Inhalt

I. Datenstrukturen für Mengen

- Disjunkte Mengen → Union / Find
  → Split / Find
- Wörterbuchproblem
  - Suchbäume } Insert, Delete, Lookup
  - Hashing }

II. Graphalgorithmen
- grundlegende Alg. (Zustandskomponenten, ...)
- Netzwerkalgorithmen
- planare Graphen

III. Algorithmische Geometrie
- grundlegende geometrische Probleme
  → Suchen

Methoden:

- Divide & Conquer
- Randomisierung → Zufallszahlen (würfeln)
  → Effizienz (Analyse)
  → Fehlerwahrscheinlichkeit ($< 1/2^n$)
- Laufzeitanalyse (normal: worst case)
  - erwartete Zeit
  - amortisierte Analyse

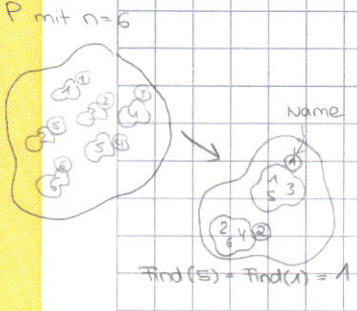
Ausgewählte Kapitel aus Algo & Data Strukt (VL)

I) Datenstrukturen für Mengen

1. Verwaltung disjunkter Mengen

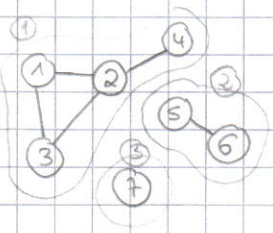
1.1. Das UNION-Find Problem

Problem Verwalte eine Partition (Menge von disj. Teilmengen) von Blöcken (Teilmengen) der Menge $U = \{1, \dots, n\}$ unter den Operationen

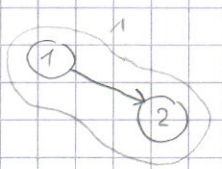


- **Init(n):** erzeugt der Partition $\{\{1\}, \{2\}, \dots, \{n\}\}$ Block $\{i\}$ erhält den Namen i
- **Find(x):** $x \in U$ liefert den Namen des Blocks, der x enthält
- **Union(A,B,C):** Vereinige die Blöcke mit Namen A und B zu einem Block C → A und B werden zerstört

1. Anwendung Zusammenhang ungerichteter Graphen $G=(V,E)$, $V = \{1, \dots, n\}$



UNION - FIND



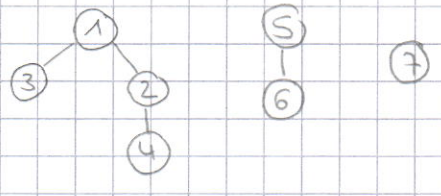

```
1. Init(n)
2. ∀(v,w) ∈ E do
3.       A ← Find(v);
4.       B ← Find(w);        ← besser: if A≠B then union(A,B,A)
5.       UNION(A,B,A)
6. od;
```

⇒ Funktion: SAME_COMPONENT(v,w)
return Find(v) = Find(w);

2. Spaning Tree → aufgespannter Baum (Wald)

berechne Teilmenge $E' \subseteq E$
$G'(V,E')$ ist azyklisch und hat die selben ZHK wie G

im Beispiel



Aufgabe: Berechne E' mit Union-Find

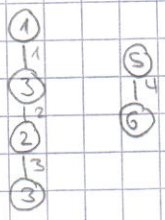
$E' \leftarrow E' \cup \{v,w\}$

3. Minimum Spanning Tree (MST)

Zusatz: Kantenkosten $\forall v,w \in E : cost(v,w)$

Berechne Spanning Tree mit minimalen Gesamtkosten
d.h. eine Menge E' (wie oben) mit $\sum_{e \in E'} cost(e)$ minimal

im Beispiel:



Kosten: 13

Lösungen für das Union-Find-Problem

1) Feld von Namen: array[1...n], das für jedes $i \in \{1,...,n\}$ den Namen des Blocks (Name[i]) angibt, das i enthält

Initialisierung: for i=1 to n do
Name[i] = i;
od
} $O(n)$

Find(x): return Name[x] } $O(1)$

Union(A,B,C): forall $x \in \{1,...,n\}$ do
if Name[x] $\in \{A,B\}$ then
Name[x] = C
} $O(n)$

2) Relabel the smaller half

zusätzliche Daten-(strukturen):
- ein Feld size: array[1,...,n] d.h. size[A] = |A|
- für jeden Block A eine lineare Liste L[A] realisiert durch ein Feld von Listenköpfen
L[A] enthält alle Elemente von A

Union(A,B,C):
- Gehe kürzere der beiden Listen A und B durch, etwa B, und ändere alle B's zu A's
- Konkateniere L[A] und L[B] (gehe in $O(1)$, falls Zeiger auf letztes Element
hier: neuer Block heißt nun A statt C
Ausweg: wir unterscheiden interne und externe Namen und benutzen 2 Felder HAPOUT und HAPIN zur Übersetzung interner Namen in externe Namen und umgedreht.
HAPOUT: intern → extern
HAPIN: extern → intern

Initialisierung:

```
Name[i] = i;
MAPIN[i] = i;
MAPOUT[i] = i;      } ∀ i ∈ {1, ..., n}      O(n)
size[i] = 1
L[i] = []
```

Find(x): return MAPOUT(Name[x]); } O(1)

Union(A,B,C): if size(A) < size(B) then

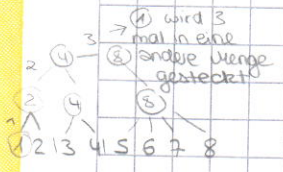
```
        x ← MAPIN[A]; y ← MAPIN[B];
    else
        x ← MAPIN[B]; y ← MAPIN[A];
    fi
    // ändere X's in Y's um
    forall z ∈ L[x] do
        Name[z] ← y;
    od
    L[y] ← L[x] conc L[y]    // hänge L[x] an L[y]

    MAPOUT[y] ← C;
    MAPIN[C] ← y;
    size[C] ← size[A] + size[B];
```

Laufzeit: $O(\min(|A|, |B|))$

Satz: Die Gesamtkosten einer beliebigen Folge von $n-1$ Unions und m Finds sind $O(m + n \log n)$

Beweis: Find hat Laufzeit $O(1)$ → m Finds haben Lz $O(m)$



Eine Operation Union(A,B,C) hat die Kosten $c \cdot$ Anzahl der Elemente $x \in \{1, \dots, n\}$ für die Name[x] geändert wird (c ist Konstante)

Bemerkung:

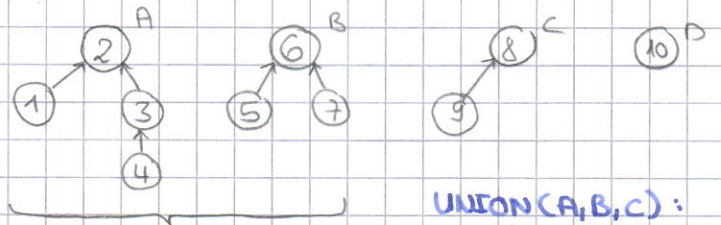
1. Falls für ein $x \in A$ Name[x] geändert wird, dann gehört x danach zu einem Block, der Größe $\geq 2 \cdot |A|$

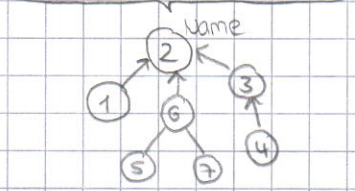
2. Am Anfang gehört jedes x zu einem Block der Größe 1, am Schluss zu einem Block der Größe $\leq n$
   ⇒ Für jedes $x \in \{1, \dots, n\}$ gilt:
   Name[x] wird höchstens $\log n$ - mal geändert
   ⇒ Insgesamt passierten höchstens $n \log n$ Namensänderungen. ⇒ Kosten aller UNIONs = $c \cdot n \log n = O(n \log n)$

Lösungen 1+2 begünstigen FINDS, die nächsten Lösungen werden Unions begünstigen.

3) Jeder Block wird durch einen Baum dargestellt, die Kinder repräsentieren die Elemente des Blocks, in der Wurzel steht der Name

Beispiel:





Find(x):
starte im Knoten des x darstellt und laufe bis zur Wurzel. Sie enthält den Namen des Blockes der x enthält
Kosten: O(Tiefe(x) im Baum)

UNION(A,B,C):
Hänge Wurzel von B zu Kind der Wurzel von A (oder umgekehrt)
Schreibe C in Wurzel (neuer Name)
⇒ Kosten: O(1)

Realisierung der Bäume durch Felder:

$$Vater[i] = \begin{cases} \text{Vater von } i \text{ im Baum} \\ 0, \text{ falls } i \text{ Wurzel} \end{cases}$$

Name[i] = Name des Blocks mit Wurzel i
(hat nur Bedeutung wenn i Wurzel ist)

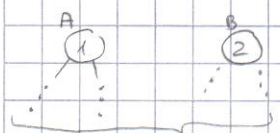
Wurzel[A] = Wurzel des Blocks mit Namen A

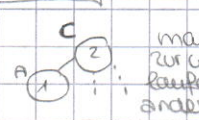
Initialisierung:

$$\left.\begin{array}{l} Vater[i] = 0 \\ Name[i] = i \\ Wurzel[i] = i \end{array}\right\} \forall i \in \{1,\dots,n\}$$

```
Find(x): while Vater[x] ≠ 0 do
             x ← Vater[x]
         od
         return Name[x]

Union(A,B,C): ω1 ← Wurzel[A];
              ω2 ← Wurzel[B];
              Vater(ω1) ← ω2;   // hier: A wird an B gehängt
              Wurzel[C] ← ω2;
              Name[ω2] ← C;
```





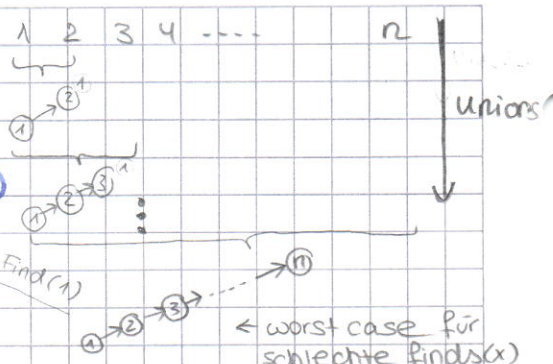
man muss bis zur Wurzel nachlaufen, da an anderen Stellen eventuell der falsche alte Name steht und nur die Wurzel den aktuellen Namen hat

Laufzeitanalyse:

Union: $O(1)$

Find(x): O(Höhe des Baumes)

worst case: $O(n)$



← worst case für schlechte finds(x) (Baum der Höhe n-1)

4) Verbesserung zu 3):

Vermeide große Tiefen dadurch, dass immer Wurzel des kleineren Baumes zum Kind der Wurzel des größeren Baumes gemacht wird (weighted union rule)

zusätzliches Feld:

size[i] = # der Knoten im Baum mit Wurzel i

Initialisierung

size[i] = 1;
Rest analog zu 3)

UNION(A,B,C):

$w_1 \leftarrow$ Wurzel[A];
$w_2 \leftarrow$ Wurzel[B];

if size[w_1] < size[w_2] then

Vater[w_1] $\leftarrow w_2$;
size[w_2] $\leftarrow$ size[w_1] + size[w_2];
Wurzel[C] $\leftarrow w_2$;
Name[w_2] $\leftarrow$ C;

else

analog mit w_1 und w_2 vertauscht

Find(x): analog zu 3)

Satz: Bei Union-Find mit weighted union rule hat eine Folge von n-1 unions und m Find's die Gesamtkosten

$O(n + m \log n)$

↖ amortisierte Laufzeit ↔ worst case

Lemma: Für alle Knoten x gilt:

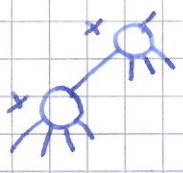
$$\text{gewicht}(x) \geq 2^{\text{Höhe}(x)}$$

Beweis
Induktion über Höhe(x)

Höhe(x) = 0 $\Rightarrow$ x ist Blatt
$\Rightarrow$ Gewicht(x) = 1 = $2^{\text{Höhe}(x)}$

Sei nun Höhe(x) > 0 : Sei y ein Kind von x mit Höhe(y) = höhe(x) − 1, (siehe Def von Höhe(x))

I.A. $\Rightarrow$ gewicht(y) $\geq 2^{(\text{höhe}(y)}$



Betrachte die Union-Operation die y zum Kind von x gemacht hat.
Seien $\overline{\text{gewicht}(x)}$, $\overline{\text{gewicht}(y)}$ die Gewichte vor dieser Operation

Dann gilt

1. $\text{gewicht}(y) = \overline{\text{gewicht}(y)}$ nur für Wurzeln kann sich Gewicht ändern
2. $\overline{\text{gewicht}(x)} \geq \overline{\text{gewicht}(y)}$ weighted union
3. nach der Union Operation

$$\text{gewicht}(x) \geq \overline{\text{gewicht}(x)} + \overline{\text{gewicht}(y)}$$
$$\geq 2 \cdot \overline{\text{gewicht}(y)} \quad \text{(wegen 2)}$$
$$= 2 \cdot \text{gewicht}(y) \quad \text{(wegen 1)}$$
$$\text{I.A.} \quad \geq 2 \cdot 2^{\text{höhe}(y)} = 2^{\text{höhe}(y)+1} \quad \text{(weil von y)}$$
$$= 2^{\text{höhe}(x)}$$

Da ja das Gewicht(x) $\leq n \ \forall x$ (insgesamt: n Knoten), folgt

$$n \geq \text{gewicht}(x) \overset{\text{Lemma}}{\geq} 2^{\text{höhe}(x)}$$

$$\Rightarrow 2^{\text{höhe}(x)} \leq n \mid \log$$

$$\Rightarrow \text{höhe}(x) \leq \log n$$

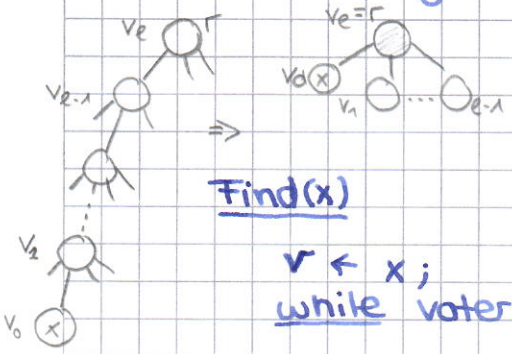
$\Rightarrow$ Kosten einer bel. FIND-Operation sind $O(\log n)$ (worst case)!
$\rightarrow$ Satz ■ m Finds kosten $O(m \log n)$

5) Verbesserung 4)

weighted union + Pfad Komprimierung (path. compression)

→ Find(x) durchläuft einen Pfad $v_0, \dots, v_e$ von Knoten wobei $v_0 = x$, v_e = Wurzel r

Idee: Hänge $v_0, \dots, v_{e-2}$ direkt an die Wurzer r



Find(x)

```
r ← x;                                   // r ist Wurzel
while vater[r] ≠ 0 do
        r ← vater[r];
od
                    (≠ r)
while vater[x] ≠ 0 do
        p ← vater[x];
        vater[x] ← r;
        x ← p;
od
```

Beobachtung:

Pfadkompremierung:

- verteuert aktuelle Find aber nur um konstanten Faktor → $O(\log n)$
- macht spätere Finds erheblich billiger

Satz (Tarjan)

Bei Union-Find mit weighted Union und path compression hat eine beliebige Folge von $n-1$ Unions und $m \geq n$ Finds die Gesamtkosten $O(m \cdot \alpha(m,n))$

Eine Variante der Ackermann-Funktion und ihre Inverse

$A : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$

$A(i,0) = 0$ für $i \geq 0$
$A(0,x) = 2x$ für $x \geq 0$
$A(i,1) = 2$ für $i \geq 0$

} Verankerung

Für $i \geq 0, x \geq 1$: $A(i,x) = A(i-1, A(i,x-1))$

Wertetabelle für A(i,x)

i \ x	0	1	2	3	4	5	6	7	8	…
0	0	2	4	6	8	10	12	14	…	
1	0	2	4	8	16	32	64	…		
2	0	2	4	16	2^{16} $=65536$	2^{65536}				
3	0	2	4							
4	0	2								

"Zeilenfunktion"

$2x = A(0,x)$

$2^x = A(1,x)$

$2^{2^{\cdot^{\cdot^{2}}}}\}x = A(2,x)$

$A(1,2) = A(0, A(1,1)) = A(0,2) = 4$

$A(3,4) = 2^{2^{\cdot^{\cdot^{2}}}}\}65536 \cdot x$

Inverse zu A

$$\alpha(m,n) = \min\{ i \mid A(i, \lfloor \tfrac{m}{n} \rfloor) > \log n \}$$

Bsp.: $\alpha(24,8) = \min\{ i \mid A(i,3) > 3 \} = 0$

$\alpha(3 \cdot 2^{16}, 2^{16}) = \min(i \mid A(i,3) > 16\} = 3$

Man sieht leicht, dass $\alpha(m,n) \leq 3$ für alle in der Praxis vorkommenden Werte für m&n (→ Übung)

Beweis des Satzes

Situation: n Knoten (Elemente)

$m > n$ Folge von n-1 Unions + m Finds

$U_1 F_1 F_2 U_2 U_3 F_3 \ldots U F \ldots F_m U_{n-1}$

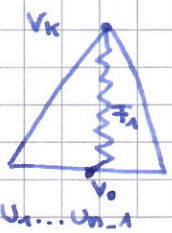
Am Ende: Einen Baum T'

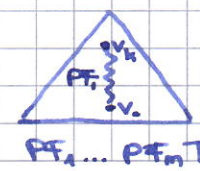
Können wir auch so erhalten: (konzeptionell)

a) Führe zunächst alle n-1 Unions aus → Baum T (log Tiefe)

b) Führe auf T m partielle Finds (partial finds) aus $PF_1, \ldots PF_m$

PF_i simuliert das echte Find F_i, indem es den selben Pfad wie F_i durchläuft (dh. i.A. nicht bis zur Wurzel)





Wir schätzen nun die Kosten der Folge von Unions und partielles Finds ab.

