

## Hinweise

Fehlende Stellen / Fehler / Unklarheiten sind mit //TODO markiert. Helfe mit diese Stellen zu verbessern, forke das Repo oder schreibe Issues!

## 1. Vorlesung

### Randomisierte Algorithmen

Vergleich mit Kochrezept, manchmal zufällig 1 oder 2 Prisen Salz. Abgrenzung zu deterministischen Algorithmen.

Ein randomisierter Algorithmus ist ein ein Algorithmus, welcher unter Nutzung einer Zufallsquelle (Münzwurf, Zufallszahlengenerator, ...) ein Problem löst:

Man könnte Algo dahingehend analysieren, wie viel Zufall er braucht. Echter Zufall z.B. gemessenes physikalisches Rauschen.

Oft sind RA (randomisierte Algorithmen) deutlich einfacher und teilweise auch effizienter als entsprechend deterministische Algorithmen. Typischerweise analysiert man Algorithmen bzgl. Platz und Zeitbedarf, RA kann man auch bzgl. "verbrauch von Zufall" analysieren (machen wir wahrscheinlich nicht).

### Problem, Closest Pair

Gegeben:  $n$  Punkte in  $\mathbb{R}^2$

//TODO Bild einfügen

Gesucht:  $P_1, P_2 \in P$  mit  $|p_1, p_2|$  minimal

**Naive Lösungsidee:**

- Betrachte  $p_1$  und berechne Distanz zu  $p_1, \dots, p_n$  merke Minimum
- Betrachte  $p_2$  und berechne Distanz zu  $p_3, \dots, p_n$
- ...
- Betrachte  $p_{n-1}$  und berechne Distanz zu  $p_n$
- Gebe Minimum aller gemessenen Distanzen aus. **Aufwand:** # gemessene Distanz

$$p_1 \text{ bzgl } n-1; \text{ etc } = n-1 + n-2 + n-3 \dots 1 = \sum i = \frac{n-1}{2} = \mathcal{O}(n^2)$$

## Beispiel

CPU mit 1 Ghz,  $10^9$  Instruktionen/Sekunde Ab welchem  $n^2$  Instruktionen für Eingabe der Größen braucht, läuft für  $n = 100 \rightarrow (100)^1 \cdot 10^{-9} = 10^{-5}$   
 $\Rightarrow 10$  Mikrosekunden  $n = 1000 \rightarrow (1000)^1 \cdot 10^{-9} = 10^{-3} \Rightarrow 1$  Millisekunde  
 $n = 100000 \rightarrow (100000)^1 \cdot 10^{-9} = 10 \Rightarrow 10$  Sekunden  $n = 1000000 \rightarrow (1000000)^1 \cdot 10^{-9} = 10^3 \Rightarrow 1000$  Sekunden

CP kann deterministisch in  $\mathcal{O}(\log n)$  Zeit gelöst werden. Man kann sogar beweisen, dass CP vergleichsbasiert nicht schneller als  $\Omega(n \log n)$ .

**Jetzt: RA der Closest Pair in erwartet  $\mathcal{O}(n)$  Zeit löst.**

X sei eine Zufallsvariable.

Der Erwartungswert einer diskreten Zufallsvariable ist  $E(X) = \sum_i i \cdot Pr(X = i)$ .

Wir entwerfe einen Algorithmus, dessen Laufzeit als Zufallsvariable  $X$  dargestellt werden kann. Bei Eingangsgröße  $n$  gilt  $E(X) = \mathcal{O}(n)$

## Ein inkrementeller Algorithmus für CP

Betrachten Punkte in Reihenfolge  $p_1, p_2, p_3, \dots, p_n$ . Sei  $\delta_i$  die CP-Distanz der Punktmenge  $\{p_1, p_2, \dots, p_i\}$ . Angenommen mit  $\delta_i$ , wie

//TODO TEXTFEHLT

Naiv: Vergleiche  $p_{i+1}$  mit  $p_1, p_2, \dots, p_i$  und setze  $\delta_i = \min(\delta_i, \min(p_{i+1}, p_\delta))$

$\rightarrow$  Gesamt ( $\delta_z$  wieder  $(1+2+\dots+n-1)$ ) =  $\mathcal{O}(n^2)$

Verbesserung: Angenommen wir haben  $\delta_i$  bestimmt und auch ein Gitter mit Maschenweite  $\delta_i$  erzeugt, in welches alle Punkte  $p_1, \dots, p_i$  eingeordnet sind.

**Berechnung von  $\delta_{i+1}$ :**

1. Lokalisiere  $p_{i+1}$  im Gitter
2. inspiziere nur Punkte in benachbarten Gitterzellen von  $p_{i+1}$  (und eigene Zelle)
3. falls neue CP-Distanz, setze  $\delta_i$ , entsprechend.; ansonsten  $\delta_{i+1} = \delta_i$

Lemma (Übung): In einer Gitterzelle liegen  $\leq 4$  Punkte.

Falls  $\delta_i = \delta_{i+1}$ , füge  $p_{i+1} > \delta_{i+1}$ , baue Gitter neu und für neues  $\delta_{i+1}$ : kostet  $\mathcal{O}(i)$ . Schlecht! Bei dieser Konfiguration und Einfügereihenfolge muss nach jedem Schritt das Gitter neu aufgebaut werden  $\rightarrow$  Laufzeit wieder  $\approx \sum_{i=1}^n i = n^2$

Abhilfe: füge Punkte in zufälliger Reihenfolge ein, d.h. jede Permutation soll mit gleicher WS  $\frac{1}{n!}$  auftauchen.



Figure 1: Bild

**Zentrale Aussage:** Wahrscheinlichkeit, dass  $\delta_{i+1} < \delta_i \leq \frac{2}{i}$ .

Dann: Erwartete Kosten des Einfügen von  $p_i \leq \frac{2}{i} \cdot \mathcal{O}(i) + \mathcal{O}(1) = \mathcal{O}(1)$

Erwartete Gesamtlaufzeit:  $E(\sum_i^n \text{Kosten für Einfügen von } p_i) = \sum_i^n E(\text{Kosten für } p_i) = \sum_i^n \mathcal{O}(1) = \mathcal{O}(n)$

Also schlechter Fall, dass Gitter

//TODO Text fehlt

---

## 2. Vorlesung

### Randomisierte inkrementelle Alogorithmen

#### Zusammenfassung CP

- Bringe Punktmenge in zufällige Reihenfolge  $p_1, \dots, p_n$
- Invariant bzgl:
  - $\delta_i :=$  Minimalabstand zweier Punkte in  $\{p_1, \dots, p_i\}$
  - haben immer Gitterstruktur mit Maschenweite  $\delta_i$ , in welcher  $p_1, \dots, p_i$  eingefügt sind.
  - wichtigste Operation: Hinzunehmen von  $p_{i+1}$  unter Aufrechterhaltung der Invarianten:
  - **Zusammen:**  $\mathcal{O}(1)$ 
    - lokalisieren von  $p_{i+1}$  in akt. Gitterstruktur  $\rightarrow \mathcal{O}(1)$
    - Untersuchen der  $\mathcal{O}(1)$  Punkte in den benachbarten Gitterzellen von  $p_{i+1}$  auf Kandidat für Closest Pair  $\rightarrow \mathcal{O}(1)$
  - $\mathcal{O}(1)$ : Fall A:  $\delta_i \leq \delta_{i+1} \rightarrow \delta_{i+1} = \delta_i \Rightarrow$  füge  $p_{i+1}$  in Gitterstruktur
  - $\mathcal{O}(i)$ : Fall B:  $\delta_i > \delta_{i+1} \Rightarrow$  baue neues Gitter mit Maschenweite  $\delta_{i+1}$ , füge  $p_1, \dots, p_{i+1}$  ein

Schlechte Punktkonfiguration und Reihenfolge führt zu  $\Theta(n^2)$  Laufzeit. Idee: Betrachte Punkte in zufälliger Reihenfolge  $\Rightarrow$  Hoffnung: "Fall B" tritt zu selten auf

Die erwartete Laufzeit des Algorithmus ist Laufzeit  $= \sum_{i=3}^n$  (Kosten für Einfügen von Punkt  $p_i$ )

Erwartete Laufzeit  $= E(\text{Laufzeit})$

Weil  $E(X_1 + X_2) = E(X_1) + E(X_2) \Rightarrow \sum_{i=3}^n E(\text{Laufzeit})$

Fall A:  $\mathcal{O}(1)$  Fall B:  $\mathcal{O}(i) = \text{WS für Fall A: } \mathcal{O}(1) + \text{WS für Fall B: } \mathcal{O}(i) \leq \mathcal{O}(1) + \text{WS für Fall B: } \mathcal{O}(i)$

**Erklärung:** Fall B tritt ein, wenn  $\delta_i < \delta_{i-1}$  d.h., wenn durch hinzufügen von  $p_0$  sich die CP-Distanz ändert.

