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:

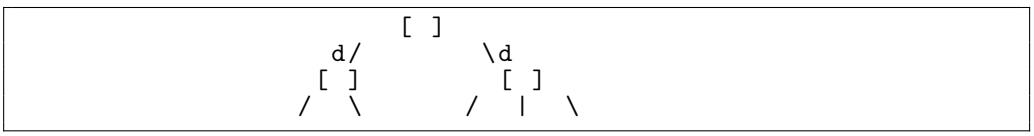
Wir verwenden hier die Buchhaltermethode. Diese kann zur Berechnung von amortisierten Kosten verwendet werden, indem die akkumulierten Kosten einer bestimmten Funktion auf viele einzelne andere aufgeteilt werden.

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 werden, wie in *b)* einen Algorithmus angeben, der Union-Find benutzt.
Die allgemeine Idee ist es, das Array von links oben nach rechts unten zu durchwandern und uns dabei einen neuen Knoten, wenn er schwarz ist, mit dem linken und dem oberen Nachbarn zu vereinigen, wenn diese auch schwarz sind. Damit wir schnell auf die Elemente im Baum zugreifen können, führt jeder Baum im Wald der disjunkten Mengen eine LinkedList, die alle Elemente speichert. Bei der Liste müssen wir nur jeweils 2 oder 4 (bei einer doppelt verketteten Liste) Pointer umgebogen werden.
Der Algorithmus sieht wie folgt aus, wir nehmen an, dass die Felder initial als einelementige Mengen in der UF-Strucktur enthalten sind:

```
maxSize := 0; //groesse der Komponente
max := null; // Root der groessten Komponente
Union := init(B);   // Macht aus jedem Feld eine Menge
for i := 1 to n do
  for j:= 1 to n do
    if B[i,j] == schwarz do
      if i>1 && B[i-1,j] == schwarz do
        // Da wir B[i,j] erstes mal betrachten, ist es nicht
            gemerged.
        Union.union(Union.find(B[i-1,j],B[i,j]);
        if maxSize < Union.getList(Union.find(B[i-1,j])).
           size() do
          maxSize := Union.getList(Union.find(B[i,j])).size
             ();
          max := Union.find(B[i,j]);
      if( j > 1 && B[i,j-1] == schwarz do
        // B[i,j] koennte gemerged sein
        Union.union(Union.find(B[i,j-1],Union.find(B[i,j]));
        if maxSize < Union.getList(Union.find(B[i,j-1])).
           size() do
          maxSize := Union.getList(Union.find(B[i,j])).size
             ();
          max := Union.find(B[i,j]);
return Union.getList(max);
```

Analyse:
Für den Speicherplatz brauchen wir zunächst einmal das Array von Pixeln, die Union-Find-Strucktur, und die Listen der Komponenten, die vereint sind.
Alle diese Struckturen haben einen Speicherplatzverbauch ovn $O(n^2)$. Das Array sollte klar sein. Die UF-Strucktur hat für jedes Feld einen Knoten und noch konstant viele Pointer. Die Zusammenhangskomponenten bestehen aus einer LinkedList, in der aber jedes Element nur einmal vorkommt und seine Pointer in die Berechnung mit übernehmen kann.
Insgesammt macht das einen Platzbedarf von $O(n^2)$.

Bei der Laufzeit müssen wir einmal über das Array iterieren. Pro Schritt führen wir maximal 2 mal **union** aus, das konstante Laufzeit hat. Da wir in bei jedem Schirtt, wenn wir vereinigen, das find auf beiden Elementen aufgerufen haben, muss ein betrachtetes Element immer unmittelbar unter der Wurzel stehen. Daher sind auch die Find Operationen konstant zu betrachten.
Zuletzt folgt noch das Zusammenlegen der Komponenten. Da dies aber wie erwähnt nur ien merge von 2 LinkedList ist, kann dies auch konstant betrachtet werden, da nur Start und Endpointer der Strucktur umgebogen werden müssen.
Insgesammt haben wir für den Algorithmus eine Laufzeit von $O(n^2)$

Dieser Algorithmus weist zusammengefasst genau die Strucktur auf, die Herr Mulzer in der Vorlesung schon erklärt hat, um Zusammenhangskomponenten in Graphen zu finden, erklärt hat.

