

Datenstrukturen und effiziente Algorithmen (Vorlesungsmitschrift)

Jens-Fabian Goetzmann

Inhaltsverzeichnis

1	Suchalgorithmen	3
1.1	Insertion Sort	3
1.1.1	Insertion Sort mit Arrays	3
1.1.2	Insertion Sort mit Liste	3
1.1.3	Laufzeitabschätzung	3
1.2	Merge Sort	4
1.2.1	Korrektheit	4
1.2.2	Implementierung	4
2	Master Theorem	4
2.1	Beweis	4
2.2	RandSelect	6
2.2.1	Erwartete Laufzeit von RandSelect	6
2.3	DetSelect	6
2.3.1	Korrektheit	6
2.3.2	Laufzeit	7
2.4	Radix Sort	7
2.4.1	Korrektheit	7
2.5	Lexikografisches Sortieren in $O(dn + m)$	7
3	Datenstrukturen für geordnete Mengen	7
3.1	Skip List	7
3.2	AVL-Baum	8
3.3	Splay-Bäume	8
3.3.1	Implementierung von $\text{splay}(x, T)$	8
3.3.2	Statische Optimalität von Splay-Bäumen	9
3.4	Priority Queue	10
3.5	Binomial Queue	11
3.5.1	Amortisierte Analyse von insert	11
3.6	Fibonacci-Heap	11
4	Datenstrukturen für ungeordnete Mengen	12
4.1	Hashing	12
4.1.1	Mittlere Laufzeit von Hashing	12
4.1.2	Universelles Hashing	13
4.1.3	Idee von Hashing	14
4.1.4	Feldgröße	14
4.1.5	Dynamisches Hashing	15
4.1.6	Perfektes Hashing	15
5	Union-Find-Datenstruktur	17
5.1	Realisierung durch Bäume	17
5.1.1	Vereinigung nach Rang mit Pfadkomprimierung	17
5.1.2	Die \log^* -Funktion	18
5.1.3	Amortisierte Analyse	18
6	Graphenalgorithmen	18
6.1	Darstellung von Graphen im Speicher	20
6.2	Topologisches Sortieren	20
6.3	Breitensuche (Bredth First Search)	21
6.4	Tiefensuche (depth first search, DFS)	21
6.5	Anwendungen von DFS (und BFS)	23
6.5.1	Algorithmus zum Berechnen der SCC	23
6.6	Artukulationspunkte	25
6.7	Kürzeste Pfade	25
6.7.1	Das Single-Source shortest path Problem	26
6.7.2	All-pairs shortest path Problem	27
7	Fehlende Vorlesungen	28

8 Algorithmen für NP-Vollständige Probleme	28
8.1 Approximationsalgorithmen	28
8.1.1 Relative Approximationsalgorithmen	28
8.1.2 Polynomial Time Approximation Scheme (PTAS)	29
8.2 Heuristiken	30
8.2.1 Konstruktive Heuristiken	30
8.2.2 Lokale Suche	31
8.3 Exakte Algorithmen für NP-Schwere Probleme → Keine polynomiellen Algorithmen	32
8.3.1 Untere Schranke	33
9 Idee der amortisierten Analyse mittels Potenzialfunktion	34

1 Suchalgorithmen

1.1 Insertion Sort

1.1.1 Insertion Sort mit Arrays

- Array der Größe n
- x_1 bis x_{i-1} sortiert
- Sortiere x_i ein
 - Suche „Richtige Stelle“ mittels binärer Suche ($O(\log n)$)
 - Einfügen von x_i an Stelle j ($O(i - j)$, im schlimmsten Falle also $O(i)$)

1.1.2 Insertion Sort mit Liste

- Liste der Länge n
- x_1 bis x_{i-1} sortiert
- Sortiere x_i ein
 - Suche „Richtige Stelle“ mittels Durchlaufen ($O(j)$, im schlimmsten Falle also $O(i)$)
 - Einfügen von x_i ($O(1)$)

1.1.3 Laufzeitabschätzung

Insgesamt: Sei $T(n)$ die Anzahl der benötigten Elementaroperationen für n Elemente im worst case. Also folgt:

$$T(n) \leq c \cdot n + T(n-1) \leq \sum_{i=1}^n c \cdot i = c \cdot \sum_{i=1}^n i \leq c \cdot \sum_{i=1}^n n = c \cdot n^2$$

Also $T(n) \in O(n^2)$. (Diese Schreibweise ist nicht ganz korrekt, weil n^2 keine Funktion ist. Wir identifizieren n^2 mit der Funktion $f: \mathbb{N} \mapsto \mathbb{R}_{\geq 0}, f(n) := n^2$).

