Max Wisniewski , Alexander Steen

Tutor: Lena Schlipf

Aufgabe 1 Caching

(a) Zeigen Sie, dass jede Offline-Caching-Strategie durch eine *reduzierte* Ersetzungsstrategie ersetzt werden kann, die nicht die Anzahl der Hauptspeicherzugriffe erhöht.

Beweis:

Sie $D = d_1...d_n$ ein nicht reduzierte Zugriffsfolge. Wir konstruieren schrittweise aus D eine *reduzierte* Zugriffsfolge D'. Sei D' zu Beginn D.

Sei j der größte Index, bis zu dem D' eine reduzierte Teilfolge ist. Nach den j Zugriffen ist der Cache bei beiden Folgen gleich und D' hat höchstens so viele Cachemisses, wie D.

Betrachten wir nun den nächsten Zugriff d_{j+1} aus D.

Fall 1: d_{j+1} passiert genau dann, wenn der Zugriff auf den Wert passiert. Dies ist nach Konstruktion von j nicht möglich.

Fall 2: d_{j+1} passiert bevor wir den Wert brauchen und wir schmeißen keinen Wert aus dem Cache, den wir bis zum Zugriff auf den Wert der bei d_{j+1} geladen wird benötigen. Dann können wir den Zugriff einfach nach hinten verschieben. Die Anzahl der Hauptspeicherzugriffe bleibt gleich.

Fall 3: d_{j+1} lädt einen Wert, der nie gelesen wird in der Zukunft. Diesen Zugriff können wir Fallen lassen. Um die Konstruktion richtig zu gestallten fügen wir ein NOP ein um die Nummerierung bei zu behalten. Die Anzahl der Hauptspeicherzugriffe wird um 1 kleiner.

Fall 4: d_{j+1} macht einen Zugriff, der einen Wert liest, der in der Zukunft benötigt wird, dafür aber einen Wert aus dem Cache schmeißt, den wir auf dem Weg noch brauchen. Wir können d_{j+1} nach hinten schieben und den Zugriff auf den ersetzen Wert (der passieren muss, spätestens bei dem Zugriff der orginal benötigten Folge) durch ein NOP ersetzen, da dieser Wert noch im Cache steht. Die Anzahl der Hauptspeicherzugriffe wird um 1 kleiner.

Jeder der auftretenden Fälle sorgt dafür, das wir gleich oder weniger Hauptspeicherzugriffe haben. Dies beduetet für uns, dass, sobald kein j mehr gefunden werden kann, dass D' eine reduzierte Strategie ist, die maximal so viele Hauptspeicherzugriffe hat wie D.

(b) Geben Sie eine allgemeine Zugriffsfolge an, so dass die LRU Strategie bei p Wörtern und einem Cache der Größe k ($p < k$), k mal so viele Misses wie die Furthest-in-the-Future Strategie erzeugt.

Lösung:

Wir konstruieren eine Folge von Werten, die wir lesen wollen, mit $Z = (1...p)^{s+1}$, wobei eine Zahl i für das i-te Wort steht und $s+1$ für die Anzahl, wie oft diese Folge wiederholt wird. Der Einfachkeit halber gelte $p = 2 \cdot k$.

Die ersten k Werte sollen bei beiden Strategien schon im Cache stehen.

Bei *Furthest-in-the-future* gilt:
Es stehen $(1..k)$ im Cache. Lesen wir nun $k+1$, werden wir k aus dem Cache schmeißen, da wir es als letztes wieder lesen. Dies zieht sich bei jedem Wert bis zu p weiter durch. Bei der ersten Folge $(1...p)$ haben wir $p-k$ Misses. Bei jeder weiteren, müssen wir auch k in den Speicher laden und danach wie beim ersten mal durch gehen. Dies führt zu $p - k + 1$ Misses.

Insgesamt haben wir $miss_{ff} = p - k + s \cdot (p - k + 1)$

Bei *LRU* gilt:
Wenn wir $k+1$ lesen, werfen wir die 1 aus dem Cache. Lesen wir $k+2$ werfen wir die 2 aus dem Speicher. Wir shiften die Wörter also immer durch den Cache. Da wir, wenn wir bei $p + 1$ ankommen, keinen der ursprünglichen Werte im Cache haben, müssen wir diese Werte alle wieder lesen.
Insgesammt haben wir, bis auf die ersten k Zugriffe nur Cachemisses.

