Zusammenfassung für EAA

Wintersemester 2013/2014

von Dagmar Sorg

Divide and Conquer

1 MergeSort

1.1 Laufzeit

1. Aufteilung der n Elemente in zwei Instanzen mit $\left\lceil \frac{n}{2} \right\rceil$ und $\left\lfloor \frac{n}{2} \right\rfloor$ Elementen
2. rekursive Lösung des Problems
3. Laufzeit von Merge ist **linear**
4. es gibt Konstanten c_1, c_2, sodass die Laufzeit der folgenden entspricht:
   $T(n) \leq T(\left\lceil \frac{n}{2} \right\rceil) + T(\left\lfloor \frac{n}{2} \right\rfloor) + c_2 \cdot n (\text{falls } n > 1), T(1) = c_1$

2 Substitutions-Methode

Raten einer Laufzeit mit Beweis durch Induktion

2.1 Raten durch Ähnlichkeit

sehen, dass eine Rekursionsformel asymptotisch ähnlich ist wie eine andere

2.2 Raten durch Verändern der Variablen

Beispiel $(T(n) = 2T(\sqrt{n}) + \log n)$**:** $n = 2^m, S(m) = T(2^m) = 2 \cdot T(2^{\frac{m}{2}}) + m = 2 \cdot S(\frac{m}{2}) + m$
$\Rightarrow S(\frac{m}{2}) \in O(m \log m)$
$\Rightarrow$ Rücksubstitution: $T(n) \in O(\log n \log \log n)$

2.3 Induktionsbehauptung stärker machen

wenn die Annahme richtig ist, aber die Induktionsvorraussetzung zu schwach ist

Beispiel $\left(T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1\right)$**:** Annahme: $T(n) \in \mathcal{O}(n)$
$\Rightarrow T(n) = c \cdot \left\lfloor \frac{n}{2} \right\rfloor + c \cdot \left\lfloor \frac{n}{2} \right\rfloor = cn + 1,$
aber das heißt noch nicht, dass $T(n) \leq cn$.
Wir nehmen das Folgende an:
$T(n) \leq c \cdot \left\lfloor \frac{n}{2} \right\rfloor - b + c \cdot \left\lceil \frac{n}{2} \right\rceil - b + 1 = cn - 2b + 1 \leq cn - b$, falls $b \geq 1$.

3 Iterative Methode

Iteratives Lösen von Rekursionsgleichungen, sodass die Rahmenbedingungen stimmen

Beispiel $\left(T(n) = \begin{cases} c_1 & \textbf{falls } n \leq 3 \\ 3 \cdot T(\left\lfloor \frac{n}{4} \right\rfloor) + c_2 \cdot n & \textbf{sonst} \end{cases}\right)$**:**

$$
\begin{aligned}
T(n) &= 3 \cdot T(\left\lfloor \frac{n}{4} \right\rfloor) + c_2 \cdot n \\
&= 3 \cdot \left(3 \cdot T(\left\lfloor \frac{n}{16} \right\rfloor) + c_2 \cdot n \left\lfloor \frac{n}{4} \right\rfloor\right) + c_2 \cdot n \\
&= 3 \cdot \left(3 \left(3 \cdot T(\left\lfloor \frac{n}{64} \right\rfloor) + c_2 \cdot n \left\lfloor \frac{n}{16} \right\rfloor\right) + c_2 \cdot \left\lfloor \frac{n}{4} \right\rfloor\right) + c_2 \cdot n \\
&= c_2 \cdot \sum_{i=0}^{k-1} 3^i \left\lfloor \frac{n}{4^i} \right\rfloor + 3^k T\left(\left\lfloor \frac{n}{4^k} \right\rfloor\right)
\end{aligned}
$$

Die Randbedingungen gelten, falls $\frac{n}{4^k} < 4$, bzw. falls $k > \log_4 n - 1$ für das kleinste k. Somit erhalten wir

$$\begin{aligned} T(n) &\leq c_2 \cdot \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + c_1 \cdot 3^{\log_4 n} \\ &\leq 4c_2 \cdot n + c_1 \cdot n^{\log_4 3} \\ &\leq (4c_2 + c_1) \cdot n \end{aligned}$$

$\Rightarrow T(n) \in \mathcal{O}(n)$

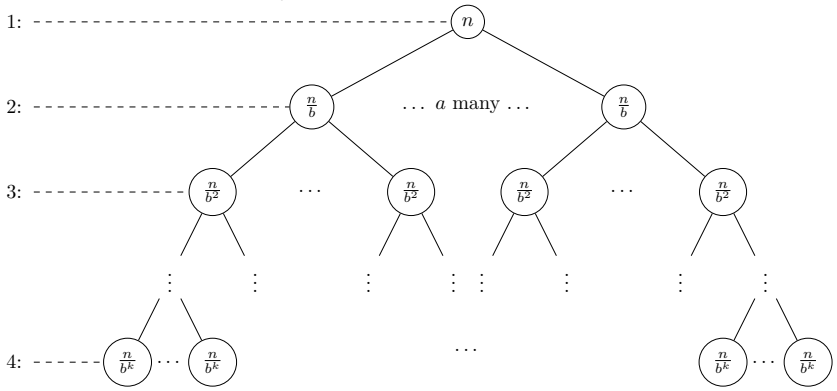
4 Master Methode (Master Theorem)

a) generelle Lösung für Rekursionsformeln der Form
$T(n) = a \cdot T(\frac{n}{b}) + f(n)$

b) $a, b \geq 1$ sind Konstanten

c) $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$

d) erste Annahme: $n = b^k$ $\left(\frac{n}{b^k} = 1 \Leftrightarrow k = \log_b n\right)$:



1. $f(n)$

2. $f(n) + a \cdot f\left(\frac{n}{b}\right)$

3. $f(n) + a \cdot f\left(\frac{n}{b}\right) + a^2 \cdot f\left(\frac{n}{b^2}\right)$

4. $f(n) + a \cdot f\left(\frac{n}{b}\right) + a^2 \cdot f\left(\frac{n}{b^2}\right) + c_0 \cdot a^k$ (wobei $k \approx \log_b n$)

Endsumme: $c_0 \cdot \underbrace{a^{\log_b n}}_{n^{\log_b a}} + \sum_{i=0}^{\log_b n - 1} a^i \cdot f\left(\frac{n}{b^i}\right)$

e) somit gilt in Rekursionsschritt i: zusätzlicher Aufwand von $a^i f\left(\frac{n}{b^i}\right)$

f) falls in Rekursionstiefe k der Wert $\frac{n}{b^k}$ klein genug ist, kann er durch die Konstante c_0 ersetzt werden

4.1 Laufzeit

$$T(n) = c_0 \cdot \underbrace{a^{\log_b n}}_{n^{\log_b a}} + \sum_{i=0}^{\log_b n - 1} a^i \cdot f\left(\frac{n}{b^i}\right)$$

4.2 Laufzeitbestimmung mit dem Master Theorem

$a \geq 1, b > 1, \epsilon > 0, f : \mathbb{N} \to \mathbb{R}_{\geq 0}$, sowie $T(n) = a \cdot T(\frac{n}{b}) + f(n)$ ($\frac{n}{b}$ ist entweder $\left\lfloor \frac{n}{b} \right\rfloor$ oder $\left\lceil \frac{n}{b} \right\rceil$)

Fall 1: Voraussetzung: $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon})$ für beliebiges $\epsilon > 0$

Folgerung: $T(n) \in \mathcal{O}(n^{\log_b a})$

Beispiel: $T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$

$\Rightarrow a = 8, b = 2, f(n) = 1000n^2, \log_b a = \log_2 8 = 3$

$\Rightarrow 1000n^2 \in \mathcal{O}\left(n^{3-\epsilon}\right)$

Fall 2: Voraussetzung: $f(n) \in \Theta\left(n^{\log_b a}\right)$

Folgerung: $T(n) \in \Theta\left(n^{\log_b a} \log n\right)$

Beispiel: $T(n) = 2T\left(\frac{n}{2}\right) + 10n$

$\Rightarrow a = 2, b = 2, f(n) = 10n, \log_b a = \log_2 2 = 1$

$\Rightarrow 10n \in \Theta\left(n^1\right)$

Fall 3: Voraussetzung: $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ für ein $\epsilon > 0$ und falls die Regularitätsbedingung gilt (ein c mit $0 < c < 1$: $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$)

Folgerung: $T(n) \in \Theta(f(n))$

Beispiel: $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

$\Rightarrow a = 2, b = 2, f(n) = n^2, \log_b a = \log_2 2 = 1$

$\Rightarrow n^2 \in \Omega\left(n^{1+\varepsilon}\right)$

Regularitätsbedingung: $2\left(\frac{n}{2}\right)^2 \leq c \cdot n^2 \Leftrightarrow \frac{1}{2}n^2 \leq cn^2$

$\Rightarrow T(n) \in \Theta(n^2)$

5 Anwendung

5.1 Matrix Multiplikation

Problem: Multiplikation zweier $n \times n$ Matrizen

Eingabe: Matrizen $A, B \in \mathbb{R}^{n \times n}$

$$\overbrace{\begin{pmatrix} a_{11} & \dots & a_{1j} & \dots & a_{1n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i1} & \dots & a_{ij} & \dots & a_{in} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nj} & \dots & a_{nn} \end{pmatrix}}^{A} \cdot \overbrace{\begin{pmatrix} b_{11} & \dots & b_{1j} & \dots & b_{1n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{i1} & \dots & b_{ij} & \dots & b_{in} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nj} & \dots & b_{nn} \end{pmatrix}}^{B} = \overbrace{\begin{pmatrix} c_{11} & \dots & c_{1j} & \dots & c_{1n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ c_{i1} & \dots & c_{ij} & \dots & c_{in} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nj} & \dots & c_{nn} \end{pmatrix}}^{C}$$

Ausgabe: Matrix C

Laufzeit: $n^3 + n^2(n-1) \in \Theta(n^3)$

Idee zur Verbesserung der Laufzeit 1 (Divide and Conquer):

1. Aufteilung der Matrizen in 4 $\frac{n}{2} \times \frac{n}{2}$ Matrizen $\Rightarrow C_{ij} = A_{i1} \cdot B_{1j} + A_{i2} \cdot B_{2j}, 1 \leq i, j \leq 2$
2. Laufzeit:
   $T(n) = 8T\left(\frac{n}{2}\right) + 4 \cdot n^2$
   $\overset{\text{Master Theorem (1)}}{\Rightarrow} \Theta(n^3)$

Idee zur Verbesserung der Laufzeit 2 (Strassen):

1. Multiplikation von nur sieben Matrizenpaaren, sowie nur 18 Additionen von Matrizen (Idee: Merken von berechneten Werten)
2. Laufzeit:
$$T(n) = \begin{cases} n^3 + n^2(n-1) & \text{falls } n \leq 2^{k_0} \text{ für eine Konstante } k_0 \geq 0 \\ 7T\left(\frac{n}{2}\right) + 18 \cdot \left(\frac{n}{2}\right) & \text{sonst} \end{cases}$$
$\overset{\text{Master Theorem (1)}}{\Rightarrow} \Theta(n^{\log_2 7}) \subset \mathcal{O}(n^{2.91})$ (wobei n eine Zweierpotenz ist)

Beste asymptotische Laufzeit: Bei einem Algorithmus von Coppersmith und Winograd (1990): $\mathcal{O}(n^{2.37\cdots})$. Es gibt auch Algorithmen mit einer geringeren asymptotischen Laufzeit, aber mit riesigen Konstanten.

5.2 Selection

- in einer Menge A mit n Elementen mit einer totalen Ordnung $\leq$ wird das k-kleinste Element gesucht
- einfacher Algorithmus: sortieren der Elemente und herausnehmen des k-ten (Laufzeit: $\mathcal{O}(n \log n)$)
- rekursiver Ansatz in $\mathcal{O}(n)$:
  1. die Menge A wird in zwei Teile A_1, A_2 geteilt, sodass $x < y$ für jedes $x \in A_1, y \in A_2$
  2. je nachdem ob $|A_1| \geq k$ arbeitet der Algorithmus auf A_1 oder A_2 weiter
  3. zuerst wird A in Gruppen der Größße 5 aufgeteilt, dann kann der Median m der Medianen der $\left\lceil \frac{n}{5} \right\rceil$ Gruppen durch den rekursiven Aufruf von SELECT berechnet werden
- Vergleich Algorithmen, 21

Amortisierte Analyse

Ein Algorithmus kann aus mehreren Operationsabfolgen bestehen. Hier kann man eine obere Grenze der Worst-Case-Laufzeit bestimmen, indem man die Worst-Case-Laufzeit einer Operation nimmt und sie mit der Anzahl an Operationen multipliziert. Die wirkliche Worst-Case-Laufzeit kann jedoch besser sein.

Beispiel (*MultiPop*)**:**

Push(*element***):** *element* wird dem Stack hinzugefügt

MultiPop(k**):** k Elemente werden vom Stack geholt (wenn weniger als k Elemente auf dem Stack sind, werden alle geholt)

1 Accounting Methode (Abrechnungsverfahren)

1. Idee: Bezahlen für mögliche kommende Operationen mithilfe von amortisierten Kosten $\hat{c}$
2. $c - \hat{c}$ (c sind die wirklichen Kosten) sind die reservierten Kosten für spätere Operationen, dessen $\hat{c}$ nicht für die wirklichen Kosten ausreichen
3. für $\hat{c}$ gilt: $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$ und ist somit eine obere Grenze der Gesamtkosten

Beispiel (*MultiPop (Fortsetzung)*)**:**

1. aktuelle Kosten für Push: 1 Einheit
2. aktuelle Kosten für MultiPop: $\min(k, |S| + 1)$
3. amortisierte Kosten für Push: 2 Einheit (1 für Push, die andere für MultiPop)
4. amortisierte Kosten für MultiPop: 1 Einheit (benötigt, falls $k > |S|$)

Alle Kosten sind konstant $\Rightarrow$ Laufzeit ist linear (in $\mathcal{O}(n)$)

2 Potentialfunktionsverfahren

1. definieren einer Potentialfunktion Φ, die jedem möglichen Zustand einer Datenstruktur einen Wert zuweist
2. bei einer Abfolge von n Operationen erhalten wir: $\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\text{Potentialdifferenz}}$
   mit D_i ist Zustand der Datenstruktur nach der i-ten Operation und D_0 Startzustand vor der ersten Operation
   $\Rightarrow \sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i + \Phi(D_0) - \Phi(D_n)$
3. wenn Φ so gewählt ist, dass $\Phi(D_n) \geq \Phi(D_0)$, dann ist $\sum_{i=1}^{n} \hat{c}_i$ eine Obergrenze der Gesamtkosten des Algorithmus.

Beispiel (*MultiPop (Fortseztzung)*)**:**

1. Φ ist die Anzahl $|S|$ der Elemente auf dem Stack S
2. amortisierte Kosten von Push: $\hat{c} = 1 + \Phi(D_1) = 1 + 1 = 2$
3. amortisierte Kosten von MultiPop(k): $\hat{c} = \min(k, |S| + 1) - \min(k, |S|) \in \{0, 1\}$

Somit ist die Laufzeit linear ($\in \mathcal{O}(n)$).

