssen wir allerdings die Elemente in der Queue beachten. Sollte diese nicht leer sein Fügen wir jedes element aus der Queue in den Baum ein. Der letzte Schritt hat eine Worstcase Laufzeit von $O(n \cdot \log n)$, somit hätten n beliebige Operationen eine Worstcase Laufzeit von $O(n^2 \cdot \log n)$. Da dieser teure Fall aber nicht oft auftreten kann wollen wir uns der amortisierten Analyse bedienen.

Suchen wir zunächst die Potenzialfunktion. Wenn man die Datenstruktur näher betrachtet, sieht man, dass nach einer langsamen Operation (Suchen oder Löschen) die Queue 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 Queue zum Zeitpunkt $i-1$ (vorher) groß und zum Zeitpunkt i (nachher) wieder 0 ist, da eine solche Operation die Queue leert. Wir folgern: Die größe der Queue sollte in die Potenzialfunktion mit einfließen. Versuchen wir einmal als Ansatz folgende Potenzialfunktion:

$$\Phi(i) = c \cdot l_i \cdot \log n_i$$

Hier ist l_i die Länge der Queue, n_i die Anzahl der Elemente in der Datenstruktur und c eine geeignete Konstante (diese dient als Hilfe um Arithmetik in der O-Notation betreiben zu können). Jetzt können wir die amortisierten Kosten für das Suchen und Löschen in der Datenstruktur abschätzen:

$$\underbrace{\overbrace{O(\log n_i)}^{\text{Suchen / Löschen}} + \overbrace{O(l_{i-1} \cdot \log n_i)}^{\text{Queue} \to \text{Baum}}}_{\text{reelle Kosten}} + \underbrace{c \cdot 0 \cdot \log n_i}_{\Phi(i)=0} - \underbrace{c \cdot l_{i-1} \cdot \log(n_{i-1})}_{\Phi(i-1)} = O(\log n_i)$$

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 O-Term in der Ungleichung lässt sich mit geeigneter Wahl für c nun ebenfalls abschätzen (Das wählbare c ist eigentlich auch in der 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 $O(\log n)$ abgeschätzt. Bleibt zu prüfen, was beim Einfügen passiert:

$$\underbrace{\overbrace{O(1)}^{\text{Einfügen}}}_{\text{reelle Kosten}} + \underbrace{c \cdot l_i \cdot \log n_i}_{\Phi(i)} - \underbrace{c \cdot l_{i-1} \cdot \log(n_i - 1)}_{\Phi(i-1)} = O(\log n_i)$$

Da $n_i \leq n$ gilt kann man **jeden** Term mit $O(\log n)$ abschätzen, wenn das c entsprechend gewählt wird. Für die Kosten von n gemischten Operationen folgt dann:

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

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

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

$$\Phi(i) = c \cdot l_i \cdot n_i$$

Dann gilt für das Suchen und Löschen:

$$\underbrace{\overbrace{O(\log n_i)}^{\text{Suchen / Löschen}} + \overbrace{O(l_{i-1} \cdot \log n_i)}^{\text{Queue} \to \text{Baum}}}_{\text{reelle Kosten}} + \underbrace{c \cdot 0 \cdot n_i}_{\Phi(i)=0} - \underbrace{c \cdot l_{i-1} \cdot n_{i-1}}_{\Phi(i-1)} < 0$$

Allerdings funktioniert diese Funktion nicht für das Einfügen:

$$O(1) + c \cdot l \cdot n_i - c \cdot (l_i - 1) \cdot (n_i - 1) = \Omega(n_i)$$

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`.

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:

$$\Phi(i) = c \cdot s_i$$

Wobei s_i die größe des Stacks zum Zeitpunkt i ist und c eine geeignete Konstante. Wir definieren uns $t_{\texttt{push}}$, $t_{\texttt{pop}}$ und $t_{\texttt{multipop(k)}}$ als die reellen Kosten von den Operationen `Push`, `Pop` und `Multipop(k)`. Probieren wir mal eine amortisierte Schranke zu finden, falls `Push` die Operation im i-ten Schritt ist:

$$t_{\texttt{push}} + \underbrace{\Phi(i)}_{=c \cdot s_i} - \underbrace{\Phi(i-1)}_{=c \cdot (s_i - 1)} = O(1) + c \cdot s_i - c \cdot (s_i - 1) = O(1)$$

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

$$t_{\texttt{pop}} + \underbrace{\Phi(i)}_{=c \cdot s_i} - \underbrace{\Phi(i-1)}_{=c \cdot (s_i + 1)} = O(1) + c \cdot s_i - c \cdot (s_i + 1) \leq 0$$

Wichtig ist zu beobachten, dass sich die Anzahl der Elemente s_i auf dem Stack mit einem `Push` vergrößert und mit einem `Pop` verkleinert!

Die letzte Operation die wir uns anschauen müssen ist das `Multipop(k)`. Hier werden k Elemente vom Stack entfernt. Falls sich weniger Elemente auf dem Stack befinden ($s_i \leq k$) als gelöscht werden sollen, terminiert die Funktion wenn der Stack leer ist, also nach s_i Schritten. Da im Zweifelsfall also der Aufwand für `Multipop(k)` 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(k)`:

$$t_{\texttt{multipop(k)}} + \underbrace{\Phi(i)}_{=c \cdot s_i} - \underbrace{\Phi(i-1)}_{=c \cdot (s_i + k)} = O(k) + c \cdot s_i - c \cdot (s_i + k) \leq 0$$

Jetzt wissen wir, dass wir alle möglichen Operationen zu jedem möglichen Zeitpunkt durch die amortisierte obere Schranke $a_i \leq O(1)$ abschätzen können. Also gilt für eine Folge von n gemischten Operationen:

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

Wichtig ist wie immer, dass $\Phi(0) = 0$ ist[1] 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 $O(1)$ beschränkt ist.

[1] Wir gehen davon aus, dass wir mit einem leeren Stack beginnen.