Sei $\mathcal{F}$ die Multi-Menge (Menge in denen Elemente auch mehrfach vorkommen können) der Kanten, die in allen PFs durchlaufen werden.

zu zeigen: $|\mathcal{F}| = O(m \cdot \alpha(m,n))$

Idee

a) Teile $\mathcal{F}$ in Gruppen nach Rang der Endpunkt der Kanten ein (Rangdifferenz)

$$\text{Rang}(x) = \text{Höhe von } x \text{ im Baum } T \quad (\text{nicht } T')$$

Rang(x) ist konstant (in Folge der PF's)

b) schätze alle Gruppen ab

Definitionen

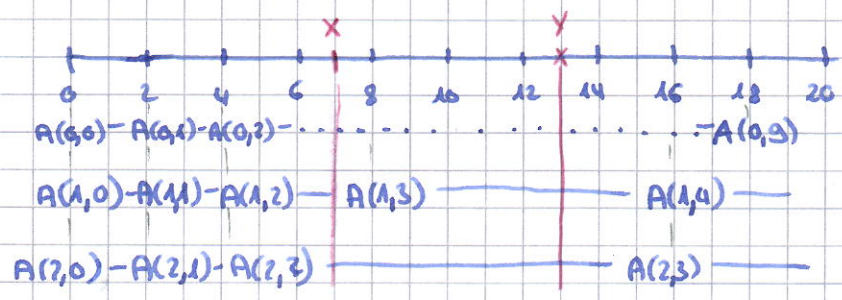
zunächst "Einteilung" der Knoten (nicht disjunkt)

Sei $z \in \mathbb{N}$ (wird später definiert $\rightarrow \alpha(m,n)$)

Für $0 \le i \le z$, $j \ge 0$ definiere

$$G_{ij} = \{\text{Knoten } x \mid A(i,j) \le \text{Rang}(x) < A(i,j+1)\}$$

Veranschaulichen $\{[A(i,j) \dots A(i,j-1)]\}$: Menge von Intervallen



Beispiel: $\text{Rang}(x) = 7$, $\text{Rang}(y) = 13$

$\Rightarrow x \in G_{0,3}, G_{1,2}, G_{2,2}, \dots$

$y \in G_{0,6}, G_{1,3}, G_{2,2}, \dots$

Einteilung der Kanten in $\mathcal{F}$

$$N_k = \{(x,y) \in \mathcal{F} \mid k = \min\{i \ge 0 \mid \exists j \text{ mit } x,y \in G_{ij}\}\}$$

für $0 \le k \le 2$ im Bsp $(x,y) \in N_2$

und

$$N_{z+1} = \mathcal{F} \setminus \bigcup_{k=0}^{z} N_k \quad (\text{Rest})$$

Intuitiv: $(x,y) \in N_i \Rightarrow$ Rangdiff$(x,y) \approx A(i,\dots)$? $\Rightarrow$ Übung)

Im Beispiel: Rang(x)=7, Rang(y)=13

$(x,y) \in N_2$ → erstes mal in selber Menge

$G_{2,2} =$ A(1,..) ... ...
A(2,2) ... A(2,3)
4 16

Schließlich definiere:

$L_k := \{(x,y) \in N_k \mid (x,y)$ ist letzte (oberste) Kante auf PF-Pfad$\}$

für $0 \le k \le z+1$

Lemma

a) $|L_k| \le m$ für $0 \le k \le z+1$

b) $|N_0 \setminus L_0| \le n$

c) $|N_k \setminus L_k| \le \frac{5}{8} n$ für $0 \le k \le z$

d) $|N_{z+1} \setminus L_{z+1}| \le n \cdot a(z,n)$, wobei $a(z,n) = \min\{i \mid A(z,i) > \log n\}$

Beweis

zu a) Für jedes PF gibt es höchstens 1 Kante in L_k (eine letzte Kante)
Da m PF's ⇒ Teil a

zu b) Sei $(x,y) \in N_0 / L_0$, dann gilt:

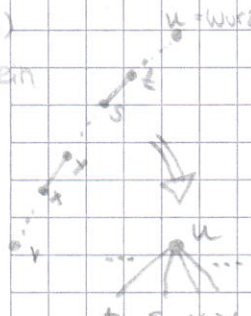
1) $\exists j$ (Spalte) mit $x,y \in G_{0,j}$

dh $A(0,j) \le$ Rang(x) < Rang(y) $\le A(0,j+1)$ (Übung)
$A(0,j) = 2^j$, $A(0,j+1) = 2^{(j+1)}$

⇒ Rang(x) = 2^j und Rang(y) = 2^j+1
⇒ dh. Rangdifferenz = 1

2) $(x,y) \notin L_0$ ⇒ war nicht letzte Kante auf PF-Pfad

PF-Pfad:
PF(v)
es kann sein y=s



Betrachte PF (dess (x,y) noch N_0 brachte

Dann $\exists (s,t) \neq (x,y)$ auf PF-Pfad mit $(s,t) \in N_0$
mit Rang(x) = 2^j
Rang(y) = 2^j+1
Rang(s) ≥ Rang(y)
Rang(t) > Rang(s) ⇒ Rang(t) ≥ 2^j+2
Rang(u) = Rang(s)+1

Nach PF: x hat neuen Vater u (u=t ist möglich)
⇒ Rangdifferenz(x,u) ≥ 2
⇒ Spätere PFs können keine Kante (x,u) zu N_0 hinzufügen

$\Rightarrow$ Für alle Knoten x gilt es höchstens eine Kante $(x,y) \in N_0 \setminus L_0$

$\Rightarrow |N_0 \setminus L_0| \leq n$

zu c)

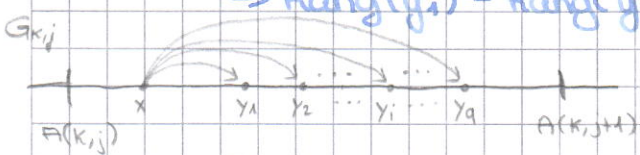
wie oben: Schätze Beitrag eines Knotens $x \in G_{k,j}$ zu $|N_k \setminus L_k|$ ab
dh. alle Kanten $(x, \dots)$ mit $\dots$

Sei $k \geq 1$ und $x \in G_{k,j}$ beliebig (Spalte j)
dh $A(k,j) \leq \text{Rang}(x) < A(k,j+1)$

und $y_1, y_2, \dots, y_q$ alle Endknoten mit $(x, y_i) \in N_k \setminus L_k$ in der Reihenfolge in der sie von PFs benutzt werden

dh. y_{i+1} ist Wurzel des PF von (x, y_i)

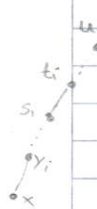
$\Rightarrow \text{Rang}(y_1) \leq \text{Rang}(y_2) \leq \dots \leq \text{Rang}(y_q) < A(k, j+1)$



Dann gilt:

i) $j \geq 2$ denn sonst wär $k = 0$ (min)

ii) $(x, y_i) \notin L_k$ für $i = 1, \dots, q$ $\Rightarrow \exists (s_i, t_i) \in N_k$ oberhalb von (x, y_i) auf PF



$\Rightarrow \text{Rang}(x) < \text{Rang}(y_i) \leq \text{Rang}(s_i) < \text{Rang}(t_i) \leq \text{Rang}(y_{i+1})$

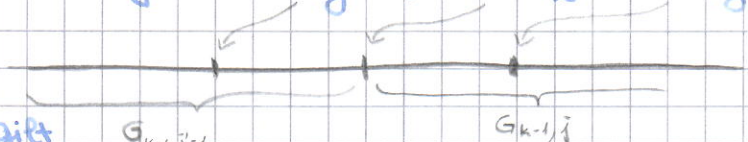
nach Pfadkompremierung: x hat Vater y_{i+1}

Definition von N_k (k minimal)

$\Rightarrow (x, y_i), (s_i, t_i) \in N_{k-1}$

$\Rightarrow \exists j'$ mit $\text{Rang}(s_i) < A(k-1, j') \leq \text{Rang}(t_i)$

Zeile $n-1$



und daher gilt

$\text{Rang}(y_i) < A(k-1, j) \leq \text{Rang}(y_{i+1})$

Iteriere: q-mal (für $y_1, \dots, y_q$)

$\Rightarrow \exists j''$ mit $\text{Rang}(y_i) < A(k-1, j'') \leq A(k-1, j''+q-1) \leq \text{Rang}(y_q)$

Umbenennung

(I) $\exists j$ mit $\text{Rang}(y_q) \geq A(k-1, j+q-1)$

Beobachtung: $j \geq 2$ (siehe oben)

Teile: $1 \leq k \leq z$

$x \in G_{n,j}$, $(x,y) \in N_T \Rightarrow y_1, \dots, y_q \in G_{k,j}$

$A(k,j) \leq Rang(x_1) \leq \dots \leq Rang(x_q) < A(k,j+1)$

(II) $Rang(x_q) < A(k,j+1)$

(I+II) $A(k-1, j+q-1) < A(k,j+1) = A(k-1, A(k,j))$

Monotonie
$\Rightarrow \quad j+q-1 < A(k,j)$

$j \geq 2$
$\Rightarrow \quad q < A(k,j)$

Das bedeutet (Zusammenfassung)

Für jedes $x \in G_{k,j}$, $k \geq 1$, $j \geq 2$ gibt es höchstens $A(k,j)$ Kanten $(x,y) \in N_k \setminus L_k$

$$\Rightarrow |N_k \setminus L_k| \leq \sum_{j \geq 2} |G_{k,j}| \cdot A(k,j)$$

* Behauptung: $|G_{k,j}| \leq \frac{2n}{2^{A(k,j)}}$

Daraus folgt:

$$|N_k \setminus L_k| \leq \sum_{j \geq 2} \frac{2n \cdot A(k,j)}{2^{A(k,j)}} \leq 2n \cdot \sum_{j \geq 2} \frac{A(k,j)}{2^{A(k,j)}}$$

Es gilt: da $k \geq 1$, $A(k,j) \geq 2^j$

$$= 2n \sum_{j \geq 2} \frac{2^j}{2^{2j}} = 2n \sum_{j \geq 2} \frac{1}{2^{2j-j}} = 2n \cdot \underbrace{\left(\frac{1}{4} + \frac{1}{32} + \frac{1}{2^{12}} + \dots \right)}_{\leq 1/16} \qquad j = 2, 3, 4$$

$$\leq 2n \cdot \frac{5}{16} = \frac{5}{8} n$$

* Beweis der Behauptung: $|G_{k,j}| \leq \frac{2n}{2^{A(k,j)}}$ mit $k \geq 1$, $j \geq 2$

$G_{k,j}$ = Knoten mit Rang im Intervall $[A(k,j) \dots A(k,j+1)[$

Sei ℓ eine bel. natürliche Zahl mit $A(k,j) \leq \ell < A(k,j+1)$

Wir zählen alle $x \in G_{k,j}$ mit $Rang(x) = \ell$

Sei $G_{k,j,\ell} = \{ x \in G_{k,j} \mid Rang(x) = \ell \}$

Es gilt:

i) Jeder Knoten x mit Rang ℓ hat $\geq 2^\ell$ Nachkommen (Gewicht von $x \geq 2^\ell$) ← weighted union

ii) Für x, y mit $Rang(x) = Rang(y)$ und $x \neq y$ sind die Nachfolgermengen disjunkt

i)+ii) $\Rightarrow |G_{k,j,\ell}| \leq \sum_{\ell = A(k,j)}^{A(k,j+1)-1} |G_{k,j,\ell}| \leq \sum_{\ell = A(k,j)}^{\infty} \frac{n}{2^\ell} = n \cdot \sum_{\ell = A(k,j)}^{\infty} \frac{1}{2^\ell}$

$$= \frac{1}{2^{A(k,j)}} + \frac{1}{2} \cdot \frac{1}{2^{A(k,j)}} + \frac{1}{4} \cdot \frac{1}{2^{A(k,j)}} = \frac{n}{2^{A(k,j)}} \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \dots\right)$$

$$\leq \frac{2n}{2^{A(k,j)}}$$

zu d) $k = z+1$ auch hier Schranke q

weighted union $\Rightarrow$ Rang$(y_q) \leq \log n$ (da T' Höhe $\leq \log n$)

(I) für $k = z+1$

$\exists j$: Rang$(y_q) \geq A(k-1, j+q-1)$ mit $j \geq 2$

$\log n \geq$ Rang$(y_q) \geq A(z, j+q-1)$

$\Rightarrow A(z, j+q-1) \leq \log n$

$\Rightarrow j+q-1 \leq \alpha(z,n)$, da $\alpha(z,n)$ minimal mit $A(z, A(z,n)) > \log n$ $j \geq 2$

$\Rightarrow q \leq \alpha(z,n)$

Also gibt es höchstens $\alpha(z,n)$ Kanten (x,y) für jeden Knoten x in $N_{z+1} \setminus L_{z+1}$

$\Rightarrow |N_{z+1} \setminus L_{z+1}| \leq n \cdot \alpha(z,n)$ ■ a, b, c, d

Zurück zu Satz (Tarjan)

Kosten aller $m \geq n$ Finds

$$|F| = \underbrace{\sum_{k=0}^{z+1} |L_k|}_{a} + \underbrace{|N_0 \setminus L_0|}_{b} + \underbrace{\sum_{k=1}^{z} |N_k \setminus L_k|}_{c} + \underbrace{|N_{z+1} \setminus L_{z+1}|}_{d}$$

$$|F| \leq \underbrace{(z+2) \cdot m + n}_{A} + \underbrace{z \cdot \frac{5}{2} n}_{c} + \underbrace{n \cdot \alpha(z,n)}_{B}$$

Das gilt für jedes z!

Betrachte $z = \alpha(m,n)$, $m \geq n$

i) $A = O(m \cdot \alpha(m,n))$

ii) $\alpha(z,n) = \alpha(\alpha(m,n), n) = \min\{j \mid A(\alpha(m,n), j) > \log n\}$ Def von α

mit $\alpha(m,n) = \min\{i \mid A(i, \lfloor \frac{4m}{n} \rfloor) > \log n\}$

$$= \leq \left\lfloor \frac{4m}{n} \right\rfloor$$

$\Rightarrow$ (B) $\leq n \cdot \left\lfloor \frac{4m}{n} \right\rfloor = O(m)$

(A) + (B) $= O(m \cdot \alpha(m,n))$

Satz: $n-1$ Unions und $m \geq n$ Finds kosten $O(m \cdot \alpha(m,n))$

i) nicht linear $\longleftrightarrow$ praktisch: $O(m)$, da $\alpha(m,n) \leq 3$ für realistische Werte

ii) praktisch sehr effizient (kleine Konstante)

iii) leicht implementierbar, Korrektheit einfach

iv) obere Schranke ist scharf d.h. es gibt Probleminstanzen mit dieser Laufzeit

v) die Komplexität des Problems ist tatsächlich

$\Omega(m \cdot \alpha(m,n))$ untere Schranke

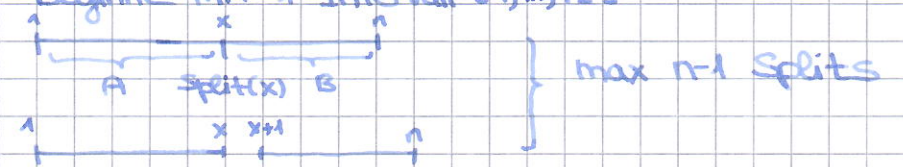
Union/Find $\longleftrightarrow$ Split/Find (→ Übung)

↓

Mengen sortiert, Spliten an einer Stelle mit $\leq, >$

⇒ Menge von Intervallen (disjunkt) von x in Teile $A \leq x$ und $B > x$

beginne mit 1 Intervall $[1, \ldots, n]$



Finds wie vorher

2. Das Wörterbuchproblem

Verwalte eine Menge S von Schlüsseln aus einem Universum U unter den Operationen:

Insert(x) : $S \leftarrow S \cup \{x\}$ mit $x \in U$

Delete(x) : $S \leftarrow S \setminus \{x\}$

LookUp(x) (Member) $\begin{cases} \text{true} & : x \in S \\ \text{false} & : \text{sonst} \end{cases}$

Eigentliches Wörterbuch: speichere mit jedem Schlüssel $x \in S$ eine bel. Information Inf(x) (Satellitendaten)

$\Rightarrow$ Insert(x,i), Delete(x), LookUp(x) $\rightarrow$ liefert Inf(x) falls $x \in S$

Man unterscheidet 2 Typen von Datenstrukturen abhängig von U

i) U ist linear geordnet
speziell: Ordnung "$\leq$" d.h. es existiert eine Vergleichs-Operation auf U
$\Rightarrow$ Dynamische Suchbäume (AVL-Bäume)

ii) Schlüssel sind ganze Zahlen aus $[1 \dots N]$ $n := |S|$
d.h. $U = \{1, \dots, n\}$ $\rightarrow$ Hashing

einfach bei $N = O(n)$ $\rightarrow$ einfaches Feld, sonst Hashing

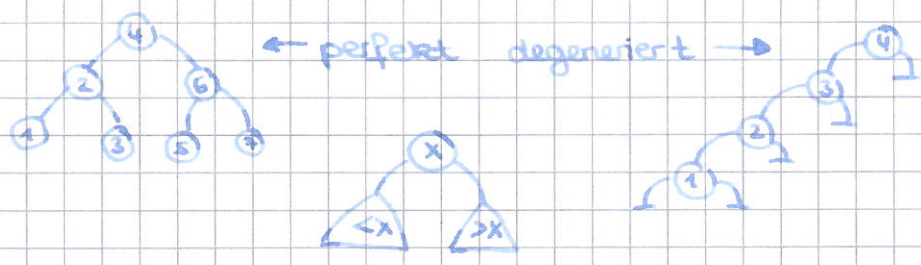
2.1 U ist linear geordnet

Randomized Search Trees (Aragon/Seidel 1990)

Idee: Benutze einen Zufallsprozess (Zufallszahlen) zur Balancierung eines binären Suchbaumes

Vorteile:
- extrem einfache Implementierung
- geringer Aufwand für Balance-Informationen
- Effizient: logarithmische Kosten pro Operation (mit sehr hoher W'keit)

Binär-Suchbaum (blattorientiert)



$\rightarrow$ Informationen Inf(x) werden erstmal Ignoriert

Definition: Randomized Search Tree (RST)

Sei $S = \{x_1, \dots, x_n\}$ eine Menge von Schlüsseln aus einem Universum U mit linearer Ordnung "$\leq$"
Jedem $x_i \in S$ wird zusätzlich eine Zufallszahl, auch Priorität genannt, $prio(x_i)$ zugeordnet.
Genauer: $prio(x_i)$ sind gleichverteilt aus $[0,\dots,1]$ oder aus einem hinreichend großen Intervall der ganzen Zahlen z.B.: $\{0,\dots,2^{32}-1\}$

Ein RST T für S ist ein knotenorientierter binärer Suchbaum für die Paare $(x_i, prio(x_i))$, $1 \leq i \leq n$, so dass:

i) T normaler binärer Suchbaum bzgl. der Schlüssel x_i
d.h. Schlüssel erfüllen die in-order-Bedingung

$\forall$ Knoten $v \in T$ gilt

a) $\forall$ Knoten $w \in T_\ell(v)$: $key(w) < key(v)$
b) $\forall$ Knoten $u \in T_r(v)$: $key(u) > key(v)$

ii) T ist ein Heap (Max-Heap) bzgl der Prioritäten,
d.h. $prio(v) \geq prio(w)$, falls v Vater von w (Vorfahr)

→ Wurzel enthält max Priorität
→ $\forall$ Suchpfade bilden Prioritäten fallender Folge

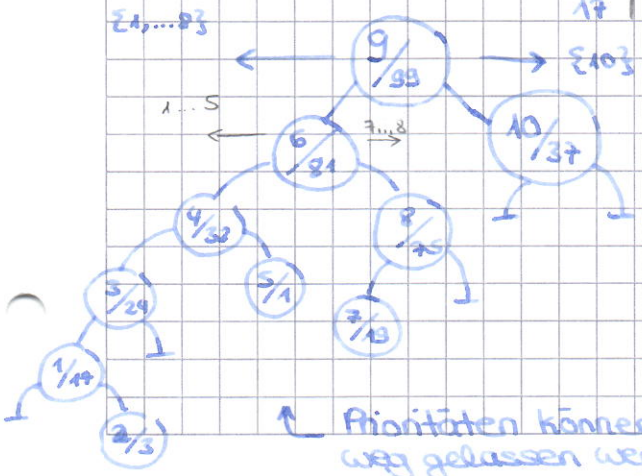
Aufbau eines RST für $S = \{(x_1, prio(x_1)=p_1), \dots, (x_n, p_n)\}$
(→ Existenzbeweis)

1. Speichere Paar (x_i, p_i) mit max. Priorität in Wurzel r
2. Baue rekursiv linken Unterbaum
$T_\ell(r)$: RST für $S' = \{(x_j, p_j) \in S \mid x_j < x_i\}$
3. Baue rekursiv rechten Unterbaum
$T_r(r)$: RST für $S' = \{(x_k, p_k) \in S \mid x_k > x_i\}$

Beispiel: Schlüssel: $\{1, \dots, 10\}$

$T_\ell(6)$ $T_r(6)$ $T_\ell(3)$ $T_r(8)$

Prioritäten:	1	2	3	4	5	6	7	8	9	10
	17	3	24	32	1	81	13	75	93	37

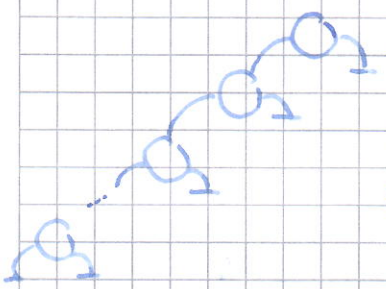


alternativ:

füge die Schlüssel in absteigender Reihenfolge ihrer Priorität in einen normalen (unbalancierten) Suchbaum ein

↑ Prioritäten können weg gelassen werden
→ werden nur für den Aufbau benötigt

Intuition: Wie sieht im schlechtesten Fall der Baum aus?