Union-Find-Datenstruktur

1. es wird eine endliche Menge X verwendet
2. Ziel: dynamische Menge $\mathcal{S}$ von disjunkten Teilmengen von X
3. vorhandene Methoden:

   MakeSet(item x): erstellt eine neue Menge nur mit dem Item x ($\{x\}$)

   Find(item x): gibt die Menge mit dem Item x zurück

   Union(set i, set j): erstellt eine neue Menge mit den Mengen i, j und löscht die beiden Mengen i, j
4. an kann annehmen dass $X = \{1, \ldots, n\}$ mit $n \in \mathbb{N}$ ist, da man für andere Mengen jedem Item eine einzigartige Zahl zuordnen kann
5. jede Menge hat einen **Repräsentanten**, FINDgibt diesen zurück, UNIONbekommt diese als Argumente

Im Folgenden betrachten wir eine Sequenz mit m Operationen MAKESET, FIND und UNION, wobei n die Anzahl an MAKESET-Operationen ist.

1 Array Darstellung

Beispiel ($\mathcal{S} = \{\{1,3,5,7\},\{2,4,8\}\}, X = \{1,\ldots,9\}$)**:**

Item x	1	2	3	4	5	6	7	8	9
Menge $A[x]$	1	2	1	2	1	0	1	2	0

Laufzeiten:

MakeSet: $\Theta(1)$

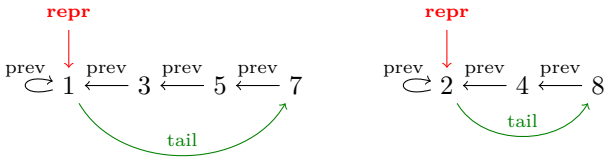
Find: $\Theta(1)$

Union: $\Theta(n)$

2 LinkedList Darstellung

Zur Reduzierung der Laufzeit von UNION

Beispiel ($\mathcal{S} = \{\{1,3,5,7\},\{2,4,8\}\}, X = \{1,\ldots,9\}$)**:**



Laufzeiten:

MakeSet: $\Theta(1)$

Find: $\Theta(n)$

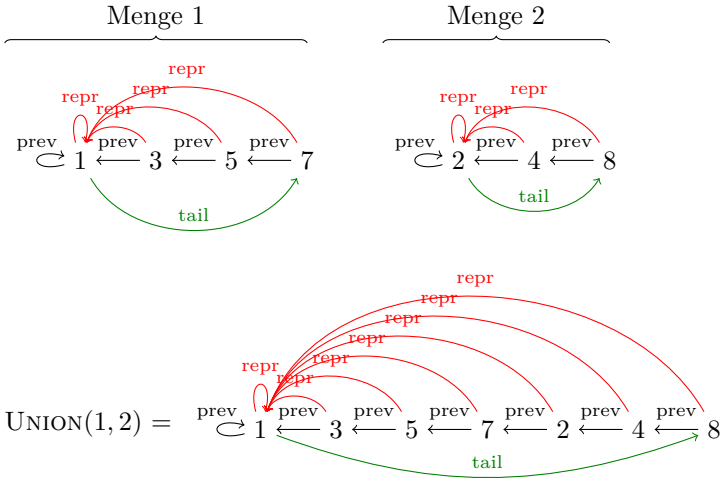
Union: $\Theta(1)$

Gesamtlaufzeit für $n-1$ Union und m Find: $\Theta(m \cdot n)$

$\Rightarrow$ keine Verbesserung der Laufzeit

2.1 Erweiterte LinkedList Darstellung

Beispiel ($\mathcal{S} = \{\{1,3,5,7\},\{2,4,8\}\}, X = \{1,\dots,9\}$)**:**



Wenn man die Länge jeder Liste speichert und immer die kürzere Liste an die Längere hängt, wird jeder Repräsentanten-Zeiger höchstens $\lfloor \log n \rfloor$-mal verändert werden.
Laufzeit von einer Sequenz mit m Operationen (MakeSet, Union, Find) liegt in $\mathcal{O}(m + n \log n)$

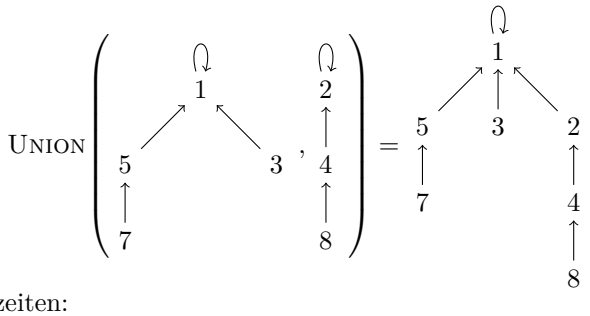
3 Rooted Tree Darstellung

Repräsentant: Wurzel des zugehörigen Baumes

Union(a, b)**:** Anhängen der Wurzel von a an Wurzel von b

Find(a)**:** Aufsteigen im Baum bis zur Wurzel von a

Beispiel (**Union**$(1,2)$)**:**



Laufzeiten:

MakeSet: $\Theta(1)$

Union: $\Theta(1)$

Find: Die Laufzeit von Find ist anhängig von der Höhe des Baumes. Wenn Union einfach ohne Überprüfung der Höhe der Bäume durchgeführt wird, liegt Find in $\Theta(n)$.

3.1 gewichtete Vereinigung (weighted Union)

Es wird der kleinere Baum an den größeren angehängt. Damit das möglich ist, wird die Größe jedes Baumes folgendermaßen gespeichert: $parent[root] = -size$.
Wenn ein Baum aus mehreren *weighted* Union Operationen entstanden ist, so gilt: $h(T) \leq \log |T|$, wobei $h(T)$ die Höhe des Baumes und $|T|$ die Anzahl der Elemente in T ist.

Baum T_j wurde an Baum T_i angehängt. Dann gilt: $h(T) = \max(h(T_j) + 1, h(T_i))$. Somit entstehen zwei Fälle:

1. $h(T_i) > h(T_j) + 1 \Rightarrow h(T) = h(T_i) \leq \log |T_i| < T$ ✔
2. $h(T_i) \leq h(T_j) + 1$
   $\Rightarrow h(T) = h(T_j) + 1 \leq \log |T_j| + 1 = \log(2 \cdot |T_j|) \leq \log(|T_j| + |T_i|) = \log |T|$ ✔

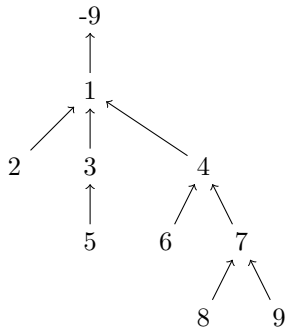
$\Rightarrow$ Eine Sequenz von n MakeSet-Operationen und m *weighted* Union- und Find-Operationen, kann in $\mathcal{O}(m \log n)$ ausgeführt werden.

3.2 Find mit "Path Compression"

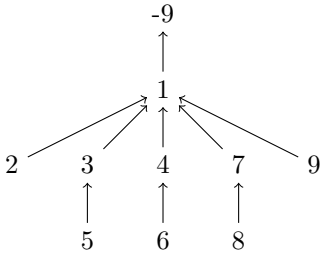
Bei der Suche nach dem Schlüssel k ändern wir für alle Knoten auf dem Pfad von *root* zu a den Zeiger zum Vorgänger ($parent[x] \leftarrow root$, x liegt auf dem Pfad von *root* zu a).

Beispiel (**Find**(9)):

Vor der Suche nach 9:



Nach der Suche nach 9:



Laufzeiten:

Find: $\Theta(\log n)$

Union: $\Theta(1)$

MakeSet: $\Theta(1)$

Mit der Anwendung der amortisierten Kosten erhält man jedoch folgendes:

Find: $\Theta(\log^* n)$

Wobei folgendes gilt (**iterativer Logarithmus**):

$$\log^* n = \min\{j \geq 0; \log^{(j)} n \leq 1\}$$

sowie

$$\log^{(i)} n = \begin{cases} n & \text{falls } i = 0 \\ \log(\log^{(i-1)} n) & \text{falls } i > 0 \text{ und } \log^{(i-1)} > 0 \text{ definiert} \\ \text{undefiniert} & \text{sonst} \end{cases}$$

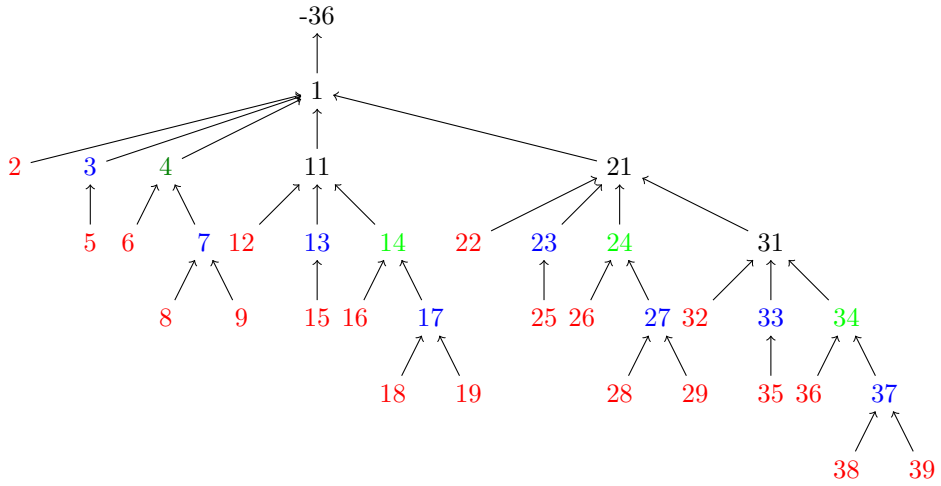
Der **rank** $r(v)$ eines Knotes v entspricht der Höhe seines Teilbaumes, gewurzelt bei v. Somit gilt

$$r(v) \leq \log n, \ \forall v \in V$$

Eine **Rank**-Gruppe R_j ist eine Menge von Knoten für die gilt:

$$R_j = \begin{cases} \{v | \log^{(j+1)} n > r(v) \leq \log^{(j)} n\} & \text{falls } \log^{(j+1)} n \text{ definiert ist} \\ \{v | r(v) = 0\} & \text{falls } \log^{(j)} n < 1 \text{ definiert ist} \\ \emptyset & \text{sonst} \end{cases}$$

Beispiel ($r(1) = 5, r(21) = 4, r(11) = r(31) = 3$, **grüne:** $r() = 2$, **blaue:** $r() = 1$, **rote:** $r() = 0$):
Sowie R_1 sind die schwarzen Knoten,
R_2 sind die grünen Knoten,
R_3 sind die blauen Knoten,
R_4 sind die roten Knoten.



Alle ranks steigen zur jeder Zeit der Sequenz auf dem Weg eines Knotens zur Wurzel strikt monoton an (auf einem Pfad vom Knoten zur Wurzel).

Beweis:

Zu einem bestimmten Punkt setzen wir für einen Knoten v: $parent[v] \leftarrow w$ durch die Pfadkompression (davor war v in einem Teilbaum von w). Somit war vorher schon $r(v) < r(w)$.

Es gibt höchstens $\frac{n}{2^r}$ Knoten vom rank r.

Beweis:

T_v ist Teilbaum gewurzelt bei v vom rank r im Wald T'. Dann gilt

$$r = h(T_v) \leq \log |T_v| \quad \Rightarrow |T_v| \geq 2^r$$

Da zwei Teilbäume mit der selben rank disjunkt sind und es insgesamt n Knoten gibt folgt daraus, dass es höchstens $\frac{n}{2^r}$ Knoten pro rank gibt.

Beginn der amortisierten Analyse:

1. Originalsequenz (σ)
2. Hinzurechnen der Kosten einer Operation FIND(x) zu der Operation für das Bewegen der Knoten (eine Einheit für das Durchlaufen der Knoten auf einem Pfad x zur Wurzel (inklusive x, ohne Wurzel und Vorgänger der Wurzel) und eine Einheit für das Bewegen der Knoten)
3. zwei Arten von Bewegungen:
   Typ A: Vor der Bewegung gilt $R_i(v), R_j(parent[v]), i \neq j$
   Typ B: Vor der Bewegung gilt $R_i(v), R_j(parent[v]), i = j$
4. es gibt höchstens $\log^* n + 1$ nicht-leere Rank-Gruppen
5. weil der rank eines Knotens auf dem Weg zur Wurzel ansteigt folgt, dass es höchstens $\log^* n$ Bewegungen vom *Typ A* gibt
6. es gibt weniger als $\log^j n$ Bewegungen in der Rank-Gruppe R_j
7. es gibt höchstens $\frac{n}{2^r}$ Knoten pro rank

Hieraus folgt:

$$\begin{aligned} |R_j| &< \sum_{i=\lceil \log^{(j+1)} n \rceil}^{\infty} \frac{n}{2^i} \\ &= \frac{n}{2^{\lceil \log^{(j+1)} n \rceil}} \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &\leq \frac{2n}{2^{\log^{(j+1)} n}} \\ &= \frac{2n}{2^{\log(\log^{(j)} n)}} \\ &= \frac{2n}{\log^{(j)} n} \end{aligned}$$

Somit gibt es $|R_j| \cdot \log^{(j)} = 2n$ Bewegungen vom Typ B pro Rank-Gruppe.
$\Rightarrow 2n \cdot \log^* n + 1$ Bewegungen vom Typ B.
Zusammenfassend:
Eine Sequenz von m Operationen MakeSet, gewichtete Union und Find mit Pfadkompression (n sind MakeSet-Operationen) kann in $\mathcal{O}(m \log^* n)$ ausgeführt werden.

3.3 inverse Ackermannfunktion

Wächst langsamer als der iterative Logarithmus, die m Operationen können in $\mathcal{O}(m\alpha(m, n))$ ausgeführt werden, wobei α eine Variante der inversen Ackermannfunktion ist.

4 Anwendung: Gleichheit von endlichen Automaten

- **witness** ist ein Beispiel, das zeigt, dass zwei Automaten nicht gleich sind.
- zwei Automaten können nur dann gleich sein, wenn ihre Startzustände gleich sind
- zwei Automaten sind gleich, wenn sie die gleiche Menge an Wörtern akzeptieren
- Algorithmus zum Testen der Gleichheit von endlichen Automaten kann dann eine **kürzeste** *witness* ausgeben, wenn die Datenstruktur zum Speichern der Zustände als Queue und nicht als Stack realisiert wird (ansonsten kann auch eine längere *witness* ausgegeben werden)
- der Algorithmus ist korrekt, weil alle möglichen Wege gespeichert und somit überprüft werden
- Laufzeit: es kann in $\mathcal{O}(|\Sigma| \cdot (|Q_1| + |Q_2|) \cdot \log^*(|Q_1| + |Q_2|)$ entschieden werden, ob zwei Automaten gleich sind oder nicht
- Vergleich Algorithmus 3.

Minimaler Spannbaum

inzident:

- ein Knoten v und eine Kante e sind inzident, falls $v \in e$
- zwei Kanten e_1, e_2 sind inzident, falls $e_1 \cap e_2 \neq \emptyset$

adjazent: zwei Knoten v, w sind adjazent, falls $\{v, w\} \in E$

Grad: $deg(v) = \#$ inzidenter Kanten

Pfad der Länge l: ist ein Teilgraph mit allen Kanten des Pfades mit $l + 1$ Knoten

verbundener Teilgraph: ist ein maximal verbundener Teilgraph (alle Kanten zwischen den Knoten $v \in V_{Teilgraph}$ sind in $E_{Teilgraph}$)

Baum: $m = n - 1$ und ist verbunden

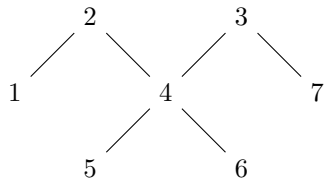
gespannter Teilgraph: ist ein verbundener Teilgraph mit $V_{Teilgraph} = V$

gespannter Teilbaum: ist ein gespannter Teilgraph, der ein Baum ist

1 Prüfer-Sequenz

Es gibt n^{n-2} beschriftete Bäume auf der Knotenmenge $\{1, \ldots, n\}$ für alle $n \in \mathbb{N}_{\geq 1}$.
Ein Baum T kann definiert werden durch $T = \text{Prüfer2Tree}(\text{Tree2Prüfer}(T))$
Beispiel (**Prüfer-Sequenz:** $(2, 4, 4, 4, 3)$)**:**

Baum:



Zurück zum Baum:

j	1	2	5	6	4	7
$parent[j]$	2	4	4	4	3	3

Vergleich Algorithmen 4 und 5 ..