Frage: Ist die Analyse „scharf“? Gibt es Instanzen für die tatsächlich $\Omega(n^2)$ Elementaroperationen gebraucht werden?
 Array: Betr. Instanz $n, n-1, \dots, 1$. Fügen wir das i -te Element $n-i+1$ in das sortierte Array ein, müssen wir alle $i-1$ bisher eingefügten Elemente verschieben. Also brauchen wir mindestens i Elementaroperationen.

Liste: Betr. Instanz $1, \dots, n$. Für jedes einzufügende Element müssen wir die komplette Liste durchlaufen. Also benötigen wir mindestens i Elementaroperationen.

Also:

$$T(n) \leq n + T(n-1) \geq \sum_{i=1}^n i \geq \sum_{i=\lceil \frac{n}{2} \rceil}^n \left\lceil \frac{n}{2} \right\rceil = \underbrace{\left(n - \left\lceil \frac{n}{2} \right\rceil + 1 \right)}_{\geq \frac{n}{2}} \cdot \underbrace{\left\lceil \frac{n}{2} \right\rceil}_{\geq \frac{n}{2}} \geq \frac{n^2}{4}$$

Also ist $T(n) \in \Omega(n^2)$ und $T(n) \in \Theta(n)$.

1.2 Merge Sort

1.2.1 Korrektheit

Beweis durch Induktion. Induktionsanfang $n = 1$: klar.

$n - 1 \rightarrow n$: Die rekursiven Aufrufe liefern $x_1, \dots, x_{\lfloor \frac{n}{2} \rfloor}$ bzw. $x_{\lfloor \frac{n}{2} \rfloor + 1}, \dots, x_n$ korrekt sortiert zurück, da $\lfloor \frac{n}{2} \rfloor \leq n - 1$ und $n - \lfloor \frac{n}{2} \rfloor \leq n - 1$. Da „merge“ korrekt ist, folgt die Korrektheit.

1.2.2 Implementierung

Idee: Für beide Felder verwalte einen Zeiger auf das kleinste noch nicht übertragene Element. Das insgesamt kleinste noch nicht übertragene Element ist eines der beiden.

Maximale Rekursionstiefe $\lceil \log_2 n \rceil$, denn jedes Problem dieser Tiefe enthält höchstens $2^{\lceil \log_2 n \rceil \cdot \lceil \log_2 n \rceil} = 1$ Element und damit gibt es keinen rekursiven Aufruf.

Aufwand pro Rekursionsebene Sei n_j^i für $j = 1, \dots, k_i$ die Größe der Teilprobleme von Tiefe i . Dann gilt $\sum_{j=1}^{k_i} n_j^i = n$, da jedes Element in genau einem Teilproblem enthalten ist.

...

Gesamtlaufzeit ist höchstens $\sum_{i=1}^{\lceil \log_2 n \rceil} c \cdot n = c \cdot n \cdot \lceil \log_2 n \rceil \in O(n \log n)$.

2 Master Theorem

2.1 Beweis

$$T(n) = \begin{cases} c & \forall n \leq N \\ cn^k + aT(\lceil \frac{n}{b} \rceil) & \forall n \geq N \end{cases}$$

Lemmata

$$c^{\log_b a} = a^{\log_b c}$$

Sei $r \neq 1$. Dann gilt:

$$\sum_{j=0}^i r^j = \frac{r^{i+1} - 1}{r - 1}$$

Beweis durch Induktion. Für $i = 0$ gilt:

$$1 = \sum_{j=0}^0 r^j = \frac{r^1 - 1}{r - 1} = 1$$

Für $i - 1 \rightarrow i$:

$$\sum_{j=0}^i r^j = r^i + \sum_{j=0}^{i-1} r^j = r^i + \frac{r^i - 1}{r - 1} = \frac{r^{i+1} - r^i + r^i - 1}{r - 1} = \frac{r^{i+1} - 1}{r - 1}$$

