Vorlesungsmitschrift

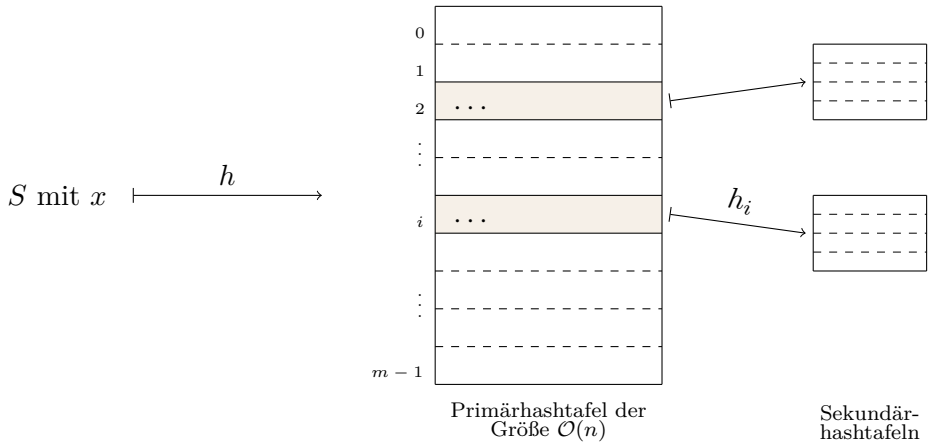
Algorithmen und Berechenbarkeit

Vorlesung 09

Letztes Update: 2017/11/25 - 12:15 Uhr

Ansatz 2: Zweistufiges perfektes Hashing

Die Idee für ein mehrstufiges perfektes Hashing ist im Folgenden aufgezeichnet:



Man wählt eine Primärhashfunktion h, sodass etwa linear viele Kollisionen erzeugt werden, also $c_S(h) = \mathcal{O}(n)$. Zusätzlich erzeugt man für jeden Primärhashtafeleintrag i, für den es mehr als einen Eintrag gibt, eine Sekundärhashfunktion h_i, die injektiv in eine Sekundärhashtafel hasht.

Des Weiteren werden festgelegt:

$m =$ Größe der Primärhashtafel
$m_i =$ Größe der i-ten Sekundärhashtafel
$l_i =$ Anzahl der Elemente, die von der Primärhashfunktion h auf i gemappt wurden.

Kollorar: Falls $m > \frac{n-1}{2} \cdot c$, dann existiert ein $h \in \mathcal{H}$ mit $c_S(h) \leq n$.

Beweis: Für ein zufälliges $h \in \mathcal{H}$ gilt

$$\begin{aligned} E(c_S(h)) &\leq \binom{n}{2} \cdot \frac{c}{m} \\ &= \frac{1}{2} \cdot \frac{n \cdot (n-1)}{\frac{n-1}{2}} \\ &= n \end{aligned}$$

□

Kollorar: Falls $m > (n-1) \cdot 2$, dann gilt für mindestens die Hälfte aller $h \in \mathcal{H}$

$$c_S(h) \leq n$$

Beweis: Wie letztes Mal. □

Damit ist gezeigt, dass eine Primärhashfunktion h in erwarteter $\mathcal{O}(n+m)$ Zeit mit $c_S(h) \leq n$ gefunden werden kann. Die Hashtafelgröße ist dabei $m = \mathcal{O}(n)$.

Seien nun alle Elemente aus S, die von der Primärhashfunktion h in den i-ten Beutel gehasht werden folgendermaßen definiert:

$$B_i(h) = h|_S^{-1}(i) = \{x \in S \mid h(x) = i\}$$

Es gilt außerdem

$$S_i(h) = |B_i(h)|$$

und

$$c_S(h) = \sum_{i=0}^{m-1} \binom{S_i(h)}{2}$$

Da h so gewählt wurde, dass $c_S(h) \leq n$, gilt auch

$$\sum_{i=0}^{m-1} \binom{S_i(h)}{2} \leq n$$

Für jeden Hashbeutel $B_i(h)$ erzeugt man sowohl eine Sekundärhashfunktion als auch eine Sekundärhashtafel mit der Größe $m_i > 2c \cdot \binom{S_i(h)}{2}$, die den Inhalt von $B_i(h)$ injektiv hasht. Der Platzverbrauch und die Konstruktionszeit für einen Hashbeutel $B_i(h)$ ist

$$\mathcal{O}\left(2c \cdot \binom{S_i(h)}{2}\right) = \mathcal{O}\left(\binom{S_i(h)}{2}\right)$$