$\forall i \neq j \quad prio(x_i) \geq prio(x_j) \iff x_i > x_j$

d.h.: für alle

Extrem unwahrscheinlich, wenn Priorität zufällig

RST = Treat + Heap → Treap

Operationen

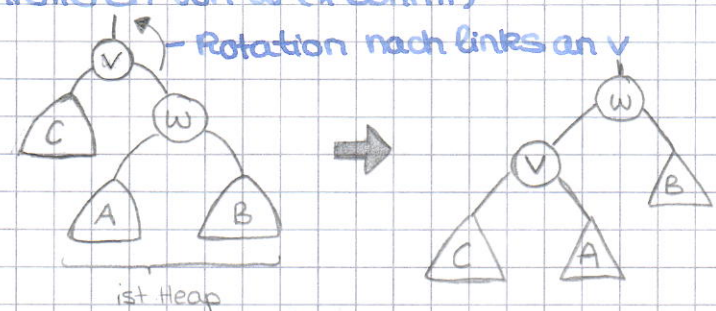
1. Look_up(x) : normale Suche in bin. Baum

   Kosten $O(\text{Höhe}(T))$ ✓

2. Insert(x) :

   1. wähle prio(x) zufällig aus [0,1] (reelle Zahlen)
      ↗ wenn schon da, tue nichts
   2. Füge einen neuen Knoten (Blatt) (x, prio(x)) gemäß dem Schlüssel x in T ein (normales Insert)
      i.A. ist hier die Heapeigenschaft verletzt
   3. Rotiere w nach oben bis die Heap-Ordnung wieder hergestellt d.h. $prio(vater(w)) \geq prio(w) \geq prio(kinder(w))$ (beide)

      Hochrotieren von w (1 Schritt)

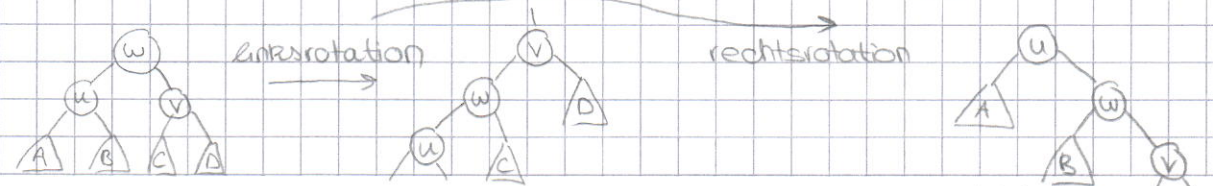


Korrektheit → Übung

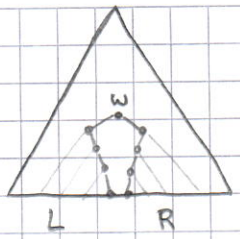
Intuition: Da prio(w) zufällig & gleichverteilt, ist die W'keit für viele Rotationsschritte sehr klein.
Insbesondere: $prob(w: \text{neue Wurzel}) \leq 1/n$
W'keit → w wird zur neuen Wurzel → viele Rotationen

3. Delete(x)

   1. Lookup liefert Knoten w mit key(w) = x (sonst tue nichts)
   2. Rotiere w nach unten "bis es ein Blatt von T ist.



⇒ man hat immer 2 Möglichkeiten unter w zu rotieren, wenn 2 Kinder vorhanden sind



Jede Rotation vermindert diese Knotenmengen

L = rechtes Rückgrat des linken Unterbaums (nach rechts)

R = linkes —"— rechten UB (nach rechts)

3. Entfernen des Blattes w

Einschub: zusätzliche Operationen

SPLIT(y) → $S_1 = \{x \in S \mid x \leq y\}$
$S_2 = \{x \in S \mid x > y\}$

1) Insert(y) mit Priorität ∞ (im Intervall $[0,1] \to 1$)
↳ nicht zufällig
=> y steht in der Wurzel von T (Heap')

Umkehroperation

→ T_1 für S_1 = linker Unterbaum
T_2 für S_2 = rechter Unterbaum

Join(T_1, T_2) : $S \leftarrow S_1 \cup S_2$ wobei S_1 Menge von T_1, S_2 Menge von T_2

Voraussetzung: $\max(S_1) < \min(S_2)$

Wähle ein Schlüssel x mit $\max(S_1) < x < \min(S_2)$

1) Konstruiere folgenden Baum T

2) Delete(x)

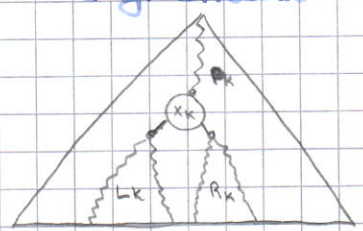
Laufzeitanalyse

Wir analysieren die erwarteten Kosten einer Delete-Operation
(Insert = Delete^{-1})

Sei T ein RST für die Menge $S = \{x_1, \dots, x_n\}$ mit $x_1 < \dots < x_n$, dh.: ist entstanden durch Insert-Operationen)

Betrachte die Operation Delete(x_k) mit $1 \leq k \leq n$.

allg. Situation:



P_k: Suchpfad nach x_k
L_k: rechter Rückgrat des linken UB von x_k
R_k: linker —"— rechten —"—

Kosten: $O(|P_k| + \underbrace{|L_k| + |R_k|}_{\text{runter Rotieren}})$

$|P_k|$: Lookup

$O(|L_k| + |R_k|)$ bis x_k ein Blatt

runter Rotieren: { links: $|R_k| - 1$, rechts: $|L_k| - 1$

Wir schätzen die Erwartungswerte ab für $|P_k|, |L_k|, |R_k|$

Lemma 1

Sei $S = \{x_1, \dots, x_n\}$ mit $x_i < x_{i+1}$ für $i = 1, \dots, n-1$ und $\{prio(x_i) \mid i = 1, \dots, n\}$ gleich verteilte Zufallszahl aus $[0,1]$ abgespeichert im RST T (durch Folge von Inserts)

Betrachte den Knoten der einen Schlüssel x_k enthält

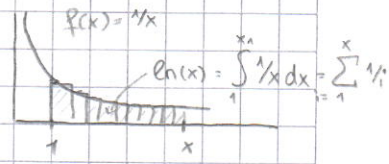
Dann gilt:

a) $E(|P_k|) = H_k + H_{n-k+1} - 1$

b) $E(|L_k|) = 1 - 1/k$

c) $E(|R_k|) = 1 - 1/(n-k+1)$

mit $H_k = \sum_{i=1}^{k} 1/i$ (k-te harmonische Zahl $\approx \ln(k)$)



Teil fehlt

I) **Spiel A**: Kandidaten $K = \{x_1, \dots, x_k\}$

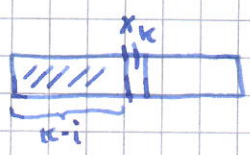
$$A^k = \frac{1}{k} \cdot 1 + \sum_{i=1}^{k-1} 1/k \, (1 + A^{k-i})$$

A^k: Spiel A mit Parameter k

$\frac{1}{k} \cdot 1$: im 1 Zug

$\sum_{i=1}^{k-1} 1/k \, (1 + A^{k-i})$: nicht x_k

$(1 + A^{k-i})$: Spiel mit $k-1$



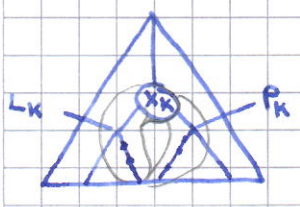
$$= \frac{1}{k} \sum_{i=1}^{k} (1 + A^{k-i}) \; \dots = H_k$$

II) $E(P_k'') = H_{n-k+1}$ Beweis: symmetrisch → Übung

Spiel B, Kandidaten $K = \{x_k, \dots, x_n\}$

zähle wie oft neues Minimum gezogen

b & c)



Seien $L_k = v_1, v_2, \dots, v_\ell$

$R_k = w_1, \dots, w_m$

Es gilt i) $key(v_i) < v_k$ $\quad H(w_i) > x_k$

ii) $key(v_i) > key(v_{i-1})$ $\quad key(w_i < key(w_{i-1})$

$E(|L_k|) =$ Erwartungswert für ℓ

$E(|R_k|) =$ ——— " ——— m

(Wie im Teil a) können wir annehmen, dass T normaler bin. Suchbaum mit zufälliger Einfügereihenfolge $x_{\pi(1)} \cdots x_k \cdots x_{\pi(n)}$

Erwartungswert von $|L_k|$ bzw. $|R_k|$

Ziehe zufällig Elemente aus $\{x_1, \dots, x_n\}$

sobald x_k gezogen werde (Trigger)

zähle wie oft an $x_i < x_k$ ($x_i > x_k$) gezogen wird, das

größer (kleiner) als alle vorher gezogenen Schlüssel

$E(|L_k|) = t(\ell)$

Spiel C K Kandidaten $\{x_1, \dots, x_k\}$ mit Auslöser x_k ($\leq x_k$)

$E(|L_k|) = C^k$ = Erwartungswert wie oft neues Maximum nach Ziehung von x_k (Auslöser) gezogen wird

$$C^k = \underbrace{1/k \cdot A^{k-1}}_{\text{im ersten Zug } x_k \text{ gezogen, dann Spiel A für die restl. } k-1 \text{ Kandidaten}} + \underbrace{\sum_{i=1}^{k-1} 1/k \cdot C^{k-i}}_{\text{im 1 Zug } x_i < x_k \rightarrow \text{Spiel C mit } k-i \text{ Kandidaten}}, \qquad C^0 := 0$$

$$= 1/k \left(H_{k-1} + \sum_{i=0}^{k-1} C^i \right) \quad \text{(Teil a)}$$

Trick: schätze $\Delta_j := C^{j+1} - C_j$ ab

$$\Rightarrow C^k = \sum_{j=1}^{k-1} \Delta_j + \underbrace{C^0}_{=0}$$

Betrachte: $(j+1) C^{j+1} - j \cdot C^j$

$$= (j+1) \cdot \frac{1}{j+1} \cdot \left(H_j + \sum_{i=0}^{j} C^i \right) - j \cdot 1/j \left(H_{j-1} + \sum_{i=0}^{j-1} C_i \right)$$

$$= H_j + \sum_{i=0}^{j} c^i - \left(H_{j-1} + \sum_{i=0}^{j-1} c^i\right) \qquad H_j = \sum_{i=1}^{j} 1/i$$

$$\Rightarrow 1/j + c^j = (j+1)\cdot c^{j+1} - j \cdot c^j \qquad 1 - c^j$$

$$1/j = (j+1)\cdot c^{j+1} - (j+1)\cdot c^j \qquad 1 \cdot \frac{1}{j+1}$$

$$\frac{1}{j\cdot(j+1)} = c^{j+1} - c^j$$

$$\rightarrow \Delta_j = \frac{1}{j\cdot(j+1)} = \frac{1}{j} - \frac{1}{j+1}$$

$$C^k = \sum_{j=1}^{k+1} \Delta_j = \sum_{j=1}^{k-1} \left(1/j - \frac{1}{j+1}\right) = 1 - 1/k \quad \blacksquare$$

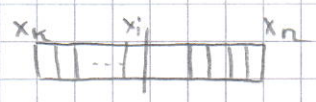
↖ Teleskop

Teil c $\quad E(|R_k|) = 1 - 1/(n-k+1)$

Symmetrisch zu b)

Spiel D : wie oft wird neues Minimum gezogen nachdem der Auslöser x_k gezogen wurde

$$D^k = \underbrace{\frac{1}{n-k+1}\cdot B^{k-1}}_{\text{im 1 Zug } x_k\text{, dann Spiel B für den Rest}} + \underbrace{\sum \frac{1}{n-k+1}\cdot D^{i-k}}_{\text{1 Zug } x_i > x_k}$$



$$D^k = \frac{1}{n-k+1}\left(H_{n-k+1} + \sum_{i=0}^{n-k} D^i\right), \quad D^0 = 0$$

Abschätzung : (üb) analog zu c)

$$= 1 - 1/(n-k+1) \quad \blacksquare$$

Satz: Sei T ein RST für eine Menge von n Schlüsseln. Dann gilt:

1. Die erwartete Laufzeit für Insert, Delete, LookUp ist $O(\log n)$.
2. Die erwartete Anzahl von Rotationen bei Insert und Delete ist < 2.

Beweis:

zu 1) Die Kosten von $\text{LookUp}(x_k) = O(|P_k|)$

Insert / Delete: $O(|P_k| + |L_k| + |R_k|)$ ($= \text{Höhe}$)

nach Lemma: $= H_k + H_{n-k+1} - 1 + 1 - 1/k + 1 - 1/(n-k+1)$

$\leq 2 H_n$ da $H_k \leq H_n$, $H_{n-k+1} \leq H_n$

$= O(H_n)$

$= O(\ln n) = O(\log n)$

zu 2) Erwartete Zahl der Rotationen:

$E(|L_k|) + E(|R_k|)$

nach Lemma: $= 1 - 1/k + 1 - 1/(n-k+1) < 2$

3. Hashing

(spezielle Datenstrukturen für Wörterbücher)

Schlüsselmenge S sind ganze Zahlen aus $[0 \dots N-1]$ für ein $N \in \mathbb{Z}$ dh. $U = [0 \dots N-1]$

Wie immer $n = |S|$

Triviale Lösung: Feld $A[0 \dots N-1]$

Jedes $x \in S$ wird in $A[x]$ abgespeichert.

Für $y \notin S$: $A[y] \leftarrow -1$ ← true (wenn nur Mengenproblem)
false

Lookup : Kostet $O(1)$

Nachteil : Platzbedarf $O(N)$, $N \gg n$

Beispiel:

i) kleines N : $N = 256$: Schlüssel von Buchstaben

=> Verfahren ok

ii) Schlüssel sind bel. int-Werte zB.: 32-Bit Zahlen
dann $N = 2^{32}$

=> viel zu großer Platzbedarf, dabei ist A ein sehr dünn besetztes Feld falls $n \ll N$

Ziel von Hashing

Laufzeit von $O(1)$ mit Feld (Tafel) der Größe $O(n)$
(Lookup, Insert, Delete)

Grundkonzepten von Hashing

Hashtafel : $T[0 \dots m-1]$
m = Tafelgröße ($m \geq n$)

Hashfunktion : $h : U \longrightarrow [0 \dots m-1]$

$U = [0 \dots N-1]$

Idee : Speichere $x \in S$ (allg. Paare $(x, \inf(x))$) an der Position $T[h(x)]$

Beispiel: $N = 100$, $U = \{0, \dots, 99\}$

$m = 7$, $S = \{3, 19, 24, 75\}$

wird oft verwendet: $h: x \mapsto x \bmod 7$ // $h(x) = x \bmod m$

	T
0	
1	
2	
3	3
4	
5	19
6	

$3 \to 3$
$19 \to 5$
$24 \to 3$ ✓
$75 \to 5$ ✓ Problem: Kollisionen

Methoden zur Behandlung von Kollisionen
(Kollisionsauflösung)

a) offene Adressierung → Verwende Folge von Hashfunktionen $h_0, h_1, \dots$

Bsp.: 2 Hashfunktionen $f, g: U \to [0 \dots m-1]$

definiere $h_i(x) = (f(x) + i \cdot g(x)) \bmod m$

$h_0 = f$, $h_1 = f + g$, $h_2 = f + 2g$, ...

↖ Lineares Probieren: $g(x) = 1$

Insert(x)

Betrachte Tafelpunkte $h_0(x), h_1(x), \dots$ bis freie Position gefunden

Lookup(x)

Probiere $T[h_0(x)], T[h_1(x)], \dots$ bis entweder x gefunden oder freie Position $\Rightarrow x \notin S$

Delete(x) → Übung

Idee: Tafelposition: 3 Zustände belegt / frei / gelöscht

Operationen werden teurer

$O(\ell(x))$, $\ell(x) = \#$ Position $h_1, \dots, h_\ell$

b) Hashing mit Verkettung

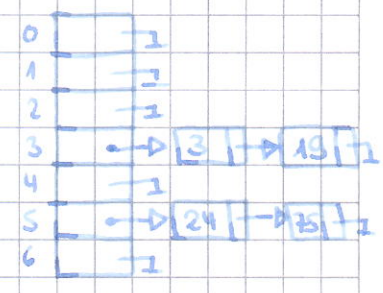
c) Perfektes Hashing → geschickte Wahl der Hashfunktion (Kollisionsvermeidung)

b) Hashing mit Verkettung

T[i] ist Kopf einer Liste, die alle Schlüssel speichert, die von h auf die Position i abgebildet werden.
Genauer: $T[i] = \{x \in S \mid h(x) = i\}$

Beispiel $S = \{3, 15, 24, 25\}$

$h(x) = x \bmod 7$



Implementierung der Operationen: Übung

↳ Lineare Suche in Liste $T[h(x)]$

Kosten $O(1 + \text{Länge von } T[h(x)])$

Schlechtester Fall: $O(n)$ alle Schlüssel in einer Liste (Bsp.: Vielfaches von 7)

Erwartete Laufzeit Mittlerer Fall

Annahmen i) h verteilt U gleichmäßig auf Tafel

$|h^{-1}(i)| = \frac{1}{m}|U|$ für $i = 0 \dots m-1$

⇒ Das gilt für $x \bmod m$!

ii) Schlüssel in S werden zufällig, gleichverteilt und unabhängig aus U gezogen

Frage: Wieviel Kosten n zufällige Einfügungen (in eure Tafel)?

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

$$\delta_h(x,S) = \sum_{y \in S} \delta_h(x,y)$$

d.h.: $\delta(x,S)$ ist Länge der Liste $T[h(x)]$ (in die x gespeichert werden soll)

Betrachte Folge der zufälligen Schlüssel $x_1, x_2, \dots, x_n$

(Wie groß ist $\delta_h(x_{i+1}, \underbrace{\{x_1, \dots, x_i\}}_{=:S})$ im Mittel?