Isgesammt haben wir $miss_{LRU} = (s + 1)p - k$

Untersuchung:
Wir sollten zeigen, dass $miss_{LRU} \geq k \cdot miss_{ff}$ gilt. Setzen wir hier einmal unsere Formeln ein.

$$\begin{aligned}
& & miss_{LRU} &\geq k \cdot miss_{ff} \\
\Leftrightarrow & & (s+1)p - k &\geq k \cdot (p - k + s \cdot (p - k + 1) \\
\Leftrightarrow & & sp + p - k &\geq kp - k^2 + ks \cdot (p - k + 1) \\
\Leftrightarrow & sp + p - k - kp + k^2 - ks \cdot (p - k + 1) &\geq 0 \\
\Leftrightarrow & k^2(1 + s) - k(2 + p + ps) + (s + 1)p &\geq 0
\end{aligned}$$

An dieser Stelle haben wir eine Quadratische Funktion in k. Diese ist wächst gegen $\pm\infty$ gegen $infty$. Also sollte rechts von der größeren der beiden Nullstellen die Gleichung immer erfüllt sein. Wie wir an dieser Stelle schon sehen können, kann man diese p und s immer konstruieren, wir können diese aber noch Versuchen näher zu bestimmen und zu zeigen, dass sie immer existieren müssen.
Nach p-q-Formel gilt:

$$\begin{aligned} n_{1/2} &= \frac{2+p+ps}{1+s} \pm \sqrt{\left(\frac{2+p+ps}{1+s}\right)^2 - \frac{s+1}{1+s}} \\ &= p + \frac{2}{1+s} \pm \sqrt{\left(p + \frac{2}{1+s}\right)^2 - 1} \end{aligned}$$

Wenn wir nun s groß genug wählen (beispielsweise gegen unendlich streben lassen), erhalten wir die Formel:
$n_{1/2} = p \pm \sqrt{p^2 - 1}$

Daraus können wir den ungefähren Wert ermitteln, dass bei $p \geq 2k$ die Anzahl der Misses immer k mal so groß ist bei LRU, wie bei Furthest-in-the-Future.

□

Aufgabe 2 Union-Find

Betrachten Sie eine Folge von **Union** und **Find** Operationen der Länge m. Gegeben ist die Startpartition $\{\{1\}, \{2\}, ..., \{n\}\}$. Verwendet werden sowohl Union-By-Rank also auch Pfadkompression.

(a) Werden alle **Union**-Operationen vor allen **Find** - Operationen durchgeführt, so ist die Gesamtlaufzeit $O(n + m)$.

Beweis:

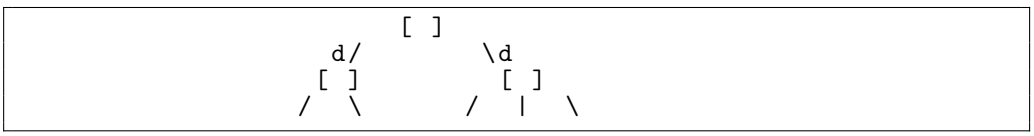
Wir verwenden hier die Buchhaltermethode. Diese kann zur Berechnung von amortisierten Kosten verwendet werden, indem die akkumulierten Kosten einer bestimmten Funktion auf viele einzelne andere aufgeteilt werden.

Die Kosten für Union sind mit $T_U(n) = O(1)$ bezeichnet. Wir setzen die konstante auf c, sodass $T_U(n) = c$.

Als nächsten nehmen wir die Kosten für das Umhängen einer einzelnen Kante im Wald. Wir biegen hier nur einen Pointer um, daher sind die Kosten auch konstant berechnet. Daher ergibt sich $T_H(n) = O(1)$ und wir wollen dies mit der konstanten d beschreiben, sodass $T_H(n) = d$.

Nach der Bankiersmethode sagen wir nun, dass Union die Kosten für das Umhängen mittragen soll : $T'_U(n) = c + d = O(1)$. Die asymptotische Laufzeit hat sich ncht geändert, aber sie tragen nun die kosten für ein Umhängen mit.

Nachdem alle Unions ausgeführt wurden, haben wir einen Baum der folgenderMaßen aussieht:



Wobei d an jeder Kante steht und für die gespeicherten Kosten einer Unionoperation steht. Wird nun ein *Find* ausgeführt so müssen wir einen solchen Pfad nach oben Laufen und für die Länge des Pfades je einen Knoten umhängen. Wenn wir einen

Pfad der Länge $k+1$ haben, müssen wir k Knoten Umhängen. Dies würde uns $d \cdot k$ Kosten verursachen.