2 Tarjan's Kantenfärbungs-Methode

- **Farbeninvariante:** Es gibt einen MST, der alle blauen und keine rote Kante enthält.
- eine Kante $e = \{v, w\} \in E$ **kreuzt** einen *Schnitt*, falls $v \in S \subsetneq V$ und $w \in V \setminus S$
- ein **einfacher Kreis** ist ein verbundener (Teil-)Graph mit $\forall v \in V : deg(v) = 2$
- wenn T ein Spannbaum ist, so gibt es für jeden Schnitt in G eine Kante, die diesen Schnitt kreuzt, sowie es in jedem Kreis eine Kante gibt, die nicht in T ist

Blaue Regel: Auswählen eines Schnittes, den keine blaue Kante kreuzt $\rightarrow$ färbe Kante mit dem kleinsten Gewicht blau

Rote Regel: Auswählen eines einfachen Kreises, der keine rote Kante enthält $\rightarrow$ färbe die Kante mit dem größten Gewicht rot

Dieser Algorithmus wird solange angewendet, bis keine Regel mehr angewendet werden kann.

Tarjan's Kantenfärbungsalgorithmus färbt alle Kanten richtig.

Beweis:

Am Anfang ist keine Kante gefärbt. Da der Graph verbunden ist, gibt es auch einen MST. Nach dem k-ten Schritt gibt es einen MST T mit allen blauen und keinen roten Kanten. Jetzt gibt es zwei Fälle:

Anwendung der blauen Regel: Falls der Algorithmus eine Kante $e \in T$ färbt, ist alles ok. Sonst gibt es eine Kante e' auf dem Schnitt $C = (S, V \setminus S)$ die nicht blau gefärbt ist und zu T gehört (sie kann nicht rot sein, sonst wäre sie nicht im Baum T). Dann färben wir die Kante e blau. Da immer die Kante mit dem kleinsten Gewicht genommen wird, gilt $w(T') \leq w(T)$.

Anwendung der roten Regel: Äquivalent zur blauen Regel mit einem Kreis C sowie der Folgerung, dass $w(e) \geq w(e')$ und $w(T') \leq w(T)$.

Zum zeigen, dass der Algorithmus auch alle Kanten färbt müssen wir folgende zwei Fälle zeigen:

$e \in T$**:** Betrachten der beiden Komponenten, die durch den Schnitt C durch e entstehen: keine blaue Kante geht über C, somit können wir e blau färben.

$e \notin T$**:** Betrachten den Kreis C (der einzigartige Pfad von v nach w, wobei $e = \{v, w\}$), dann gibt es keine rote Kante auf C und wir können die rote Regel anwenden.

3 Kruskal's Algorithmus

- wird mit n blauen disjunkten Bäumen gestartet
- Kanten werden in nicht-absteigender Reihenfolge (bezogen auf ihr Gewicht) abgearbeitet
- falls eine Kante e inzident zu zwei Knoten in **verschiedenen** Bäumen ist, wird die Kante **blau** gefärbt, sonst rot
- Anwendung der Färbungsregeln von Tarjan
  Beweis:
  Falls e in zwei unterschiedlichen blauen Bäumen endet, kann man S als die Menge an Knoten definieren, die v enthält. Dann kreuzt keine blaue Kante den Schnitt $C = (S, V \setminus S)$ und durch das Ordnen der Kanten ist e die Kante mit dem geringsten Gewicht.
  Falls $e = \{v, w\}$ inzident zu zwei Knoten im selben Baum ist, ist der Pfad P zwischen v und w zusammen mit e ein einfacher Kreis ohne rote Kanten. Somit wird e rot gefärbt (e ist die einzige ungefärbte Kante).
- Laufzeit:
  - Sortieren der Kanten in $\mathcal{O}(m \log n)$
  - Union-Find-Datenstruktur in $\mathcal{O}(m \log^* n)$
  - Gesamtlaufzeit somit in $\mathcal{O}(m \log n)$

Vergleich Algorithmus 6.

4 Matroide und der Greedy Algorithmus

4.1 Matroid

Unabhängigkeitssystem: endliche Menge X und eine Menge $\mathcal{I}$ von Teilmengen von X für die gilt:

1. $\emptyset \in \mathcal{I}$
2. falls $I_2 \in \mathcal{I}$ und $I_1 \subseteq I_2$ dann gilt $I_1 \in \mathcal{I}$

Austauscheigenschaft: falls $I_1, I_2 \in \mathcal{I}$ und $|I_1| < |I_2|$ dann gibt es ein $x \in I_2 \setminus I_1$ sodass $I_1 \cup \{x\} \in \mathcal{I}$

Matroid: Unabhängigkeitssystem mit Austauscheigenschaft

Beispiel $(Matroid)$**:**

- ein endlicher Vektorraum mit der Menge an unabhängigen Teilmengen
- Kantenmenge eines Graphs zusammen mit der Menge von kreisfreien spannenden Teilgraphen

Kreis eines Unabhängigkeitssystems: kleinste Teilmenge von X, die nicht in $\mathcal{I}$ ist

Basis eines Unabhängigkeitssystems: größtes Element aus $\mathcal{I}$; alle Basen eines Matroids haben die gleiche Größe (Folgerung aus Austauscheigenschaft)

4.2 Greedy Algorithmus

Vergleich Algorithmus 7.

Voraussetzungen:

1. Unabhängigkeitssystem $(X, \mathcal{I})$ mit Gewichtsfunktion $w : X \to \mathbb{R}$
2. $w(X') = \sum_{x \in X'} w(x)$ ist das Gewicht einer Teilmenge $X' \subseteq X$

Nutzen: berechnet Basis mit kleinstem Gewicht

Wenn $M = (X, \mathcal{I})$ ein Matroid ist, so berechnet der Greedy-Algorithmus die kleinste Basis im Bezug auf die Gewichtsfunktion.
Beweis fehlt.

5 Der Algorithmus von Prim

Datenstruktur:

- *Priority Queue*
- jedes Element hat einen Schlüssel, der die Priorität des Elementes abbildet
- kleinster Schlüssel entspricht höchster Priorität
- Implementation in als Heap dargestellten Bäumen oder Wäldern
- Laufzeit verschiedener Heaps:

	Binär-Heap	d-Heap	Fibonacci-Heap
INSERT	$\mathcal{O}(\log n)$	$\mathcal{O}(\log_d n)$	$\mathcal{O}(1)$
DECREASEKEY	$\mathcal{O}(\log n)$	$\mathcal{O}(\log_d n)$	$\mathcal{O}(1)^*$
EXTRACTMIN	$\mathcal{O}(\log n)$	$\mathcal{O}(d \log_d n)$	$\mathcal{O}(\log n)^*$
MAKEHEAP	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

$*$ amortisierte Kosten

Operationen:

Insert(item x, key k): Einfügen eines Elementes x mit Schlüssel k in die *Priority Queue*

DecreaseKey(item x, key k): Setzen des Schlüssels von x auf k

ExtractMin: gibt das Element mit dem kleinsten Schlüssel zurück und löscht es aus der *Priority Queue*

MakeHeap: erstellt eine *Priority Queue* mit allen Elementen

Während der Algorithmus läuft enthält die *Priority Queue* alle Kanten, die nicht im blauen Baum enthalten sind. Der Schlüssel eines Knotens ist das Gewicht der leichtesten Kante e, die inzident zu v ist und einem Knoten des blauen Baumes. Durch umhängen der Elternzeiger wird der blaue Spannbaum erzeugt.

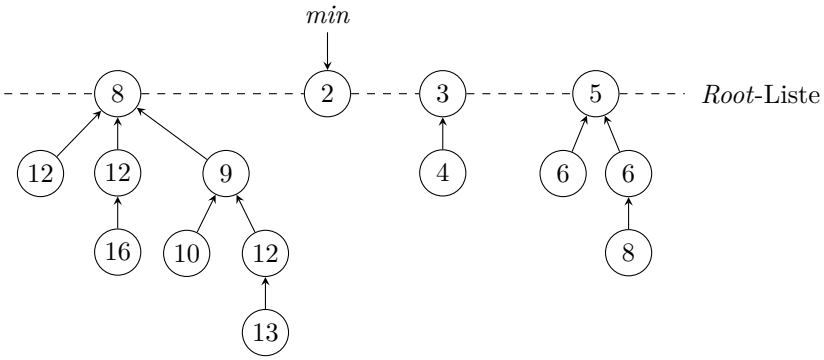
Laufzeit:

- n EXTRACTMIN-Operationen
- höchstens $m + 1$ DECREASEKEY-Operationen
- mit Fibonacci-Heaps kann der Algorithmus somit in $\mathcal{O}(m + n \log n)$ ausgeführt werden

Vergleich Algorithmus 8.

Fibonacci-Heaps

- Wald aus (Min-)Heaps
- Element mit dem kleinsten Schlüssel ist die Wurzel
- Min-Zeiger auf kleinste Wurzel
- Wurzeln sind in einer *Root*-Liste gespeichert
- Knotennamen sind die Schlüssel der Elemente



Operationen:

Insert(item x, key k): Einfügen des Elementes x mit Schlüssel k als neue Wurzel in der *Root*-Liste, eventuelles Updaten des Min-Zeigers

ExtractMin:

1. alle Kinder des Minimums werden in die *Root*-Liste eingefügt
2. das Minimum wird entfernt
3. Funktion Consolidate wird auf der *Root*Liste aufgerufen

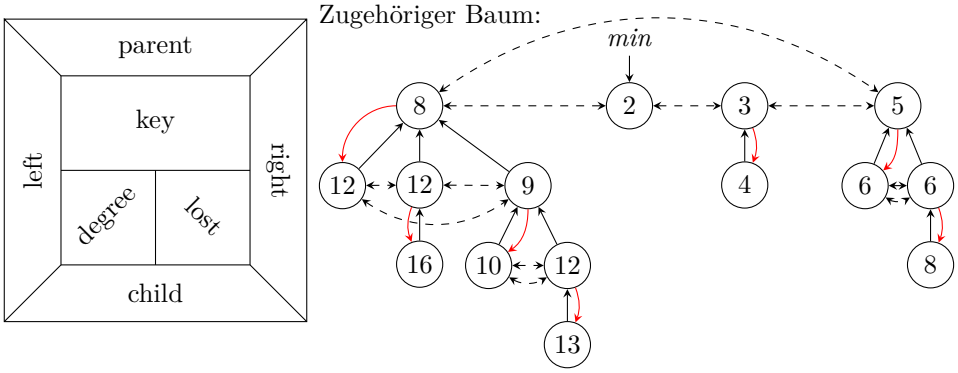
DecreaseKey(item x, key k):

1. k wird der neue Schlüssel von x
2. falls $k < key[parent]$ wird der Teilbaum T_x mit Wurzel x abgeschnitten und die x in die *Root*-Liste eingefügt
3. Update des Min-Zeigers
4. falls der Elternknoten von x schon ein Kind verloren hat, werden alle übrig gebliebenen Teilbäume (deren Elternknoten $parent[x]$ ist) in die *Root*-Liste eingefügt (**cascading cut**)

Consolidate: solange es zwei Wurzeln gibt mit der gleichen Anzahl an Kindern, wird der Baum mit dem größeren Schlüssel an den Baum mit dem kleineren Schlüssel angehängt, hiernach muss der Min-Zeiger erneuert werden
Vergleich Algorithmus 9.

1 Notwendige Datenfelder

- für DecreaseKey speichern wir für jedes Element eine Boolean-Variable *lost*, zum Speichern, ob bereits ein Kind abgeschnitten wurde
- für ExtractMin speichern wir für jeden Knoten das Kind mit dem kleinsten Schlüssel
- zu jedem Knoten wird das linke und das rechte Kind gespeichert
- für Consolidate speichern wir die Anzahl der Kinder in der Variablen *degree*



2 Laufzeit Analyse

Consolidate:

1. $r = \#$ Elemente in *Root*-Liste vor einer CONSOLIDATE-Operation
2. in jeder Iteration über die Anzahl der Knoten des aktuellen Knoten x werden zwei Bäume verschmolzen (das kann maximal r-mal passieren)
3. für jedes original Element in der *Root*-Liste gibt es höchstens **eine** Null-Anfrage für die innere Schleife (Iteration aus Punkt 2) geben
4. $\Rightarrow \mathcal{O}(r)$

Insert: Bei jeder INSERT-Operation zahlen wir 2 Einheiten. Die zweite Einheit ist für eine spätere (erste) CONSOLIDATE-Operation.

DecreaseKey: Die Worst-Case Laufzeit ist proportional in der Höhe des Baumes. In amortisierter Analyse ist sie aber konstant: 4 Einheiten pro Operation.

- für das Bewegen des aktuellen Elementes
- falls das Label *lost* von (höchstens) einem Element gesetzt wird (genau das Element, des letzten bewegten Elementes): für das Bewegen in einem späteren *cascading cut*
- zwei Einheiten für eine spätere CONSOLIDATE-Operation der beiden bewegten Elemente für die die Operation bezahlt hat

ExtractMin:

- Worst-Case-Laufzeit ist in $\mathcal{O}(n)$ (präziser: proportional zu der Anzahl an Elementen in der *Root*-Liste)
- für viele Elemente in der *Root*-Liste wurde schon bezahlt
- Unterscheidung der folgenden Knoten:
  1. für Knoten, die in den Heap seit der letzten EXTRACTMIN-Operation eingefügt wurden, wurde für die erste CONSOLIDATE-Operation bezahlt
  2. für Knoten, die während einer DECREASEKEY-Operation seit der letzten EXTRACTMIN-Operation in die *Root*-Liste eingefügt wurden, wurde schon für die CONSOLIDATE-Operation bezahlt
  3. für Knoten, die direkt nach der letzten EXTRACTMIN-Operation eingefügt wurden, wurde noch nicht bezahlt
  4. für die Kinder der Wruzel mit kleinstem Schlüssel wurde noch nicht bezahlt

  Für 3 und 4 zeigen wir, dass die maximale Anzahl der Elemente in der *Root*-Liste nach einer CONSOLIDATE-Operation, sowie die maximale Anzahl an Kindern eines Knotens in $\mathcal{O}(\log n)$ liegt.

Beweis:

Zuerst definieren wir die Zahlen S_k, welche die minimale Anzahl an Knoten in einem (Teil-)Baum eines Fibonacci-Heaps mit Wurzel k definieren:

$$\begin{aligned} S_0 &= 1 \\ S_1 &= 2 \\ S_k &= \underbrace{1}_{Wurzel} + \underbrace{1 + \sum_{i=0}^{k-2} S_i}_{\substack{Teilbbäume\ mit\ Kind-\\ knoten\ als\ Wurzel}} , k \geq 1 \end{aligned}$$

Diese Zahl S_k entspricht genau F_{k+2} für alle $k \geq 0$. Des weiteren gilt:

$$F_{k+2} = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{k+2} - \left(\frac{1-\sqrt{5}}{2} \right)^{k+2} \right) \geq \left(\underbrace{\frac{1+\sqrt{5}}{2}}_{\substack{goldener\\ Schnitt\ \phi}} \right)^k , \quad \forall k \geq 0$$

Aus $n \geq S_k$ folgt, dass der Grad einer Wurzel höchstens $\frac{1}{\log \frac{1+\sqrt{5}}{2}} \cdot \log n < 1.5 \log n$ ist.