Beobachtung:

Sei $p_i = h(x_i)$ (Position (Liste) von x_i in Tafel), dann nimmt p_i die Zahlen aus $[0 \dots m-1]$ gleichwahrscheinlich an

W'keit, dass ℓ der Zahlen $p_1, \dots, p_i$ gleich einer bestimmten Zahl p sind.

$$= \binom{i}{\ell} \cdot \left(\frac{1}{m}\right)^{\ell} \left(\frac{m-1}{m}\right)^{i-\ell} \quad \text{(Bernoulli-Formel)}$$

- $\binom{i}{\ell}$: Anzahl der Möglichkeiten ℓ Elemente aus i auszuwählen (dass x_{i+1} in Liste mit ℓ Elementen eingefügt wird)
- $\left(\frac{1}{m}\right)^{\ell}$: W'keit, dass $h(x) = p$ für ausgewähltes Element $x = 1/m$ (Trefferw'keit)
- $\left(\frac{m-1}{m}\right)^{i-\ell}$: W'keit, dass Position $\neq p$ ist $= \frac{m-1}{m} = (1 - 1/m)$, $i-\ell$ mal

$$\text{prob}(|T[h(x_{i+1})]| = \ell)$$

Erwartungswert für die Länge der Liste

$$E(\delta_h(x_{i+1}, \{x_1, \dots, x_i\})) = \sum_{\ell=0}^{i} \ell \cdot \text{prob}(\text{"Länge } \ell\text{"})$$

$$= \sum_{\ell \geq 0} \ell \cdot \binom{i}{\ell} \cdot \left(\frac{1}{m}\right)^{\ell} \cdot \left(\frac{m-1}{m}\right)^{i-\ell} \qquad \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

$$= \sum_{\ell \geq 0} \frac{i!}{\ell!(i-\ell)!} \cdot \left(\frac{1}{m}\right)^{\ell} \left(\frac{m-1}{m}\right)^{i-\ell} \cdot \ell$$

$$= \frac{i}{m} \sum_{\ell \geq 0} \frac{(i-1)!}{(\ell-1)!(i-\ell)!} \cdot \left(\frac{1}{m}\right)^{\ell-1} \cdot \left(\frac{m-1}{m}\right)^{i-\ell}$$

$$= \frac{i}{m} \underbrace{\sum_{\ell \geq 0} \binom{i-1}{\ell-1} \cdot \left(\frac{1}{m}\right)^{\ell-1} \cdot \left(\frac{m-1}{m}\right)^{i-\ell}}_{\overset{\text{Üb}}{=} \left(\frac{1}{m} + \frac{m-1}{m}\right)^{i-1} = 1} = \frac{i}{m} \qquad \blacksquare$$

$\Rightarrow$ Die erwarteten Kosten der $(i+1)$-ten Einfügung sind $O\left(\frac{i}{m}\right)$

$\Rightarrow$ n Einfügungen kosten dann

$$O\left(\sum_{i=1}^{n} \frac{i}{m}\right) = O\left(\frac{n \cdot (n+1)}{2m}\right) = O\left(\frac{n}{m} \cdot n\right) = O(\beta \cdot n)$$

$\beta := \frac{n}{m}$ heißt Belegungsfaktor

Für $\beta = O(1)$ z.B. $\beta = 2$ sind die erwarteten Kosten pro Operation $O(1)$
(eigentlich $O(\beta)$)

c) Perfektes Hashing

Zunächst: Schlüsselmenge $S \subseteq U$ ist fest (statisch)

2 Operationen:

- INIT(S) baue Datenstruktur Platz $O(n)$
  Lookup(x) suche in $O(1)$

$m = O(n)$ ($m \ll N$)

keine Kollision $\Leftrightarrow$ Hashfunktion $h: [0 \dots N-1] \to [0 \dots m-1]$ ist injektiv auf S

Aufgabe: Finde injektive Hashfunktion für S

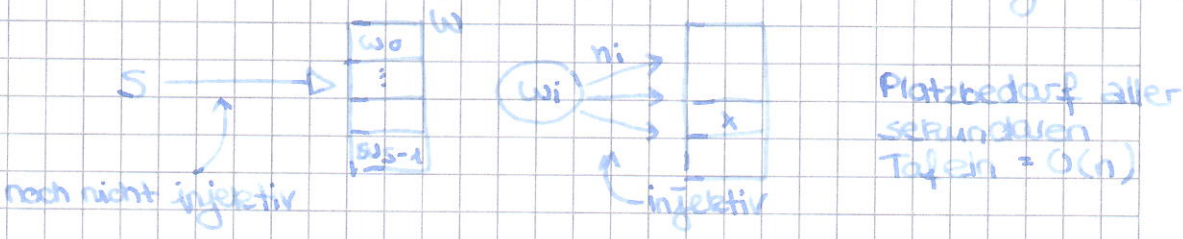
Wir behandeln ein sehr einfaches randomisiertes Verfahren

Paper: Storing a Sparse Table with $O(1)$ worst case Access Time

Fredman & Komlós Journal of ACM, Vol. 31 No 3 1984

Idee 2-stufiges Hashing-Schema (injektiv)

Auswahl der Hashfunktion durch Randomisierung (würfel)



Platzbedarf aller sekundären Tafeln $= O(n)$

Sei p eine Primzahl mit $p > N$ und $s \in \mathbb{N}$ (Tafelgröße)

Betrachte folgende Hashfunktion $\{h_1, \dots, h_p\}$ mit

$$h_k : [0 \dots N-1] \to [0 \dots s-1]$$

$$h_k(x) = (k \cdot x \bmod p) \bmod s \quad \text{für } 1 \le k \le p-1$$

Jede Funktion h_k, $0 \le k \le p-1$ verteilt die Elemente von S auf s Buckets $w_0^k, \dots w_{s-1}^k$

d.h. $w_i^k = \{x \in S \mid h_k(x) = i\}$

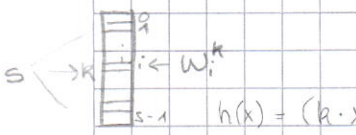
Für jede Menge $S \subseteq \{0 \dots N-1\}$, $|S| = n$

Lemma 1: $\exists k$, $1 \le k \le p-1$ mit

$$\sum_{i=0}^{s-1} \binom{|w_i^k|}{2} < \frac{n^2}{s}$$

↑ Anz. der Paare x, y, $x \neq y$ die im selben Bucket landen.

$\Rightarrow$ Zahl der Kollisionen



Beweis

Zunächst zeigen wir die Behauptung:

$$\underbrace{\sum_{k=1}^{p-1} \sum_{i=0}^{s-1} \binom{|w_i^k|}{2}}_{①} < (p-1) \cdot \frac{n^2}{s}$$

Daraus folgt Lemma 1 (indirekt)

Annahme: Lemma 1 gilt nicht

$$\forall\, 1 \le k \le p-1 \quad \sum_{i=0}^{s-1} \binom{|w_i^k|}{2} \ge \frac{n^2}{s}$$

$$= \sum_{k=1}^{p-1} \quad \ge p-1 \cdot \frac{n^2}{s}$$ ▶

① = Anzahl der Paare $(k, \{x,y\})$ $1 \le k \le p-1$ mit $x, y \in S$, $x \neq y$ und $h_k(x) = h_k(y)$

Beitrag eines festen Paares $x \neq y$ zu ①

= Anzahl aller k's mit $(kx \bmod p) \bmod s$

$= (ky \bmod p) \bmod s$

$\Leftrightarrow ((kx \bmod p) - ky \bmod p) \bmod s = 0$

$\Leftrightarrow (kx \bmod p) - (ky \bmod p) = i \cdot s \quad i \in \mathbb{Z}$

$(kx \bmod p) \in [0 \dots p-1]$, $(ky \bmod p) \in [0 \dots p-1]$

$\in [-p-1 \dots p-1] / 0$

$\triangle\ (k \cdot (x-y) \bmod p)$

Möglichkeiten $(x \neq y)$!

$k \cdot (x-y) \bmod p = 1s, 2s, \dots, -s, -2s$ $\Big\}$ $\le \frac{2 \cdot (p-1)}{s}$ Gleichungen

Da p eine Primzahl (Z_p Körper → Inverse zur Multiplikation) hat jede Gleichung höchstens eine Lösung für k

$\Rightarrow$ Beitrag eines festen Paares (x,y), $x \neq y$ zu (1) ist höchstens $\frac{2 \cdot (p-1)}{s}$

Summiere über alle Paare (x,y), $x \neq y$

$$(1) \leq \binom{n}{2} \cdot \frac{2(p-1)}{s}$$

$$= \underbrace{\frac{n \cdot (n-1)}{2}}_{< \frac{n^2}{2}} \cdot \frac{2(p-1)}{s}$$

$$< \frac{n^2 (p-1)}{s}$$ ∎

Folgerung 1 (Behauptung)

Tafel der Größe n : $\exists k$, $1 \leq k \leq p-1$ mit $\sum_{i=0}^{n-1} |w_i^k|^2 < 3n$ $= O(n)$

Beweis

dh. es existiert eine Hashfunktion, so dass Summe der Quadrate der Bucketgrößen linear (für Tafeln der Größe n

$\Rightarrow$ Betrachte demma 1 für $s = n$

$$\sum_{i=0}^{n-1} \binom{|w_i^k|}{2} < n$$

$$\sum_{i=0}^{n-1} \frac{|w_i^k| \cdot (|w_i^k| - 1)}{2} < n$$

$$\sum_{i=0}^{n-1} |w_i^k|^2 - |w_i^k| < 2n$$

$$\sum_{i=0}^{n-1} |w_i^k|^2 < 2n + \underbrace{\sum_{i=0}^{n-1} |w_i^k|}_{=n} = 3n$$ ∎

Folgerung 2: Für Tafel der Größe $s = n^2$

$\exists k'$, $1 \leq k' \leq p-1$, so dass die Hashfunktion

$h_{k'} : x \longmapsto (k' \cdot x \bmod p) \bmod s$

injektiv auf S ist dh. $|w_i^{k'}| \leq 1$ für $i = 0 \dots n^2 - 1$ ← keine Konflikte

$\Rightarrow$ für quadratisch große Tafel existiert eine perfekte (injektive) Hashfunktion $h_{k'}$

Beweis

Betrachte Lemma 1 mit $S = n^2$

$\exists k', 1 \le k' \le p-1$ mit $\sum_{i=0}^{n^2-1} \binom{|w_i^{k'}|}{2} < 1$

$\Rightarrow |w_i^{k'}| \le 1$ (keine 2 Elemente in einem Bucket $w_i^{k'}$)

$\Rightarrow h_{k'}$ ist injektiv auf S

Vermeidung des quadratischen Speicherplatzes durch ein 2-stufiges Hashing-Schema.

1. Stufe: Wähle ein k gemäß Folgerung 1 (Tafelgröße $S = n$)

d.h. $\sum_{i=0}^{n-1} |w_i^k|^2 < 3n$

genauer: Die Hashfkt. $h_k : x \mapsto (k \cdot x \bmod p) \bmod n$ verteilt S auf eine Tafel der Größe n, wobei die Summe der Quadrate der Bucketgrößen kleiner als $3n$ ist. d.h. linear $|w_i^k|$

2. Stufe: Für jeden nicht-leere Bucket w_i^k der ersten Stufe, wobei ein k_i gemäß Folgerung 2

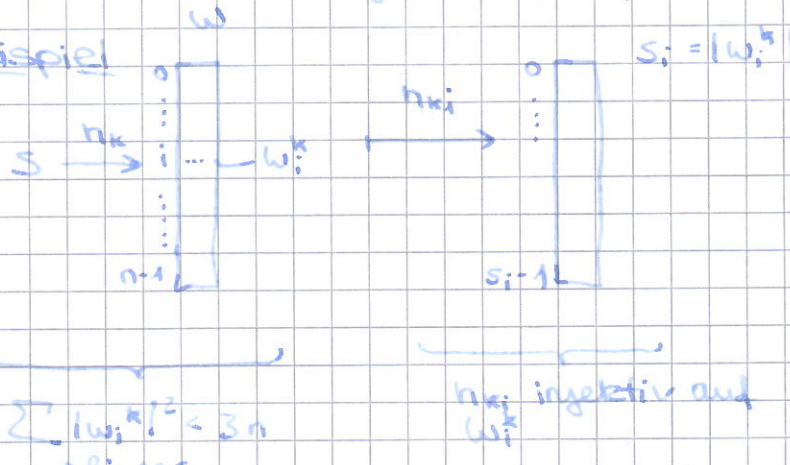
genauer: Für $i = 0 \dots n-1$ ist die Hashfunktion

$h_{k_i} : x \mapsto (k_i \cdot x \bmod p) \bmod s_i$ mit

$s_i = |w_i^k|^2$ (jeweils quadrat. große Tafeln

injektiv auf w_i^k

Beispiel



Injektion: $h_{k_i}(h_k(x))$ injektiv

... und benötigt Platz: $3n + 1 + \sum_{i=0}^{n-1} |w_i^k|^2 < 6n + 1 = O(n)$

Datenstruktur : 3 Felder + Variable k

K [0 ... n-1] $\in \{0 \dots p-1\}$

size [0 ... n-1] $\in \mathbb{N}$

B [0 ... n-1] Pointer auf Hashtafeln (der Stufe 2)
B[i] → Tafel der Größe size[i]

```
i ← (kx mod p) mod n ;
k' ← K[i];
s' ← size[i];
j ← (k'x mod p) mod s'
B[i][j] ← xj
```

Abspeichern der Elemente nach initialisierender Füllung der Felder

Problem : Wie findet man k und K[i], i = 0 ... n-1, die die Bedingungen von Folgerung 1 & 2 erfüllen, effizient?

Triviale Lösung :

Probiere alle möglichen Werte $\{1, \dots, p-1\}$ aus.

Effiziente Lösung :

wir brauchen verbesserte Folgerungen
(∃ viele k's + zufällige Wahl)

Beispiel zum Aufbau der Datenstruktur

N = 30 , p = 31 $S = \{2, 4, 5, 15, 18, 13\}$ n = 6

wähle k = 2 (erfüllt Bedingung von Folgerung 1)

$h_k = (2x \bmod 31) \bmod 6$

	0	1	2	3	4	5
size	1	0	1	0	2	2
	15		4		2	18
					5	13
K	1	0	1	0	1	1
B	↓	–	↓	–	↓	↓

B[0] B[2] B[4] B[5]

Summe der Quadrate : 10 = 3n

Stufe 2 (durch ausprobieren)

B[4] : (k'x mod 31) mod 4
mit k' = 1 ✓

B[5] : (k'x mod 31 mod 4
mit k' = 1 ✓

Lookup(x)

O(1)

```
i ← (kx mod p) mod n
s' ← size[i];
k' ← k[i];
j ← (k'·x' mod p) mod s'
if (B[i][j] = x) then true;
```

Kosten für den Aufbau der Datenstruktur (Init)

Wie findet man ein k für die erste Stufe und k[i], $i = 0(1)n-1$, für die zweite Stufe mit den geforderten Eigenschaften.

1. Stufe: $\sum_{i=0}^{n-1} |w_i^k|^2 < 3n$

2. Stufe: $h_{k[i]}$ injektiv auf w_i^k

Folgerung 1+2: Solche k-Werte existieren!

Naive Methode k zu finden: Probiere mögliche Werte für die k's aus, dh. jeweils Schleife:

for k=1 to p-2 do

Ein einzelner Test kostet (in Stufe 1 & 2) O(n)
→ verteile Elemente von S in h_k und überprüfe ob Eigenschaften erfüllt sind.

Gesamtlaufzeit (worst case): $O(p \cdot n^2) = O(n^2 \cdot N)$ ($n+1$ k-Werte, $p = O(N)$)
⇒ sehr schlecht

Verbesserung: Randomisierte Berechnung der k-Werte (suche nach geeigneter Hashfunktion)

(wenn ein Bruchteil aller möglichen k-Werte (zB. die Hälfte) die Bedingungen erfüllt. Dann ist ein zufälliges k mit (W'keit 1/2 ok.

Algorithmus

ziehe zufällig k aus $\{1, \dots, p-1\}$

bis die entsprechenden Bedingungen erfüllt.

Übung ⇒ erwartete Anzahl der Ziehungen bei einer Trefferwahrscheinlichkeit 1/2 ist 2

Mit einer "kleinen" Erhöhung der Speicherplatzschranke können wir erreichen, dass mind. die Hälfte der k-Werte ok sind.

⇒ Erwartete Aufbauzeit : $O(n^2)$

Wir brauchen n+1 k-Werte,
ein Test kostet kostet $O(n)$
→ wir machen 2 Tests pro k

Modificierte Folgerung aus Lemma 1

Folgerung 3 für $s = n$

Für mind. die Hälfte aller k, $1 \le k \le p-1$ gilt

$$\sum_{i=0}^{n-1} |w_i^k|^2 < 5n$$

Beweis

Betrachte Gleichung (1) im Beweis von Lemma 1

$$\sum_{k=1}^{p-1} \sum_{i=0}^{s-1} \binom{|w_i^k|}{2} < p-1 \cdot \frac{n^2}{s}$$

für $s = n$

$$\sum_{k=1}^{p-1} \sum_{i=0}^{n-1} \binom{|w_i^k|}{2} < (p-1) \cdot n$$

⇒ Für mind. die Hälfte aller k's, $1 \le k \le p$ gilt

$$\sum_{i=0}^{n-1} \binom{|w_i^k|}{2} < 2 \cdot n$$

Zum Beweis

Annahme für mehr als die Hälfte aller k gilt

$$(*) \quad \sum_{i=0}^{n-1} \binom{|w_i^k|}{2} \ge 2 \cdot n$$

$$\Rightarrow \sum_{}^{p-1} (*) \ge \frac{p-1}{2} \cdot 2n = (p-1)\,n$$

→ Beitrag der Hälfte

$$\Rightarrow \sum_{i=0}^{n-1} \frac{|w_i^k| (|w_i^k| - 1)}{2} < 2n \quad \Big) \cdot \tfrac{1}{2}, \text{ ausmultiplizieren}, + \sum |w_i^k|$$

$$\sum_{i=0}^{n-1} |w_i^k|^2 < 4n + \underbrace{\sum_{i=0}^{n-1} |w_i^k|}_{n} < 5n$$

Folgerung 4 ($s = 2n^2$)

Für mind. die Hälfte aller k', $1 \le k' \le p-1$ gilt die Abb.: $h_{k'} : x \mapsto (k' \cdot x \bmod p) \bmod 2n^2$ ist injektiv auf S

Betrachte (1) mit $s = 2n^2$

$$\sum_{k=1}^{p-1} \sum_{i=0}^{2n^2-1} \binom{|w_i^k|}{2} < (p-1) \cdot \frac{n^2}{2n^2} = \frac{1}{2}(p-1)$$

$\Rightarrow$ Für mind. die Hälfte aller k gilt: $\sum_{i=0}^{2n^2-1} \binom{|w_i^k|}{2} < 1 = 0$

$\Rightarrow$ für mind. die Hälfte aller k gilt $|w_i^k| \le 1$, $0 \le i \le 2n^2-1$

$\Rightarrow$ $h_{k'}$ injektiv ✓

Datenstruktur

1. Stufe: wähle k, so dass Summe der Quadrate der Bucketgrößen $\le 5n$ (die Hälfte aller k-Werte erfüllt Bedingung)

2. Stufe: Für jedes Bucket W_i^k der ersten Stufe verwende Tafel der Größe $2 \cdot \text{size}[i]^2$ (die Hälfte aller k'-Werte liefert injektives h_k)

$\Rightarrow$ Platzbedarf aller Tafeln der 2. Stufe $< 10 \cdot n$

$\Rightarrow$ Insgesamt: Platz $\le 13n$ (1 Stufe: 3 Felder der Länge n)

Zusammenfassung

Jede Menge $S \subseteq \{0, \dots, N-1\}$ mit $|S| = n$ kann so abgespeichert werden, dass gilt

1. Platzbedarf $O(n)$, genauer $\le 13n$
2. Erwartete Aufbauzeit: $O(n^2)$
3. Lookup(x): $O(1)$ im worst case

} perfektes Hashing (↔ speichern einer dünn besetzten Tafel)

Dynamisierung: Dynamic Perfekt Hashing

→ möglich, aber nicht mehr Inhalt der VL

4. Warteschlangen (Priority Queues)

Verwalte eine Menge S von Prioritäten, $S \subseteq U$
U linear geordnet (z.B. Zahlen)
Operationen: PQ.insert(p)
PQ.findmin()
PQ.delmin()

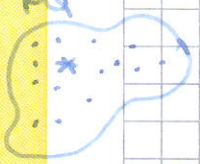
PQ.insert(p,i) p Priorität, i Information
PQ.delmin() { liefert Paar (p,i) mit min
PQ.findmin() { Priorität p (und entfernt es)

PQ.decrease_p((p,i), q) vermindert die Priorität von (p,i) auf q
(p,i) → (q,i)

Anwendung:

Kürzeste (billigste) Wege in Graphen

Argumente von PQ.decrease_p(*, q)



sollte eine Position in der Datenstruktur sein (Pointer, Ref, Index, Iterator)

Einfache Implementierung → Übung

1. Doppeltverkettete Liste

Variante a) unsortiert

PQ.insert → vorne einfügen
PQ.findmin → lineare Suche, O(n)
PQ.delmin → lineare Suche, O(n)
PQ.decrease_p → ersetze p durch q, O(1)

Variante b) (genauer: aufsteigend nach Priorität)

PQ.findmin() O(1), erstes Element
PQ.delmin() O(1)
PQ.insert(p,i) O(n), lin. Suche + einfügen
PQ.decrease_p(i,q) O(n), ——"——

2. Binäre Bäume (siehe 1b)
(balanciert) → Übung

3. Minimums-Heap → Übung

Für diese einfachen Datenstrukturen gelten die Laufzeitschranken für den schlechtesten Fall (worst case):

Für die Anwendung (Graph. Alg.) ist oft eine amortisierte Analyse interessanter:
Beispiel: zur Lösung eines Problems sind n Inserts

n Delmin und n Decrease erforderlich.

Dann sind die Kosten der Gesamtfolge der Operationen entscheidend.

Amortisierte Analyse von Datenstrukturen

Datenstrukturen D + Folge von n Operationen

$D_0 \xrightarrow{op_1} D_1 \xrightarrow{op_2} D_2 \cdots D_{i-1} \xrightarrow{op_i} D_i \cdots \xrightarrow{op_n} D_n$

Amortisierte Analyse schätzt die Gesamtkosten der Folge (worst case) ab

triviale Abschätzung $O(n \cdot \text{worst case}(op))$ ist oft sehr schlecht

Beispiel Binäre Zähler

Operation: Reset (setze auf Null)
Increment (erhöhe um 1)

Implementierung: Bitstring $a_k \ldots a_3 a_2 a_1 a_0$

Zählerwert: $\sum_{i=0}^{k} a_i 2^i$

Reset Bitstring $a_0 \leftarrow 0$ $(O(1))$

Increment
```
a_0 ← a_0 + 1;
i ← 0
while a_i = 2 do
    a_i ← 0;
    a_{i+1} ← a_{i+1} + 1   // Übertrag
    i ← i+1;
od
```

Kosten: $O(1 + \#\text{Schleifendurchläufe})$

Was kostet nun eine Folge von 1 × Reset ($\to D_0$) und n × Increment.