Damit kann nun die Gesamtgröße und -konstruktionszeit aller Sekundärhashtabellen und -funktionen eingeordnet werden

$$\begin{aligned}\mathcal{O}\left(\sum_{i=0}^{m-1} c \cdot \binom{S_i(h)}{2}\right) &= \mathcal{O}\left(c \cdot \underbrace{\sum_{i=0}^{m-1} \binom{S_i(h)}{2}}_{c_S(h) < n}\right) \\ &= \mathcal{O}(c \cdot n)\end{aligned}$$

Hashing Zusammenfassung

Es sind $S \leq U, |S| = n$ gegeben. Außerdem kann man eine c-universelle Hashfunktion $h : U \to \{0, 1, \cdots, m\}$ universell samplen. Perfektes Hashing zusammengefasst ist

Primärhashing	1. Finde $h : U \to \{0, 1, \cdots, c \cdot n\}$, die auf S maximal n Kollisionen produziert.	$\mathcal{O}(c \cdot n)$-Zeit
	2. Bestimme für $i = \{0, 1, \cdots c \cdot n\}$ das $B_i(h) = \{x \in S \mid h(x) = i\}$.	
Sekundärhashing	3. Für jedes $i = 0, 1, \cdots, c \cdot n$ mit $\lvert B_i(h)\rvert > 1$ finde h_i mit $U \to \left[0, 1, \cdots, \binom{S_i(h)}{2} \cdot c\right]$, sodass h_i injektiv ist für $B_i(h)$.	$\mathcal{O}(c \cdot n)$-Zeit

Operationen auf einer gehashten Datenstruktur

Zugriff auf ein $x \in U$

Man hasht x mit der Primärhashfunktion $h : h(x) = i$.

- Falls der errechnete Platz ≤ 1 Elemente enthält ($B_i(h) \leq 1$), ist der Zugriff fertig (Man gibt das Element zurück oder man gibt zurück, dass sich kein Element an diesem Platz befindet).
- Sonst wendet man die entsprechende Sekundärhashfunktion h_i auf x an. Unter $h_i(x)$ ist sicher ein Hashtafeleintrag mit ≤ 1 Elementen.

$\Rightarrow$ Die Zugriffskosten sind in $\mathcal{O}(1)$.

Einfügen eines $x \in U$

Falls ein `insert` die Anzahl an Kollisionen der Primär- oder Sekundärhashfunktion zu sehr erhöht, wird alles komplett neu gehasht (Neukonstruktion). Das ist sehr teuer. Man kann jedoch zeigen, dass das bei universellen Hashfunktionen nicht allzu oft passiert (etwa einmal pro Verdopplung der Schlüsselmenge).
$\Rightarrow$ Die amortisierten Kosten pro `insert` sind in $\mathcal{O}(1)$.

Entfernen eines $x \in U$

Falls ein `remove` den Platzverbrauch relativ zu $|S|$ zu groß werden lässt, wird ebenfalls alles komplett neu gehasht. Das passiert etwa einmal pro Halbierung der Schlüsselmenge.
$\Rightarrow$ Die amortisierten Kosten pro `remove` sind in $\mathcal{O}(1)$.

Cuckoo-Hashing

Cuckoo-Hashing ist eine einfache Alternative zu Zweistufigem perfekten Hashing. Gegeben sind eine Hashtafel der Größe m sowie zwei Hashfunktionen h_1, h_2 aus der c-universellen Familie von Hashfunktionen $\mathcal{H}$. Jeder Hashtafeleintrag hat Platz für genau ein Element.

Einfügen eines Elements - `insert`

Man hasht ein Element x mit h_1 und überprüft, ob der Platz in der Hashtafel frei ist. Falls der Platz frei ist, fügt man x ein und ist fertig. Falls ein Element y den Platz schon belegt ($h_i(y) = h_1(x)$), wird y rausgeworfen und x nimmt seinen Platz ein. Das Element y wird nun mittels $h_{3-i}(y)$ (also jeweils der anderen Hashfunktion) ein neuer Platz zugewiesen. Falls der errechnete Platz frei ist, fügt man y ein und ist fertig. Falls ein Element z den Platz schon belegt (es also gilt: $h_j(z) = h_{3-i}(y)$), wird z rausgeworfen und y nimmt seinen Platz ein. Nun wird wieder versucht, mit $h_{3-j}(z)$ einen neuen Platz für das Element z zu finden ...
$\Rightarrow$ Die amortisierten Kosten pro `insert` sind in $\mathcal{O}(1)$.