$\delta_i < \delta_{i-1}$  kann nur passieren, wenn  $p_i$  einer der beiden Punkte aus  $\{p_1, p_2, \dots, p_i\}$  ist, welches das CP definiert. Seien  $p_a, p_b$  2 Punkte aus  $\{p_1, p_2, \dots, p_i\}$ . Fall B tritt ein, falls  $p_i = p_a$  oder  $p_i = p_b$  WS ist  $\leq \frac{2}{i}$

Erwartete Laufzeit für Einfügen von  $p_i$  ist  $\leq \mathcal{O}(1) \frac{2}{i} \cdot \mathcal{O}(i) = \mathcal{O}(1) \Rightarrow$  erw.  
Gesamtlauftzeit:  $\delta_z \sum_{i=3}^n \mathcal{O}(1) = \mathcal{O}(n)$

**Wichtig:** Die Laufzeit des Algorithmus hängt *nicht* von der Eingabe ab - sie ist immer erwartet  $\mathcal{O}(n)$ , egal wie die Eingabe aussieht.

Annamhe zur Gitterstruktur: Angenommen wir haben Gitterstruktur mit Maschenweite  $\delta$   $p(p_x, p_y)$  fällt in Gitterzelle  $(\lfloor \frac{p_x}{\delta} \rfloor, \lfloor \frac{p_y}{\delta} \rfloor)$  Annahme: Wir haben eine Datenstruktur, welche in  $\mathcal{O}(1)$  ein Element in Zelle  $(i, j)$  hinzufügen und in  $\mathcal{O}(1)$  den Inhalt einer Zelle zurückgeben kann. Größe der Datenstruktur soll  $\mathcal{O}(\#$  der eingefügten Elemente )

Im obigen (*weit oben*) Algorithmus hängt nur die Laufzeit vom Zufall ab, das Ergebnis ist immer korrekt.  $\Rightarrow$  LAS VEGAS ALGORITHMUS

Im Gegensatz dazu: **MONTE CARLO ALGORITHMEN** \* haben Laufzeit unabhängig vom Zufall \* Korrektheit des Ergebnisses hängt vom Zufall ab.

### MinCut-Problem

Lässt sich mit einem Monte Carlo (MC) Alg. lösen

Gegeben: ungerichteter, ungewichteter Multigraph mit  $|V| = n, |E| = m$

Eine Teilmenge  $A \subseteq V$  der Knoten induziert einen sogenannten **Cut**.

$\text{cut } (A, G) = \{e = v, w \in E | v, w \cap A = 1\}$  Der Wert des Cuts ist seine Kardinalität  $|\text{cut } (A, G)|$ .

Das MinCut-Problem für einen Graphen  $G(V, E)$ : Finde  $A \subsetneq V$  mit  $|\text{cut } (A, G)|$  ist minimal:

Anmerkung: Minimalgrad eines Knotens in G ist immer die obere Schranke für den Wert des MinCuts.

Anwendungen z.B.: Netzwerk & Fehlertoleranz; Stromnetze; Redundanzprüfung

### Naiver Algorithmus

Teste alle möglichen Partitionen von  $V$   $\sum_{i=1}^{n-1} \binom{n}{i} = 2^n - 2$  für  $n = |V| \rightarrow$  aber nicht praktikal, da schon für einen Graph mit 128 Knoten mehr Partitionen möglich sind (340282366920938463463374607431768211456) als es Anzahl Atome im Universum gibt. > Was zu beweisen wäre ~

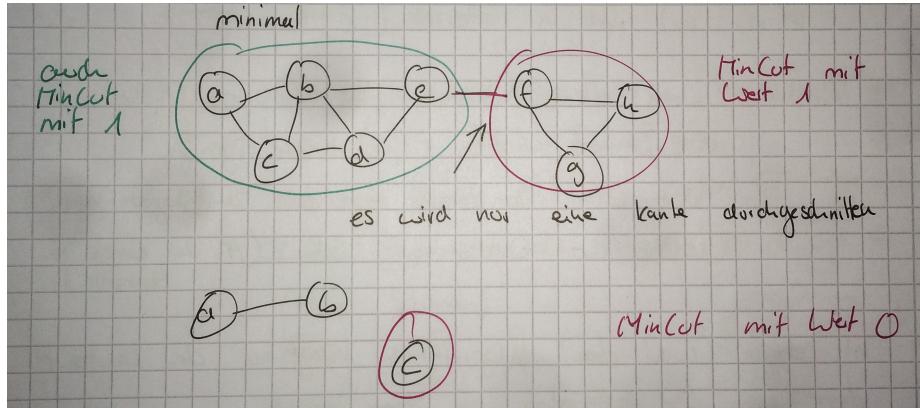


Figure 2: Bild

### Kargers MinCut Algorithmus

Randomisierter Algorithmus der mit WS  $\sim \frac{1}{n^2}$  das richtige Ergebnis berechnet.  
**Zentrale Operation: Kantenkontraktion** 1. wähle eine Kante  $\{v, w\}$  aus 2. ersetze Knoten  $v$  und Kante  $w$  durch Knoten  $vw$  der alle Kanten von  $v$  und  $w$  erbt (ohne Schlingen).

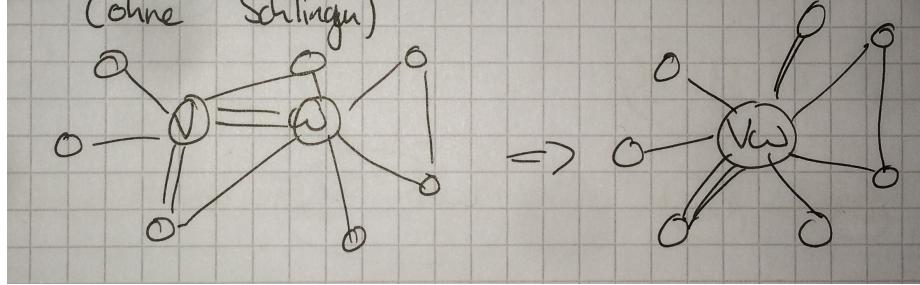


Figure 3:

### Algorithmus

**for** 1 to  $n - 2$  **do:** Kontrahiere zufällige Kante **rof** Gib Kantenmenge entsprechend einen der beiden verbleibenden Knoten aus.

Annahme: Es gibt genau einen MinCut und dessen Wert ist K.

*Beobachtung:* Alg. berechnet den MinCut  $\Leftrightarrow$  In keiner der  $n - 2$  Kontraktionen wurde einer der K MinCut-Kanten kontrahiert.

Output =  $\{e, f\}$  induziert Cut mit Wert 2.

Alternativer Ablauf:

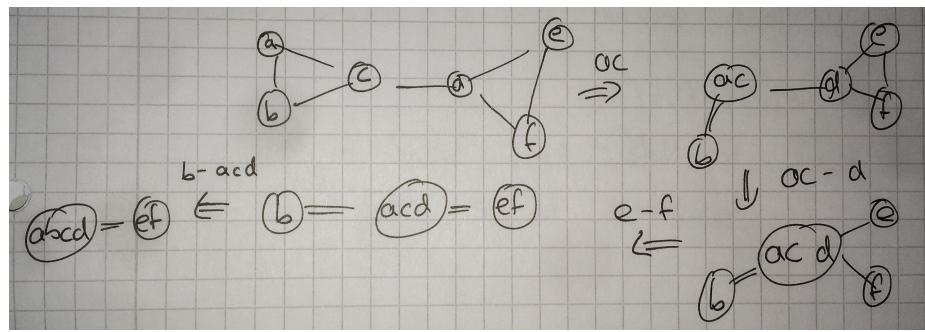


Figure 4:

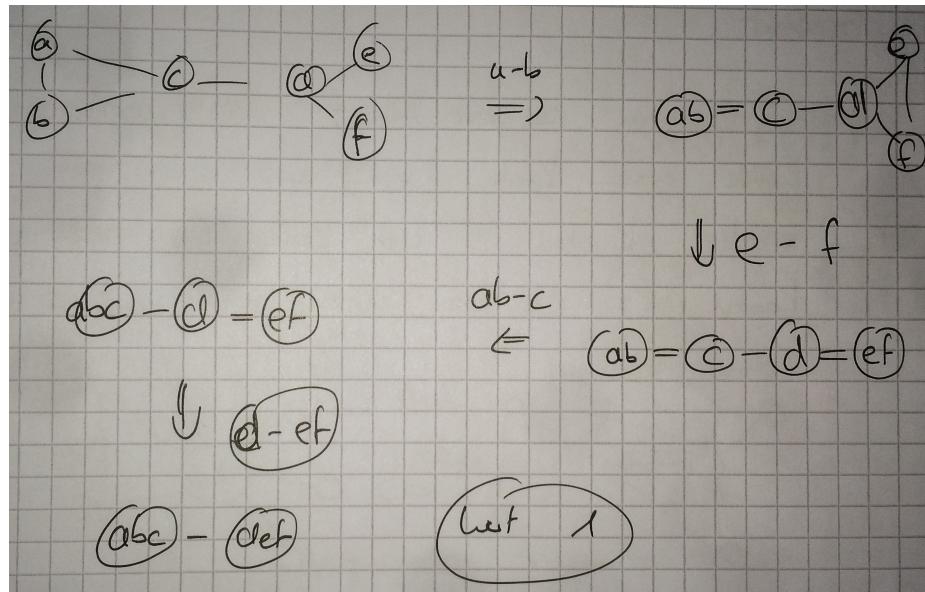


Figure 5:

---

### 3. Vorlesung

//TODO Anfang fehlt

#### Repeate MinCut

Zeige, dass WS des Kargers MinCut Alg. das richtige Ergebnis berechnet, mindestens  $\frac{1}{n^2}$  ist.

Das klingt nicht besonders gut, aber wir können die Erfolgswahrscheinlichkeit beliebig erhöhen durch Wiederholung.

```
RepMinCut: bestVal = ∞; bestSol = ∅  
for i=1 to r do  
    p = KargerMinCut(G)  
    if (val((p)) < bestVal)  
        bestVal = val((p))  
        bestSol = (p)  
    endif  
    rof  
return bestSol
```

Angenommen KargersMinCut berechnet MinCut mit WS  $\geq \frac{2}{n(n-1)}$ .

Die WS, dass in jedem der  $r$  Versuche nicht der MinCut gefunden wird ist:

$$(1 - \frac{2}{n(n-1)})^r < (e^{-\frac{2}{n(n-1)}})^r > \text{weil } < \text{bedeutet: } 1 \cdot x < e^{-x} = e^{-\frac{rr}{n(n-1)}}$$

Obere Schranke für WS, dass sie jedes Mal Pech haben z.B.  $r = \frac{n(n-1)}{2} \Rightarrow$  WS, dass falsch  $1 \leq \frac{1}{e} \approx 0,36$  oder auch für  $r \approx n^2 \log n \Rightarrow$  Fehler WS  $\leq \frac{1}{n^e}$

**Jetzt: Beweisen, dass Erfolgs WS in der Tat  $\geq \frac{2}{n(n-1)}$**

Wir //TODO Wort fehlt hierzu einen MinCut und zeigen, dass die WS nie eine Kante des MinCuts zu kontrahieren  $\geq \frac{2}{n(n-1)}$  ist.

**Lemma:** Betrachte einen Multigraph  $G(V, E)$  mit einem MinCut mit Wert  $k$ . Dann gilt:  $G$  hat mindestens  $\frac{k \cdot n}{2}$  Kanten. **Beweis:** Beobachtung: Jeder Knoten hat mindestens Grad  $k$ .

Sei  $\varepsilon$  Ereignis, dass in i-tem Kontraktionsschritt keine MinCut-Kante erwischte.

Wir betrachten einen fixen MinCut mit Wert  $k$  (bzw. dessen  $k$  Kanten). Die WS im ersten Kontraktionschritt eine MinCut-Kante zu erwischen ist:

$$\frac{k}{m} \leq \frac{2}{n} \Rightarrow Pr(\varepsilon_1) \geq 1 - \frac{2}{n} > \text{die erste ungleichung gilt weil: } m \geq \frac{k \cdot n}{2}$$

Falls  $\varepsilon_1$  eingetreten ist, existieren vor dem zweiten Schritt mindestens  $\frac{k \cdot (n-1)}{2}$  Kanten.

Die Ws im zweiten Schritt eine MinCut-Kante zu erwischen (unter der Voraussetzung, dass  $\varepsilon_1$  eingetreten) ist:

$$\leq \frac{k}{\frac{k(n-1)}{2}} = \frac{2}{n-1}$$

$$\Rightarrow Pr(\varepsilon_1 | \varepsilon_2) \geq 1 - \frac{2}{n-1}$$

Falls  $\varepsilon_1$  und  $\varepsilon_2$  eingetreten sind, existieren vor dem dritten Schritt mindestens  $\frac{k(n-2)}{2}$  Kanten.  $\Rightarrow$  WS, im dritten %%TODO ( - " - ) (unter der Voraussetzung, dass  $\varepsilon_1 \& \varepsilon_2$  eingetreten sind) ist:

$$\leq \frac{k}{\frac{k(n-2)}{2}} = \frac{2}{n-2}$$

$$\Rightarrow Pr(\varepsilon_3 | \varepsilon_1 \cap \varepsilon_2) \geq 1 - \frac{2}{n-2}$$

**Allgemein:** Vor dem i-ten Schritt haben wir noch  $n - i + 1$  Kanten. Falls wir bisher noch keinen Fehler gemacht haben ( $\varepsilon, \dots, \varepsilon_{i-1}$  sind eingetreten) hat der MinCut noch Größe k  $\Rightarrow$  wir haben noch  $\geq k(n - i + 1)$  Kanten  $\Rightarrow Pr(\varepsilon_i | \cap_{j=1}^{i-1} \varepsilon_j) \geq 1 - \frac{2}{n-i+1}$

Uns interessiert  $Pr(\cap_{j=1}^{n-2} \varepsilon_j)$

$$= Pr(\varepsilon_1) \cdot Pr(\varepsilon_2 | \varepsilon_1) \cdot Pr(\varepsilon_3 | \varepsilon_1 \cap \varepsilon_2) \cdot Pr(\varepsilon_4 | \dots) \leq \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) \\ = \prod_{i=1}^{n-2} \frac{n-i+1-2}{n-i+1} = \prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1} = \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \cdot \frac{n-6}{n-4} \cdot \dots \cdot \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{n(n-1)}$$

$$\Rightarrow Pr(\text{Algorithmus berechnet wirkliche MinCut}) \geq \frac{2}{nn-1}$$

$$k = 2$$

$\Rightarrow$  Wahrscheinlichkeit dass wir mehr als  $\log_2 n$  mal werfen müssen, bis Kopf kommt ist  $\leq \frac{1}{2}^{\log_2 n} = \frac{1}{n}$

## 4. Vorlesung

### Komplexität Randomisierter QuickSort

#### Laufzeit Randomisierter QuickSort

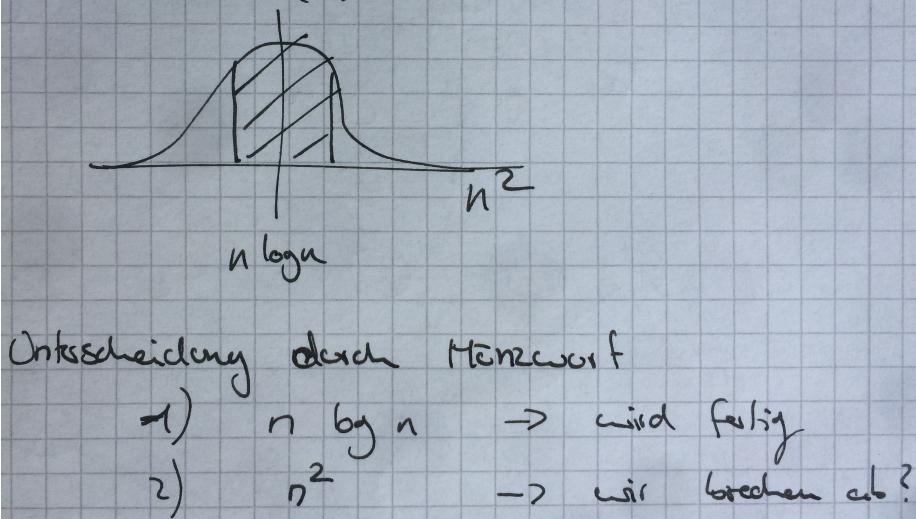
Mit hoher Wahrscheinlichkeit braucht Randomisierter Quicksort mit Wrapper  $\leq 2n \cdot (\log n) \cdot (\log n) \leq 2n \cdot \log^2 n$  Schritte.