1. Antwort

Der Zähler hat höchstens $\log n$ Bits (bei max Wert n)
$\Rightarrow$ die Schleif wird höchsten $\log n$ mal durchlaufen

$\Rightarrow$ Gesamtlaufzeit $O(n \log n)$

($n \times$ worst-case)

Antwort 2

genauere Betrachtung → Kosten $O(n)$
(Übertrag läuft nur selten über $\log n$ Stellen)

Beweis durch amortisierte Analyse (Potentialmethode)

Ein Potential ist eine Funktion, die jedem Zustand (Instanz) der Datenstruktur eine Zahl aus $\mathbb{N}_0$ zuordnet.
(nicht negativ)
$(pot(D_i) \in \mathbb{N}_0^+)$

Im Beispiel $pot(\dots a_3 a_2 a_1 a_0) := \sum_{i \geq 0} a_i$
d.h. Anzahl der Einsen

Eine Operation op (im Bsp. Increment)
überführt die Datenstruktur von einem Zustand zu einer anderen $D \xrightarrow{op} D'$

Definition

$T_{tats}(op)$ = tatsächliche Ausführungszeit von op

$T_{amort}(op) := T_{tats}(op) + \underbrace{pot(D') - pot(D)}_{\Delta pot}$

Nun betrachte die Folge

$D_0 \xrightarrow{op_1} D_1 \dots D_{i-1} \xrightarrow{op_i} D_i \dots \xrightarrow{op_n} D_n$

Ziel obere Schranke für $\sum_{i=1}^{n} T_{tats}(op_i)$

Betrachte

$$\sum_{i=1}^{n} T_{amort}(op_i) \overset{def}{=} \sum_{\text{Teleskopsumme}} (T_{Tats}(op_i) + pot(D_i) - pot(D_{i-1}))$$

$$= \sum_{i=1}^{n} T_{Tats}(op_i) + pot(D_n) - pot(D_0)$$

oder

$$\sum_{i=1}^{n} T_{tats}(op_i) = \sum_{i=1}^{n} T_{amort}(op_i) + pot(D_0) - pot(D_n)$$

möchten wir abschätzen — lässt sich oft leicht abschätzen

$$\Rightarrow \sum_{i=1}^{n} T_{tats}(op_i) \leq \sum_{i=1}^{n} T_{amort}(op_i) + pot(D_0)$$

Spezialfall $pot(D_0) = 0$ (gilt im Bsp.)

Spezialfall

$$\sum_{i=1}^{n} T_{tats}(op_i) \leq \sum_{i=1}^{n} T_{amort}(op_i)$$

gilt für alle Potentiale mit

i) $pot(D) \geq 0$
ii) $pot(D_0) = 0$

gute amortisierte Analyse

→ Finde ein Potential, so dass $\sum_{i=1}^{n} T_{amort}(op_i)$ eine gute Schranke ist (dh. möglichst klein)

Im Beispiel (Binärzähler)

$pot(D_i)$ = # Einsen in D_i

Zähler [. . . . 0 | 1 . . . 1] $k \geq 0$
(hat ja die Form) (erste 0 | k Einsen)

Dann gilt: 1) Increase kostet $k+1$ ($O(k+1)$)
dh. $T_{tats}(op_i) = k+1$

$D_{i-1} \xrightarrow{Inc} D_i$
... 0 1 ... 1 ... 1 0 ... 0

2.) Potentialdifferenz $\Delta_{pot} = 1-k$

$$\Rightarrow T_{amort}(Inc) = \underbrace{k+1}_{t_{tats}} + \underbrace{(1-k)}_{\Delta_{pot}} = 2 = O(1)$$

Bed. dass ~~Fehlerfall~~ erfüllt

$$\Rightarrow \sum_{i=1}^{n} T_{tats} \leq \sum_{i=1}^{n} T_{amort} \leq 2n = O(n)$$

Warteschlangen implementiert durch

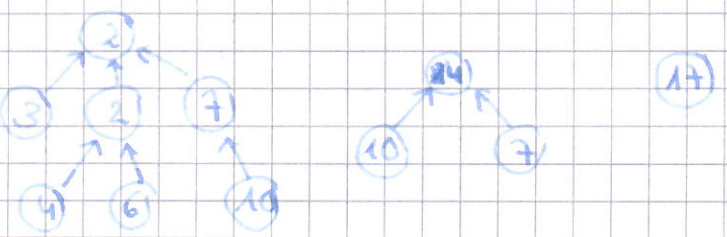
Fibonacci-Heaps

Ein Fib-Heap besteht aus einer Menge von Heaps (Min-Heaps!)

Heap: Baum, dessen Knoten mit Werten (hier Prioritäten) beschriftet sind, so dass

$$prio(vater(v)) \leq prio(v)$$

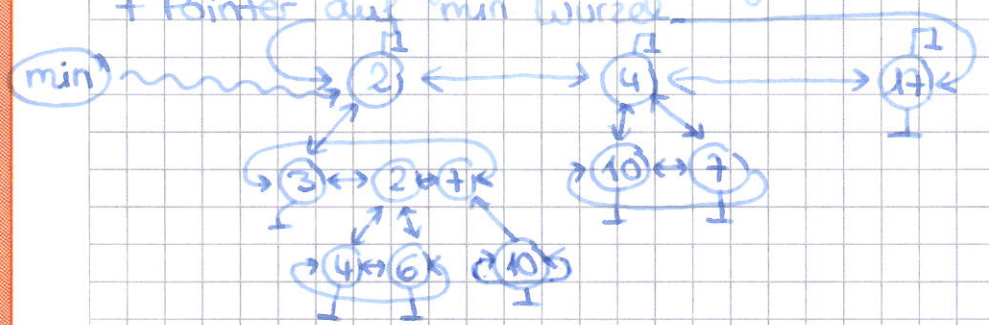
Bsp.:



Knoten werden durch folgende Struktur (Klasse) realisiert

```
class node {  prio p;
              inf i;
              node parent, child, leftsib, rightsib;
              int rank;        // Anz der Kinder
              bool mark;
              : später mehr
}
```

Fib-Heap Menge von Heaps genauer Liste der Wurzeln + Pointer auf min Wurzel



Realisierung der PQ-Operationen

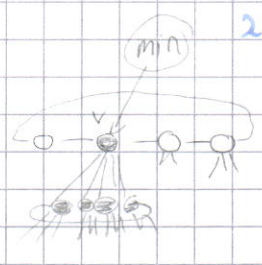
Init PQ ← ∅ min ← NULL, O(1)
(Konstruktor)

FINDMIN(): Zugriff über min-Pointer O(1)

Insert(p,i): erzeuge Heap, der aus einem Knoten v besteht mit v.prio = p, v.inf = i und füge v in die Wurzelliste ein.
O(1)
Falls p < min.prio dann min ← v

DELMIN 1) Resultat Knoten, auf dem min-Pointer zeigt

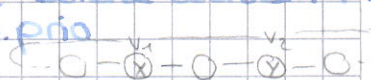
2) Streiche v aus Wurzelliste und füge alle Kinder von v in diese Liste ein:



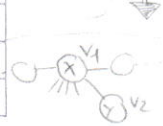
3) Sorge dafür dass alle Wurzeln paarweise verschiedenen Rang (#Kinder) besitzen:

while 2 Wurzeln mit gleichem Rang existieren

seien v_1, v_2 solche Wurzeln mit $v_1.prio \leq v_2.prio$



mache v_2 zum Kind von v_1
↳ verschmelze v_1, v_2



od

4) Durchsuch die Liste der Wurzeln linear nach dem neuen Minimum. Das kostet $O(\text{max Rang})$, da alle alle Wurzeln paarweise versch. Rang (siehe 3)

Lemma Kosten von Schritt 3

Schritt 3 kostet $O(\text{maxRang} + \#\text{Verschmelzungen})$

Beweis durch Algorithmus

```
A[0...maxRang] von Knoten
1.  for i=0 to maxRang do A[i] ← NULL; od;
2.  Gehe alle Wurzeln nacheinander durch
3.  sei r ← nächste Wurzel
4.  while A[r.rank] ≠ NULL do
5.      r' ← A[r.rank];
6.      A[r.rank] ← Null;
7.      r ← Verschmelzen (r,r')   // liefert die neue
8.  od                            // Wurzel, r o. r'
9.  A[r.rank] ← r;
10 od
```

Analyse

Seien $\#w_0$ = Anzahl der Wurzeln von Schritt 3
$\#w_1$ = —"— nach —"—
$\#V$ = Anzahl der Verschmelzungen

Dann gilt

a) Kosten von Schritt 3 : $O(\text{maxRang} + \#w_0 + \#V)$
b) $\#w_1 \leq \text{maxRank}$ (da alle versch. Rang)
c) $\#V = \#w_0 - \#w_1$ (je Verschmelzung vermindert $\#w$ um 1)

$\Rightarrow \#w_0 = \#V + \#w_1$ ($\#w_0 \geq$ c)

$\Rightarrow$ **Laufzeit** : $O(\text{maxRang} + \#V + \#w_1 + \#V)$

b) $\rightarrow$ $O(\text{maxRang} + \#V)$

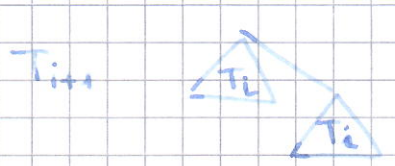
Wir schätzen den max. Rang ab

Beobachtung: Insert + Delmin

erzeugen spezielle Bäume

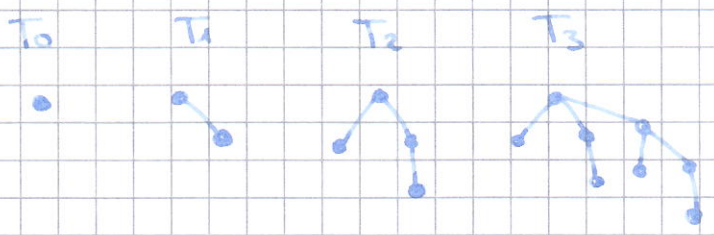
Binomische Bäume Folge von Bäumen $T_0, T_1, \ldots$

T_0 • einzelner Knoten (Insert)



Verschmelze

von Baum zu Baum:



2. Knoten Rang + 1

Lemma

Der maximale Rang eines binomischen Baumes mit n Knoten ist $\log n$ (n = # Knoten)

$\Rightarrow$ max Rang im Fib Heap $\leq \log n$

Beweis

1. Die Wurzel von T_i hat Rang i und T_i hat 2^i Knoten (# Knoten verdoppelt sich)

2. Wurzeln hat max Rang in T_i, da alle Nachkommen Wurzeln von Bäumen T_j mit $j < i$ sind

Amortisierte Analyse der Operationen (Insert, Delmin, Findmin)

Zur Erinnerung: $T_{amort}(op) = T_{tats}(op) + \Delta pot$

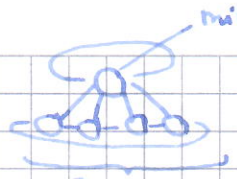
Definiere Potential = Anzahl der Wurzeln mit i) immer positiv ii) am Anfang 0

Offensichtlich: Insert, Findmin: $O(1)$

T_{tats} $O(1)$
$\Delta pot = 1$

Delmin $T_{tats} = O(\underbrace{\text{max Rang}}_{\log n \text{ nach Lemma 2}} + \#\text{ Verschmelzung})$

$T_{amort} = T_{tats} + \Delta pot$ →



$\Delta pot \leq \log n - \#$ Verschmelzung

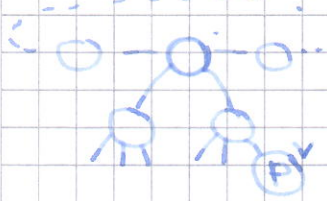
$\leq$ maxRang $\leq \log n$

$T_{amort} = \underbrace{\log n + \#\text{Verschmelzung}}_{t_{ats}} + \underbrace{\log n - \#\text{Versch.}}_{\Delta pot}$

$= O(\log n)$

genauer: In einer beliebigen Folge von Insert, Findmin, und Delmin Operationen betragen die amort Kosten für Insert $O(1)$, für Findmin $O(1)$ (worst case) und Delmin $O(\log n)$

Nun zur Operation PQ.decrease_p(v,q)



ersetze Priorität p durch q ($q < p$)

$\Rightarrow$ Heap

Realisierung:

i) v.prio ← q
ii) Streiche die Pointer zum Vater
iii) Füge v in Wurzelliste ein

Problem: Entstehende Bäume sind nicht mehr binomisch d.h. die $\log n$-Schranke für maxRang gilt auch nicht mehr. (Beispiel → Übung)

Durch eine Modifikation kann man jedoch den maxRang immernoch durch $O(\log n)$ beschränken.

Idee: Setze das Streichen von Parent-Verweisen fort, sobald ein 2-tes Kind entfernt wird.

Schritt iv

iv) Falls Vater u von v existiert: Markiere(u)

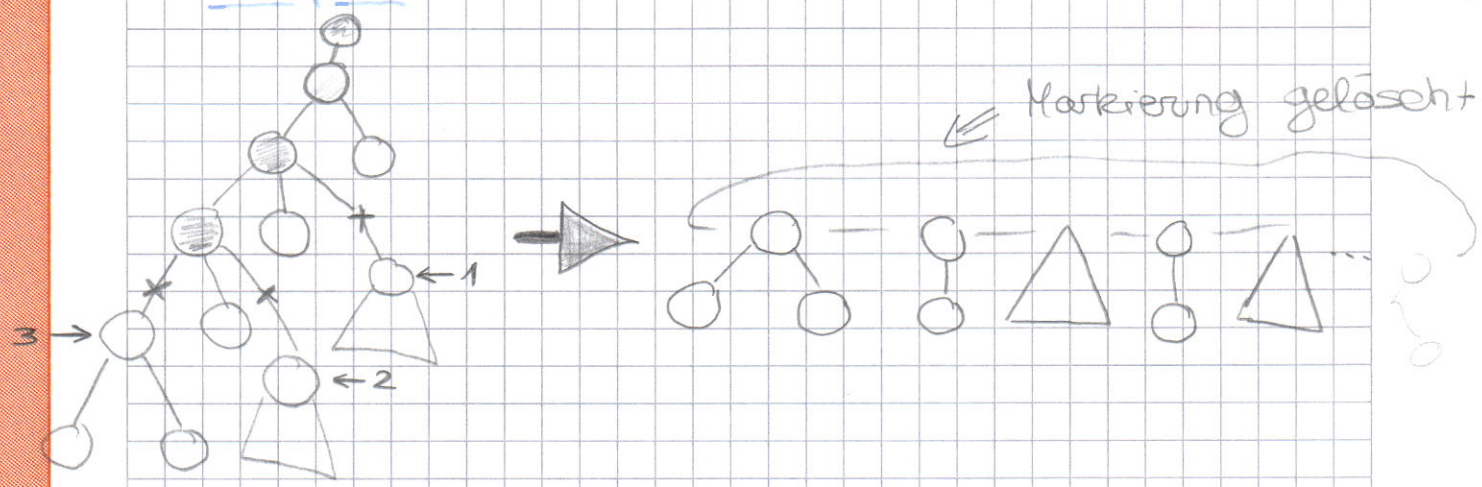
$\Rightarrow$ markiere(u)

while u ist markiert ∧ u ist nicht Wurzel do
  streiche Verweis zum parent(u) ← [cut]
  u ← u.parent; { u ← u.parent; v.parent ← NULL; u ← u }
  Füge u in Wurzelliste ein
  u.marke ← false
od
markiere u;

$\Rightarrow$ Übung genauer Pseudocode

zum Markieren benutze u.mark - Feld

Beispiel



Ein einzelner Decrease_Operation kann sehr viele Cuts auslösen d.h. teuer sein

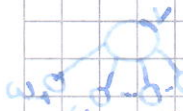
Aber die amort. Kosten bleiben O(1)

Wir zeigen nun, dass durch die Strategie der maxRang durch $O(\log n)$ beschränkt bleibt.

Lemma 3 $\quad maxRang \leq 1{,}4404 \log n$

Beweisidee Teilbaum mit Wurzelrang r enthält $\geq 2^r$ Knoten.

Beweis : sei v bel. Knoten mit Rang i (d.h. v hat i Kinder)



Ordne die Kinder von v nach der Zeit, zu der sie zu v kamen (verschmelzen).

Sei w_j das j-te Kind

Behauptung : $Rang(w_j) \geq j-2$

Beweis : Als w_j zum Kind von v wurde galt : $Rang(w_j) = Rang(v) = j-1$
(gilt bei jeder Verschmelzung)

w_j hat seitdem höchstens 1 Kind verloren da es keine Wurzel ist (siehe Decrease-p)

$\Rightarrow Rang(w_j) \geq j-2$ ∎

Für einen bel. Knoten v mit Rang i sei

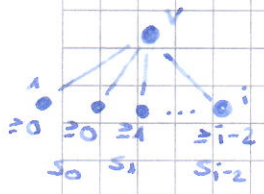
S_i = minimale Anzahl von Knoten im Unterbaum mit Wurzel v (inkl. v)

Dann gilt:

$S_0 = 1$ // Knoten mit Rang 1

$S_1 = 2$

für $i \geq 2$ gilt: $S_i = 2 + S_0 + \dots + S_{i-2}$



$\Rightarrow S_0 = 1,\ S_1 = 2,\ S_i = 2 + \sum_{j=0}^{i-2} S_j$ für $i \geq 2$

Fib-Zahlen (Üb)

$F_0 = 0,\ F_1 = 1,\ F_i = F_{i-2} + F_{i-1}$ für $i \geq 2$

Übung: es gilt $F_{i+2} = 2 + \sum_{j=2}^{i} F_j$

Beh. $S_i \geq F_{i+2}$ für $i \geq 0$

Bew. durch Induktion über i

$i = 0$: $S_0 = 1 = F_2$ ✓

$i > 0$ $S_i = 2 + \sum_{j=0}^{i-2} S_j$

$\overset{IV}{\geq} 2 + \sum_{j=2}^{i-2} F_{j+2} = 2 + \sum_{j=2}^{i} F_j$

$\overset{Üb.}{=} F_{i+2}$ ∎

d.h. die Anzahl der Knoten in einem Unterbaum mit Wurzelrang i ist $\geq F_{i+2}$

Übung: $F_{i+2} \geq \left(\frac{1+\sqrt{5}}{2}\right)^i \approx 1{,}618^i$

$\Rightarrow S_i \geq 1{,}618^i$

Da wir n Knoten haben gilt für maxRang r

$$\boxed{1{,}618^r \leq n}$$

$$\Rightarrow r \leq \frac{\log n}{\log 1{,}618} \leq 1{,}4404 \log n$$

$\Rightarrow$ maxRang $= O(\log n)$

genauer: um etwa 50% erhöht ∎

Feld im Deletemin-Alg. muss nun Länge von $\geq 1{,}4404 \cdot \log n$ haben

Amortisierte Analyse aller Operationen

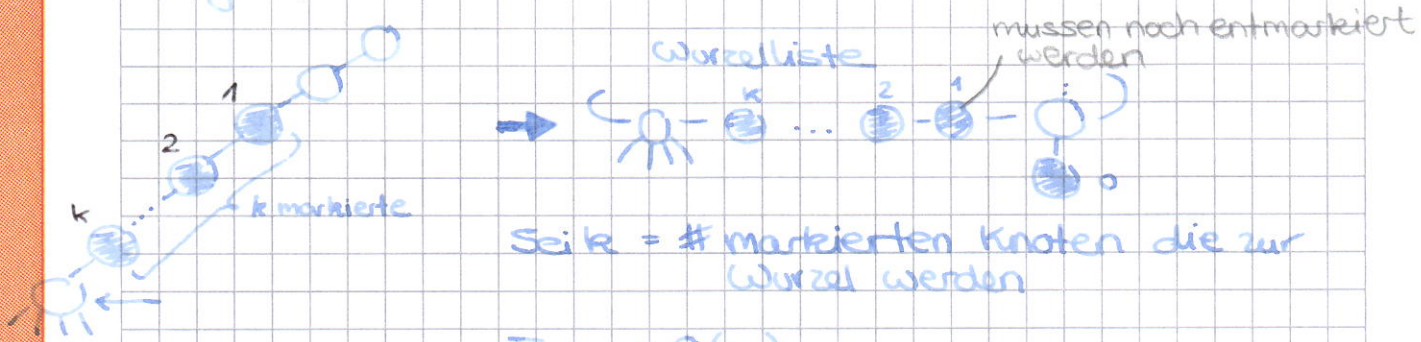
Wir modifizieren das Potential (Wurzeln sind nicht markiert)

$$pot := \#\text{Wurzeln} + 2 \cdot \#\text{markierte Knoten}$$

Analyse von Delmin, Insert, Findmin bleibt gleich
(keine Änderung der markierten Knoten)