Für $r > 1$:

$$\sum_{j=0}^i r^j \leq \frac{r^{i+1}}{r - 1} \in O(r^i)$$

Für $r < 1$:

$$\sum_{j=0}^i r^j = \frac{r^{i+1} - 1}{r - 1} = \frac{1 - r^{i+1}}{1 - r} \leq \frac{1}{1 - r} \in O(1)$$

Intuition Jeder rekursive Aufruf teilt die Problemgröße etwa um den Faktor b . Rekursionstiefe ist etwa $\log_b n$. Wieviel Laufzeit brauchen wir in Tiefe j ?

Wir haben a^j Teilprobleme der Größe etwa $\frac{n}{b^j}$. Die Laufzeit ist also $\leq a^j \left(\frac{n}{b^j}\right)^k$.

Ist $\left(\frac{a}{b^k}\right)^j = 1$, brauchen wir in jeder Tiefe $O(n^k)$, also insgesamt $O(n \log n)$.

Ist $\left(\frac{a}{b^k}\right)^j < 1$, ist $\sum_{j=0}^i \left(\frac{a}{b^k}\right)^j \in O(1)$, also Gesamtlaufzeit $O(n^k)$.

Ist $\left(\frac{a}{b^k}\right)^j > 1$, ist $\sum_{j=0}^i \left(\frac{a}{b^k}\right)^j \in O\left(\left(\frac{a}{b^k}\right)^i\right)$. Die Rekursionstiefe ist etwa \log_b^n . Gesamtlaufzeit $O\left(n^k \cdot \left(\frac{a}{b^k}\right)^{\log_b n}\right) = O\left(n^k \frac{a^{\log_b n}}{(b^k)^{\log_b n}}\right) = O\left(n^k \frac{n^{\log_b a}}{n^k}\right) = O(n^{\log_b a})$.

Beweis Sei $G(j)$ die Problemgröße in Tiefe j . Behauptung: $G(j) \leq b^{\lceil \log_b n \rceil - j}$. Der Beweis erfolgt durch vollständige Induktion.

Also ist die Rekursionstiefe höchstens $\lceil \log_b n \rceil$, da Probleme in dieser Tiefe eine Größe von höchstens 1 haben und somit keinen rekursiven Aufruf machen. Damit ist die Laufzeit in Tiefe j höchstens $c \cdot a^j G(j)^k \leq c \cdot a^j \cdot (b^{\lceil \log_b n \rceil - j})^k$.

Gesamtlaufzeit ist also

$$T(n) \leq \sum_{j=0}^{\lceil \log_b n \rceil} c \cdot a^j \cdot \left(\underbrace{b^{\lceil \log_b n \rceil - j}}_{\leq b \cdot b^{\log_b n - j}} \right)^k \leq \sum_{j=0}^{\lceil \log_b n \rceil} \underbrace{c \cdot b^k}_{c'} \cdot a^j \left(\frac{b^{\log_b n}}{b^j} \right)^k$$

$$= \sum_{j=0}^{\lceil \log_b n \rceil} c' n^k \left(\frac{1}{b^k} \right)^j$$

Rest wie bei der Induktion!

Erweiterung

$$T(n) = \begin{cases} c & \forall n \leq N \\ cn^k + aT(\lceil \frac{n}{b} \rceil + d) & \forall n \geq N \end{cases}$$

c, k, a, b, N wie vorher. $d \leq \frac{N}{3} - 1$.

Sei $G(j)$ die Problemgröße in Tiefe j . Dann gilt: $G(j) \leq b^{\lceil \log_b n \rceil - j} + 3d$.

Beweis durch Induktion.

$j = 0$: klar.