Also  $Pr(X > 2 \cdot n \cdot \log^2 n) \leq \frac{1}{n}$

**Zum Vergleich: Mit Markov-Ungleichung konnten wir zeigen:**

$$Pr(X > 2 \cdot n \cdot \log^2 n) \leq \frac{1}{2 \log n}$$

Konzentration um den Erwartungswert  
für Algorithmen mit  $O(n \log n)$  oder  
Worst Case  $O(n^2)$ .



Unterscheidung durch Häufewurf

1)  $n \log n \rightarrow$  wird fällig

2)  $n^2 \rightarrow$  wie brechen ab?

Figure 6:

Genauso gilt für Wrapper Algorithmus:

$$Pr(X > 2 \cdot 5 \cdot n \cdot \log^2 n) \leq \frac{1}{n^5}$$

Hinweis: 5 is hier Exponent und Faktor.

### **Zero-Knowledge-Proof**

#### **Anwendung:**

Online-Banking. Sie gehen auf die Webseite ihrer Bank. Wie können Sie sicher sein, welche auf deren Webseite zu sein und nicht auf dem Mockup ihres WG-Genossen, der ihre TANs abgraben will?

#### **Ansatz:**

Die Bank soll ihnen beweisen, dass sie ein Geheimnis kennt, welches nur die Bank und Sie kennen kann, allerdings ohne das Geheimnis zu verraten.

#### **In Cryptosprache:**

Alice möchte Bob beweisen, dass sie ein Geheimnis kennt, ohne das Bob etwas über das Geheimnis lernt (Zero-Knowledge-Proof).

#### **Gewünschte Eigenschaften des Protokolls:**

- Falls Alice das Geheimnis nicht kennt, wird sie mit hoher Wahrscheinlichkeit von Bob ertappt.
- Alice gibt nichts preis, was Bob nicht sowieso schon weiß.

#### **Protokoll basiert auf dem Graphisomorphieproblem:**

#### **Geg:**

Graphen  $G_1(V_1, E_1)$  und  $G_2(V_2, E_2)$

#### **Frage:**

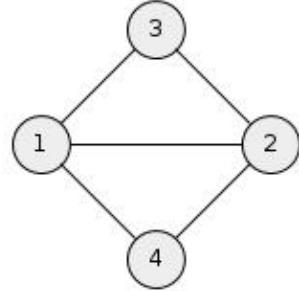
Sind  $G_1$  und  $G_2$  isomorph?  $G_1$  und  $G_2$  heißen isomorph zueinander falls  $\exists$  eine Bijektion:  $\phi : v_1 \rightarrow v_2$

Sodass  $\forall (u, v) \in E_1 \Leftrightarrow (\phi(u), \phi(v)) \in E_2$

Beispiel 1:

Ja, sind isomorph, Beweis:

G1



G2

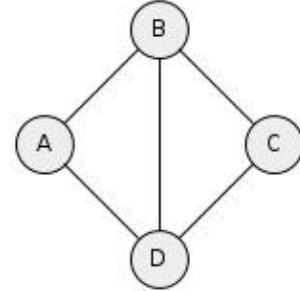
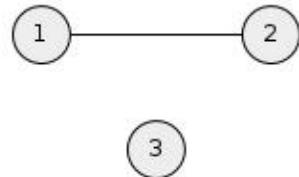


Figure 7:

$v$	$\phi(v)$
1	D
2	B
3	A
4	C

Beispiel 2:

G1



G2

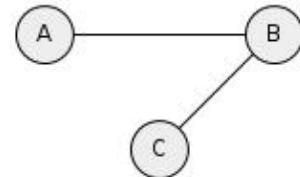
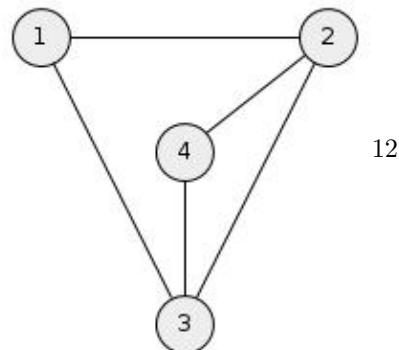


Figure 8:

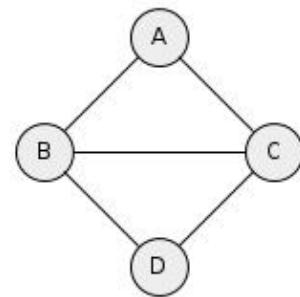
Nein, weil identische Anzahl Knoten und Kanten notwendig für Existenz einer Isomorphie.

Beispiel 3:

G1



G2



$v$	$\phi(v)$
1	A
2	C
3	B
4	D

Beispiel 4:

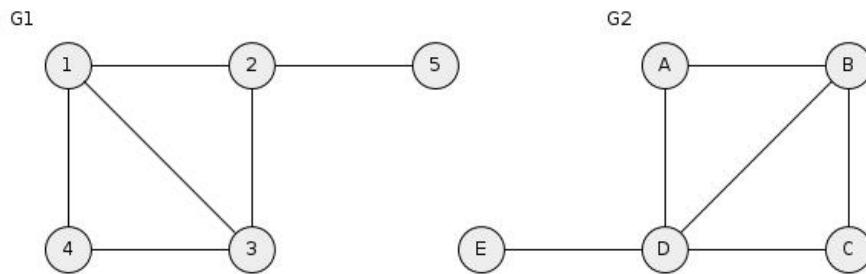


Figure 10:

Nein, da  $G_2$  einen Grad-4-Knoten enthält und  $G_1$  nicht.

Bislang gibt es keinen Polynomzerlegungsalgorithmus, der Graphenisomorphie entscheidet. NP-Hart ist nicht bekannt.

Geheim der Bank (Alice): Isomorphismus zwischen zwei öffentlich bekannten Graphen  $G_1$  und  $G_2$ .

Alice (Bank) möchte Bob (Sie) davon überzeugen, dass sie  $\phi$  kennt, ohne  $\phi$  zu verraten.

### Setup (nur 1x):

Alice erzeugt zufällig einen riesigen Graph  $G_1$  und durch zufällige Permutation der Knotenmenge (das ist der geheime Isomorphismus  $\phi$ ) auch einen isomorphen Graph  $G_2$ .

$G_1$  und  $G_2$  werden öffentlich gemacht,  $\phi$  bleibt das Geheimnis von Alice.

### Protokoll

Protokoll durch welches Alice Bob überzeugt, das Geheimnis zu kennen:

Alice permutiert  $G_j$  (mit  $j \in \{1, 2\}$  zufällig) mit einer zufälligen Permutation  $\pi$  zu  $H$  (und stellt sicher, dass  $H \neq G_1, G_2$ ) und schickt  $H$  zu Bob.

Bob: Möchte sich davon überzeugen, dass Alice Isomorphie zwischen  $G_1$  und  $G_2$  kennt, indem er  $k \in \{1, 2\}$  zufällig wählt und Alice bittet, die Isomorphie zwischen  $G_k$  und  $H$  offenzulegen.

Alice: Zerlegt entweder  $\pi$  (falls  $j = k$ ) oder  $pi \cdot \phi$  bzw.  $pi \cdot \phi^{-1}$

Alice erzeugt  $H$  aus  $G_1$

//TODO BILD

Alice antwortet mit  $\pi$ , falls Bob  $k = 1$  sagt bzw. mit  $\phi \cdot \phi^{-1}$  falls  $k = 2$ .

Alice erzeugt  $H$  aus  $G_2$

Falls Bob  $k = 1$  sagt:  $\pi \cdot \phi$

Falls Bob  $k = 2$  sagt:  $\pi$

Falls Alice  $\phi$  kennt, kann sie immer korrekt antworten.

Falls Alice  $\phi$  nicht kennt und  $H$  aus  $G_1$  konstruiert hat, kann sie Bobs Anfrage nach Isomorphie zwischen  $G_1$  und  $H$  einfach beantworten, allerdings wird es Alice schwerfallen, eine Isomorphie zwischen  $G_2$  und  $H$  zurück zu geben, es sei den, sie kann das Graphisomorphieproblem lösen.

Theorem:

Alice verrät nichts über  $\phi$

Beweis:

Bob lernt Isomorphie zwischen z.B.  $G_1$  und einer zufälligen Permutation von  $G_1$ . Die Isomorphie hätte er sich auch selber basteln können.

Theorem:

Bob bekommt probalitischen Beweis, dass Alice  $\phi$  kennt. Bei  $f$ -maliger Wiederholung ist Wahrscheinlichkeit, dass Bob immer den Graph  $G_j$  wählt, den Alice zur Erzeugung des jeweiligen  $H$  verwendet hat  $\frac{1}{2^f}$ .