Decrease-p

allgemeine Situation



Sei k = # markierten Knoten die zur Wurzel werden

$T_{tats} = O(k)$

#Wurzeln erhöht sich um k+1

#markiert Knoten nimmt um k-1 ab

$\Delta pot = k + 1 - 2(k-1) = 3 - k$ o. $-k+3$

$T_{amort} = T_{tats} + \Delta pot$

$= O(3) = \underline{O(1)}$

Zusammenfassung

Fibonacci-Heaps unterstützen die Operationen Insert, Findmin, Decrease_p in Zeit $O(1)$ (Insert, Findmin sogar im worst case) und Delmin in Zeit $O(\log n)$ jeweils amortisiert

Auch Operation Delete(v), die Knoten v entfernt ist möglich:

PQ.decrease_p(v, $-\infty$) + PQ.delmin()

$\Rightarrow O(\log n)$

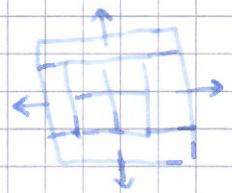
Anwendung: kürzeste Wege in Graph

Dijkstra: n × (Insert, Delete) + n × Decrease_p

Kosten der Folge → amortisierte Analyse

Gesamtlaufzeit: $O(n \log n + m)$ n = # Knoten, m = # Kanten

Platz: $O(n)$ aber mit relativ großen Konstanten



8 Speicherzellen pro Knoten

II. Graphalgorithmen

1. Planare Graphen

Literatur: Nishizeki / Chiba: Planare Graphen

(Wir betrachten hier ungerichtete Graphen $G = (V, E)$

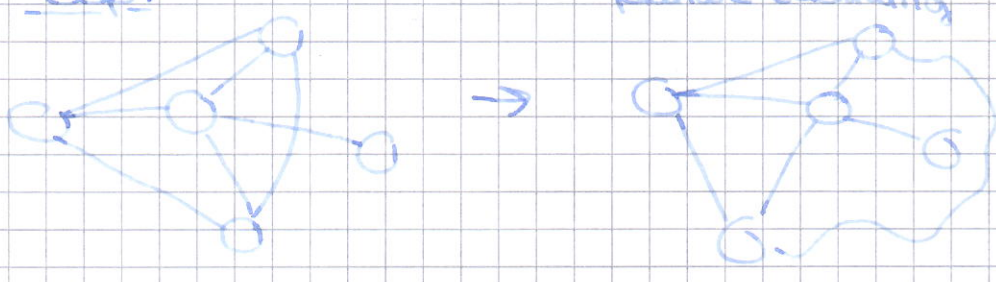
Definition

a) Ein Graph $G = (V, E)$ ist planar, wenn G eine planare Zeichnung besitzt.

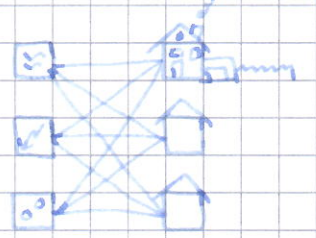
b) Eine planare Zeichnung von G ordnet jedem Knoten v einen Punkt $p \in \mathbb{R}^2$ (Ebene) zu (seine Position: $pos(v)$) und jeder Kante (v,w) eine stetige Kurve mit den Endpunkten $pos(v)$ und $pos(w)$, so dass sich diese Kurve paarweise nicht schneidet, außer in ihren Endpunkten.

G planar $\Leftrightarrow$ $\exists$ planare Zeichnung

Bsp:



nicht planar: Vollständiger Bipartiter Graph $K(3,3)$ o. $K_{3,3}$



$\Rightarrow$ offensichtliches Problem: Teste ob ein Graph planar ist.

Beobachtung: G planar ⇔ G besitzt planare Zeichnung auf Kugeloberfläche

Planare Zeichnung zerlegt die Oberfläche in Gebiete → Faces

In Ebene: 1 äußeres Face

Wichtige Begriffe und Definitionen

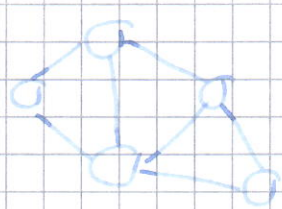
1. Der Zusammenhang eines Graphen (genauer Knotenzusammenhang)

a) G heißt (einfach) zusammenhängend wenn er für jedes Paar (v, w) von Knoten einen Pfad zwischen v und w gibt

zusammenhängend

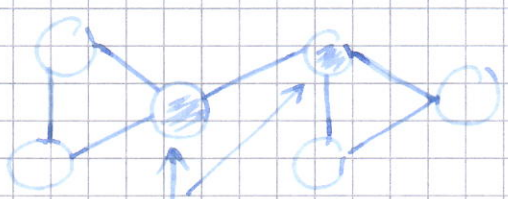
b) G heißt zweifach zusammenhängend falls $G \setminus \{v\}$ für jeden Knoten $v \in V$ zusammenhängend

Bsp:



→ egal welchen Knoten man wegnimmt, der Graph bleibt zusammenhängend

nicht zweifach zusammenhängend



Cut-Vertex
oder
Artikulationspunkt

21.12.2011

1-fach zusammenhängend (Nummerierung der Knoten)

2-fach zusammenhängend

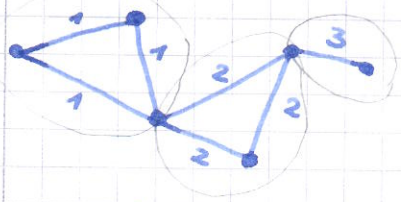
$G \setminus \{v\}$ für bel. Knoten v ist einfach zusammenhängend

CUT-Vertex: G nicht 2-fach zusammenhängend
∃ Knoten v, dessen Entfernung G zerlegt
↳ Artikulationspunkt

Komponenten: maximale Teilgraphen mit entspr. Eigenschaft

↳ Zusammenhangskomponente

↳ 2-fach-Zusammenhangskomponente → Blöcke



← Nummerierung der Kanten

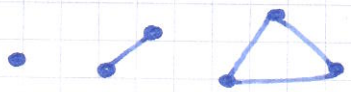
Ein 2-fach zusammenhängender Graph heißt 3-fach zusammenhängend, wenn für bel. Knoten $v, w \in V$ gilt

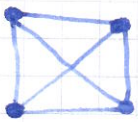
$G \setminus \{v, w\}$ ist zusammenhängend.

Ein nicht 3-fach zusammenhängender Graph besitzt Kantenpaare $\{v, w\}$ (mind 1), deren Entfernung ihn zerlegen. Diese heißen Seperationspaar (seperation pair)

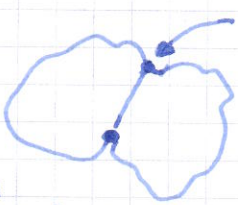
Beispiele

trivial:





nicht 3-fach zusammenhängend



→ es entstehen 2 split Graphen

Sei G 2-fach zushäng. und
(v,w) ein Seperationspaar von G
(dh. G ist nicht 3-fach-zus.-häng.)
Dann heißen die beiden Teilgraphen G_1 und G_2, die durch Entfernen von (v,w) entstehen Splitgraphen, genauer G_1 und G_2 enthalten beide v und w sowie die Kante (v,w), falls sie existiert.

k-fach zusammenhängend

man kann k-1 Knoten entfernen und er bleibt trotzdem zusammenhängend

Beobachtung -

Ein nicht 2-fach zusammenhängender Graph G ist planar (dh. besitzt eine planare Zeichnung) genau dann, wenn alle Blöcke (2-fach-zus.häng Komponenten) von G planar sind.

Idee

Konstruire planare Zeichnung für jeden Block, so dass Artikulationspunkte außen liegen und klebe die Zeichnung dort zusammen.

Folgerung -

Beim Test auf Planarität & bei der Konstruktion einer Zeichnung, kann man sich auf 2-fach zush. Graphen beschränken.

Planare Einbettung -

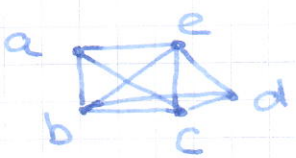
→ abstrakte planare Zeichnung (Plane Graph's)

keine Position
Faces (Fläche)

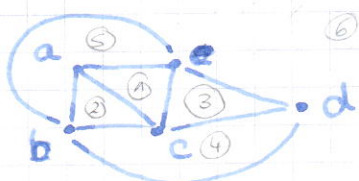
Definition Sei G ein planarer Graph

Eine planare Einbettung von G ist eine zyklische Sortierung der adjazenten Kanten für jeden Knoten v, so dass eine planare Zeichnung für G existiert, in der die Kanten in dieser Reihenfolge gegen den Uhrzeigersinn den Knoten v verlassen.

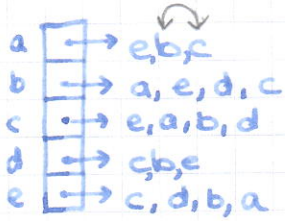
Bsp.



pl. Zeichnung



Einbettung -



6 Flächen

1. ace
2. abc
3. cde
4. bdc
5. aeba
6. bed

2 Darstellungen

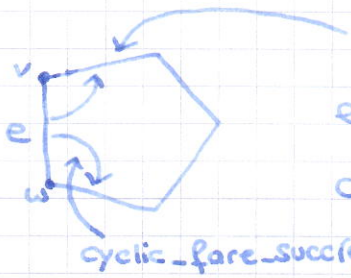
I) Ordnung der Kanten um jeden Knoten (Def)

II) Ränder der Faces

Darstellungen sind äquivalent

i) => ii) :

Wie komme ich zur nächsten Kante des Face-Zyklus?



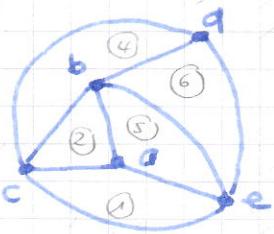
x = cyclic_succ(e)

e = (v, w)

cyclic_face_succ(e) = cyclic_pred(e) im Knoten w

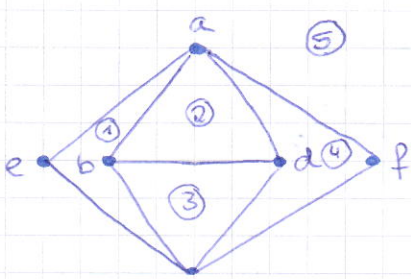
ii) → i) : symmetrisch (→ Übung)

Eine planare Einbettung stellt verschiedene Zeichnungen dar, insbesondere kann jedes Face außen liegen.
Im Beispiel Fläche 3 außen



Im Allgemeinen existieren verschiedene Einbettungen (falls G nicht 3-fach zusammenhängend)

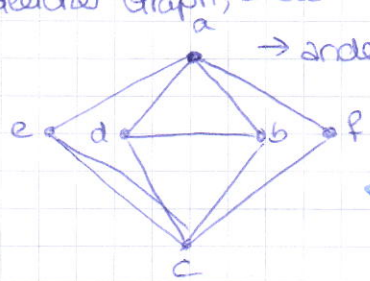
neues Beispiel



Planare Einbettung
hat mehrere Zeichnungen
- genaue Position der Knoten
- Kanten zB gerade Linien, Polygone, Kurven

wichtig → äußere Flächen
kann jedes Face sein

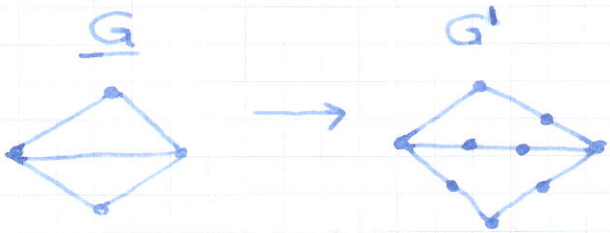
Gleicher Graph, andere Flächen → andere Einbettung



← Beobachtung: G nicht 3-fach-zusammenh.
→ (a,c) seperationspaar

Definition

Sei G ein ungerichteter Graph
G' heißt **Unterteilung** (subdivision) von G wenn G' aus G durch Ersetzen von Kanten durch einfache Pfade entsteht. (Plazieren von neuen Knoten auf die Kanten von G.)

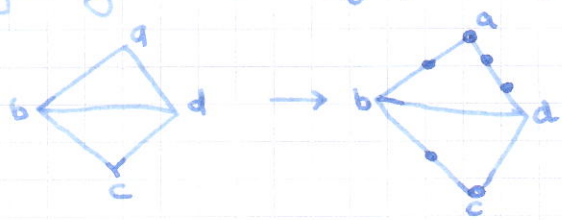


Satz 1

Die planare Einbettung eines 2-fach zusammenhängenden planaren Graphen ist eindeutig, genau dann, wenn G eine Unterteilung eines 3-fach-zusammenh. Graphen ist.

Beweis: später

Folgerung: eindeutig für 3-fach. zush. Graphen



auch hier eindeutig

Die Euler-Formel

Satz 2 (Euler 1750)

Sei G eine zusammenhängende planare Einbettung (plane graph) mit n Knoten, m Kanten und f Faces. Dann gilt

$$n - m + f = 2$$

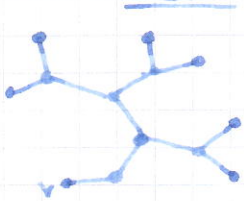
Beweis durch Induktion über $n \geq 1$

IA: $m = 0 \Rightarrow n = 1, f = 1$

IV: $n - m + f = 2$

IS: $m \geq 2$

Fall 1 G ist ein Baum (d.h. azyklisch)



$\Rightarrow f = 1$

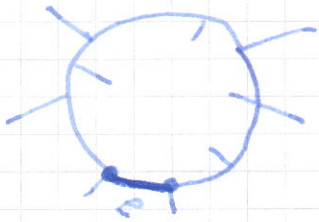
Dann besitzt G einen Knoten v mit degree(v) = 1 (Blatt)

Die planare Einbettung $G \setminus \{v\}$ ist zusammenhängend und hat n-1 Knoten & m-1 Kanten

$$(n-1) - (m-1) + f = 2$$

$$n - m + f = 2 \quad \checkmark$$

Fall 2: G ist kein Baum → $f > 1$



sei e eine Kante auf einem Kreis

Betrachte die planare Einbettung $G \setminus \{e\}$ für die gilt

$m-1$ Kanten
n Knoten
$f-1$ Faces

$$n-(m-1)+(f-1) = 2$$

$$\rightarrow \quad n-m+f = 2 \quad \blacksquare$$

Folgerung Ein planarer Graph besitzt planare Einbettungen für alle diese gilt die Euler Formel

Folgerung Sei G ein planarer Graph mit $n \geq 3$ Knoten und m Kanten, dann gilt

$$m \leq 3n-6$$

dh. $m = O(n)$ ⇒ planarere Graphen sind dünn

Algorithmus mit $O(m+n)$

Def

Beweis ✓ Ein maximal planarer Graph ist ein planarer Graph G, der durch Hinzufügen einer bel. Kante $(v,w) \notin E$ nicht-planar wird

Beobachtung:

Die Faces eines planaren Einbettung eines max. planaren Graphes sind alle Dreiecke.

Zeige für max. planare Graphen $m = 3n-6$
Daraus folgt für allg. planare Graphen $m \leq 3n-6$

Alle Faces sind Dreiecke (Zyklen der Länge 3)
Jedes Face ~~besteht~~ besitzt 3 Kanten und jede Kante gehöret zu 2 Faces.

$$3 \cdot f = 2m$$

$$f = \frac{2}{3} m$$

einsetzen in Eulerformel

$$n - m + \frac{2}{3} m = 2$$

$$3n - m = 6$$

$$m = 3n-6$$

maximal planar ↔ Triangulierung

~~Beweis~~

Folgerung 2

Sei G ein planarer bipartiter Graph mit $n \geq 3$ Knoten und m Kanten. Dann gilt

$$m \leq 2n - 4$$

Beweis analog zur Folgrung 1

kleinst mögliche Fläche : 4-Eck

□

$$\Rightarrow 4f \leq 2m$$

$$f \leq \tfrac{1}{2} m$$

nach einsetzen in der Eulerformel => Beh. ■

Folgerung 3

Sei $G = (V, E)$ ein planarer Graph, dann besitzt G einen Knoten v mit $\deg(v) \leq 5$

Bew: indirekt

Annahme: $\forall v \in V : \deg(v) \geq 6$

$$m = \sum_{v \in V} \deg(v) / 2$$

$$\geq \frac{6 \cdot n}{2} = 3n$$ ↙

Das Färbarkeitproblem für planare Graphen

Knotenfärbung: k Farben $\{1, \ldots, k\}$

Finde Abb. $f: V \to \{1, \ldots, k\}$ so dass $f(v) \neq f(w)$ für alle Kanten $(v, w) \in E$

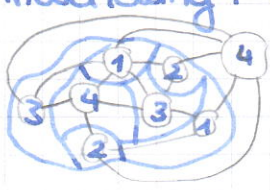
$f(v)$: Farbe von v

Frage: Wie viele Farben (k?) braucht man?

Trivial Vollständige Graph K_n => $k = n$

Für planare Graphen gilt $k = 4$ (berühmtes Problem, kompl. Beweis)

Anwendung: Farben von Ländern in Karten



↓

Faces in planarer Einbettung

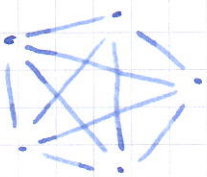
Dualer Graph

Dualer Graph: Knoten: repräsentiert die Faces

Kante (v,w) $\Leftrightarrow$ Face v und w grenzen aneinander

Flächenfärbung $\rightarrow$ Knotenfärbung des dualen G

$m \leq 3n-6 \Rightarrow K_5$ nicht planar



$n = 5 \rightarrow m \leq 9$

$m = 10$

$m = 2n-4 \Rightarrow K_{3,3}$ nicht planar

$m = 9$, $n = 6$

$2n-4 = 8$ ↙

Satz: Jeder planare Graph ist 5-färbbar

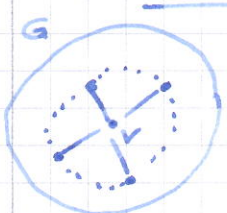
Bew (algorithmisch) $\rightarrow$ Induktion über n

IA: für $n \leq 5$

Sei $n > 5$: IA: Jeder Planare Graph mit weniger als n Knoten ist 5 färbbar

G enthält einen Knoten mit Grad ≤ 5 (Folgerung aus Euler)

Fall 1: G besitzt Knoten v mit $\deg(v) \leq 4$



IA. $\exists$ 5-Färbung für $G' = G \setminus \{v\}$

Betrachte die r Farben der Nachbarn von v $\Rightarrow r \leq 4 \rightarrow$ mind. einer Farbe frei

Färbung für G. Färbung von G' + Farbe für v

Fall 2: alle Knoten haben mind. Grad 5

Sei $v \in V$ mit $\deg(v) = 5$

Beobachtung: v besitzt 2 Nachbarknoten x, y mit $(x,y) \notin E$

Beweis (indirekt)

Annahme : Alle Paare (x,y) aus den Nachbarknoten von v sind durch Kante $(x,y) \in E$ verbunden.

$\Rightarrow$ G enthält einen K_5 als Teilgraph

$\Rightarrow$ G nicht planar ↙

Konstruire G' aus G durch Verschmelzung von x,y zu einem Knoten z und Entfernung von v.

G' ist planar und hat $n-2$ Knoten

IA : G' ist 5-färbbar

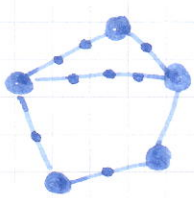
Betrachte die 4 Nachbarn von v in G' gib v eine freie Farbe.

Expandiere den Knoten z wieder zu x,y mit der Farbe von z.

■

Der Satz von Kuratowski (1930)

Satz Ein Graph G ist genau dann planar, wenn er keine Unterteilung des K_5 oder $K_{3,3}$ enthält.