$j - 1 \rightarrow j$:

$$G(j) \leq \frac{\lceil G(j-1) \rceil}{b} + d \leq \frac{\lceil b^{\lceil \log_b n \rceil - (j-1)} + 3d \rceil}{b} + d$$

$$= b^{\lceil \log_b n \rceil - j} + \underbrace{\frac{\lceil 3d \rceil}{b}}_{\leq 2d} + d \leq b^{\lceil \log_b n \rceil - j} + 3d$$

Laufzeit in Tiefe j :

$$a^j c \left(\underbrace{b^{\lceil \log_b n \rceil - j}}_{\geq \frac{N}{b}} + \underbrace{3d}_{\leq N} \right)^k$$

Jedes Teilproblem hat Größe von mindestens $\frac{N}{b}$, da nur rekursive Aufrufe für Probleme mit Größe mindestens N erfolgen. Also ist die Laufzeit in Tiefe j höchstens

$$a^j c \left(2b^2 b^{\lceil \log_b n \rceil - j} \right)^k \leq a^j c (2b^2 b^{\log_b n - j})^k = c 2^k b^k \left(\frac{n}{b^k} \right)^j$$

Rest wie vorher.

2.2 RandSelect

2.2.1 Erwartete Laufzeit von RandSelect

Zufallsexperiment: Element mit Rang i wurde gewählt. $p(i) = \frac{1}{n}$.
Ist der Rang des Pivotelementes i , gilt:

$$|x_{<}| = i - 1, |x_{>}| = n - i$$

Sei $T(n, k)$ die erwartete Anzahl von Vergleichen. Dann gilt:

$$T(n, k) = n - 1 + \frac{1}{n} \left(\sum_{i=1}^{k-2} T(n-i, k-(i-1)-1) + 0 + \sum_{i=k}^n T(i-1, k) \right)$$

Behauptung: $T(n, k) \leq 12 \cdot n$.

Beweis: $n \leq 12$: $T(n, k) \leq 12n$, da $T(n, k) \leq n^2$.

$n > 12$:

$$\begin{aligned} T(n, k) &\stackrel{(IA)}{\leq} n - 1 + \frac{1}{n} \left(\sum_{i=1}^{k-2} 12(n-i) + \sum_{i=k}^n 12(k-1) \right) \\ &= n - 1 + \frac{12}{n} \cdot \left(\sum_{i=n-k+2}^{n-1} i + \sum_{i=k-1}^{n-1} i \right) \\ &= n + 1 + \frac{12}{n} \cdot \left(\underbrace{\frac{n(n-1)}{2}}_{\leq \frac{n^2}{2}} - \underbrace{\frac{(n-k+2)(n-k+1)}{2}}_{(1)} + \underbrace{\frac{n(n-1)}{2}}_{\leq \frac{n^2}{2}} - \underbrace{\frac{(k-1)(k-2)}{2}}_{(2)} \right) \end{aligned}$$

1.Fall: $k \leq \frac{n}{2}$. Dann gilt:

$$\frac{(n-k+2)(n-k+1)}{2} \geq \frac{\frac{n}{2} \frac{n}{2}}{2} = \frac{n^2}{8}$$

2.Fall: $k > \frac{n}{2}$. Dann gilt:

$$\frac{(k-1)(k-2)}{2} \geq \frac{(\frac{n}{2}-1)(\frac{n}{2}-2)}{2} = \frac{n(n-1)}{8} - \frac{n-2}{2} \geq \frac{n(n-1)}{12}$$

Insgesamt:

$$T(n, k) \leq n - 1 + \frac{12}{n} \left(\frac{n^2}{2} + \frac{n^2}{2} - \frac{n(n-1)}{12} \right) = 12n$$

2.3 DetSelect

2.3.1 Korrektheit

Zeige: $\text{rang}(x) \geq \frac{3}{10}n$ für alle n , die durch 5 teilbar sind. Bild für $n = 25$. Wir können annehmen (durch Ummummerieren), dass $m_1 < m_2 < \dots < m_5$

Die drei kleinsten Elemente der Gruppen $X_1, \dots, X_{\lceil \frac{n}{10} \rceil}$ sind höchstens so groß wie x . Das sind $3 \lceil \frac{n}{10} \rceil \geq \frac{3}{10}n$ viele.

Ist n nicht durch 5 teilbar, finden wir zumindest $\frac{3}{10}(n-4)$ (eine Zahl zwischen n und $n-4$ ist auf jeden Fall durch 5 teilbar) $\geq \frac{3}{10}n - 4$ Elemente $\leq x$.