Wir nehmen nun an, dass nach einer CONSOLIDATE-Operation r Wurzeln in der *Root*-Liste sind. Alle Grade der Wurzeln sind disjunkt (CONSOLIDATE-Voraussetzung). Somit haben wir

$$n \geq \sum_{i=0}^{r-1} S_i = S_r - 2 + S_{r-1} \geq S_r \geq \left(\frac{1+\sqrt{5}}{2} \right)^r$$

Die vorletzte Ungleichung gilt, falls $r \geq 2$. Somit gibt es maximal $\max\{1, 1.5 \log n\}$ Wurzeln nach einer CONSOLIDATE-Operation.

Gesamtaufstellung:

	Worst-Case	amortisiert
INSERT	$\Theta(1)$	$\mathcal{O}(1)$
DECREASEKEY	$\Theta(n)$	$\mathcal{O}(1)$
EXTRACTMIN	$\Theta(n)$	$\mathcal{O}(\log n)$

Minimaler Schnitt in ungerichteten Graphen

- gegeben ist ein Graph mit Gewichtsfunktion $w : E \to R_{\geq 0}$
- gesucht ist ein Schnitt $C = (S, V \setminus S)$ mit minimalem Gewicht $w(C) = \sum_{e \in E(C)} w(e)$ in Bezug auf alle Schnitte im gegebenen Graphen
- das Problem ist $\mathcal{NP}$-vollständig mit negativen Kantengewichten
- ein $s-t$-Schnitt trennt s und t ($s \in S, t \notin S$)
- entweder gibt es einen minimalen Schnitt oder s und t sind in der gleichen Menge
- **Definition:** Ein Graph $G/_{st} = (V/_{st}, E/_{st})$ mit $w/_{st} : E/_{st} \to R_{\geq 0}$ ist erstellt worden aus G durch vereinigen von s und t, falls
  1. $V/_{st} = V \setminus \{t\}$
  2. $E/_{st} = (E \setminus \{\{t, v\}; v \in V\}) \cup \{\{s, v\}; \{t, v\} \in E \text{ und } v \neq s\}$
     in anderen Worten: die Kantenmenge $E/_{st}$ enthält alle Kanten von t zu allen v (die schon in E vorhanden gewesen sind) als Kanten von s zu v (ausgenommen die Kante von s zu t)
  3. eingeschränkt auf die Kantenmenge $E \cap \binom{V \setminus \{s,t\}}{2}$ setzen wir die Gewichtsfunktion $w/_{st} \equiv w$ und

$$w/_{st}(\{s, v\}) = \begin{cases} w(\{s, v\}) & \text{falls } \{s, v\} \in E \text{ und } \{t, v\} \notin E \\ w(\{t, v\}) & \text{falls } \{s, v\} \notin E \text{ und } \{t, v\} \in E \\ w(\{s, v\}) + w(\{t, v\}) & \text{falls } \{s, v\} \in E \text{ und } \{t, v\} \in E \end{cases}$$

- **Algorithmus von Stoer und Wagner:**
  - λ (das Gewicht des minimalen Schnittes) sowie der minimale Schnitt selbst kann in $|V| - 1$ Phasen berechnet werden
  - Auswählen eines Schnittes zwischen zwei Knoten u und v
  - u und v zusammenfügen zu einem Knoten
  - Gewichte der Kanten neu berechnen
  - s und t werden in jeder Iteration neu gewählt
  - **Laufzeit:** $\mathcal{O}(n^2 \log n + m \cdot n)$
- es gilt für $S \subset V, v \in V \setminus S$: $w(S, v) = \sum_{e \in E(S, \{v\})} w(e)$
- **modifizierter Algorithmus von Stoer und Wagner (Vergleich Algorithmus 10.):**
  - s und t können nicht gewählt werden, der Algorithmus berechnet sie
  - ein minimaler s-t-Schnitt für zwei geeignete Knoten s und t kann mit einer ähnlichen Methode berechnet werden, wie der MST-Algorithmus von Prim
  - Start ist ein zufällig gewählter Knoten $a \in V$ und eine Menge $A = \{a\}$
  - es wird immer der am engsten verbundene Knoten zu A zu A hinzugefügt (der Knoten $v \in V \setminus A$ mit $w(A, v)$ ist maximal), bis nur noch t übrig ist
  - angenommen s wurde zuletzt zu A hinzugefügt, dann ist $(A, \{t\})$ ein minimaler s-t-Schnitt
  - Implementation mithilfe einer *Priority Queue*
  - **Laufzeit:** mit Fibonacci-Heaps: $\mathcal{O}(m + n \log n)$

- Min-Cut-Phase-Algorithmus berechnet zwei Knoten s, t mit minimalem Schnitt $C = (V \setminus \{t\}, \{t\})$:

  Beweis:

  1. s, t ist Ausgabe des Algorithmus
  2. $C = (S, V \setminus S)$ ist beliebiger $s-t$-Schnitt mit $s \in S$
  3. Knoten aus V werden in der Reihenfolge betrachtet, in der sie aus der *Priority Queue* genommen wurden, t ist der letzte
  4. ein Knoten v ist **aktiv**, falls v und sein Vorgänger auf zwei verschiedenen Seiten von C sind
  5. t ist **aktiv**
  6. für jeden Knoten v gibt es eine Menge von Knoten A_v mit allen Knoten, die vor v aus der *Priority Queue* herausgeholt wurden, sowie die Menge $S_v = S \cap (A_v \cup \{v\})$
  7. es gilt für jeden **aktiven** Knoten v (insbesondere für t): $w(A_v, v) \leq w(S_v, (A_v \cup \{v\}) \setminus S_v)$
     (für t: $w(A_t, t) \leq w(S_t, (A_t \cup \{t\}) \setminus S_t) \Rightarrow w(V \setminus \{t\}, t) \leq w(S, V \setminus S)$)

     Beweis:

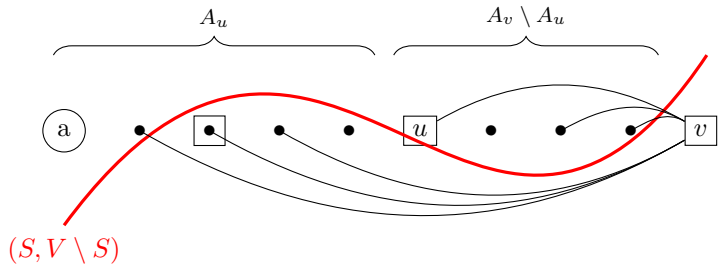
     Induktionsanfang: v ist der erste **aktive** Knoten

     a) $v \in S$: $S_v = \{v\}$

     b) $v \notin S$: $S_v = A_v$

     in beiden Fällen gilt: $w(A_v, \{v\}) = w(S_v, (A_v \cup \{v\}) \setminus S_v)$

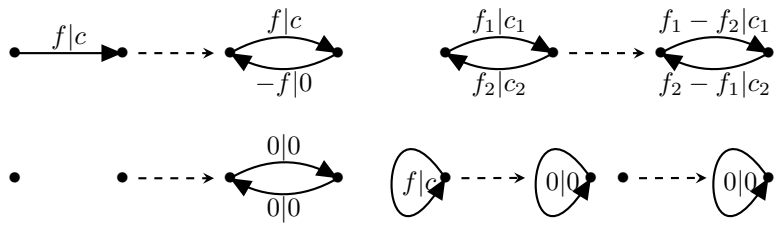
     Induktionsschritt: u war der letzte **aktive** Knoten vor v



     - durch die Wahl des am engsten verbundenen Knoten, wissen wir: $w(A_u, v) \leq w(A_u, u)$
     - durch (IA) wissen wir, dass $w(A_u, u) \leq w(S_u, (A_u \cup \{u\}) \setminus S_u)$
     - alle Kanten zwischen $A_v \setminus A_u$ und v gehen über den Schnitt $(S, V \setminus S)$
     - die Kantenmengen $E(S_u, (A_u \cup \{u\}) \setminus S_u)$ und $E(A_v \setminus A_u, \{v\})$ sind disjunkte Teilmengen von $E(S_v, (A_u \cup \{v\}) \setminus S_v)$
     - durch die Annahme, dass alle Kantengewichte positiv sind, erhalten wir: $w(A_v, v) \leq w(S_u, (A_u \cup \{u\}) \setminus S_u) + w(A_v \setminus A_u, v) \leq w(S_v, (A_v \cup \{v\}) \setminus S_v)$

Network Flows und minimale Schnitte

- Kantenmenge $E \subseteq (V \times V) \setminus \{(v, v); v \in V\}$
- für eine Kante $e = \{v, w\}$ gilt
  - v ist *tail* von e
  - w ist *head* von e
- ein gerichteter Weg $P : v_0, \ldots, v_l$ ist ein Graph $P = (\{v_0, \ldots, v_l\}, \{(v_{i-1}, v_i); i = 1, \ldots, l\})$ mit $l + 1$ Knoten
- ein *Schnitt* in einem gerichteten Graphen ist ein **geordnetes** Paar $C = (S, V \setminus S)$
- ein *s-t-Schnitt* ist ein *Schnitt* wobei $s \in S, t \notin S$ gilt
- für $S, T \subseteq V$ gilt $E(S, T) = (S \times T) \cap E$ ($E(S, T)$ enthält alle Kanten mit *tail* in S und *head* in T)
- Kapazitäten $c : E \to \mathbb{R}_{\geq 0}$ sind Kantengewichte mit $c(S, V \setminus S) = \sum_{e \in E(S,T)} c(e)$
- der Algorithmus von Stoer und Wagner kann nicht auf gerichtete Graphen angewendet werden, stattdessen kann man einen minimalen Schnitt mit dem dualen *maximalen flow*-Problem lösen
- Flussnetzwerk $\mathcal{N} = (D, s, t, c)$ mit
  - gerichteter Graph D
  - eine Quelle (*source*) $s \in V$
  - ein Ziel (*sink*) $t \in V$
  - Kapazitäten $c : E \to \mathbb{R}_0^+$
- ein *s-t*-flow in einem Flussnetzwerk ist eine Funktion $f : E \to \mathbb{R}_0^+$ mit
  1. **Kapazitätsbeschränkung:** $f(e) \leq c(e), \quad \forall e \in E$
  2. **Flusskonservierung:** $\sum_{(w,v) \in E} f(w, v) - \sum_{(v,w) \in E} f(v, w) = 0, \quad \forall v \in V \setminus \{s, t\}$
- der *Wert* eines Flussnetzwerkes ist die Differenz zwischen dem eingehenden und dem ausgehenden Fluss, oder $w(f) = \sum_{(s,v) \in E} f(s, v) - \sum_{(v,s) \in E} f(v, s)$
- ein Fluss ist maximal, wenn der *Wert* maximal ist
- ein Fluss *sättigt* eine Kante e, falls $f(e) = c(e)$
- für eine einfachere Darstellung fügen wir Kanten hinzu:



- für eine endliche Menge V mit $s,t \in V$ und $c : V \times V \to \mathbb{R}_0^+$ ist die Funktion $f : V \times V \to \mathbb{R}$ ein s-t-Fluss in (V, s, t, c), falls

  Kapazitätsbeschränkung: $f(v,w) \leq c(v,w), \quad \forall v, w \in V$

  Skew-Symmetrie: $f(v,w) = -f(w,v), \quad \forall v, w \in V$

  Flusserhaltung: $\sum_{v \in V} f(v,w) = \sum_{v \in V} f(w,v) = 0, \quad \forall w \in V \setminus \{s,t\}$

- der Wert eines Flusses in (V, s, t, c) ist $\sum_{v \in V} f(s,v)$

- die betrachtete Kantenmenge eines Flussdiagrammes ist $E = \{(u,v) \in V \times V; c(u,v) \neq 0 \text{ oder } c(v,u) \neq 0\} \setminus \{(v,v); v \in V\}$, also alle Kanten, die einen Fluss ungleich 0 haben können

- für eine Kante $e = (v,w)$ bezeichnen wir die Rückwärtskante $(w,v) = -e$

- der Wert eines s-t-Flusses ist Summe aller Flüsse über die Kanten eines s-t-Schnittes:

  Beweis:

  $$\begin{aligned} w(f) &= \sum_{v \in V} f(s,v) && \text{//Hinzufügen einer Doppelsumme, die sich aufhebt (Flusserhaltung)} \\ &= \sum_{u \in S} \sum_{v \in V} f(u,v) && \text{//Aufteilen der Summe in Schnittkanten und andere} \\ & && \text{//der Wert der Kanten, die nicht zum Schnitt gehören, fällt weg (Skew-Symmetrie)} \\ &= \sum_{u \in S} \sum_{v \notin S} f(u,v) \\ &= \sum_{e \in E(S, V \setminus S)} f(e) \end{aligned}$$

- $w(f) = \sum_{v \in V} f(v,t)$ folgt direkt aus vorherigem Beweis mit $S = V \setminus \{t\}$

- **Cut-Lemma:** der Wert eines Flusses kann nicht größer sein als die Kapazität eines minimalen Schnittes:
  $$w(f) = \sum_{e \in E(S, V \setminus S)} f(e) \leq \sum_{e \in E(S, V \setminus S)} c(e) = c(S, V \setminus S)$$

- ein s-t-Fluss ist maximal, wenn es einen s-t-Schnitt gibt mit $w(f) = c(S, V \setminus S)$

- ein *augmenting path* ist ein Kantenzug in einem Flussnetzwerk, auf dem keine Kante gesättigt ist

- **Augmenting Path Theorem:** ein Fluss ist maximal $\Leftrightarrow \nexists$ *augmenting* s-t-*path* in Bezug auf f:

  Beweis:
  $\Rightarrow$ P ist ein *augmenting path* mit $\Delta = \min_{e \in P}(c(e) - f(e))$
  Dann kann der Fluss f erhöht werden mit der folgenden Funktion:

  $$f'(e) = \begin{cases} f(e) + \Delta & \text{falls } e \in P \\ f(e) - \Delta & \text{falls } e \notin P \\ f(e) & \text{sonst} \end{cases}$$

  daraus folgt $w(f') > w(f)$ ↯

  $\Leftarrow$ $S = \{v \in V;$ es gibt einen augmenting s-t-path im Bezug zu $f\}$
  da $t \notin S$ folgt, dass $(S, V \setminus S)$ ein s-t-Schnitt ist und alle Kanten in $E(S, V \setminus S)$ sind gesättigt
  $\Rightarrow w(f) = \sum_{e \in E(S, V \setminus S)} f(e) = \sum_{e \in E(S, V \setminus S)} c(e) = c(S, V \setminus S)$
  $\Rightarrow$ f ist maximaler Fluss

- **Min-Cut Max-Flow Theorem:** der Wert eines maximales s-t-Flusses entspricht der Kapazität eines minimalen s-t-Schnittes:

  $$S = \{v \in V;\ \text{es gibt einen augmenting s-t-path im Bezug zu } f\}$$

  Mit einem minimalen s-t-Schnitt $(S^*, V \setminus S^*)$ und dem **Cut-Lemma** gilt:

  $$w(f) = c(S, V \setminus S) \geq c(S^*, V \setminus S^*) \geq w(f)$$

- mit dem *augmenting path*-Theorem kann man direkt den Algorithmus von *Ford und Fulkerson* ableiten (Vergleich Algorithmus 11.)