Unterteilung :  ist Unterteilung von 

Allgemein

Definition von Graph-Klassen durch verbotene Teilstrukturen.

anderes Beispiel

G ist bipartit $\Leftrightarrow$ G enthält keinen Kreis ungerader Länge

Beweis

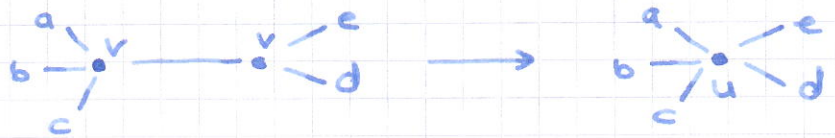
" => " : offensichtlich, da K_5 und $K_{3,3}$ nicht planar. ✓

" <= " :

Zutaten: (Übung)

Lemma 1

Jeder 3-fach zusammenhängende Graph G mit ≥ 5 Knoten besitzt eine Kante e, so dass G/e 3-fach zusammenhängend, wobei G/e ist der Graph der durch Kontraktion der Kante $e=(v,w)$ entsteht, dh. v und w werden zu einem Knoten verschmolzen und e entfernt.



Lemma 2

Sei e eine beliebige Kante in einem Graph G. Falls G/e eine Unterteilung des K_5 oder des $K_{3,3}$ enthält, dann gilt das auch für G.

Sei $G=(V,E)$ ein Graph mit $n=|V|$ der keine Unterteilung des K_5 o. $K_{3,3}$ enthält.

Induktion über n

IA: Behauptung ist wahr für $n \leq 5$

$n<5$ alle Graphen planar

$n=5$ nur K_5 ist nicht planar (u. die einzige Möglichkeit eine Unterteilung des K_5 mit $n=5$, ist der K_5 selbst und dieser ist oben ausgeschlossen.

IV: für alle Graphen mit max. n Knoten gilt die Behauptung, dh. keine Unterteilung des K_5 oder $K_{3,3}$ in G, dann G planar.

Fallunterscheidung:

1. Fall G ist nicht 3-fach zusammenhängend

Wir wissen schon: G ist planar <=> alle Blöcke von G (2-fach ZHK) sind planar

azyklische Struktur ↵
(sonst ja 3-fach ZHK)

oBdA: G ist 2-fach zusammenhängend
(und nicht 3-fach —"—)

=> G besitzt ein Split-Pair (x,y)

Entfernen von $\{x,y\}$ zerlegt G in zwei Split-Graphen G_1 und G_2

G_1 und G_2 haben beide weniger als n Knoten und enthalten keine Unterteilung des K_5 oder $K_{3,3}$

IA: => G_1 und G_2 sind planar

G_1 & G_2 sind 2-fach zusammenhängend

=> ∃ planare Einbettung für G_1 und G_2 mit x & y jeweils im äußeren Face

=> G ist planar

Fall 2 G ist 3-fach zusammenhängend

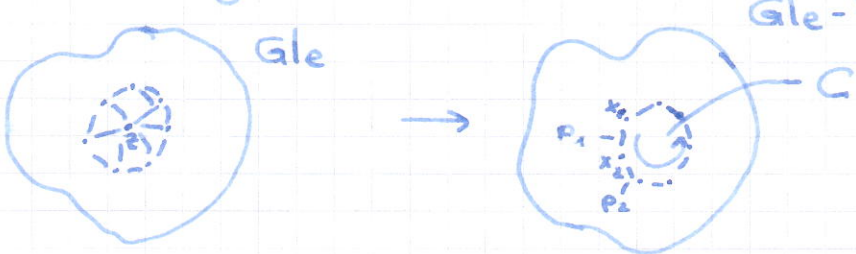
Lemma 1 => ∃ Kante e mit $G|e$ ist 3-fach zush.

Sei e eine solche Kante ($e=(x,y)$) und z der entsprechende Knoten in $G|e$

Lemma 2 => $G|e$ enthält auch keine Unterteilung des K_5 oder $K_{3,3}$ und besitzt n-1 Knoten

IA => $G|e$ ist planar

Betrachte eine planare Einbettung von $G|e$ und die planare Einbettung, die daraus durch Entfernung von z entsteht.



Sei C der Kreis / Fläche in dem z lag.

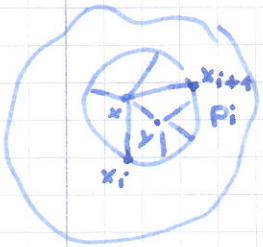
In G: Alle Nachbarn von x, y liegen auf C (außer x & y)

Seien $x_1,\dots,x_k$ die Nachbarn von x auf C in der Reihenfolge der planaren Einbettung von $G|e$. und $P_1,\dots,P_k$ die entsprechenden Teilpfade auf C

genauer: P_i Pfad von x_i zu x_{i+1} auf C (zyklisch)

Nun betrachte die Nachbarn von y auf C

Fall 1 Alle Nachbarn von y ($\neq x$) liegen auf dem selben Pfadstück P_i



Dann kann man leicht aus der Einbettung für G|e eine planare Einbettung für G konstruiren

$\Rightarrow$ G ist planar

Fall 2 Nicht alle Nachbarn von y ($\neq x$) liegen im selben P_i:

Fall 2.1 y hat ≥ 3 gemeinsame Nachbarn x_i, x_j, x_ℓ mit x
(Fall 1: höchstens 2)

→ Teilgraph mit Knoten $\{x, y, x_i, x_j, x_\ell\}$ ist ein K_5

→ G enthält Unterteilung des K_5

Fall 2.2 y hat einen Nachbarn $u \in P_i \setminus \{x_i, x_{i+1}\}$ und einen Nachbarn $v \notin P_i$

→ G enthält eine Unterteilung von $K_{3,3}$

Fall 2.3 y hat 2 Nachbarn x_i und x_j gemeinsam mit x mit $j \neq i+1$, und $i \neq j+1$ (zyklisch)
(nicht benachbart auf C)

$\Rightarrow$ G enthält eine Unterteilung von $K_{3,3}$

$\Rightarrow$ Das sind alle möglichen Fälle!

Planaritätstest

Algorithmus basierend auf Satz von Kuratowski, dh. sie testen, ob G eine Kuratowski-Unterteilung enthält.
Falls ja => G nicht planar (Ausgabe: K_5, $K_{3,3}$
sonst G ist planar (Ausgabe: planare Einbettung)

Im Prinzip liefert der Beweis (Induktion) einen rekursiven Algorithmus.

Effiziente Algorithmen (skizzen)

1. DFS (Alg. von Tarjan)

Idee Verwalte für den schon besuchten Teil des Graphen eine planare Einbettung

Fallunterscheidung: Sei e die nächste besuchte Kante $e \in \{T, F, B, C\}$

Laufzeit : $O(n+m) = O(n)$

(falls $m > 3n-6$ => nicht planar)

2. Vertex-Addition Algorithmus (Lempel & Godechaum)

Durchlaufe die Knoten in einer speziellen Reihenfolge $v_1, \ldots, v_n$

Planare Einbettung $Q \leftarrow \emptyset$

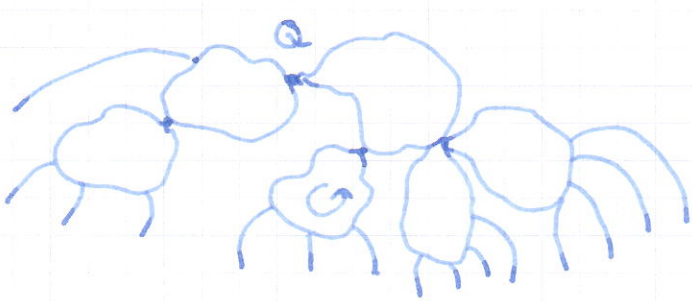
for $i = 1$ to n do

// Q ist planare Einbettung für $\{v_1, \ldots, v_{i-1}\}$

Konstruire aus Q eine Einbettung für $\{v_1, \ldots v_j\}$ (füge v_i zu Q hinzu)

od

Q ist zerlegt in seine 2-fach-Zusammenhangs-Komp.



Laufzeit $O(n)$

~~Planare~~

Planare Graphen

1. Zeichen von Graphen & Diagrammen (mit wenig Kreuzungen)

2. Effiziente Algorithmen

   es existieren oft effiziente Alg. für planare Graphen

3. Geometrie

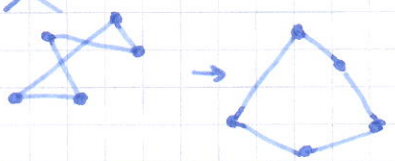
   → planare Unterteilung der Ebene

   - Menge von Strecken :

   Schnittgraph → planarer Graph

   - Polygone sind planare Graphen

   (auch Schnittgraph von Polygonen)

Algorithmische Geometrie

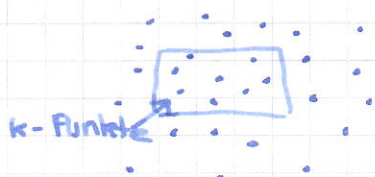
Probleme für geometrische Objekte

Beispiele :

- Schnittprobleme
- Vereinfachung
  (Approximation von komplizierten Formen)

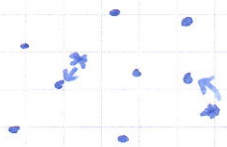
  z.B.: konvexe Hülle
- Suchen → Datenstrukturen

  i) Wörterbuchproblem für Punkte $\in \mathbb{R}^d$
  dh. d Schlüssel → $\log n$

  ii) Bereichssuche (Range - Query)
  → $\log n + k$

  iii) nächste Nachbar
  (nearest Neighbor)

  oder Post Office Problem

Hier: Schnitt einer Menge von Strecken
(Line Segment Intersection)

Eingabe: S = Menge von n Strecken (Segmente)
$\{s_1, \dots, s_n\}$ im $\mathbb{R}^2$

s_i = 

Ausgabe: Varianten

i) true / false
→ schneiden sich mind. 2 g. nicht

ii) Menge aller Schnittpunkte

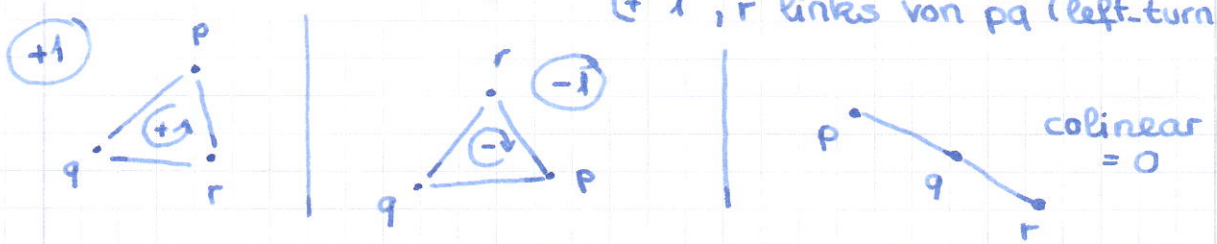
iii) planare Unterteilung der Ebene durch S (Graph)

Geometrische Grundoperationen (Primitive)

i) Test, ob ein Punkt links/rechts/auf einer Geraden liegt (im $\mathbb{R}^2$)

• Seien $p,q,r \in \mathbb{R}^2$

$$\text{orientation}(p,q,r) = \begin{cases} -1 & \text{, r rechts von } \overline{pq} \text{ (right\_turn)} \\ 0 & \text{colinear} \\ +1 & \text{, r links von } \overline{pq} \text{ (left\_turn)} \end{cases}$$

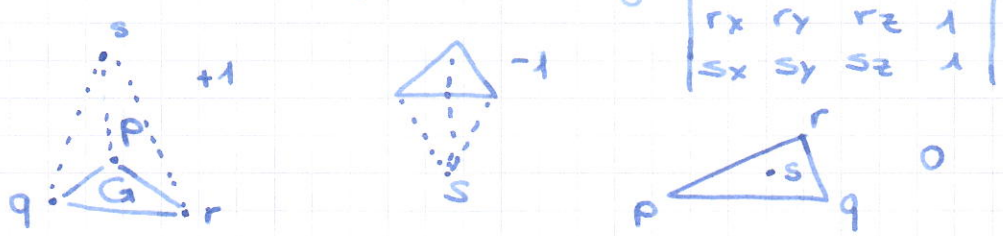


Lineare Algebra

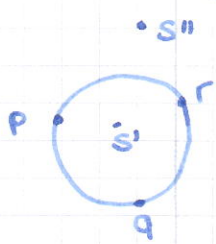
$$\text{orientation}(p,q,r) = \text{sign} \begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} \quad \text{, det}$$

• seien $p,q,r,s \in \mathbb{R}^3$

$$\text{orientation}(p,q,r,s) = \text{sign} \begin{vmatrix} p_x & p_y & p_z & 1 \\ q_x & q_y & q_z & 1 \\ r_x & r_y & r_z & 1 \\ s_x & s_y & s_z & 1 \end{vmatrix}$$



ii) Test ob 3 Punkte einen Kreis bilden und ein 4 Punkt im Kreis ist



in_circle (p,q,r,s)

Kann mit orientation realisiert werden

Projeziere p,q,r,s auf Paraboloiden

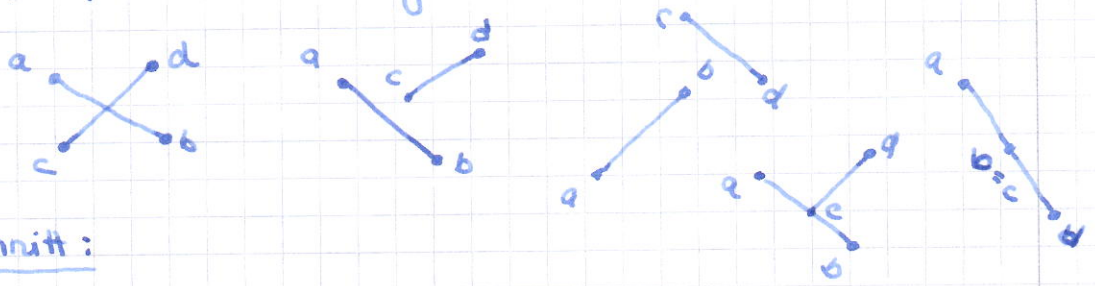
$a_x = p_x$, $a_y = p_y$ $a_z = p_x^2 + p_y^2$

$b_x = q_x \ldots$

c_x :

$\Rightarrow$ in_circle $(p,q,r,s) \Leftrightarrow$ orientation $(a,b,c,d) \geq 0$

iii) Test, ob sich 2 Segmente $\overline{ab}$, $\overline{cd}$ schneiden



Schnitt:

orientation(a,b,c) ≠ orientation(a,b,d)
∧ orientation(c,d,a) ≠ orientation(c,d,b)

(Ausnahmen werden vorher abgefragt, zB. a==c ?, ...)

iv) Schnittpunkte berechnen

→ Geradengleichung aufstellen & berechnen → (Üb)

Sonderfälle: vertikale Geraden

Zusammenfassung:

Man kann in Zeit $O(1)$ testen, ob sich 2 Segmente $\overline{ab}$, $\overline{cd}$ schneiden und den Schnittpunkt berechnen.

Sei nun S Menge von n Segmenten $S = \{s_1, \dots, s_n\}$

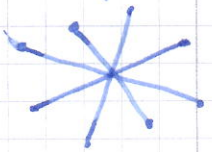
gesucht: alle Schnittpunkte zwischen Segmenten S

Naive Methode: teste alle Paare

```
forall s ∈ S do
    forall t ∈ S do
        if s ≠ t then
            p ← s ∧ t ;
            print(p) ;
        fi
    od
od
```

Laufzeit: $O(n^2)$

optimal für $\Omega(n^2)$ Schnittpunkte



schlecht wenn $k \ll n^2$

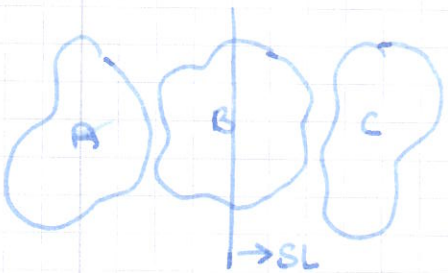
Ziel: Laufzeit abhängig von k

hier: $O((n+k) \cdot \log n)$ mit Plane-Sweep-Alg.

optimal: $O(n \log n + k)$

Plane-Sweep

Idee: Bewege eine vertikale Gerade SL (Sweepline) über die Objekte

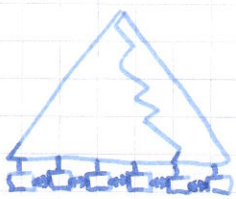


A: Berechnung abgeschlossen

B: aktive Objekte werden von SL geschnitten sind bekannt, aber Berechnung nicht abgeschlossen

C: sind noch unbekannt

1. Datenstruktur: Y-Struktur (blattorientierter balancierter Baum)



→ Objekte von unten nach oben (dh nach y-koord. der Schnittpunkte mit SL) sortiert

→ dynamisch: beim Sweep werden neue Objekte eingefügt, andere gelöscht

Zur Erkennung der Positionen (x-Koord.) der SL, an denen sich Y ändert verwendet man sog. Events (iA Punkte) Diese Events speichert wir in einer X-Struktur X nach x-Koordinaten aufsteigend sortiert. Die Sweepline SL wird vom Algorithmus immer zur nächsten Event-Position in X bewegt (dh. keine kontinuierliche Bewegung)

2. Datenstruktur: X-Struktur (Priority Queue)

→ Events nach x-Koordinaten sortiert

→ a) statisch: alle Eventpunkte bekannt

b) dynamisch: ein Teil der Events wird während des Sweep berechnet

Segmentschnitt

S = Menge von Strecken $s_1, \dots, s_n$ in allgemeiner Lage

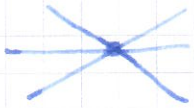
i) keine vertikalen Segmente
(dh. SL trifft zuerst linken und später rechten Endpunkt → 2 Events)

ii) Endpunkte aller Segmente sind paarweise verschieden (dh. höchstens ein Event pro Punkt)

nicht

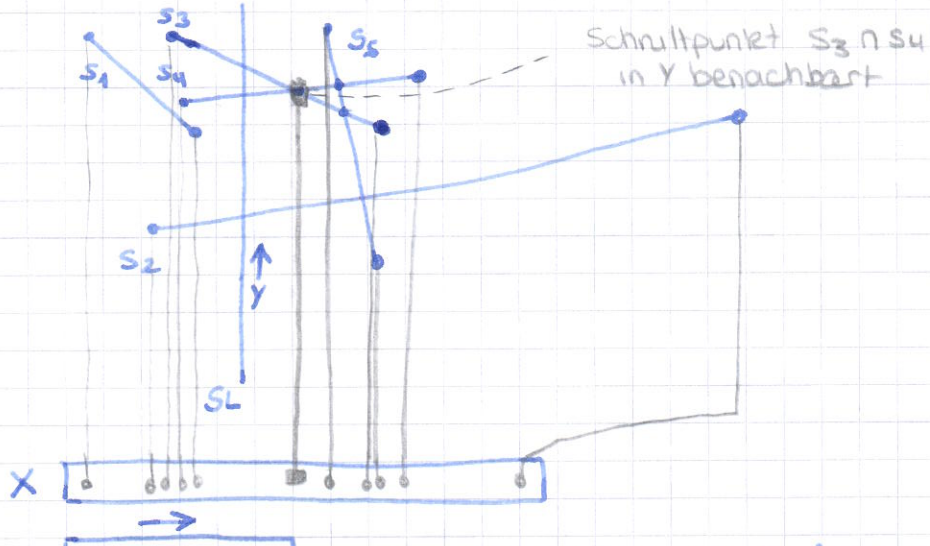
iii) Schnittpunkte auch paarweise verschieden
dh: jeder Punkt schneidet höchstens 2 Segmente

nicht:



Diese Annahmen sollen nur die Idee des Algorithmus vereinfachen - Die Fälle müssen später noch behandelt werden.