Analog zeigen wir $\text{rang}(x) \leq \frac{7}{10}n + 4$.

Für $n > 80$ gilt:

$$\frac{1}{4}n \leq \frac{3}{10}n - 4 \leq \text{rang}(x) \leq \frac{7}{10}n \leq \frac{3}{4}n$$

2.3.2 Laufzeit

Sei $T(n)$ die Laufzeit von DetSelect. Dann gilt:

$$T(n) \leq \begin{cases} c & \text{für } n < 80 \\ cn + T\left(\frac{n}{5}\right) + T\left(\frac{3}{4}n\right) & \text{sonst} \end{cases}$$

Intuitiv sind die zwei Teilprobleme der Größen $\frac{n}{5}$ und $\frac{3}{4}n$ leichter zu lösen als ein Teilproblem der Größe $\frac{n}{5} + \frac{3}{4}n = \frac{19}{20}n$. Wir erwarten also, dass $T(n) \leq T'(n)$ mit

$$T'(n) = \begin{cases} c & \text{für } n < 80 \\ cn + T\left(\frac{19}{20}n\right) & \text{sonst} \end{cases} \in O(n) \text{ (Master Theorem)}$$

Beweis Daher zeigen wir induktiv: $T(n) \leq 20cn$.

Für $n < 80$: klar.

Für $n \geq 80$:

$$T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{3}{4}n\right) \leq cn + 20c\frac{n}{5} + 20c\frac{3}{4}n = 20cn$$

2.4 Radix Sort

2.4.1 Korrektheit

Sei $x = (k_1, \dots, k_d, i)$ und $x' = (k'_1, \dots, k'_d, i')$. Wir zeigen: nach der Sortierung bzgl. der i -ten Komponente liegt x vor x' , falls $(k_1, \dots, k_d) <_{lex} (k'_1, \dots, k'_d)$.

Induktionsbeweis: IA: $i = d$. Da nach der d -ten Komponente korrekt sortiert wird, klar.

$i + 1 \rightarrow i$:

1. Fall: $k_i < k'_i$: Da wir nach der i -ten Komponente sortieren, stimmt die Aussage.

2. Fall: $k_i = k'_i$ und $(k_{i+1}, \dots, k_d) <_{lex} (k'_{i+1}, \dots, k'_d)$. Da die Sortierung bezüglich der letzten $d - i - 1$ Schlüssel korrekt ist (nach IA) ist x vor x' bevor nach dem i -ten Schlüssel sortiert ist. Da die Sortierung bezüglich des i -ten Schlüssels stabil erfolgt, ist die Aussage korrekt.

2.5 Lexikografisches Sortieren in $O(dn + m)$

Algorithmus:

- 1) Erstelle Array mit Tupeln (k_i^j, i) mit $1 \leq j \leq n, 1 \leq i \leq d$. Laufzeit: $O(nd)$.
- 2) Sortiere Tupel mit Bucket-Sort nach 1. Eintrag, dann nach 2. Eintrag. Laufzeit: Erster Eintrag: $O(n + d)$, Zweiter Eintrag: $O(n + d)$
- 3) Eliminiere Doppelte in Bucket-Struktur B . Laufzeit: $O(n)$
- 4) Radix-Sort, wobei bei den Ausgaben nur die Buckets ausgegeben werden, die in B vorkommen. Laufzeit: d -mal alle Elemente in Buckets: $O(dn)$. d -mal alle Buckets betrachten, die Elemente beinhalten (maximal n): $O(dn)$

Gesamtlaufzeit: $O(dn + m)$.

3 Datenstrukturen für geordnete Mengen

3.1 Skip List

Erwartete Größe der i -ten Liste Wahrscheinlichkeit dass ein Element x in die i -te Liste aufgenommen wird ist gerade 2^{-i+1} .

Aufwand zum Suchen Die $\lfloor \log n \rfloor$ -te Liste hat erwartete Größe

$$\frac{n}{2^{\lfloor \log n \rfloor - 1}} = \frac{n}{2^{\lfloor \log n \rfloor}} 2 \leq 4$$

Im Erwartungswert brauchen wir in der $\lfloor \log n \rfloor$ -ten Liste $O(1)$ Zeit.