Beweis:

//TODO

## 5. Vorlesung

Bis jetzt: Randomisierte Algorithmen

Jetzt: Randomisierte Datenstruktur Skiplisten - eine randomisierte Alternative zu (2, 4)-, AVL-, Rot-Schwarz-, ... Bäumen

---

## Skiplisten

Gegeben:

$S = x_1, \dots, x_n \subseteq U$  aus einem total geordneten Universum.

Ziel: Baue Datenstruktur für  $S$ , sodass man effizient - für gegebenes  $x \in U$  das maximale Element  $x' \in S$  mit  $x' \leq x$  bzw. das minimale Element  $x'' \in S$  mit  $x'' \leq x$  (Suche / Lookup) - Elemente hinzufügen kann - Elemente löschen kann

Naiv: Sortierte Liste: - Suchzeit:  $\sim n$  -  $\mathcal{O}(1)$  falls Position bekannt, sonst  $\mathcal{O}(n)$  für Einfügen - dito für Löschen Platz:  $\mathcal{O}(n)$

Besser: (2,3,4)-Baum: - Platz:  $\mathcal{O}(n)$  - Suchzeit:  $\mathcal{O}(\log n)$  - Einfügen / Löschen:  $\mathcal{O}(\log n)$

C'T: Skiplisten:

Beispiel:

//TODO GRAFIK

Bei Einfügen eines Elements wird zufällig eine "Turmhöhe"  $h$  gewürfelt mit  $Pr(h = i) = 2^{-(i+1)}$ .

Verbinde jede Turmetage mit dem nächsten Turm, der auf gleicher Höhe rechts davon sichtbar ist.

Suche nach einem Element mit  $x \in U$

```
$v := $"-infinity" - Turm
$h := v.height$
while $h \geq 0$ do
    while (x > v.forward[h] -> key) do
        v = v.forward[h]
    od
    h := h - 1
od
return
```

Übung: Erklären von Einfügen und Löschen zeigen, dass Platzverbrauch  $O(n)$  erwartet.

Zu zeigen: Erwartete Suchzeit nach einem Element  $x \in U$  ist  $O(\log n)$ .

Standardbeweis: Definiere Zufallsvariable  $X_{i,k} = \begin{cases} 1 & \text{falls such nach } x \text{ Turm } i \text{ in Höhe } k \text{ bes} \\ 0 & \text{sonst} \end{cases}$

Laufzeit =  $\sum_{i=0}^{j-1} \sum_{h \geq 0} X_{i,k}$

falls x Rang j in der Schlüsselmenge hat.

//TODO Prüfen

Alternativer Beweis: Lemma: Die erwartete Höhe eines Turms über einem Element  $X_i$  ist  $O(n)$  Beweis: Übung

Was ist die erwartete Maximalhöhe eines vorkommenden Turms?

- Mit hoher Wahrscheinlichkeit ist ein fixer Turm nicht höher als  $\sim \log n$ .  
 $Pr(h_i \geq k) = 2^{-k} \Rightarrow Pr(h_i \geq 2 \cdot \log n) = \frac{1}{n^2}$
- Wahrscheinlichkeit, dass irgendein Turm Höhe  $\geq 2 \cdot \log n$  hat, ist  $Pr(h_1 \geq 2 \cdot \log n \vee h_2 \geq 2 \cdot \log n \vee \dots \vee h_n \geq 2 \cdot \log n) \leq n - \frac{1}{n^2} = \frac{1}{n}$  (wegen Unionband)

Also sind mit hoher Wahrscheinlichkeit ( $\geq 1 - \frac{1}{n}$ ) alle Türme nicht größer als  $2 \cdot \log n$ .

Idee für Analyse:

Konstruiere Routine, die genau die selben Zellen (Turm + Etage) besucht wie Suchroutine, aber einfacher zu analysieren ist.

```

v := x
h := 0
while v \neq -\infinity \and h \neq h_max do
    if v.height > h then
        h := h + 1
    else
        v = v.backward[h]
    fi
od

```

Besucht offensichtlich gleiche Zellen wie die Suche.

Beobachtung: Wenn man als Männchen, der dem obigen Algorithmus ausführen muss, in einer Zelle eines Turms sitzt, dann muss man mit einer Wahrscheinlichkeit  $1/2$  ein Stockwerk hoch gehen und mit einer Wahrscheinlichkeit von  $1/2$  einen Turm nach links.

$\Rightarrow$  Erwartete Anzahl Links-Schritte = erwartete Anzahl Hoch-Schritte  $\Rightarrow$  Erwartete Links-Schritte =  $O(\log n)$   $\Rightarrow$  Gesamlaufzeit erwartet  $O(\log n)$

### Das Wörterbuchproblem (Hashing)

Gegeben: Ein Universum  $U$  (sehr groß) und eine Teilmenge  $S \subseteq U$  (eher klein) sowie ein  $m \in \mathbb{N}$  ist das Ziel die Bestimmung einer Funktion  $h : U \rightarrow \{0, 1, \dots, m-1\}$  sodass ||  
//TODO Foto

Beispiel: 1) Telefonbuch Aufgabe: Für gegebenen Namen und Vornamen die entsprechende Telefonnummer finden.

$|U|$  = sehr groß (Anzahl aller Nachnamen und Vornamen)  $|S| = 500000$  (Anzahl der Nachnamen und Vornamen in Stuttgart) Ideal:  $m = 500000$  Dann speichere 500 000 Telefonnummern in Array sodass  $h$  für gegebenen Nachnamen und Vornamen genau den Index im Array liefert, wo entsprechende Telefonnummer steht.

Falls  $h()$  in  $O(n)$  ausgewertet werden kann, hat man  $O(n)$  Zugriff auf die Information.

## Vorlesung 6

### Das Wörterbuchproblem

Gegeben:

Universum  $U$  (sehr groß)  
 Teilmenge  $S \subseteq U$  (mittelgroß)  $|S| = n$   
 $m \in \mathbb{N}$

Ziel: Finde  $h : U \rightarrow \{0, \dots, m-1\}$  sodass

$$\forall i \leq m : |\{x \in S | h(x) = i\}| \leq \frac{n}{m}$$


---

Bspw:  $U = \mathbb{N}$

$S = 1, 77, 23, 99$  ( $n = 4$ )

Gute Funktion  $h$  fpr  $S$  wäre zum Beispiel  
 $h(x) = x \bmod 5$

$$h(1) = 1$$

$$h(2) = 2$$

$$h(23) = 3$$

$$h(99) = 4$$

Gleiche Hachfunktion für:  $S = 12, 22, 17, 32, 52$

Anwendungsbeispiele: 1) Telefonbuch:

Mit Name und Vorname die zugehörige Telefonnummer rausfinden

$U \cong$  Menge aller Zahlen die irgendeine Name und Vorname Kombination entsprechen

$S \subseteq U \cong$  Name und Vorname in Stuttgart  $|n| = 500000$

Wähle  $m = 500000$

Könnten wir eine Hashfunktion  $h : U \rightarrow 0, \dots, m-1$  abbildet mit:

$\forall 0 \leq i < m : |\{x \in S | h(x) = i\}| \leq \frac{n}{m} = 1$  > Das ist toll Wir können ein Array anlegen mit 500000 Einträgen. Die Telnr. einer "Name und Vorname" speichern wir an Position  $h(\text{Name und Vorname})$ , des Arrays.

- 2) Stowasser (Latein-Deutsch Wörterbuch)  
 $U$ ... Menge aller Wörter  
 $S$ ... Menge aller Lateinwörter  
 $\dots$
- 3) Beim randomisierten CP-Alg:  
 $U$ ... Menge aller Paare  $(i, j)$ ,  $i \in \mathbb{Z}, j \in \mathbb{Z}$   
 $S$ ... Menge der Paare  $(i, j)$ , für die in enspr. Gitterzelle mind. 1 Punkt liegt.
- 4) OpenStreetMapDaten  
basiert auf Knoten & Kanten Anzahl der Knoten ungefähr 4-5Mrd.  
 $U \dots \mathbb{N}$   
 $S$ ... Menge der Knoten IDs im aktuellen Kartenausschnitt.  
Bei guter Hashfunktion  $h$  könnte man in  $\mathcal{O}(1)$  Information mit  $S$  assoziieren oder auslesen.
- 

### Betrachten wir ein Wörterbuch als abstrakte Datenstruktur.

Diese sollte folgende Operationen unterstützen:

- $S = \text{MAKESET}()$  ... (ein Wörterbuch erzeugen)
- $\text{inSet}(x, S)$  ... fügt item  $x$  in  $S$  ein  
> item  $x$ : besteht aus  $(K, \text{inf})$   
>  $K$  = Schlüssel;  $\text{inf}$  = Information
- $\text{delete}(x, S)$  ... löscht item  $x$  aus Wörterbuch
- $\text{lookup}(K, S)$  ... gibt item  $x = (K, \text{inf})$  aus, falls vorhanden

**Naive Implementierungen** 1) Array/Feld für die Items, Zähler für Anzahl Items in  $S$

0	1	$\dots$	$n - 1$
$i_0 = (K_0, \text{inf}_0)$	$i_1 = (K_1, \text{inf}_1)$	$\dots$	$i_{n-1} = (K_{n-1}, \text{inf}_{n-1})$

- $\text{insert}(x, S)$  ... Überprüfe ob Schlüssel von  $x$  schon enthalten, falls ja überschreiben des entsprechenden items, falls nein, Item hinten

anhängen & Zähler eins erhöhen

- $\text{delete}(x, S)$  ... item finden, austausch mit letztem Item, Zähler eins runter
  - $\text{lookup}(k, S)$  ... alle Items durchgehen, Item mit passendem Schlüssel zurückgeben
  - $\text{MAKESET}()$  ...  
> Vorteile: Einfach, platzsparend
- 2) Verkettete Liste (einfach oder doppelt)  
... ähnlich, aber braucht nicht unbedingt zusätzlichen Speicher
- 3) Direkte Adressierung  
Annahme: Schlüsseluniversum endlich,  $[0, \dots, n - 1]$   
Wir spendieren ein Array der Größe  $K$ .  
An Position  $i$  des Arrays steht das Item mit Schlüssel  $i$  (falls vorhanden) oder Sonderwert "NIL" falls keines vorhanden.  
Insert/Delete/Lookup haben alle Zeitverbrauch  $\mathcal{O}(1)$ .  
*Problem:* Platzverbrauch ~Größe des Schlüsseluniversums  
> zB bei verketteter Liste war Platzverbrauch ~Anzahl zu verwaltender Elemente

**Ziel:** Datenstruktur für Wörterbuch, welches

- $\mathcal{O}(1)$  Zugriffszeit für insert/delete/lookup -  $\mathcal{O}(n)$  Platzverbrauch garantiert.
- 

**Achtung:** er mixt Schlüssel und Item

Idealierweise hätten wir gerne für ein gegebenes

$S \subseteq U = 0, \dots, 2^64 - 1, |S| = n$  eine hashfunktion  $h$ , welches  $S$  injektiv auf  $\{0, \dots, n-1\}$  abbildet, d.h.  $\forall x, y \in S : h(x) \neq h(y)$  Mit einem solchen  $h$  könnten wir  $S$  wie folgt verwalten:

- lege Array mit  $n$  Einträgen für die Items an - Item mit Schlüssel  $x$  wird an Position  $h(x)$  gespeichert ( an keiner Stelle wird mehr als 1 Item gespeichert ) -  $\text{lookup}(x, S)$ : Schaue an Position  $h(x)$  nach:

**Vorteile:** - Zugriffszeit  $\mathcal{O}(1)$  - Größe  $\mathcal{O}(|S|)$

---

- $U$  ... Schlüsseluniversum
- $T[0, \dots, t - 1]$  ... Feld auch hashfeld genannt

- $S \subseteq U$  ... Teilmenge von Schlüsseluniversum, welche gehasht werden soll

**Gesucht:**  $h : U \rightarrow [0, \dots, t - 1]$

$C_x^h(S) := y \in S | h(y) = h(x)$  ... die Menge an Elementen in  $S$ , die von  $h$  auf derselben Hashtafeleintrag gemappt werden wie  $x$

Idealierweise  $C_x^h(S) \leq 1$  ( $\Rightarrow h$  injektiv für  $S$ )

Falls  $t < |S|$ , hätte man gerne  $|C_x^h(S)|$

**Frage:** Gibt es eine Superhashfunktion  $h^*$ , welche für jedes  $S \subseteq U$  garantiert, dass  $|C_x^h(S)| \leq \frac{\lceil |S| \rceil}{t}$  ?

**Bsp:**  $U = \mathbb{N}$

$S = 20, 13, 6, 18, 28, 31, t = 5$

$h(x) = x \bmod 5$

#### TABELLE

Für  $S = 11, 33, 22, 19, 5$  wäre daraus  $h$  perfekt.

Für  $S = 16, 5, 1, 26, 31, 13$  ist dieses Katastrophal.

**Satz:** Seien  $U, t$  und  $h$  gegeben,  $|U| = K$ .

Dann gibt es für alle  $n$  mit  $1 \leq n \leq \frac{K}{t}$  ein  $S \subseteq U$  mit  $|S| = n$

Schreibe auch  $S \in \binom{K}{n} \quad \binom{70}{12}$

mit  $|C_x^h(S)| = n \quad \forall x \in S$ , d.h. alle Elemente aus  $S$  werden in den gleichen Hashtafeleintrag gehasht.

Eventuell Prüfungsrelevant

#### Beweis:

Nach Schubfachprinzip existiert ein  $i$

$0 \leq i < t$  sodass  $(h^{-1}(i)) \geq \frac{|U|}{t} > x \in U | h(1) = i$

$\Rightarrow \frac{|U|}{t} = \frac{K}{t} \geq n$ . Wähle  $S$  aus  $\binom{h^{-1}(i)}{n}$

$\Rightarrow$  Es gibt also keine Superhashfkt die für jedes  $S$  gut ist; man sollte also die Hashfkt.  $h$  randomisiert oder unter Berücksichtigung von  $S$  bestimmen.

## Vorlesung 7

### Das Wörterbuchproblem

#### Annahme:

Universum  $\mathcal{U} = \{0, 1, \dots, N - 1\}$

Bspw. alle OSM IDs.

Abzuspeichernde Schlüsselmenge  $S \subseteq \mathcal{U}$  mit  $|S| = n$   
 Bspw. alle im Raum Stuttgart vorkommende OSM IDs.

$$n \ll N$$

Hashtafel:

$$T : |0|1|2|3|4| |-|-|-|-|$$

**Gesucht:**

Gute Hashfunktion  $h : \mathcal{U} \rightarrow \{0, \dots, m-1\}$  für  $S$

Wir wollen dann Informationen zu  $x \in S$  in Hashfeldeintrag  $h(x)$  speichern.

**Letztes Mal:**

**Theorem:**

Es gibt keine immer ( $\forall S \subseteq \mathcal{U}$ ) "gute" Hashfunktion.

**Beispiel:**

$$U = \{0, \dots, 2^{64} - 1\}$$

$$S = \{3, 7, 9, 10\}$$

$$h(x) = x \bmod 5$$

$$T : |0|1|2|3|4| |-|-|-|-| |10||7|3|9|$$

Für dieses  $S$  ist  $h$  sehr gut.

Für  $S = \{6, 11, 31, 46\}$  aber nicht.

Problem, wenn  $h(x_1) = h(x_2)$  für  $x_1 \neq x_2$ .

### Hashing mit Verkettung

Jede Tafelposition ist Kopf einer verketteten Liste. Alle  $x \in S$  mit  $h(x) = i$  werden in  $i$ -ter Liste gespeichert.

Platzbedarf:  $\mathcal{O}(|T| + |S|) = \mathcal{O}(m+n) = \mathcal{O}(n(1 + \frac{1}{\beta}))$  mit  $\beta = \frac{n}{m}$  (Belegungsstufe).

Je kleiner  $\beta$ , desto platzineffizienter wird das Wörterbuch (aber dann evtl. einfacher, gute Hashfunktion zu finden).

Zugriffszeiten:

> unter Annahme dass  $h(x)$  in  $\mathcal{O}(1)$  ausgewertet werden kann.

Zugriff auf  $x \in S$

$\mathcal{O}(1 + \text{Position von } x \text{ in Liste } L_{h(x)})$

Zugriff auf  $x \notin S$

$\mathcal{O}(1 + |L_{h(x)}|)$

---

### Zum Aufwärmen:

#### Annahme:

$h$  verteilt  $\mathcal{U}$  gleichmäßig über  $T$ , d.h.  $\forall i |\{x \in \mathcal{U} | h(x) = i\}| \leq \lceil \frac{N}{m} \rceil$

Bsp:  $h(x) = x \bmod m$

#### Satz:

Sei  $x$  ein zufälliges Element aus  $\mathcal{U} \setminus S$  und  $n \leq \frac{N}{2}$ . Dann ist die erwartete Suchzeit für  $x$  in  $\mathcal{O}(1 + \beta)$ .

#### Beweis:

$l_i = \#$  Elemente aus  $S$ , die in  $L_i$  gespeichert werden, mit  $i = 0, \dots, m - 1$ .