Problem: Das Einfügen kann lange Tauschsequenzen beziehungsweise Zyklen zur Folge haben. Wenn das passiert, muss alles neu gehasht werden. Falls h_1, h_2 aus dem Beutel c-universeller Hashfunktionen gesampelt und die Hashtafel hinreichend groß gewählt wurde ($m > 2n$), ist die Wahrscheinlichkeit sehr gering, dass neu gehasht werden muss.

Entfernen eines Elements - `remove`

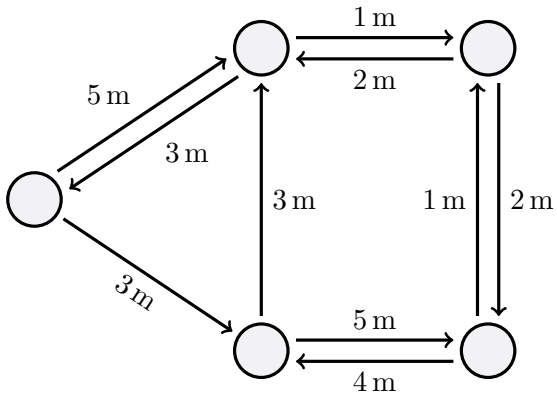
Man entfernt die Elemente bei $h_1(x)$ und $h_2(x)$. Falls $|S|$ im Vergleich zum aktuellen m zu groß wird, muss neu gehasht werden.
$\Rightarrow$ Die amortisierten Kosten pro `remove` sind in $\mathcal{O}(1)$.

Lookup eines Elements - `lookup`

Man schaut bei $h_1(x)$ und $h_2(x)$ nach.
$\Rightarrow$ Die Kosten pro `lookup` sind in $\mathcal{O}(1)$.

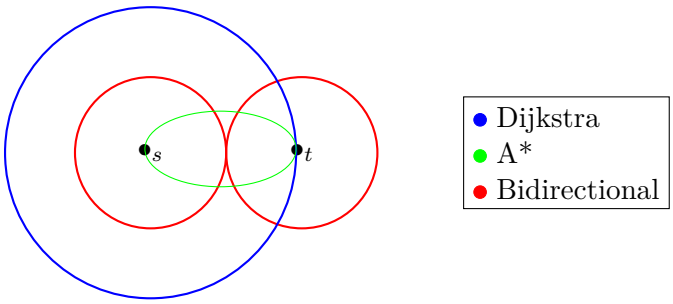
Routenplanung in Straßengraphen

Gegeben ist ein Graph $G(V, E, c)$, bei dem wie gehabt V die Menge der Knoten und E die Menge der Kanten darstellt. Im Gegensatz zum herkömmlichen Graph kommt hier noch die Reisezeit c hinzu. Ein Straßengraph kann wie im Folgenden dargestellt werden



Eine Anfrage beinhaltet einen Startknoten s und einen Targetknoten t wobei $s, t \in V$. Das Ziel ist der kürzeste/schnellste Weg von s nach t. Für dieses Problem gibt es verschiedene Algorithmen, das Standardverfahren wäre *Dijkstra*. Bei einem Straßengraphen von Deutschland (Größenordnung: $|V| \approx 20$ Millionen, $|E| \approx 40$ Millionen) dauert eine Anfrage etwa 5 s, wenn der Dijkstra-Algorithmus ordentlich implementiert wurde.

Im Nachfolgenden ist skizzenhaft dargestellt, welche Ausdehnung ausgewählte Graphalgorithmen erreichen, die dieses Problem lösen können.



Google wird wohl keinen dieser Algorithmen verwenden, da sie alle viel zu teuer wären.

Suchalgorithmen in den letzten 15 Jahren für dieses Problem

Eine Anfrage wie oben beschrieben soll in wenigen ms, µs bzw. ns beantwortet werden. Das Ganze soll auch beweisbar sein. Da Straßennetze sich nicht sehr häufig ändern, wird ein Vorverarbeitungsschritt durchgeführt, der für den Straßengraphen in einigen Minuten Hilfsinformationen berechnet. Anfragen können mit diesen Hilfsinformationen dann in 500 µs − 1 ms beantwortet werden.