Da aber jeder Knoten, den wir Umhängen müssen, durch ein Union einmal vereinigt worden sein musste, besitzt dieser an seiner Kante, die ihn mit seinem Vater verbindet die gespeicherten Kosten d. Dies gilt für alle Knoten, bis auf die direkt unter einer Wurzel, da diese schon umgehangen worden sein können. Diese müssen aber nicht erneut umgehangen werden.

Bei einem Find braucht also jeder Knoten, der umgehangen wird, die Kosten die das Union auf ihm gespart hat auf. Somit verursacht das Find keine Kosten um den Weg nach oben zu gehen, da diese Kosten nun alle schon vom Union bezahlt worden sind.
$\Rightarrow T_F(n) = O(1)$.
Alle diese überlegungen beziehen sich auf eine Operation der Folge von Operationen, wenn wir die gesamtkosten durch die Anzahl der Kosten teilen.

□

Dieser Vorteil würde verloren gehen, wenn wir ein nach einem Find, Union zulassen, da wir nun eine nicht Wurzelkante haben, die keine gespeicherten Kosten hat. Wir könnten also wieder Pfade erzeugen, deren Kosten noch nicht bezahlt sind.

(b) Wenn alle **Find** - Operationen auf Mengen mit mindestens $\log\ n$ Elementen durchgeführt werden, so ist die Gesamtlaufzeit $O(n+m)$.

Beweis:

Tipp: Im $\log * n$ Teil steht es. Hauptlemma + andere Aufteilung (nicht der Rang)

Aufgabe 3 Bitmaps

Sie n eine natürtliche Zahl. Eine $n \times n$ Bitmap ist ein Array $B[1...n, 1...n]$ vom Typ `Boolean`, welches ein Schwarz-Weiß-Bild darstellt.

(a) Entwerfen und analysieren Sie einen effizienten Algorithmus, der eine größte Zusammenhangskomponente von schwarzen Pixeln in B bestimt.

Lösung: Wir

(b) Beschreiben und analysieren Sie eine Funktion schwärze(i,j), die das Pixel an der Stelle $B[i,j]$ schwarz färbt und die Größe einer größten Zusammenhangskomponente von schwarzen Pixeln zurückgibt. Nehmen Sie an, dass zu Beginn alle Pixel weiß sind. Die Gesamtlaufzeit für jede Folge von m Aufrufen von schwärze sollte so gering wie möglich sein.

Lösung:

(c) Was ist die worst-case Laufzeit für einen Aufruf Ihrer Funktion schwärze aus (b)?

Lösung:

Aufgabe 4 Matroide

Sei S eine endliche, nichtleere Menge und sei $\mathfrak{I} \subseteq 2^S$ eine nichtleere Menge von Teilmengen von S. Das Paar $M = (S, \mathfrak{I})$ soll ein Matriod, nach Definition aus der Aufgabe sein.

(a) Eine inklusionsmaximale unabhängige Menge heißt *Basis* von M. Zeigen Sie, dass alle Basen von M die gleiche Anzahl von Elementen haben.

Beweis: (Widerspruch)
Angenommen A, B sind inklusionsmaximale unabhängige Mengen von M, mit o.B.d.A. $|A| < |B|$.
Dann wissen wir, dass $B \setminus A \neq \emptyset$ ist, die mindestens die überzähligen im Schnitt liegen und wenigestens ein weiteres Element, da A sonst nicht inklusionsmaximal wäre.

Nun können wir nach Austauscheigenschaft $x \in B \setminus A$ nehmen und $A \cup \{x\}$ bilden, so dass $A \cup \{x\} \in \mathfrak{I}$ sein muss. Nun ist aber A nicht inklusionsmaximal, da $A \subset A \cup \{x\}$ ist.

□

(b) Sei $w : S \longrightarrow \mathbb{R}^+$ eine Gewichtsfunktion. Gesucht ist eine Basis von von M mit maximalem Gewicht, wobei das Gewicht einer Teilmenge die Summer der Einzelgewichte ist. Der Algorithmus funktioniert wie folgt:

- Sortiere S absteigend.
- Setzt $B := \emptyset$.
- Gehe S Elementweise durch (nach Ordnung.
  Füge ein $x \in S$ zu B hinzu, wenn B dadurch unabhängig bleibt.
- Gib B zurück.

Zeigen Sie, dass der gierige Algorithmus eine Basis von maximalem Gewicht bestimmt. Was können Sie zur Laufzeit sagen?

Korrektheit:

bla

Laufzeit:

blub