(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:
Unsere Grundidee bleibt die selbe. Die Erweiterung mit den Listen der Größten

Zusammenhangskomponente bleibt gleich. Abgesehen davon speichern wir uns diesmal *max* und *maxSize* persistent, damit diese nach einem Funktionsaufruf erhalten bleiben. Beide sind zu Begin wie in *a)* initialisiert.

```
blacken(i,j) IS
for x := -1 to 1 do
  for y := -1 to 1 do
    if x+i < 1 || x + i >n || j+y < 1 || j + y > n || abs(x)
        + abs(y) /= 1 do
      continue;
    // Wir sind nicht uebers Feld hinweg und direckt unter
       oder neben dem Feld
    if B[i+x , j + y] == schwarz do
      Union.union(Union.find(B[i + x,j - y],Union.find(B[i,j
         ]));
      if maxSize < Union.getList(Union.find(B[i + x,j + y]))
         .size() do
        maxSize := Union.getList(Union.find(B[i,j])).size();
        max := Union.find(B[i,j]);
return Union.getList(max);
```

Analyse:
Den Speicher können wir wie eben berechnen. Dieser gilt nun nicht für einen Aufruf, weil er über längere Zeit gehalten werden muss, aber über die komplette dauer, die eine Algorithmus darauf ausgeführt wird, ist der Speicherplatzbedarf wiederum $O(n^2)$.

Da wir auf die Laufzeit schon in *b)* eingehen noch etwas zu Korrektheit.

Wenn wir einen Knoten schwarz färben, d.h. wir gehen davon aus, dass er davor nicht schwarz war, gehen wir alle seine Nachbern durch und betrachten, ob diese schon schwarz sind. Sollten sie es sein, vereinigen wir sie in der UF-Strucktur. Dies hat genau den gewünschten Effekt für uns, da diese Mengen danach vereinigt sind. Nun vegleichen wir bei jedem vereinigen, ob wir eine neue größte Menge erreichen. Sollte die neu vereinigte Menge größer als die gespeicherte sein, speichern wir diese. Am Ende geben wir die Liste der Elemente aus, die in der Menge vom Maximum liegen.

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

Lösung:

Wir konstruieren uns folgendes Schema, nach dem wir die Elemente in der ersten Runde schwärzen:

```
[s s] w [s s] w [s s] ...
 w w   w   w w   w   w w ...
[s s] w [s s] w [s s] ...
...
```

Alle diese Mengen haben den Rank 1, da immer ein Element am nächste hängt. In den folgenden Runden, mergen wir ersteinmal die Zeilen bis diese vollständig gefüllt sind. Dabei machen wir aus den 2er Mengen zunächst 5er, dann 11er und so weiter. Sind wir damit fertig fangen wir nach dem Schema an Zeilen zu mergen.

Initial haben wir $\frac{n}{6}$ Tupel von Elementen, die einen Rang von 1 haben. Die restlichen Elemente müssen frei bleiben, damit diese nicht gemerget werden müssen.
In jeder Runde halbiert sich nun die Anzahl der Menge, solange bis nur noch eine Komponente haben.

Nun sollten wir noch Elemente haben, die wir mergen können. (zumindest in den initial freien Zeilen, da wir in diesen nur 1 Element zum Mergen brauchten). In jeder Runde können wir die Mengen, die alle den gleichen Rang haben, so mergen, dass der Rang jeweils um 1 steigt.

Am Ende haben wir also eine Menge vom Rank $\log \frac{n}{6}$. Nun sollte sich also ein Element finden lassen, dass ganz unten im Baum steht. Färben wir einen Nachbern dieses Knotens schwarz, so wird bei einem find auf diesem Knoten, der komplette Baum bis zur Wurzel durchlaufern werden müssen. Da dieser Rank $\log \frac{n}{6}$ hat erhalten wir eine Laufzeit für diesen Worst-Case von $O(\log n)$.

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 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:

Sei A eine inklusionsmaximale Basis maximalen Gewichts und B die Menge, die der Algorithmus zurück gibt.

Schritt 1: B ist inkulsionsmaximale Basis.
Annahme: B wäre nicht inklusionsmaximale Basis.
Dann gäbe es noch eine Element s in der Menge S, so dass $B \cup \{s\}$ unabhängig ist. Wäre der Algorithmus an diesem s vorbei gekommen, wäre nach Vererbungslemma die Untermenge zu diesem Zeitpunkt nach der Vereinigung mit s immer noch unabhängig gewesen.