  Laufzeit:

  - in $\mathcal{O}(w^*)$, wobei w^* = Wert des maximalen Flusses (kann bei hohen Kapazitäten sehr hoch sein)
  - terminiert nicht bei irrationalen Kapazitäten
  - falls immer die kleinste Anzahl an Kanten für einen *augmenting path* gewählt wird, ist der Algorithmus in $\mathcal{O}(nm)$ (*Edmonds und Karp*)

- *Goldberg und Tarjan*: wenn kein Fluss über einen *augmenting path* geschickt wird, sondern nur über einzelne Kanten (mit lokalen Entscheidungen) läuft der Algorithmus z.B. in $\mathcal{O}(n^2\sqrt{m})$ bzw. in $\mathcal{O}(nm \log \frac{n^2}{m})$

- **integrality-Theorem:** wenn alle Kapazitäten Integer sind, berechnet der Algorithmus von *Ford und Fulkerson* einen ganzzahligen maximalen Fluss

Geometrische Algorithmen

1 Grundbegriffe

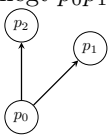
konvexe Kombination (von p_1, p_2): irgendein Punkt zwischen den beiden Punkten p_1 und p_2 oder auch $p = (1-\alpha) \cdot p_1 + \alpha \cdot p_2$ wobei α den Abstand zum Punkt p_1 beschreibt

(Linien-)Segment (mit den Endpunkten p_1, p_2): die Menge $\overline{p_1 p_2} = \{(1-\alpha) \cdot p_1 + \alpha \cdot p_2; 0 \leq \alpha \leq 1\}$ aller konvexen Kombinationen von p_1, p_2

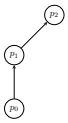
gerichtetes (Linien)-Segment (von p_1 nach p_2): (Linien-)Segment definiert durch die Abbildung $\overrightarrow{p_1 p_2} : [0,1] \longrightarrow \mathbb{R}$ mit $\alpha \mapsto (1-\alpha) \cdot p_1 + \alpha \cdot p_2$

1.1 Probleme

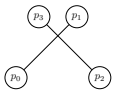
1. liegt $\overrightarrow{p_0 p_1}$ rechts von $\overrightarrow{p_0 p_2}$:



2. muss man auf dem von p_0 nach p_2 rechts abbiegen:

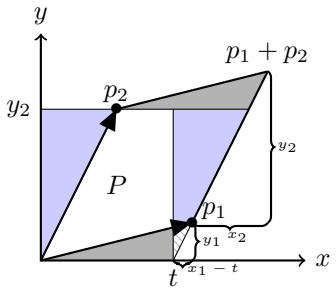


3. schneiden sich die beiden Liniensegmente $\overline{p_1 p_2}$ und $\overline{p_3 p_4}$:



Lösungsansätze

1. Problem:
   - bei der Transformation $p \mapsto p - p_0$ nehmen wir an, dass $p = (0,0)$ der Ausgangspunkt ist
   - wir nehmen an, dass $x_1, x_2, y_1, y_2 \geq 0$
   - wir nehmen an, dass $\overrightarrow{p_0 p_1}$ rechts von $\overrightarrow{p_0 p_2}$ liegt
   - auf dem folgenden Parallelogramm ist $(t, 0)$ der Schnittpunkt der x-Achse und der Linie durch p_1 sowie $p_1 + p_2$:



   - Fläche $P = t \cdot y_2$: die beiden grauen Dreiecke und die beiden blauen Dreiecke sind *deckungsgleich*
   - Berechnung der Steigung der Linie durch p_1 und $p_1 + p_2$ kann auf zwei Arten berechnet werden, welche die folgenden Gleichung ergeben:
     $\frac{y_2}{x_2} = \frac{y_1}{x_1 - t}$
     $\Rightarrow$ Flächeninhalt von P:
     $0 < y_2 \cdot t = x_1 \cdot y_2 - x_2 \cdot y_1 = p_1 \times p_2$
   - wenn $\overrightarrow{p_0 p_1}$ links von $\overrightarrow{p_0 p_2}$ liegt, werden die Indizes ausgetauscht, dass folgendes gilt: $0 < x_2 \cdot y_1 - x_1 \cdot y_2 = -(p_1 \times p_2)$
   - durch die Übertragung zum Ursprung erhalten wir:
     $\overrightarrow{p_0 p_1}$ rechts von $\overrightarrow{p_0 p_2} \iff (p_1 - p_0) \times (p_2 - p_0) > 0$
   - die Segmente $\overrightarrow{p_0 p_1}$ und $\overrightarrow{p_0 p_2}$ sind **kollinear**, wenn $(p_1 - p_0) \times (p_2 - p_0) = 0$

2. Problem: auf dem Weg von p_0 nach p_2 über p_1 muss man rechts abbiegen, falls $\overrightarrow{p_0p_2}$ rechts von $\overrightarrow{p_0p_1}$ liegt (Reduzierung des Problems auf Problem 1)

3. Problem:
   - wenn sich zwei Segmente schneiden, schneiden sich auch ihre Begrenzungsboxen
   - die **Begrenzungsbox** einer Menge von Punkten ist das kleinste zu den Achsen parallele Rechteck, das die Punkte enthält
   - für ein Segment $\overline{p_1p_2}$ sind die folgenden Punkte definiert:
     - linkester unterster Punkt der Begrenzungsbox: $\hat{p}_1 = (\underbrace{\min\{x_1, x_2\}}_{\hat{x}_1}, \underbrace{\min\{y_1, y_2\}}_{\hat{y}_1})$
     - rechtester oberster Punkt der Begrenzungsbox: $\hat{p}_2 = (\underbrace{\max\{x_1, x_2\}}_{\hat{x}_2}, \underbrace{\max\{y_1, y_2\}}_{\hat{y}_2})$
   - für ein zweites Liniensegment $\overline{p_3p_4}$ haben wir die folgenden Punkte $(\hat{x}_3, \hat{y}_3)$ und $(\hat{x}_4, \hat{y}_4)$ dann schneiden sich die Begrenzungsboxen gdw.
     $$\hat{x}_1 \leq \hat{x}_4 \text{ und } \hat{x}_3 \leq \hat{x}_2 \text{ und } \hat{y}_1 \leq \hat{y}_4 \text{ und } \hat{y}_3 \leq \hat{y}_2$$
   - die Methode der **schnellen Verwerfung (quick rejection)** basiert nur auf Vergleichen und keinen arithmetischen Operationen: wenn die Begrenzungsboxen sich nicht schneiden, können es die Liniensegmente auch nicht, trotzdem kann es sein, dass die Segmente sich nicht schneiden, aber die Begrenzungsboxen es tun
   - ein Liniensegment $\overline{p_1p_2}$ **straddles** ein Liniensegment $\overline{p_3p_4}$, wenn
     - p_1 und p_2 sich auf verschiedenen Seiten der Geraden durch p_3, p_4 befinden *oder*
     - mindestens einer der beiden Punkte p_1, p_2 liegt auf der Geraden durch p_3, p_4

     in anderen Worten:
     a) $\overrightarrow{p_3p_1}$ liegt rechts von $\overrightarrow{p_3p_4}$ und $\overrightarrow{p_3p_2}$ liegt links von $\overrightarrow{p_3p_4}$ *oder*
     b) $\overrightarrow{p_3p_1}$ liegt links von $\overrightarrow{p_3p_4}$ und $\overrightarrow{p_3p_2}$ liegt rechts von $\overrightarrow{p_3p_4}$ *oder*
     c) $\overrightarrow{p_3p_1}$ und $\overrightarrow{p_3p_4}$ sind kollinear *oder*
     d) $\overrightarrow{p_3p_2}$ und $\overrightarrow{p_3p_4}$ sind kollinear

     in der Summe ergibt sich dann:
     $$((p_1 - p_3) \times (p_4 - p_3)) \cdot ((p_2 - p_3) \times (p_4 - p_3)) \leq 0$$
   - zwei Liniensegmente $\overline{p_1p_2}, \overline{p_3p_4}$ schneiden sich $\iff$
     a) sich die Begrenzungsboxen von $\overline{p_1p_2}$ und $\overline{p_3p_4}$ schneiden *und*
     b) $\overline{p_1p_2}$ straddles $\overline{p_3p_4}$ *und*
     c) $\overline{p_3p_4}$ straddles $\overline{p_1p_2}$

     Beweis:

     Fall 1 (p_1, p_2, p_3, p_4 liegen auf einer Linie): die Segmente schneiden sich nur dann, wenn sich ihre Begrenzungsboxen schneiden

     Fall 2 (p_1, p_2, p_3, p_4 liegen nicht alle auf einer Linie):
     a) l ist die Gerade durch p_3, p_4
     b) mindestens einer der Punkte p_1, p_2 liegen nicht auf l
     c) da $\overline{p_1p_2}$ straddles $\overline{p_3p_4}$ schneiden sich das Segment $\overline{p_1p_2}$ und die Gerade l höchstens in einem Punkt (s)
     d) da $\overline{p_3p_4}$ straddles $\overline{p_1p_2}$ schneiden sich das Segment $\overline{p_3p_4}$ und die Gerade durch p_1, p_2 höchstens in einem Punkt, der gleich dem Punkt aus (c) entsprechen muss (s)
     e) somit ist s sowohl in $\overline{p_1p_2}$ als auch in $\overline{p_3p_4}$ enthalten

2 Sweep-Line-Methode

- eine *sweep line* ist eine imaginäre Linie, die eine Menge von geometrischen Objekten abarbeitet (z.B.: eine vertikale Linie, die die Objekte von links nach rechts abarbeitet)
- verwendete Daten:

  Status der*sweep line*: Beziehung zwischen den Objekten im Bezug auf die aktuelle Position der *sweep line*

  Ereigniszeitplan (event point schedule): Sequenz von Positionen (z.B. die x-Koordinaten von links nach rechts) an denen sich der *sweep line* Status ändern kann

2.1 Schneiden von Segmenten

Problem: Gibt es in einer Menge von n Liniensegmenten mindestens ein Paar von sich schneidenden Liniensegmenten?

mögliches Auftreten: beim Übereinanderlegen von mehreren Schichten von Informationen auf einer Karte

Lösen des Problemes:

- durch Testen aller $\binom{n}{2}$ Paaren von Liniensegmenten ob sie sich schneiden
  $\Rightarrow \mathcal{O}(n^2)$
- die Linien können sich jedoch nur schneiden, falls sich ihre Projektionen auf die x-Achse schneiden
- Algorithmus von *Shamos und Hoey* löst das Problem mit einem (verikalen) *sweep line*-Ansatz in $\mathcal{O}(n \log n)$
- erste Annahmen (einfacher):
  1. kein Liniensegment ist vertikal
  2. kein Liniensegment besteht nur aus einem Punkt
  3. Liniensegmente schneiden sich nicht in einem ihrer Endpunkte
  4. höchstens zwei Liniensegmente schneiden sich in einem Punkt
- aus **1.** folgt, dass ein Liniensegment die *sweep line* in höchstens einem Punkt schneidet
- der Status der *sweep line* ist die Ordnung der Liniensegmente, die die *sweep line* schneiden (entsprechend ihrer y-Koordinate des Schnittpunktes mit der *sweep line*):
  - s_1, s_2 sind zwei Liniensegmente welche die vertikale Linie l schneiden:
    $\Rightarrow s_1 <_l s_2 \iff s_1$ schneidet l strikt unter s_2
  - Änderung des *sweep line* Status:
    1. *sweep line* ist auf dem linken Endpunkt eines Segmentes (dann wird ein neues Segment in die Ordnung eingefügt), *oder*
    2. *sweep line* ist auf dem rechten Endpunkt eines Segmentes (dann wird das entsprechende Segment aus der Ordnung entfernt)
    3. zwei Segmente schneiden sich (die Ordnung der Segmente wird vertauscht)
- Algorithmus stoppt, wenn zwei Segmente gefunden wurden, die sich schneiden $\Rightarrow$ der Ereigniszeitplan entspricht der Sequenz von $2n$ Endpunkten der n Segmente, geordnet in nicht-abnehmender Reihenfolge in Bezug auf ihre x-Koordinate
- **Annahme:** zwei Liniensegmente schneiden sich, da es keinen Schnittpunkt mit drei Segmenten gibt $\Rightarrow$ es muss ein x geben, sodass s_1 ist der direkte Vorgänger oder Nachfolger von s_2 in Bezug auf $<_x$
- **Idee von Shamos und Hoey:** wenn man zwei aufeinanderfolgende Segmente findet $\rightarrow$ testen, ob sie schneiden

- wenn die *sweep line* l den linken Endpunkt p eines Liniensegmentes s müssen wir für ein anderes Liniensegment s', das die *sweep line* schneidet, entscheiden, ob s die *sweep line* ober- oder unterhalb von s' schneidet
  $\Rightarrow$ kann in konstanter Zeit entschieden werden:
  1. p'_l: linker Endpunkt von s'
  2. p'_r: rechter Endpunkt von s'

  dann gilt: s schneidet l strikt unter $s' \iff \overrightarrow{p'_l p}$ ist rechts von $\overrightarrow{p'_l p'_r}$
- für die Implementation wird der Status der *sweep line* in der Datenstruktur T repräsentiert, welche die folgenden Operationen zulässt:

  T.**Insert**(s)**:** fügt ein Segment s in den *sweep line* Status ein

  T.**Delete**(s)**:** löscht ein Segment s aus dem *sweep line* Status

  T.**Pred**(s)**:** gibt das Segment zurück, das die *sweep line* direkt unter s schneidet

  T.**Succ**(s)**:** gibt das Segment zurück, das die *sweep line* direkt über s schneidet

  mit balancierten binären Suchbäumen können diese Operationen in $\mathcal{O}(\log n)$ ausgeführt werden, falls es $\mathcal{O}(n)$ Elemente gibt
- der Algorithmus von *Shamos und Hoey* kann in $\mathcal{O}(n \log n)$ ausgeführt werden:
  - $2n$ Endpunkte $\Rightarrow$ können in $\mathcal{O}(n \log n)$ sortiert werden
  - jede der $2n$ Iterationen der For-Schleife braucht eine konstante Anzahl an Suchbaum-Operationen
  - Vergleich Algorithmus 12.