Beispiel: $S = \{s_1, s_2, s_3, s_4, s_5\}$



Y $[s_2, s_4, s_3]$

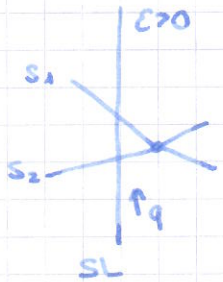
alle Segmente die im Moment von SL geschnitten werden, nach Y-Koordinaten der Schnittpunkte mit SL sortiert (von unten nach oben)

X-Struktur (Event-Queue)

Alle Endpunkte von Segmenten (statischer Teil) und Schnittpunkte von in der Y-Struktur benachbarten Segmenten rechts von SL.

Folgerung:

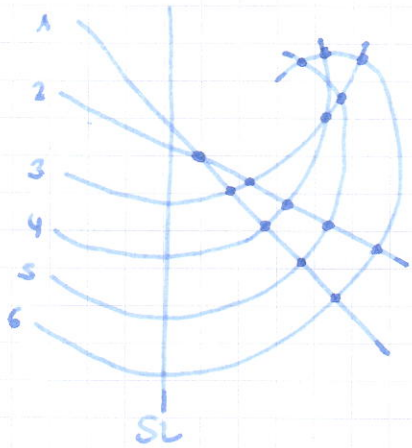
i) Alle Schnittpunkte werden als Events vorkommen dh werden in X eingefügt => Korrektheit



sei $q = s_1 \cap s_2$, dann $\exists\, \varepsilon > 0$, so dass in Y-Struktur bei SL-Position $q_x - \varepsilon$ s_1 und s_2 benachbart sind

Aus Invariante, dass X alle Schnittpunkte von in Y benachbarten Segmenten enthält folgt Korrektheit.

ii) Platzbedarf für die X-Struktur ist $O(n)$ denn es gibt maximal n benachbarte Paare in Y



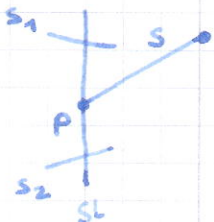
nicht $O(n^2)$

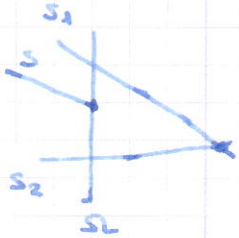
Der Sweep-Algorithmus für den Segmentschnitt

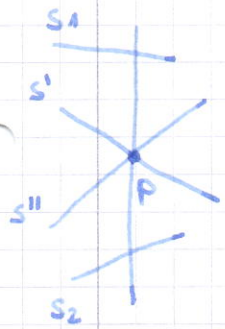
```
xpos ← -∞        // Position von SL
X-Struktur X ← ∅  // PQ von x-Koord.
Y-Struktur Y ← ∅  // sortierte Folge von Segm. (bin Baum)

forall s ∈ S do
    X.insert(s.left);
    X.insert(s.right);
od

while X ≠ ∅ do          // verarbeite Events
    p ← X.delmin();
    xpos ← p.x          // schiebe SL in Punkt p
    // Fallunterscheidung
    case p linker Endpunkt von s
        Y.insert(s);
        s1 ← Y.succ(s);
        s2 ← Y.pred(s);
        X.delete(s1 ∩ s2);
        X.insert(s ∩ s1);
        X.insert(s ∩ s2);
        // Achtung: s1, s2, s1 ∩ s2, s ∩ s1, s ∩ s2
        //          existieren nicht immer
    case p rechter Endpunkt von s
        s1 ← Y.succ(s);
        s2 ← Y.pred(s);
        Y.delete(s);
        X.insert(s1, s2);
        // auch hier Achtung!
```







case p Schnittpunkt von s' und s''

$s_1 \leftarrow Y.succ(s')$;
$s_2 \leftarrow Y.pred(s'')$;

$Y.swap(s', s'')$;

$X.delete(s' \cap s_1)$;
$X.delete(s'' \cap s_2)$;
← vorher testen ob Schnittpunkte existieren

$X.insert(s'' \cap s_1)$;
$X.insert(s' \cap s_2)$;

// Ausgabe:

$p = s' \cap s''$

od

Analyse des Planesweep-Algorithmus

Beobachtung: Die Hauptschleife bearbeitet alle Endpunkte und jeden Schnittpunkt genau einmal.

$\Rightarrow 2n + k$ Durchläufe (k = #Schnittpunkte)

Im Rumpf werden je nach Fall konstant viele Operationen ausgeführt

- Schnitttest
- Operationen auf Y
  (Y.insert, Y.succ, Y.delet, Y.swap)
- Operationen auf X
  (X.delmin, X.insert, X.delete)

Alle Operationen auf X- & Y-Struktur kann in Zeit $O(\log n)$ realisieren (beachte, dass beide höchstens $O(n)$ Objekte enthalten)

z.B.: durch binäre Suche

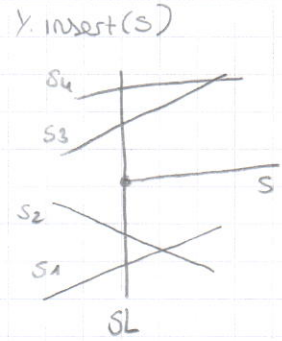
Alternative: Heap für X-Struktur (stat PQ)

Gesamtlaufzeit: $O((n+k) \cdot \log n)$

Grundoperationen (Primitive)

- Schnitte (Üb) → orientation
- Vergleiche in Y-Struktur

zu Vergleiche in Y



if (s<s') then

sei f(x) die Geradengleichung
y' ← f(xpos)
y ← s.left.y

if y<y' then

Es lohnt sich hier, alle Geradengleichungen in einem Reprocessing zu bestimmen (keine Vertikalen!)

Einige Ideen zur Verallgemeinerung

Eingabe nicht in allgemeiner Lage:

i) mehrere Events an einer Position

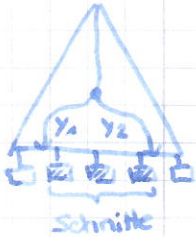
1. Verwende lexikogr. xy-Ordnung für die Events in x-Struktur

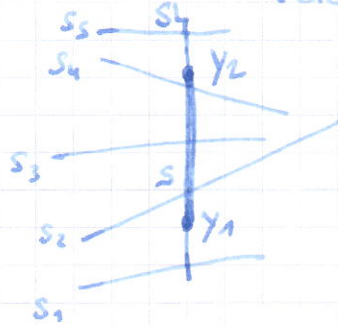
2. Bearbeitung eines Events p

keine Fallunterscheidung, sondern gleichzeitig:
- Einfügen aller startenden
- Löschen aller endenden
- Swap aller durchgehenden

ii) Vertikale Segmente

können nicht in Y eingefügt werden (da keine Geradengleichung für Vertikale)

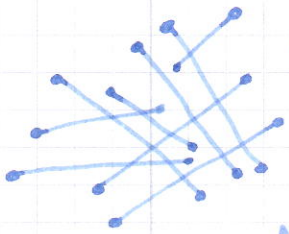




Sei $s = (x_{pos}, y_1, y_2)$

s schneidet alle Segmente in Y zwischen y_1 und y_2

Erweiterung: Berechne den planaren Graphen $G = (V, E)$ bzw. seine planare Einbettung



V: Menge der End- und Schnittpunkte und zwischen jedem auf einem Segment benachbarten Paar von Knoten (v,w) existiert eine Kante $(v,w) \in E$

Anwendung:
- Schnitt von Polygonen $P \cap Q$

S ← Menge aller Segmente aus P & Q
Graph ← sweep(S)
Finde alle Faces aus P & Q

Berechnung von $G = (V, E)$

1. Jeder Eventpunkt p erzeugt einen Knoten v
2. Speichere mit jedem Segment $s \in Y$

   $last(s)$ = der zuletzt erzeugte Knoten auf s (linker Endknoten oder Schnittpunkt)

3. für Schnittpunkte und rechte Endpunkte von s

   - erzeuge Knoten v
   - erzeuge Kante $(v, last(s))$
   - $last(s) \leftarrow v$

! erzeuge die Kanten im Uhrzeigersinn !

Sonderfälle (Übung)

- nur vertikale oder horizontale Segmente (achsen-parallel)

  → einfacherer Algorithmus

- Red-Blue-Segmentschnitt

  2 Mengen von Segmenten S_1, S_2 (disjunkt)

  → es interessieren nur die Schnitte zwischen blau (S_1) und rot (S_2)

  Anwendung: Polygonschnitt

Point Location

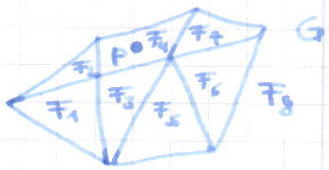
Gegeben: Planare Unterteilung des $\mathbb{R}^2$ dh. durch planaren Graphen $G = (V, E)$

mit i) jeder Knoten $v \in V$ besitzt Position $pos(v) \in \mathbb{R}^2$ (einfach $v \in \mathbb{R}^2$)

ii) jede Kante $(v, e) \in E$ ist Strecke $\overline{pos(v)pos(w)}$

Beispiele
- Schnittgraph

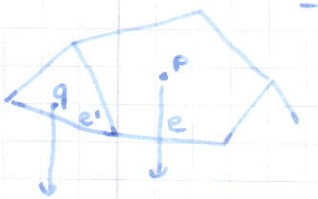
- planare Unterteilung durch geschlossen Flächen (speziell: Triangulierung)



Problem: Für viele Anfragen nach Punkt $p \in \mathbb{R}^2$

Locate(p): liefert die Fläche (Face) von G, die p enthält

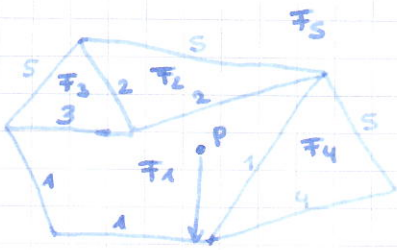
äquivalent: vertikale Sichtbarkeit



Für $p \in \mathbb{R}^2$ finde die Kante e in G, die von einem vertikalen nach unten gerichteten Strahl als erstes getroffen wird.

Dazu beschrifte alle Kante $e \in E$ (nicht vertikal) mit Face oberhalb von e

Beispiel



Für viele Anfragen lohnt es sich eine Datenstruktur aufzubauen (Preprocessing), die dann die einzelnden Fragen effizient beantwortet.

Bemerkung: für wenige (zB.: O(1)) Anfragen ist die triviale Lösung optimal
→ Lineare Suche nach der Kante mit kleinsten vertikalen Abstand => O(n)

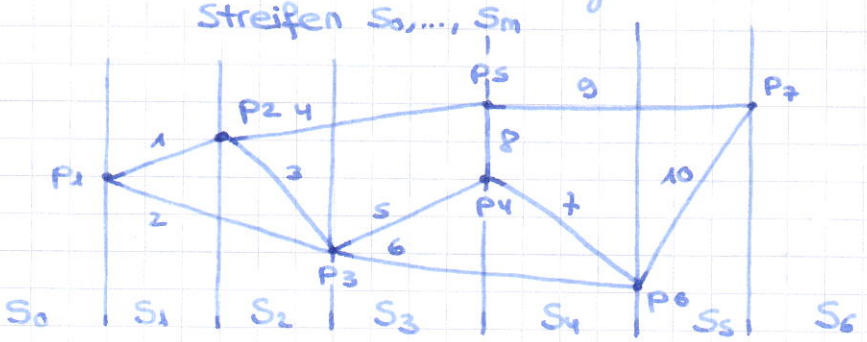
Lösungen für Point Location

(vertikale Sichtbarkeit)

1. Die Streifenmethode

Sei $G=(V,E)$ die Unterteilung mit $V=\{p_1,\dots,p_n\}$, $p_i \in \mathbb{R}^2$ ($\Rightarrow |E| \leq 3n-6 = O(n)$)

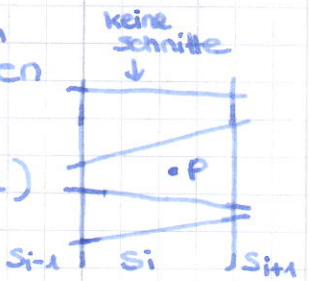
Idee: Ziehe durch jeden Knoten p_i eine vertikale Gerade. Dies zerlegt die Ebene in (max) $n+1$ Streifen $S_0,\dots,S_m$



$p_1,\dots,p_n$ sind in xy-ordnung sortiert

Locate(p) mit $p=(x,y)$

1. Finde den Streifen S_i der p enthält durch binäre Suche nach x auf den x-Koordinaten der Punkte $p_1,\dots,p_n$
2. Dann sortiere den Punkt p (→binäre Suche) in Liste der Kanten des Streifen S_i ein



Datenstruktur: Feld von Feldern

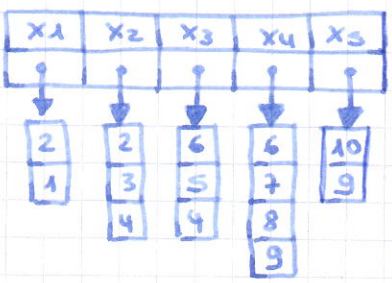
Primäres Feld: $A[1,\dots,n]$

$A[i]$: x Koordinaten von p_i + Verweis auf Feld $B[i]$ (realisiert Streifen S_i)

Sekundäres Feld:

$B_i[j]$: Folge von den Kanten in Streifen S_i von unten nach oben sortiert

Beispiel



Streifen − 2

Aufbau durch einfachen Sweep

locate(p)

i ← A.binary_search(p_x)

B ← A[i].Streifen ← Referenz auf sekundäres Feld

j ← B.binary_search(p)

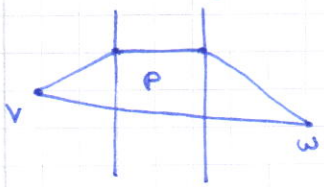
return j

Binare Suche nach p in B_i ~~Feld~~

als Vergleich p unterhalb von e <=> orientation(v,w,p) < 0

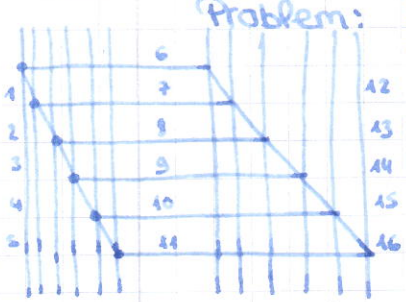
p auf e <=> orientation(v,w,p = 0

p überhalb von e <=> orientation(v,w,p) > 0



Laufzeit: $O(\log n)$, da 2x binare Suche (optimal)

Problem: Speicherplatz quadratisch sein dh. $\Omega(n^2)$
=> Aufbauzeit



wenn fast alle sekundären Felder B_i Länge $O(n)$ haben

=> $n/2$ der Kanten schneiden ca die Hälfte der n streifen. $\approx 1/4\, n^2$

=> Platz $\Omega(n^2)$
=> Aufbauzeit $O(n^2 \log n)$

Zusammenfassung

Bei nicht degenerierter Eingabe (dh. allgemeine Lage)

Preprocessing: $O(n \log n)$

Platz : $O(n)$

Suchzeit : $O(\log n)$

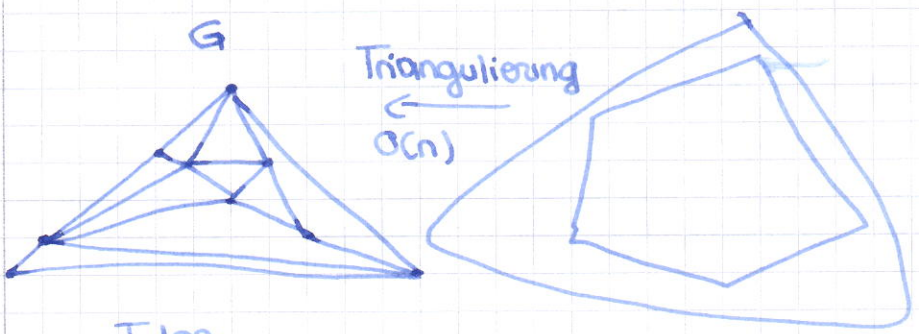
$O(n^2 \log n)$

$O(n^2)$ ↑ degeneriert

2. Lösung : Triangulierungsverfeinerung

garantiert optimales Verhalten aber viel aufwendiger und daher praktisch evtl. langsamer

Gegeben Eine Triangulierung der Ebene $G = (V, E)$ dh. planare Unterteilung, die nur Dreiecke als Flächen besitzt, auch die äußere Fläche



Idee

Konstruiere Folge von Triangulierung $S_1, \dots, S_k$ so dass gilt

i) $S_1 = G$

ii) S_k = äußeres Dreieck von G

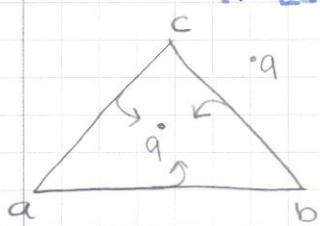
iii) $k = O(\log n)$

iv) S_{i+1} besteht aus einem konstanten Bruchteil der Kanten von S_i (zB 1/2 o. 1/16)

v) für jeden Punkt q, den man in S_{i+1} lokalisiert hat (dh. Dreieck von q bekannt) kann man in $O(1)$ zeit das entsprechende Dreieck in S_i finden.

Locate(q)

1. Lokalisiere q in S_k



2 Fälle dh. 2 mögliche Δ (außen t_0, innen t_1)

orientation(a,b,q) ≥ 0 ∧
orientation(b,c,q) ≥ 0 ∧
orientation(c,a,q) ≥ 0
} t_1.contains(q) ↳ Dreieck t

2. for i = k-1 to 1 do

lokalisiere q in S_i (siehe v)) in $O(1)$
$t \leftarrow S_i$.locate(S_{i+1}, t)
} $O(\log n)$ da $k = O(\log n)$ siehe ii)

Konstruktion der Datenstrukturen dh. Folge $S_1, \dots, S_k$
$S_1 \leftarrow G$

Frage: Wie kommt man von S_i nach S_{i+1}?

Sei $S_i = (V_i, E_i)$. Wir entfernen eine Knotenmenge $I \subset V_i$ mit folgenden Eigenschaften:

1. $|I| \geq c \cdot |V|$ z.B. $c = 1/10$ konstanter Bruchteil
2. I enthält keinen der 3 äußeren Kanten a,b,c
3. I ist eine unabhängige Knotenmenge von S_i d.h. $E_i \cap (I \times I) = \emptyset$

(keine Kante innerhalb von I)

wenn man nur solche unabhängigen Kanten entfernt, dann entstehen nur sternförmige (star shaped) Flächen (Polygone)

Diese kann man einfach in Zeit $O(\deg(v))$ retriangulieren.

Beobachtung: Wenn t Dreieck in S_{i+1} mit $q \in t$, dann gibt es in S_i nur $\deg(v)$ Kandidaten.

4. Für alle Knoten $v \in I$: $\deg(v) \leq c'$, c' konstante

(zur Erinnerung in planaren Graphen existiert immer ein Knoten von Grad ≤ 5)

Beobachtung: Schritt von $S_{i+1} \to S_i$ in Zeit $O(1)$

Lemma: Jede Triangulierung S_i enthält eine unabhängige I mit $|I| \geq \lceil 1/12 \cdot (n/2 - 3) \rceil$ $(n = |V_i|)$ so dass jeder Knoten $v \in I$ einen Grad von maximal 11 besitzt und $v \notin \{a,b,c\}$

Beweis durch greedy Algorithmus

1. markiere die äußeren Knoten a,b,c
2. $I \leftarrow \emptyset$
3. repeat

   $v \leftarrow$ bel. nicht markierter Knoten mit Grad ≤ 11
   $I \leftarrow I \cup \{v\}$
   markiere v und alle Nachbarn von v
   until $\nexists$ unmarkierten Knoten mit Grad ≤ 11

$\Rightarrow$ I unabhängig
Grad ≤ 11
zz.: $|I| \geq \lceil 1/12 \cdot (n/2 - 3) \rceil$ $\in$ Euler-Formel

$S_{i+1} \rightarrow S_i$

- berechne I und entferne die Knoten
- triangulière die sternförmigen Flächen
  → neue Dreiecke
- speichere für jedes neue Dreieck t in S_{i+1} Pointer auf die (≤ 11) alten Dreieck t' in S_i mit $t \cap t' \neq \emptyset$

Datenstruktur

besteht aus Knoten (DAG)

Wurzel: Dreieck von S_k

Blätter: alle Dreiecke von S_1 ($= G$)

innere Knoten: Dreiecke durch Retriangulierung

Platz: $O(n)$

Zeit: $O(\log n)$

Aufbau: $O(n)$