Demnach muss der Algorithmus s genommen haben und s muss Teil von B sein. Demnach muss B inklusionsmaximal sein.
Dies sagt uns nach *a)*, dass $|A| = |B|$ gelten muss.

Schritt 2:
Wir sortieren A und B absteigend und nummerieren so ihre Elemente durch, mit $B = b_1 b_2 ... b_n$ und $A = a_1 a_2 ... a_n$.
Sei nun i der kleinste Index, ab dem sich die beiden Folgen unterscheiden.

Wir zeigen nun, dass $w(a_i) = w(b_i)$ gelten muss.

Fall 1: $w(a_i) < w(b_i)$
Nach Vererbungsaxiom muss auch $A \setminus \{a_i\}$ unabhängig sein. Da nun $|A| < |B|$ (gefolgert aus Aufgabenteil *a)*) gilt, können wir b_i aus B nehmen, das nach Annahme nicht in A sein kann und benutzen darauf das Austauschaxiom.
Nun hat aber $(A \setminus \{a_i\}) \cup \{b_i\}$ ein größeres Gewicht als A, damit kann A nicht maximales Gewicht gehabt haben.

Fall 2: $w(a_i) > w(b_i)$
Nach Vererbungsaxiom ist $B \setminus \{b_i\}$ unabhängig. Nun gilt wiederum $|B| > |A|$. Nach Austauschlemma ist nun $(B \setminus \{b_i\}) \cup \{a_i\}$ eine inklusionsmaximale Menge mit einem größeren Gewicht als B. Nach den selben Überlegungen aus dem ersten Schritt war nun allerdings im Algorithmus (da wir die Menge nach größe geordnet haben) die Menge B zu einem Zeitpunkt $B = b_1 ... b_{i-1}$. Da $a_i > b_i$, muss der Algorithmus diesen Wert vorher geprüft haben und da die Menge unabhängig ist, muss er es genommen haben. Damit ist dieser Fall unmöglich.

Fall 3: $w(a_i) = w(b_i)$
Dies ist der einzigste verbliebene Fall. Wir können wieder nach Vererbungs- und Austauschaxiom die Elemente austauschen, aber diesmal bleibt das Gewicht gleich. Damit haben A und B das gleiche Gewicht und B hat maximales Gewicht.

□

Laufzeit:

Wir haben 3 Schritte des Algorithmus, die wir durchgehen müssen.

Um S zu sortieren benötigt man bekanntlich $O(|S| \log |S|)$. In einzelfällen kann es schneller gehen, wenn die Soriterung irrelevant ist.

Die Schleife wird $|S|$ mal durchlaufen.
Je nachdem, wie die Teilmengen aussehen, können wir die Überprüfung ob ein B unabhängig ist beschleunigen. Beschreibe hier $t(|S|)$ diese Laufzeit.
Wir können diese Laufzeit schon einmal nach unten beschränken:

Nehmen wir zum speichern der Teilmengen einen Trie. Da wir die Mengen der Größe nach sortieren, können wir die Reihenfolge der Elemente sehr gut abschätzen.
Von einem Trie haben wir gelernt, dass alle Operationen so lange dauern, wie unser Wort lang ist, dies bedeutet für uns, dass der vergleich $t(|S|) = |B| \leq |S| = O(|S|)$ ist. Sollten alleunabhängigen Menge total verscheiden sein, kann uns das viel Speicherplatz kosten, sollten diese Mengen aber sehr viele gemeinse Prefixe besitzen sinkt der Platzverbrauch.

Um B mitzuführen und immer zu erweitern benötigen wir $O(1)$ wenn wir z.B. eine LinkedList nehmen.

Insgesammt erhalten wir somit eine obere Schranke von:
$T(n) = O(T_{sort}(n) + n \cdot T_{check}(n)) = O(n \cdot \log n + n^2) = O(n^2)$

Diese Laufzeit kann in speziellen Anwendungen gedrückt werden, wenn entweder Sortierung weggelassen oder optimiert werden kann (Countingsort o.Ä.) oder der Check Effektiver geht (Union-Find bei Kruskal und Prim).