Spezialfälle:

vertikale Segmente:

- man kann die Richtung der *sweep line* stören, sodass die *sweep line* mit keinem anderen Liniensegment kollinear ist
  $\Rightarrow$ kann fehleranfällig sein
- stattdessen wird die *sweep line* "virtuell gestört" durch betrachten des tiefsten Endpunktes eines vertikalen Liniensegmentes als linken Endpunkt und den obersten als seinen rechten Endpunkt

Punktsegmente: zweimaliges Hinzufügen des einzelnen Punktes in den Ereigniszeitplans: einmal als linker Endpunkt und einmal als rechter Endpunkt

Schnitt im Endpunkt:

- p ist Endpunkt eines Segmentes s
- 1. Annahme: p ist auch in einem Segment $s' = \overline{p'_l p'_r}$ enthalten
- 2. Annahme: s' wurde in den *sweep line* Status eingefügt vor dem Betrachten des Ereignispunktes p
- falls p links von s liegt: Beginn mit Einfügend von s in den *sweep line* Status
- notwendiger Vergleich: liegt $\overrightarrow{p'_l p}$ rechts von $\overrightarrow{p'_l p'_r}$
  $\Rightarrow \overrightarrow{p'_l p}$ und $\overrightarrow{p'_l p'_r}$ sind kollinear (bedeutet s und s' sind "gleich" im Bezug auf die aktuelle Ordnung)
  $\Rightarrow$ s und s' schneiden sich (bzw. **allgemein:** Einfügen von s in T übereinstimmend mit der Ordnung $s \leq s'$ falls $(p - p'_l) \times (p'_r - p'_l) \geq 0$)
- nach Einfügen von s muss s' der Vorgänger oder Nachfolger von s sein und wir finden einen Schnittpunkt
- ist p der linke Endpunkt von s hätte der Algorithmus schon im vorherigen Ereignispunkt einen Schnittpunkt gefunden

mehr als zwei Segmente schneiden sich in einem Punkt: für zwei dieser Segmente ist es schon wahr, dass sie Vorgänger und Nachfolger sind für eine geeignet *sweep line* links des Schnittpunktes

2.2 Voronoi-Diagramme

Problem: "Telefonzellenproblem":

gesucht: eine Unterteilung der Ebene in n (Anzahl von Knoten) Zellen mithilfe einer Distanzfunktion $d : \mathbb{R}^2 \to \mathbb{R}_{\geq 0}$

mathematisch: $V(p_i) = \{p \in \mathbb{R}^2; d(p, p_i) \leq d(p, p_j), j = 1, \ldots, n\}, \quad i = 1, \ldots, n$

- Distanzfunktion d ist die euklidische Distanz:
  $d\left(\begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}\right) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
- $V(p_1), \ldots, V(p_n)$ sind *Voronoi-Zellen*
- ein Voronoi-Diagramm $Vor(P)$ besteht aus den Grenzen der Voronoi-Zellen
- Knoten V von $Vor(P)$ sind die Punkte, die auf den Grenzen von mindestens drei Zellen liegen
- Kanten E von $Vor(P)$ sind die verbundenen Teilmengen von $Vor(P)$ ohne V
- manche Kanten von $Vor(P)$ können unendliche Länge haben
- Knoten aus P bezeichnen wir als *site*s
- ein Punkt p ist ein Knoten in $Vor(P) \iff p$ ist Zentrum eines Kreises mit mindestens drei *site*s auf seinem Umkreis und keiner *site* innerhalb des Kreises
- ein Punkt p ist auf einer Kante in $Vor(P) \iff p$ ist Zentrum eines Kreises mit genau zwei *site*s auf seinem Umkreis und keiner *site* innerhalb des Kreises
- eine Voronoi-Zelle eines Punktes $p_i, i = 1, \ldots, n$ kann wie folgt konstruiert werden:
  - für alle $p_j, j \neq i$ teilt die Mittelsenkrechte $\overline{p_i p_j}$ die Fläche in zwei Halbebenen
  - $H_{p_j}(p_i)$ ist die Halbfläche, die p_i enthält
    $\Rightarrow V(p_i) = \bigcap_{j \neq i} H_{p_j}(p_i)$
- wenn das Voronoi-Diagramm auf diese Weise konstruiert wird, müssen die Schnittpunkte der $n - 1$ Mittelsenkrechten berechnet werden, allerdings ist die Anzahl der Knoten und Kanten linear zur Anzahl der *site*s

Lemma: Ein Voronoi-Diagramm einer Menge vin $n \geq 3$ *site*s hat höchstens $2n - 5$ Knoten und $3n - 6$ Kanten

Beweis:

- Annahme: alle *site*s liegen auf einer Linie
  $\Rightarrow$ Voronoi-Diagramm besteht aus $n - 1$ parallelen Linien
  $\Rightarrow n - 1$ Kanten, keine Knoten
- sonst ist das Diagramm verbunden und alle Kanten sind Segmente oder Halblinien
- zur Betrachtung des Diagramms als normalen planaren Graphen:
  - Hinzufügen eines Knotens v_∞ als künstlicher Endknoten der Halblinien
  - beinhaltet dann einen planaren verbundenen Graphen mit n Flächen, gleich vielen Kanten wie das Voronoi-Diagramm und einem Knoten mehr als das Voronoi-Diagramm
  - mir der eulerschen Formel erhalten wir: #Flächen $= f = |E| - |V| + 1 + k$, wobei k der Anzahl der Zusammenhangskomponenten entspricht (bei einem verbundenen Graphen gilt $k = 1$)
  - jede Kante ist inzident zu zwei Knoten
  - jeder Knoten (auch v_∞) ist mindestens zu drei Kanten inzident
    $\Rightarrow 2 \cdot |E| \geq 3 \cdot |V| \Longrightarrow |V| \leq \frac{2}{3} \cdot |E| \Longrightarrow n = |E| - |V| + 1 + k \geq |E| - \frac{2}{3} \cdot |E| + 2$
    $\Longrightarrow |E| \leq 3n - 6 \Longrightarrow |V| \leq \frac{2}{3}|E| \leq 2n - 4$

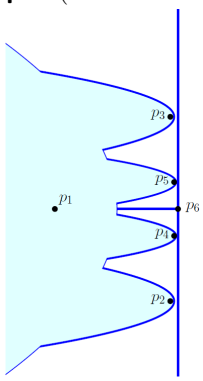
- da V den fiktiven Knoten v_∞ enthält, wissen wir jetzt, dass ein Voronoi-Diagramm höchstens $|V| - 1 \leq 2n - 5$ Knoten enthält

- **Algorithmus von Fortune:** zur Einfachheit nehmen wir an, dass es keinen Kreis gibt mit vier *site*s auf seinem Umkreis und kein *site* in seinem Inneren

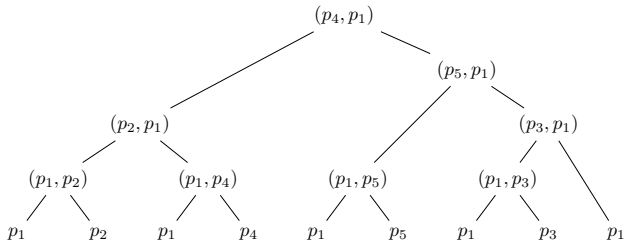
  ***sweep line* Status:**

  - jeder Punkt, der näher zu einem Punkt links der *sweep line* l ist als zu l selbst, kann nicht in einer Voronoi-Zelle einer *site* sein, die rechts der *sweep line* liegt
  - die ***beach line*** ist eine Kurve, die die Menge der Punkte, die näher zu einem Punkt links von l als zu l selbst sind, von den Punkten, die näher an l sind, trennt
  - die *beach line* ist **y-monoton** (d.h. jede horizontale Linie schneidet die *beach line* in genau einem Punkt)
  - wenn es genau eine *site* links der *sweep line* gibt, dann ist die *beach line* eine (gedrehte) Parabel
  - allgemein: die *beach line* ist eine Sequenz von Parabelbögen, wobei jeder Bogen zu einer *site* gehört
  - manche Parabel können mehrere Teile der Teilstrecken der *beach line* beisteuern
  - ein Schnittpunkt zwischen zwei aufeinanderfolgenden Parabelbögen auf der *beach line* wird ***break point*** genannt
  - der Status der *sweep line* ist die geordnete Sequenz von Parabelbögen und *break point*s auf der *beach line*
  - Speicherung des *sweep line* Status in einem binären Suchbaum

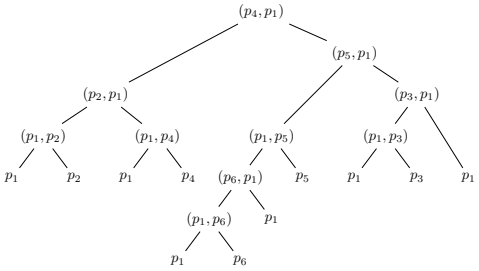
  Beispiel (*beach line* & binärer Suchbaum):



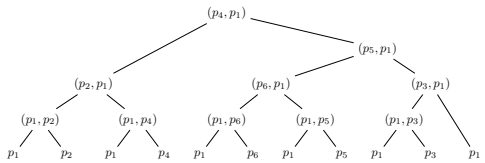
Baum vor dem Einfügen von p_6:



Baum nach dem Einfügen von p_6:



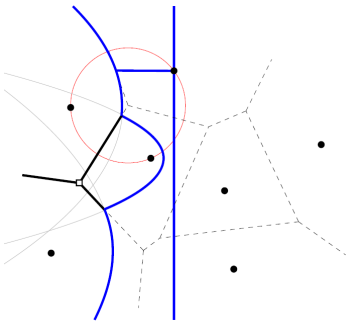
Baum nach der Neuausrichtung:



Ereigniszeitplan:

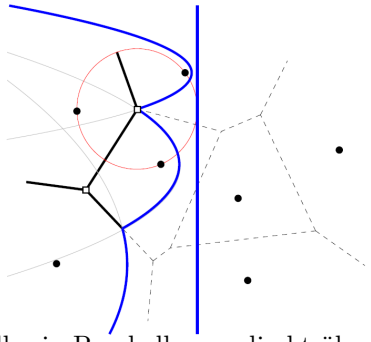
- der Status der *sweep line* ändert sich, wenn ein neuer Parabelbogen auf der *beach line* auftaucht bzw. wenn ein Parabelbogen von der *beach line* verschwindet
- verwendete Datenstruktur: *Priority Queue*

***site*-Event:**



- ein neuer Parabelbogen kann **nur** dann auf der *beach line* auftauchen, wenn die *sweep line* einen *site* p enthält
- somit sind alle *site*s, sortiert in nicht-absteigender Reihenfolge ihrer x-Koordinate eine Teil-Sequenz des Ereigniszeitplans
- Einfügen eines neuen Parabelbogens in den Suchbaum:
  1. traversieren des Suchbaums
  2. am Ende wird die y-Koordinate von p mit der Koordinate der Positionen der aktuellen *break point* verglichen
  3. die Suche endet im Parabelbogen α links von p
- q ist das Label von α
- α ist ein Fragment der Parabel, welches die Punkte näher bei der *site* q von den Punkten, die näher bei der *sweep line* liegen trennt
- Ersetzten von α in Suchbaum durch einen Teilbaum, der die Sequenz $q, (q,p), p(p,q), q$ von drei Parabelbögen und ihren *break point*s repräsentiert (Parabelbögen sind die Blätter)
- p kann direkt rechts von einem *break point* liegen:
  1. Aufteilen von α in zwei Teile, wobei ein Teil die Länge 0 hat
  2. dieser Bogen mit Länge 0 wird sofort als verschwindender Bogen betrachtet

***circle event*:**

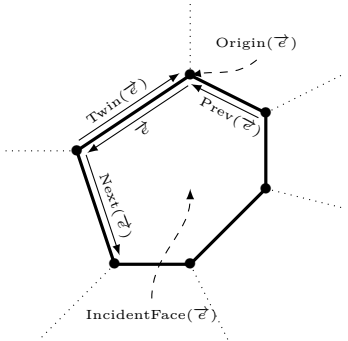


- ein Parabelbogen α mit Label q verschwindet dann aus dem *sweep line* Status, wenn der *break point* (p_1, q) unter α mit dem *break point* (q, p_2) über α übereinstimmt
  $\Rightarrow$ der Punkt p_0, an dem α verschwindet, ist der Punkt auf der *beach line*, der gleich weit entfernt ist von den *site*s p_1, q, p_2, wie zu der *sweep line*
  $\Rightarrow$ p_0 ist das Zentrum eines Kreises, der die *sweep line* berührt, die *site*s p_1, q, p_2 sind auf seinem Umkreis und kein *site* liegt innerhalb
- ein Parabelbogen α wird nicht verschwinden, falls ein Parabelbogen direkt über und unter α die gleiche *site* als Label haben
- ein Punkt ist der Ort, an dem ein Parabelbogen verschwindet
  $\Longleftrightarrow$ der Punkt ist ein Knoten im Voronoi Diagramm
- um ein *circle event* in den Ereigniszeitplan einzubinden tun wir folgendes:
  1. an jedem Ereignispunkt gilt: die drei Parabelbögen mit den Labeln p_1, p_2, p_3 erscheinen neu auf der *beach line*
  2. bei einem *site* liegt einer der drei oben genannten Bögen auf der *sweep line*, bei einem *circle event* ist gerade ein Bogen verschwunden bei (p_1, p_2) oder bei (p_2, p_3)

3. azwei *break point*s (p_1, p_2) und (p_2, p_3) konvergieren, wenn
   a) $p_1 \neq p_3$,
   b) p_1, p_2, p_3 liegen nicht auf einer Linie und
   c) der rechteste Punkt r eines eindeutigen Kreises durch p_1, p_2, p_3 ist rechts der aktuellen *sweep line*, oder r ist die *site*, die gerade rechts von einem *break point* eingefügt wurde

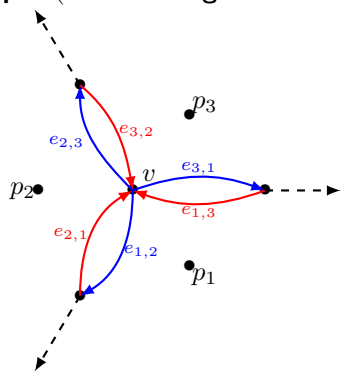
- wenn zwei *break point*s konvergieren, wird ein "potentieller" *circle event* in den Ereigniszeitplan eingefügt
- ein *circle event* bei r kann ein falscher Alarm gewesen sein (kann passieren, wenn die Bögen p_1 oder p_3 nicht vor p_2 durch ein *circle event* verschwindet, oder weil p_1, p_2 oder p_3 in zwei Teile durch ein *site*-Event aufgeteilt wurde)
- im letzten Fall wird der der *circle event* wieder aus dem Ereigniszeitplan entfernt
- hierfür speichern werden die Zeiger der Parabelbögen zu den *circle event* gespeichert, in denen sie beteiligt waren
- zum Löschen eines verschwundenen Parabelbogens α aus dem Suchbaum, wird ein Zeiger vom *circle event* zu α verwendet:
  1. Löschen von α
  2. falls α nicht beides hat (Vorgänger und Nachfolger) im *sweep line* Status (beides wären *break point*, falls es sie gibt): Löschen des bestehenden (Vorgänger oder Nachfolger) aus dem Suchbaum
  3. sonst: Löschen des Vorgängers von α und Ersetzen des Nachfolgers von α mit dem *break point* zwischen dem Parabelbogen direkt unter α und dem Parabelbögen direkt über α
  4. die einzige Änderung im Suchbaum während einer Löschoperation ist, dass das gelöschte Element durch seinen Nachfolger ersetzt wird
  5. da α schon gelöscht wurde, kann höchstens der Vorgänger von α durch den Nachfolger von α ersetzt werden
  6. Eigenschaft der inneren Knoten des Suchbaums (die *break point*s) bleibt erhalten

- **Konstruktion des Voronoi-Diagrammes:**
  - Algorithmus mit der *sweep line* basiert auf:
    1. ein Punkt ist ein Knoten in einem Voronoi-Diagramm $\Longleftrightarrow$ der Punkt ist das Zentrum eines *circle event*
    2. die *break point*s stecken die Kanten des Voronoi-Diagrammes ab
  - Speicherung des Voronoi-Diagrammes: doppelt-verkettete Kantenliste verwendet
  - immer, wenn ein neuer *break point* in die *beach line* eingefügt wird, erstellen wir zwei neue Kanten e und $Twin(e)$, mit den zugehörigen *site*s der *break point*s als ihre inzidenten Flächen
  - immer, wenn ein *circle event* behandelt wird, wird ein neuer Knoten v erstellt
  - bei jedem *circle event* verschwinden zwei *break point*s und ein neuer *break point* entsteht



  - die Halbkanten werden mit den *break point*s zu v und zu der anderen Halbkante verlinkt
  - nachdem die *sweep line* alle Punkte durchlaufen hat, wurde eine Box berechnet, die alle *site*s und alle Knoten des Voronoi-Diagrammes beinhaltet, sowie alle Halbkanten, die den Halblinien entsprechen, die in der Box verankert sind