Es gilt:  $\sum_{i=0}^{m-1} l_i = n$

#### Die erwartete Suchzeit ist:

$$(\sum_{i=0}^{m-1} Pr(h(x) = i) \cdot l_i) + 1$$

$\mathcal{U}_i \subseteq \mathcal{U}$  Menge aus  $\mathcal{U}$ , welche von  $h$  auf  $i$  gemappt wird.

$$Pr(h(x) = i) = \frac{\mathcal{U}_i \setminus S}{\mathcal{U} \setminus S} \leq \frac{\mathcal{U}_i}{\mathcal{U} \setminus S} \leq \frac{\lceil \frac{N}{m} \rceil}{\frac{N}{2}} \leq \frac{\frac{N}{m+1}}{\frac{N}{2}} = \frac{2}{m+1} \leq \frac{2}{m} + \frac{2}{N} \leq \frac{2}{m} + \frac{1}{n}$$

#### Beweis:

//TODO Beweis vom Foto

Das heißt für zufällige Elemente nicht aus  $S$  ist erwartete Zugriffszeit ok.

#### Problem:

In der Praxis sind Zugriffe nicht zufällig.

**Satz:**

Sei  $x$  ein zufälliges Element aus  $S$ . Dann ist die erwartete Zugriffszeit für  $x$ :  
 $\mathcal{O}(1 + \frac{1}{n} \sum_{i=1}^m l_i - 1(\frac{l_i(l_i+1)}{2}))$ .

Anmerkung:

Satz nicht besonders aussagekräftig, da  $\sum l_i = n$  fast nichts aussagt über  $\sum \frac{l_i(l_i+1)}{2}$

Beweis:

Wenn  $x$  das  $j$ -te Element in seiner Liste ist, dann ist Suchzeit  $\mathcal{O}(1 + j)$ . Also ist die erwartete Suchzeit:  $\mathcal{O}(\frac{1}{n} \sum_{i=0}^{m-1} \sum_{j=n}^{l_i} (1 + j)) = \mathcal{O}(\frac{1}{n} \sum_{i=0}^{m-1} \frac{l_i(l_i+1)}{2})$ .

Satz: Sei  $S$  eine zufällige gleichverteilte Teilmenge aus  $\mathcal{U}$  der Größe  $n$ : Dann ist die erwartete Zugriffszeit auf ein zufälliges  $x \in S$ .

$$\leq 1 + \beta \cdot \frac{3}{2} \cdot e^\beta \text{ (für nicht zu großes } \beta, \text{ z.B. } \beta = 1, \text{ ist das } \mathcal{O}(1)).$$

Wieder unbefriedigend: In der Anwendung ist weder  $S \subseteq \mathcal{U}$  zufällig, noch greift man typischerweise zufällig auf die Elemente in  $S$  zu.

**Universelles Hashing**

Sei  $\mathcal{H}$  eine Menge von Funktionen von  $\mathcal{U}$  nach  $\{0, 1, \dots, m-1\}$

**Definition:**

Für  $c > 1$  heißt  $\mathcal{H}$   $c$ -universell falls für alle  $x, y \in \mathcal{U}, x \neq y$  gilt:

$$\frac{|\{h \in \mathcal{H} : h(x) = h(y)\}|}{|\mathcal{H}|} \leq \frac{c}{m}$$

**Satz:**

Für  $a, b \in \{0, \dots, N-1\}$  sei  $h_{a,b} : x \rightarrow ((ax + b) \bmod N) \bmod m$  und sei  $N$  prim

**Dann ist die Klasse:**

$$\mathcal{H} = \{h_{a,b} : 0 \leq a, b \leq N-1\}$$

$$c\text{-universell mit } c = \frac{\lceil \frac{N}{m} \rceil}{\frac{N}{m}} \approx 1$$

**Beweis: Siehe Übung.**

/TODO Lösung der Übung hier einfügen.

**Satz:**

Benutzt man Hashing mit Verkettung und wählt  $h \in \mathcal{H}$  zufällig gleichverteilt (mit  $\mathcal{H}$   $c$ -universell), dann ist die **erwartete** Zugriffszeit für Zugriffe:

$$\mathcal{O}(1 + c \cdot \beta) \text{ für beliebige Mengen } S \subseteq U, |S| = n$$

**Beweis:**

Zeit für Zugriff auf  $x$

$$\leq 1 + \#\{y \in S \mid h(x) = h(y)\}$$

$$\text{Definiere } \delta_h(x, y) = \begin{cases} 1 & \text{falls } h(x) = h(y) \\ 0 & \text{sonst} \end{cases}$$

**Uns interessiert:**

$$\frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \sum_{y \in S} \delta_h(x, y) = \frac{1}{|\mathcal{H}|} \sum_{y \in S} \sum_{h \in \mathcal{H}} \delta_h(x, y)$$

Falls  $x \neq y$ : Anzahl der Hashfunktionen  $h \in \mathcal{H}$  die  $x$  in gleichen Hashtafeleintrag mappen wie  $y$   
 Sonst (falls  $x = y$ ):  $|\mathcal{H}|$

Da  $\mathcal{H}$   $c$ -universell:

$$\leq \sum_{y \in S} [\text{falls } x = y \text{ then } 1 \text{ else } \frac{c}{m}] \leq \left\{ \begin{array}{ll} 1 + \frac{c(n-1)}{m} & \text{für } x \in S \\ \frac{c}{m} \cdot n & \text{für } x \notin S \end{array} \right\} \leq 1 + c \cdot \beta$$

Noch besser wäre nicht nur **erwartete** Zugriffszeit  $\mathcal{O}(1)$ , sondern **worst-case** Zugriffszeit  $\mathcal{O}(1)$

=> Perfektes Hashing

**Vorlesung 8****Bisher:**

Wenn wir  $h$  aus einer geeigneten Familie von  $c$ -universellen Hashfunktionen zufällig wählen, ist erwartete Zugriffszeit  $\mathcal{O}(1)$  - bei Platz  $\mathcal{O}(n)$ .

**Egal** wie  $S$  aussieht.

**Jetzt:**

Garantierte Worst-Case Zugriffszeit von  $\mathcal{O}(1)$  bei Platz  $\mathcal{O}(n)$ .

## Perfektes Hashing

### Good to know:

$c$  bedeutet Anzahl der Kollisionen.

### Ziel:

Möchte  $h$  finden mit  $h(x) \neq h(y) \forall x, y \in S, x \neq y$  und  $h$  bildet in eine Hashtafel der Größe  $\mathcal{O}(|S|)$  ab.

### 1. Versuch:

#### Einstufiges Perfektes Hashing

$c_S(h) = |\{\{x, y\} \in \binom{S}{2} : h\}|$  //TODO hier fehlt was

$\mathcal{H}$  ...  $c$ -universelle Familie von Hashfunktionen.

$h : U \rightarrow \{0, \dots, m - 1\}$

Sei  $h \in \mathcal{H}$  zufällig gewählt.

### Satz:

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

//TODO Formel

### Beweis:

Definiere  $\delta_h(x, y) = \begin{cases} 1 & \text{falls } h(x) = h(y) \\ 0 & \text{sonst} \end{cases}$

$$c_s(h) = \sum_{\{x, y\} \in \binom{S}{2}} \delta_h(x, y)$$

$$\begin{aligned} E(c_S(h)) &= \sum_{\{x, y\} \in \binom{S}{2}} E(\delta_h(x, y)) = \sum_{\{x, y\} \in \binom{S}{2}} \underbrace{\Pr(\delta_h(x, y) = 1)}_{\leq \frac{c}{m}, \text{ da nur ein } \frac{c}{m} \text{ Anteil der Hashfunktion in } \mathcal{H} \text{ für fixe } x, y \text{ auf den gleichen Hashtafeleintrag mappen darf (Definition von } c\text{-Universalität)}} \leq \\ &\quad \sum_{\{x, y\} \in \binom{S}{2}} \frac{c}{m} = \binom{n}{2} \frac{c}{m} \end{aligned}$$

### Korollar:

Für  $m > c \binom{n}{2}$  gibt es ein  $h \in \mathcal{H}$  mit  $h|_S$  injektiv.

**Beweis:**

Durch Einsetzen folgt

$$E(c_S(h)) < \binom{n}{2} \frac{c}{c \binom{n}{2}} = 1$$

Da Erwartungswert  $< 1$ , muss mindestens eine Hashfunktion mit 0 Kollisionen existieren.