– **Beispiel (Bestimmung der einzelnen Kanten für eine Kante $e_{1,2}$):**



$\text{Face}(e_{1,2}) = p_1$
$\text{Twin}(e_{1,2}) = e_{2,1}$
$\text{Prev}(e_{1,2}) \leftarrow e_{1,3}$
$\text{Next}(e_{1,3}) \leftarrow e_{1,2}$
$\text{Origin}(e_{1,2}) \leftarrow v$

- **Laufzeit:**
  - Ereignispunkte werden zugehörig zu einer *site* oder einem Knoten des Voronoi-Diagrammes behandelt
  - bei n *site*s gibt es **höchstens** $3n - 5$ Schritte des *sweep line* Algorithmus
  - ein *site*-Event erhöht die Anzahl an Parabelbögen und *break point*s auf der *beach line* um höchstens 4
  - ein *circle event* erhöht die Anzahl auf der *beach line* nicht
  - auf der *beach line* sind höchstens $4n$ Elemente
  - bei jedem Schritt des Algorithmus benötigt man eine konstante Anzahl an Suchbaum- und *Priority Queue*-Operationen, die jede in logarithmischer Zeit erfolgen
  - Gesamtlaufzeit: $\mathcal{O}(n \log n)$

2.3 Anwendung: Minimale Spannbäume

- benutzen eines beliebigen MST-Algorithmus (mit $\Theta(n^2)$ Kanten)
- brauchen Graphen mit weniger Kanten aber allen MSTs: ***Delaunay Graph*/Triangulation** mit den Knoten P
- der *Delaunay Graph* ist der Dual-Graph des Voronoi-Diagrammes von P
- ein MST von $G = (P, \binom{V}{2})$ ist ein Teilgraph des *Delaunay Graph*:

  Beweis:
  - $\{p_1, p_2\}$ ist eine Kante eines MST in G
  - betrachten des kleinsten Kreises, der p_1, p_2 enthält (der Kreis mit Durchmesser $\overline{p_1 p_2}$)
  - wenn es einen Punkt p auf diesem Kreis gibt gilt:
    $$d(p_1, p_2) > d(p_1, p) \text{ und } d(p_1, p_2) > d(p_2, p)$$
  - p_1, p, p_2 ist ein Kreis in G und eine Kante, die das größte Gewicht in einem Kreis hat, ist niemals Teil eines MST (rote Regel)
    $\Rightarrow$ es gibt keinen Punkt p
  - somit gibt es nur einen Kreis mit p_1, p_2 ohne einen Punkt aus P innerhalb oder auf dem Kreis
  - da die Voronoi-Zellen von p_1, p_2 durch eine Kante getrennt sind
    $\Rightarrow \{p_1, p_2\}$ ist eine Kante des *Delaunay Graph*
- ein EMST mit n Punkten in der Ebene, kann in $\mathcal{O}(n \log n)$ berechnet werden

  Beweis:
  - Voronoi-Diagramm kann in $\mathcal{O}(n \log n)$ berechnet werden
  - der *Delaunay Graph* kann in $\mathcal{O}(n)$ berechnet werden
  - der *Delaunay Graph* hat n Knoten und höchstens $3n - 6$ Kanten
  - mit Kruskal's Algorithmus kann der MST in $\mathcal{O}(n \log n)$ berechnet werden

3 Konvexe Hülle

- ein Polygon ist gegeben durch eine Sequenz von Punkten $\langle p_1, \ldots, p_n \rangle$ mit den Seiten $\overline{p_1, p_2}, \ldots, \overline{p_{n-1} p_n}, \overline{p_n p_1}$
- einfaches Polygon: keine zwei Seiten schneiden sich
- eine Menge $S \subseteq \mathbb{R}^2$ ist **konvex**, falls $\overline{q_1 q_2} \subseteq S$ für alle Punkte $p_1, p_2 \in S$
- ein Polygon ist konvex, wenn die Vereinigung seiner Begrenzung und seinem Inneren konvex ist
- ein Polygon ist konvex $\iff$ der Innenwinkel ist höchstens π
- eine konvexe Hülle $H(Q)$ einer Menge Q von Punkten ist ein konvexes Polygon mit der minimalen Anzahl an Knoten, sodass
  1. die Knoten aus $H(Q)$ sind eine Teilmenge von Q
  2. Q ist enthalten im Inneren oder auf der Begrenzung von $H(Q)$
- die Vereinigung des Inneren und der Begrenzung der konvexen Hülle einer Menge Q ist der Schnitt aller konvexen Mengen, die Q enthalten
- Knoten und Kanten, die inzident zur Außenfläche des *Delaunay Graph* von Q sind, bilden die konvexe Hülle
- eine konvexe Hülle kann in $\mathcal{O}(n \log n)$ berechnet werden

3.1 Untere Laufzeitschranke

- in einem algebraischen Entscheidungsbaum-Modell der d-ten Ordnung kann gefragt werden, ob Polynome des Grades höchstens d positiv, null oder negativ bei der Eingabe
  $\Rightarrow$ Worst-Case-Laufzeit des Sortierens ist immer noch in $\Omega(n \log n)$
- betrachten für eine Menge $X = \{x_1, \ldots, x_n\}$ von reellen Zahlen die Menge $Q = \{(x_1, x_1^2), \ldots, (x_n, x_n^2)\}$ von Punkten in der Ebene
  $\Rightarrow$ alle Punkte aus Q sind in der konvexen Hülle enthalten
- X wird sortiert, indem man $H(Q)$ entgegen dem Uhrzeigersinn abarbeitet
- $T(n)$ ist die Laufzeit zum Berechnen der konvexen Hülle
  $\Rightarrow$ Sortierung kann in $\mathcal{O}(T(n) + n)$ erfolgen
- Worst-Case-Laufzeit zum berechnen der konvexen Hülle ist $\Omega(n \log n)$
- durch das Voronoi-Diagramm und den *Delaunay Graph* können wir die konvexe Hülle in asymptotisch optimaler Zeit konstruieren
- dieser Ansatz ist komplizierter als er nutzt
- folgende Algorithmen basieren auf einer rotierenden *sweep line*

3.2 Graham's Scan

- Q ist die Menge von n Punkten in der Ebene$q_0 \in Q$
- $q_0 \in Q$ ist der am weitesten links liegende unterste Punkt aus Q
- die Punkte werden mithilfe eines Strahls, ausgehend aus q_0 und rotierend entgegen des Uhrzeigersinnes, abgearbeitet
- der Algorithmus arbeitet die Punkte $q \in Q \setminus \{q_0\}$ in ansteigender Ordnung der Winkel zwischen $\overrightarrow{q_0 q}$ und der horizontalen Linie durch q_0 in positiver Richtung

  (in Bezug auf die Ordnung $q <_{q_0} q' \iff \overrightarrow{q_0 q}$ liegt rechts von $\overrightarrow{q_0 q'}$)
- wenn es eine Linie durch q_0 und mehrere Punkte aus $Q \setminus \{q_0\}$ gibt, wird nur der am weitesten entfernte Punkt betrachtet (nur dieser kann ein Punkt von $H(Q)$ sein)
- als Vorverarbeitung kann man alle Punkte außer den am weitesten entfernten Punkt löschen (bei mehreren Punkten auf einer Linie)
- iteratives Berechnen der konvexen Hülle von $\{q_0, \ldots, q_i\},\ i = 3, \ldots, m$

- durch die Vorverarbeitung ist $\langle q_0, q_1, q_2 \rangle$ die konvexe Hülle von $\{q_0, q_1, q_2\}$
- $S = \langle q_0 = p_1, p_2, \ldots, p_{|S|} \rangle$ ist die konvexe Hülle von $\{q_0, \ldots, q_{i-1}\}$
- beim Hinzufügen von q_i muss der Innenwinkel geprüft werden (müssen wir in dem Polygon $S + q_i = \langle p_1, \ldots, p_{|S|}, q_i \rangle$ an der Stelle $p_{|S|}$ auf dem Weg von q_i nach $p_{|S|-1}$ rechts abbiegen)
  ist der Innenwinkel größer als π, ist $p_{|S|}$ nicht Teil der konvexen Hülle und wird entfernt und wird nicht wieder als potentieller Teil der konvexen Hülle betrachtet
- iteratives Wiederholen des letzten Schrittes bis der Innenwinkel wieder kleiner als π ist
- hieraus erhalten wir ein Polygon $\langle q_0 = p_1, p_2, \ldots, p_j, q_i \rangle$, wobei der Innenwinkel bei $p_2, \ldots, p_{j-1} < \pi$, weil es schon ein konvexes Polygon war
- somit liegt q_i links von
  1. $\overrightarrow{p_{j-1}p_j}$ durch Konstruktion
  2. $\overrightarrow{p_1p_j}$ durch die Ordnung
  3. $\overrightarrow{p_1p_2}$ durch die Wahl $p_1 = q_0$
- **Laufzeit:** (Vergleich Algorithmus 13.)
  1. Sortieren in $\mathcal{O}(n \log n)$
     $\Rightarrow$ alle Punkte von Q, die im Inneren eines Segmentes zwischen q_0 und einem Punkt $q \in Q$ liegen, sind direkt nach q einsortiert
  2. Löschen der Punkte in (gesamt betrachtet) linearer Zeit
  3. alle übrigen Punkte werden einmal auf den Stack gepushed und höchstens einmal gepopped
  4. die for-Schleife liegt in linearer Zeit
  5. somit wird die Laufzeit dominiert vom Sortiervorgang und der Algorithmus liegt in $\mathcal{O}(n \log n)$

3.3 Jarvis' march

- um die $\Omega(n \log n)$ Laufzeit zur Berechnung einer konvexen Hülle von n Punkten zu erhalten, wird benutzt, dass alle Eingabepunkte auf der konvexen Hülle liegen
- ist besser, als die $\Omega(n \log n)$-Grenze, wenn nur wenige Punkte auf der konvexen Hülle liegen
- Start: am weitesten links liegender unterster Punkt
- der analoge Algorithmus in 3D heißt "gift wrapping algorithm"
- die ersten k Knoten der konvexen Hülle $(q_0 = p_1, \ldots, p_k)$ sind schon entgegen des Uhrzeigersinnes berechnet worden
  $\Rightarrow p_{k+1}$ ist der nächste Punkt, den man besucht, wenn man kreisförmig um p_k scannt (Start bei $\overline{p_kp_{k-1}}$)
  wenn es mehrere dieser nächsten Punkte gibt, dann ist p_{k+1} der am weitesten von p_k entfernte Punkt
- da p_k ein Punkt der konvexen Hülle war, weiß man, dass der Winkel entgegen dem Uhrzeigersinn zwischen $\overline{p_kp_{k-1}}$ und $\overline{p_kq}$, $q \in Q \setminus \{p_k\}$ größer als π ist
  $\Rightarrow p_{k+1}$ ist der minimale Punkt aus $Q \setminus \{p_k\}$ im Bezug auf die Ordnung $<_{p_k}$
- **Laufzeit:** (Vergleich Algorithmus 14.)
  - eine Iteration der *repeat*-Schleife für jeden Knoten der konvexen Hülle
  - jedes Minimum kann in linearer Zeit bestimmt werden
  - Gesamtlaufzeit: $\mathcal{O}(nh)$ mit $h = \#$ Knoten auf der konvexen Hülle
  - falls $h \in o(\log n)$ ist dieses Algorithmus schneller als Graham's Scan

3.4 Algorithmus von Chan

- kombiniert Graham's Scan und Jarvis' march
- berechnet die konvexe Hülle in $\mathcal{O}(n \log h)$, mit $h = \#$ Knoten auf der konvexen Hülle
- es kann gezeigt werden, dass die Berechnung für die Entscheidung, ob eine Menge mit n Punkten h Elemente auf der konvexen Hülle hat, in $\Omega(n \log h)$ geschehen kann (*Kirckpatrick und Seidel*)
- der Algorithmus ist optimal *output-sensitive*

- berechnet die konvexe Hülle einer Punktemenge durch anwenden des Jarvis' march und die folgenden beiden Tricks:

  Trick 1:
  1. vorläufige Annahme: h ist bekannt
  2. Vorverarbeitung: Aufteilen von Q in $\left\lceil \frac{n}{h} \right\rceil$ Teilmengen der Größe $\leq h$
  3. Benutzen von Graham's Scan um alle konvexen Hüllen der Teilmengen zu berechnen
     $\Rightarrow \mathcal{O}(\frac{n}{h} h \log h)$
  4. Anwenden von Jarvis' march
  5. in jedem Schritt von Jarvis' march muss der nächste Knoten auf der konvexen Hülle aus den Knoten der konvexen Hüllen der Teilmengen kommen
  6. der Kandidat aus jeder Teilmenge kann in $\mathcal{O}(\log h)$ mithilfe der binären Suche gefunden werden
  7. Laufzeit von Jarvis' march liegt in $\mathcal{O}(h \frac{n}{h} \log h)$

  Trick 2:
  1. wenn man h nicht im Voraus weiß, arbeitet der Algorithmus in verschiedenen Phasen
  2. in jeder Phase wird ein Parameter m (anstelle von h) verwendet, der die Menge von Punkten in Teilmengen der maximalen Größe m teilt
  3. im Schritt des Jarvis' march werden nicht notwendigerweise alle Knoten der konvexen Hülle berechnet, aber bis zu $m+1$ Knoten der konvexen Hülle
  4. Start des nächsten Schrittes
  5. Parameter m wird erhöht durch Quadrieren von 2 bis hinzu $h \leq m < h^2$

- detailliertere Version des Algorithmus:
  - Start mit $m = 2$, Aufrechterhaltung der Bedingung $m < h^2$
  - folgende Schritte werden in jeder Phase durchgeführt:
    1. Aufteilen der Menge Q von Punkten in $r = \left\lceil \frac{n}{m} \right\rceil$ Mengen $Q_1, \ldots, Q_r$ mit jeweils höchstens m Elementen
    2. für alle $i = 1, \ldots, r$ wird Graham's Scan angewendet, um in $\mathcal{O}(m \log m) \subseteq \mathcal{O}(m \log h)$ die konvexe Hülle von Q_i zu berechnen und die geordnete Sequenz ihrer Knoten in einem Array $H(Q_i)$ zu speichern
       $\Rightarrow$ insgesamt ergibt sich eine Laufzeit von $\mathcal{O}(rm \log m) = \mathcal{O}(n \log m)$ pro Phase
    3. mit Jarvis' march werden in $\mathcal{O}(mr \log m) = \mathcal{O}(n \log m)$ höchstens $m+1$ Knoten der konvexen Hülle von Q wie folgt berechnet:
       - ▷ Beginn mit $k = 1$, dem am weitesten links liegende Punkt der untersten Punkte p_1 von Q und $H(Q) = \langle p_1 \rangle$
       - ▷ Durchführen, solange $k \leq m$ und $H(Q)$ das Folgende nicht abgeschlossen hat:
         a) nächster Knoten p_{k+1} von $H(Q)$ ist das Minimum im Bezug auf $<_{p_k}$
            da p_{k+1} der nächste Knoten aus einer der $H(Q_i)$, kann p_{k+1} in zwei Schritten berechnet werden:
            i. für $i = 1, \ldots, r$ benutzt man binäre Suche, um in $\mathcal{O}(\log m)$ den Punkt q_i zu berechnen, der das Minimum im Bezug auf $<_{p_k}$ist, aus allen Knoten aus $H(Q_i)$
               $\Rightarrow$ pro Phase braucht dieser Schritt insgesamt $\mathcal{O}(mr \log m) = \mathcal{O}(n \log m)$
            ii. Berechnen des minimalen Punktes p_{k+1} in Bezug auf die Ordnung $<_{p_k}$ aus allen Punkten $q_1, \ldots, q_r$
               $\Rightarrow$ pro Phase braucht dieser Schritt insgesamt $\mathcal{O}(mr) = \mathcal{O}(n)$
         b) falls $p_{k+1} = p_1 \Rightarrow H(Q)$ ist vollständig
         c) sonst wird p_{k+1} zu $H(Q)$ hinzugefügt und k um eins erhöht
    4. falls $H(Q)$ noch nicht vollständig ist, wird m auf m^2 erhöht und eine neue Phase begonnen