**Probabilistische Methode:**

Also  $E(c_S(h)) < 1$ . Da  $c_S(h)$  nur ganzzahlige Werte annimmt, muss es mindest ein  $h \in \mathcal{H}$  geben mit  $c_S(h) = 0$ . Andernfalls wäre  $E(c_S(h)) \geq 1$ .

Diese Korollar ist leider nicht besonders konstruktiv, auf den ersten Blick bleibt einem nichts anderes übrig, als alle  $h \in \mathcal{H}$  zu testen um ein  $h$  mit  $h|_S$  ( $h$  eingeschränkt auf  $S$ ) injektiv zu finden.

Laufzeit dafür wäre  $\mathcal{O}(|\mathcal{H}| \cdot n + m)$ .

Bislang zwei Nachteile:

1. Platzbedarf  $m \approx n^2$ , damit so eine Funktion überhaupt sicher existiert.
2. Die gewünschte Hashfunktion finden eher schwierig.

**Korollar:**

Falls  $m > 2c \binom{n}{2}$ , können wir in erwartet  $\mathcal{O}(n + m)$  Zeit ein  $h \in \mathcal{H}$  finden mit  $h|_S$  injektiv.

**Beweis:**

$$E(c_S(h)) \leq \binom{n}{2} \cdot \frac{2}{m} \leq \frac{1}{2}$$

Markovungleichung:  $Pr(X \geq c) \leq \frac{E(X)}{c}$  für nicht ... X //TODO ... fehlt

$$\Rightarrow Pr(c_S(h) \geq 1) \leq \frac{\frac{1}{2}}{\frac{1}{2}} = \frac{1}{2} //TODO kann das stimmen?$$

$$\Rightarrow Pr(c_S(h) = 0) \geq \frac{1}{2}$$

//TODO Text von bildet wähle  $h \in X$  zufällig - falls  $h|_S$  injektiv → fertig - sonst wiederhole

**Intuitiv:**

Für groß genug Hashtafeln gibt es injektive  $h$  für  $S$  bzw. für noch ein wenig größere kann man solche auch effizient finden.

Leider ist diese Konstruktion nicht so gut, da sie eine Hashtafel der Größe  $\sim n^2$  voraussetzt. Hätten jedoch gerne Platz  $\mathcal{O}(n)$ .

**Zweistufiges Perfektes Hashing**

//TODO Bild Zweistufiges Hashing

**Korollar:**

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

**Beweis:**

$$E(c_S(h)) \leq \binom{n}{2} \frac{c}{m} < \frac{1}{2} \frac{n(n-1)}{\frac{n-1}{2}} = n$$

Wieder nur Existenzbeweis, Konstruktion aber möglich durch leicht größere Hashtafel.

**Korollar:**

Falls  $m > (n-1)c$ , gilt für mindestens die Hälfte der  $h \in \mathcal{H}$ :  $c_S(h) \leq n$

**Beweis:**

Wie oben.

$\Rightarrow$  Wir können in erwartet  $\mathcal{O}(n+m)$  Zeit ein  $h$  finden mit  $s_S(h) \leq n$  und wir brauchen nur Hashtafel der Größe  $\mathcal{O}(n)$ . //TODO stimmt  $s_S$ ?

---

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

Inhalt des Hasheimers, der alle  $x \in S$  bekommt mit  $h(x) = i$

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

$$\text{Es gilt: } c_S(h) = \sum_i \binom{|B_i(h)|}{2}$$

Wir haben  $h$  so gewählt, dass  $c_S(h) \leq n$

$$\Rightarrow \text{es gilt auch } \sum_i \binom{|B_i(h)|}{2} \leq n$$

Für jeden Hasheimer  $B_i(h)$  erzeugen wir eine Sekundärhashfunktion & -tafel der Größe:

$$m_i > 2 \cdot c \binom{S_i(h)}{2}$$

welche den Inhalt von  $B_i(h)$  injektiv in  $T_i$  hasht.

Der Platzverbrauch als auch die notwendige Zeit zur Konstruktion ist  $\mathcal{O}(2 \cdot c \cdot \binom{S_i(h)}{2})$

Die Größe der Sekundärhashtabellen (& deren Konstruktionszeit) ist:

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


---

### **Algorithmus zur Konstruktion einer Zweistufigen perfekten Hashfunktion**

#### **Annahme:**

$S \subseteq U$ ,  $|S| = n$ , können  $c$ -universelle Familie von Hashfunktionen  $h : U \rightarrow [0, \dots, m-1]$  samplen.

1. Finde  $h : U \rightarrow \{0, \dots, c \cdot (n-1)\}$  welches auf  $S \leq n$  Kollisionen produziert [geht in erwartet  $\mathcal{O}(n)$  Zeit].
  2. Bestimme für  $i = 0, \dots, c \cdot (n-1)$   
 $B_i(h) = \{x \in S \mid h(x) = i\}$
  3. Für jedes  $i = 0, \dots, c \cdot (n-1)$  mit  $B_i(h) > 1$  finde  $h : U \rightarrow [0, \dots, \binom{S_i(h)}{2} \cdot c - 1]$  welches injektiv ist für  $B_i(h)$ . Das geht in  $\mathcal{O}\left(\binom{S_i(h)}{2}\right)$  pro Eimer  $i$
- 

Zugriff auf ein  $x \in S$

Wende  $h$  auf  $x$  an:  $h(x) = i$  (falls  $|B_i(h)| \leq 1 \Rightarrow$  fertig)

Wende  $h_i$  auf  $x$  an:  $h_i(x) \rightarrow$  Hashtafeleintrag der  $\leq 1$  Elemente enthält  
 $\Rightarrow$  Zugriffszeit  $\mathcal{O}(1)$

## Vorlesung 10

### 2. VL Abschnitt: Berechenbarkeit (und etwas Komplexität)

Uns interessieren Fragestellungen der Art:  
- Was kann man überhaupt berechnen?  
- Was sind geeignete Rechenmodelle?

---

Etwas Notation:

eindeutiges Alphabet  $\Sigma$   
oft  $\Sigma = \{0, 1\}$

Die Menge aller Wörter der Länge  $k$  über  $\Sigma$  bezeichnen wir als  $\Sigma^k$   
bspw.  $\{0, 1\}^3 = \underbrace{\{000, 001, 010, 011, 100, 101, 110, 111\}}$

Kanonische Reihenfolge (Lexikalische Reihenfolge)

Probleme: "Aufgaben" oder "Fragestellungen", die wir mit dem Rechner lösen / beantworten wollen.

#### Problem 1.1:

Eingabe: Eine binär kodierte Zahl  $q \in \mathbb{N}$ ,  $q \geq 2$

Ausgabe: ein binär kodierter Primfaktor von  $q$

Beispiel: Eingabe: 110 (die Zahl 6) Ausgabe: 010 (die Zahl 2) oder 011 (die Zahl 3)

Das Problem samt Lösung entspricht Relation

$R \subseteq \{0, 1\}^* \times \{0, 1\}^*$  mit  
 $R = \{(x, y) \in \{0, 1\}^* \times \{0, 1\}^* \mid x = \text{bin}(q), y = \text{bin}(p), p, q \in \mathbb{N}, q \geq 2, p \text{ prim, } p \text{ teilt } q\}$

Relation: da für //TODO

#### Problem 1.2

Eingabe: Zwei binär kodierte Zahlen  $a, b \in \mathbb{N}$

Ausgabe: binär kodierte Zahl  $c \in \mathbb{N}$  mit  $c = a \cdot b$

Dieses Problem entspricht einer Funktion  $f : \Sigma^* \rightarrow \Sigma^*$

Um die Zahlen  $a$  und  $b$  in der Eingabe unterscheiden zu können, führen wir ein Trennsymbol ein, also

$$\Sigma = \{0, 1, \#\}$$

Die kodierte Eingabe ist dann  $\text{bin}(a)\#\text{bin}(b)$

Die kodierte Ausgabe ist  $f(\text{bin}(a)\#\text{bin}(b)) = \text{bin}(c)$

---

Wir werden uns im folgenden mindestens mit JA / NEIN - Fragen beschäftigen, sogenannte Entscheidungsprobleme. Diese haben die Form:

$$f : \Sigma^* \rightarrow \{0, 1\}$$

wobei wir 0 als *NEIN* und 1 als *JA* interpretieren.

$$L = f^{-1} \subseteq \Sigma^*$$

Menge der Eingaben, die mit 1 beantwortet werden.

$L$  nennen wir auch eine Sprache

Problem 1.3 Graphenzusammenhang

Eingabe: Kodierung eines Graphen  $G(V, E)$

Ausgabe: das Zeichen 1, falls  $G$  zusammenhängend, ansonsten 0.

Graph könnte //TODO SIEHE VL RWTH AACHEN