- **Laufzeit:**
  - für jede Phase: $\mathcal{O}(n \log m)$ (mit $m = 2^{2^i}$)
  - Algorithmus stoppt, sobald $m \geq h$
  - k wird so gewählt, dass $2^{2^{k-1}} < h \leq 2^{2^k}$ gilt
  - Gesamtlaufzeit: $\mathcal{O}(\sum_{i=0}^{k} n \log 2^{2^i})$ mit $\sum_{i=0}^{k} n \log 2^{2^i} = n \sum_{i=0}^{k} 2^i = n(2^{k+1} - 1) < 4n2^{k-1} < 4n \log h$

Zeichenkettensuche

- **Problem:** in einem Text sollen die Vorkommnisse eines Patterns gesucht werden (Länge des Textes ist echt größer als die Länger des Patterns)
- ein Pattern P taucht mit der **Verschiebung** s im Text T auf, falls $0 \leq s \leq n - m$ und $T[s+1, \ldots, s+m] = P[1, \ldots, m]$
- eine Verschiebung s ist *gültig*, falls P mit der Verschiebung s auftaucht
- **Ziel:** Finden aller gültigen Verschiebungen
- typische Anwendungen:
  - Textbearbeitungsprogramme mit beispielsweise ASCII ($|\Sigma| = 128$)
  - suchen nach einem Pattern in DNA-Sequenzen ($\Sigma = \{A, C, G, T\}$)
- $T[i, \ldots, j]$ ist der leere String, falls $i > j$

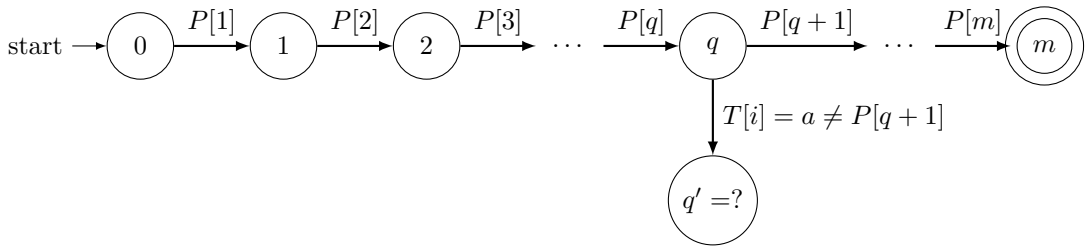
1 Naiver String-Matcher

- Testen für alle möglichen Verschiebungen, ob sie gültig sind
- **Worst-Case-Laufzeit:** $\Theta((n - m + 1)m)$ (Beispiel: $P = a^m, T = a^n$)
- ineffizient, weil die erhaltenen Informationen ignoriert werden

2 endlicher Automat-Matcher

- Idee: Benutzen der Informationen (in dem bisher gelesenen Text), die man durch die letzten Zeichen erhalten hat
- der Automat-Matcher konstruiert einen endlichen Automat $\mathcal{A}_P$ für das Pattern P, mit den folgenden Eigenschaften:
  1. $\mathcal{A}_P$ liest den Text T einmal
  2. P kommt mit der Verschiebung s in T vor $\iff$ $\mathcal{A}_P$ ist in einem Endzustand nachdem das Zeichen $T[s+m]$ gelesen wurde
- die Überführungsfunktion $\delta : Q \times \Sigma \to Q$ kann auf das Alphabet mit dem leeren Wort erweitert werden auf $\delta : Q \times \Sigma^* \to Q$ mit
  - $\delta(q, \epsilon) = q$
  - $\delta(q, wa) = \delta(\delta(q, w), a)$
- $\mathcal{A}$ **akzeptiert** alle Wörter $w \in \Sigma^*$ mit $\delta(q_0, w) \in F$ ($F \subseteq Q$ ist die Menge der Endzustände)
- der String-Matching-Automat $\mathcal{A}_P$ für das Pattern $P[1, \ldots, m]$ ist wie folgt definiert:
  - $Q = \{0, \ldots, m\}$
  - $q_0 = 0$
  - $F = \{m\}$

  sowie:



  wobei $\delta(q_0, [1, \ldots, i]) = q'$ die Länge des längsten Präfixes von P ist, das ein Suffix von dem bisher gelesenen Text ist, dh.:

  $T[1, \ldots\ldots\ldots\ldots, i-1, i]$ bekannter Teil von T

  $P[1, \ldots, q']$
- $x \in \Sigma^*$ ist ein **Suffix** von $w \in \Sigma^*$, falls $w = yx$ für ein $y \in \Sigma^*$
- $x \in \Sigma^*$ ist ein **Präfix** von $w \in \Sigma^*$, falls $w = xy$ für ein $y \in \Sigma^*$

- $\text{suf}_P : \begin{array}{ccc} \Sigma^* & \to & \{0,\dots,m\} \\ w & \mapsto & \max\{k; P[1,\dots,k] \text{ ist Suffix von } w\} \end{array}$
- $\delta(q_0, T[1,\dots,i]) = m \iff P$ tritt in T auf mit der Verschiebung $i-m$
- die Definition der Überführungsfunktion hängt nicht vom Text T ab, sondern nur vom Pattern P
- $q = \text{suf}_P(T[1,\dots,i-1]) = \delta(q_0, T[1,\dots,i-1])$,
  $a = T[i]$,
  $q' = \delta(q,a)$
  $\Rightarrow P[1,\dots\dots,q]a$ und $P[1,\dots,q']$ sind Suffixe von $T[1,\dots,i]$:
  $$\begin{array}{r} P[1,\dots,q'] \\ T[1,\dots\dots\dots,i-1,i] \\ P[1,\dots\dots,q]a \end{array}$$
- somit kann man folgendes definieren: $\delta(q,a) = \text{suf}_P(P[1,\dots,q]a)$
- **Lemma:** $\delta(q_0, T[1,\dots,i]) = \text{suf}_P(T[1,\dots,i]), \quad i = 0,\dots,n$

  Beweis:

  IA: $\delta(q_0,\epsilon) = q_0 = 0 = \text{suf}_P(\epsilon)$

  IS ($i>0$): $$\begin{array}{rcl} \delta(q_0, T[1,\dots,i]) & = & \delta(\delta(q_0, T[1,\dots,i-1], T[i]) \\ & = & \delta(\underbrace{\text{suf}_P(T[q,\dots,i-1])}_{=q}, T[i]) \\ & = & \underbrace{\text{suf}_P(P[1,\dots,q]T[i])}_{=q'} \\ & \overset{!}{=}_{(1)} & \underbrace{\text{suf}_P(T[1,\dots,i])}_{=q''} \end{array}$$

  Beweis von (1):

  Teil 1: ($q' \le q''$)

  1. $P[1,\dots,q]$ ist Suffix von $T[1,\dots,i-1]$
  2. $P[1,\dots,q']$ ist Suffix von $P[1,\dots,q]T[i]$:
     $$\begin{array}{r} T[1,\dots\dots\dots\dots,i-1,i] \\ P[1,\dots,\dots,q]T[i] \\ P[1,\dots,q'] \end{array}$$
  3. $P[1,\dots,q']$ ist ein Suffix von $T[1,\dots,i]$
  4. q'' ist maximal mit der Eigenschaft, dass $P[1,\dots,q'']$ ein Suffix von $T[1,\dots,i]$
     $\Rightarrow q' \le q''$

  Teil 2: ($q'' \le q'$)

  1. Annahme: $q'' > q'$
  2. da q' maximal mit der Eigenschaft, dass $P[1,\dots,q']$ ein Suffix von $P[1,\dots,q]T[i]$
     $\Rightarrow P[1,\dots,q'']$ ist Suffix von $T[1,\dots,i]$ aber nicht von $P[1,\dots,q]T[i]$
     $\Rightarrow q'' > q+1$ und die folgende Situation:
     $$\begin{array}{r} T[1,\dots\dots\dots\dots,i-1,i] \\ P[1,\dots\dots\dots,q''] \\ P[1,\dots\dots,q]T[i] \\ P[1,\dots,q'] \end{array}$$
  3. aber dann wäre $P[1,\dots,q''-1]$ ein größerer Suffix von $T[1,\dots,i-1]$ als $P[1,\dots,q]$, was der Maximalität von q widerspricht

- muss $(m+1)|\Sigma|$ Einträge der Überführungsfunktion berechnen
- **Laufzeit:**
  - Naive Überführungsfunktion: $\mathcal{O}(m^3|\Sigma|)$ (kann auf $\mathcal{O}(m|\Sigma|)$ reduziert werden)
    Vergleich Algorithmus 15.
  - endlicher Automat-Matcher: $\mathcal{O}(n)$
    Vergleich Algorithmus 16.

3 Knuth-Morris-Pratt-Matcher

- Linearzeit String-Matching Algorithmus
- berechnet statt der Überführungsfunktion eine

$$\pi(q) = \max\{k < q; P[1,\dots,k] \text{ ist ein Suffix von } P[1,\dots,q]\} = \text{suf}_P(P[2,\dots,q])$$

  mit $q = 1,\dots,m$ abhängig von P (Vergleich Algorithmus 17.)
- ein Prefix eines Wortes w, das auch ein Suffix von w ist, wird **Begrenzung** von w genannt
- $\pi(q)$ ist die Länge der größten Begrenzung von $P[1,\dots,q]$, die nicht $P[1,\dots,q]$ selbst ist
- $q - \pi(q)$ zeigt an, um wie viel die Verschiebung von P vergrößert werden kann, falls es ein Mismatch an der Stelle $P[q+1]$ gibt (abhängig von dem Zeichen im Text)
- für $a \in \Sigma, 0 \le q \le m$ mit $q = m$ oder $P[q+1] \neq a$ und $1 \le q \le m$ gelten die folgenden beiden Gleichungen

  1. $\pi(q) + 1 \ge \delta(q, a)$
     Beweis:
     - $q' = 0$ (trivial): $\pi(q) + 1 \ge q'$
     - $q' \neq 0$: da $P[1,\dots,q']$ ein Suffix von $P[1,\dots,q]a$ ist und $P[q+1] \neq a$
       $\Rightarrow P[1,\dots,q'-1]$ ist auch ein Suffix von $P[2,\dots,q]$
       $\Rightarrow q' - 1 \le \pi(q)$
  2. $\delta(q, a) = \delta(\pi(q), a)$
     Beweis:
     - aus der Definition von π folgt, dass $P[1,\dots,\pi(q)]$ ein Suffix von $P[1,\dots,q]$ ist
     - aus (1) folgt die folgende Situation

       $P[1,\dots,q] \quad a$
       $P[1,\dots,\pi(q)] \quad a$
       $P[1,\dots,q'-1,q']$

       $\Rightarrow P[1,\dots,q']$ ist ein Suffix von $P[1,\dots,\pi(q)]a$
     - falls es ein größeres Präfix von P gäbe, das auch ein Suffix von $P[a,\dots,\pi(q)]a$, dann wäre das auch auch Suffix von $P[a,\dots,q]a$, was ein Widerspruch zur Maximalität von q' wäre
       $\Rightarrow \delta(\pi(q), a) = q'$
- mit der wird die Überführungsfunktion simuliert
- die letzte Zeile des Algorithmus ($q \leftarrow \pi(m)$, Vergleich Algorithmus 18.) ist notwendig, damit nicht an Stelle $m+1$ in der nächsten Iteration weitergemacht wird
- **Lemma:** $(q_i = \delta(q_{i-1}, T[i]), \quad i = 1,\dots,n)$
  Beweis:
  - k ist die Anzahl der Verringerungen von q im Algorithmus (letzte Zeile) in der $(i-1)$-ten Iteration, bzw. in der i-ten Iteration in der 5. Zeile
  - q wird nur verringert, falls $q = m$ oder $P[q+1] \neq T[i]$
  - q wird verringert durch $q \leftarrow \pi(q)$
  - $q = \pi^k(q_{i-1})$ ist der Wert nach der letzten der k Verringerungen
    $\Rightarrow q_i = \begin{cases} q+1 & \text{falls } T[i] = P[q+1] \\ 0 & \text{falls } q = 0 \text{ und } T[i] \neq P[1] \end{cases}$
  - aus beiden Fällen folgt unmittelbar $q_i = \delta(q, T[i]) = \delta(\pi^k(q_{i-1}), T[i])$
    $\Rightarrow \delta(\pi^k(q_{i-1}), T[i]) = \cdots = \delta(q_{i-1}, T[i])$
- $q_i = \delta(q_0, T[1,\dots,i]) = \text{suf}_P(T[1,\dots,i]), \quad i = 1,\dots,n$
- der Algorithmus findet alle Vorkommen eines Patterns $P[1,\dots,m]$ in einem Text $T[1,\dots,n]$ in $\mathcal{O}(n)$
  Beweis:
  - anfangs: $q = 0$
  - in Zeile 5: q wird höchstens einmal reduziert
  - in Zeile 7: q wird um eins erhöht
  - q wird höchstens n mal erhöht und wird nie negativ $\Rightarrow q$ wird höchstens n-mal reduziert
  - Laufzeit (nach der Berechnung der $\mathcal{O}(n)$)
  - die wird mithilfe des KMP-Matcher für $T = P[2,\dots,m] \Rightarrow \mathcal{O}(m) \subseteq \mathcal{O}(n)$

4 Algorithmus von Boyer und Moore

Cheat-Sheet

goldener Schnitt: $\frac{a}{b} = \frac{b}{a-b}$

Algorithmen

Algorithmus 1: Selection

Input: Menge A, Integer k
Output: k-kleinstes Element in A

begin
  if $|A| \leq 10$ **then**
    return *direkt das k-kleinste Element aus A*
  $(A_1, A_2) \leftarrow$ Split(A)
  if $|A| \geq k$ **then**
    return Select(A_1, k)
  else
    return Select$(A_2, k - |A_1|)$

Algorithmus 2: Split

Input: Menge A
Output: geteilte Menge A als Mengen A_1, A_2

begin
  teile A in $\left\lfloor \frac{n}{5} \right\rfloor$ Gruppen von 5 Elementen (und einer möglichen übrigen Gruppe)
  $M \leftarrow$ Menge der $\left\lceil \frac{n}{5} \right\rceil$ (oberen) Mediane aller Gruppen
  $m \leftarrow$ Select $\left(M, \left\lceil \frac{|M|}{2} \right\rceil\right)$
  $A_1, A_2 \leftarrow \emptyset$
  for $x \in A$ **do**
    if $x \leq m$ **then**
      $A_1 \leftarrow A_1 \cup \{x\}$
    else
      $A_2 \leftarrow A_2 \cup \{x\}$
  return (A_1, A_2)

Algorithmus 3: a

c

Algorithmus 4: a

b

Algorithmus 5: a

b

Algorithmus 6: test

testt

Algorithmus 7: arg1

arg2

Algorithmus 8: arg1

arg2

Algorithmus 9: arg1

arg2

Algorithmus 10: arg1

arg2