Erwartete Schritte in einer anderen Liste: Sei $P(\text{„Stop“})$ die Wahrscheinlichkeit, dass das Element, dass wir treffen, \geq dem Schlüssel ist, den wir suchen.

$$P(\text{„Stop“}) \geq P(\text{„Das Element ist in der } i+1\text{-ten Liste“}) = \frac{1}{2}$$

3.2 AVL-Baum

Behauptung Ein AVL-Baum der Höhe h hat mindestens $\left(\frac{3}{2}\right)^h$ Knoten. Dies lässt sich durch Induktion über h beweisen.

Induktionsanfang: $h = 0$ Der Baum hat genau ein Blatt.

Induktionsschritt: $h - 1 \rightarrow h$ Sei $K(h)$ die minimale Knotenanzahl eines AVL-Baumes der Höhe h . Dann gilt:

$$\begin{aligned} K(h) &\geq 1 + \min(\underbrace{K(h-1) + K(h-2)}_{\text{hier ist das Minimum}}, K(h-1) + K(h-1)) \geq 1 + \left(\frac{3}{2}\right)^{h-1} + \left(\frac{3}{2}\right)^{h-2} \geq \frac{3}{2} \left(\frac{3}{2}\right)^{h-2} \left(\frac{3}{2}\right)^{h-2} \\ &= \left(1 + \frac{3}{2}\right) \left(\frac{3}{2}\right)^{h-1} \geq \left(\frac{3}{2}\right)^2 \left(\frac{3}{2}\right)^{h-2} \end{aligned}$$

Also hat ein AVL-Baum mit n Elementen höchstens die Höhe $\log_{\frac{3}{2}} n$, da $\left(\frac{3}{2}\right)^h \leq n$ sein muss.

3.3 Splay-Bäume

Binärer Suchbaum mit vielen interessanten Zusatzeigenschaften, von denen wir nur eine zeigen (vgl. Tarjan: „Self-Adjusting Binary Search Trees“ Journal of the ACM 32(3)).

Zusätzliche Operation: $\text{splay}(x, T)$: Verändert den Baum so, dass x zur Wurzel wird, falls x existiert. Andernfalls wird der direkte Nachfolger oder Vorgänger zur Wurzel.

Mit Hilfe von einer oder zwei splay -Operationen lassen sich die Befehle „lookup“, „search“, „insert“ und „delete“ in sonst konstanter Zeit implementieren:

lookup(x, T) : $\text{splay}(x, T) \rightarrow$ Zugriff auf die Wurzel

search(x, T) : Analog zu lookup

insert(x, T) : $\text{splay}(x, T) \rightarrow y$ ist Wurzel von x , wobei y direkter Vorgänger oder Nachfolger ist $\rightarrow x$ als neue Wurzel, y als linkes oder rechtes Kind, vorher rechtes oder linkes Kind von y als rechtes oder linkes Kind.

delete(x, T) : $\text{splay}(x, T) \rightarrow$ Lösche $x \rightarrow \text{splay}(\infty, T_l)$ (größtes Element in T_l wird Wurzel, hat also kein rechtes Kind) \rightarrow Füge T_r als rechtes Kind der Wurzel von T_l ein.

3.3.1 Implementierung von $\text{splay}(x, T)$

Eine Rotation am Vater von x bringt x ein Stück näher zur Wurzel. Wir betrachten aber immer 2 Schritte auf einmal, d.h. wir rotieren zuerst am Großvater und dann am Vater. Wir erhalten dann 6 Fälle:

Fall 1, 2: x hat keinen Großvater, d.h. x ist Kind der Wurzel.

1. Fall: x ist linkes Kind (Zick). Dann wird eine Rechtsrotation an der Wurzel durchgeführt und x wird zur Wurzel.

2. Fall: x ist rechtes Kind (Zack). Dann wird eine Linksrotation an der Wurzel durchgeführt und x wird zur Wurzel.

3. Fall: x ist linkes Kind, Vater von x ist auch linkes Kind (Zick Zick). Dann wird zunächst am Großvater eine Rechtsrotation und dann am Vater eine Rechtsrotation durchgeführt. Dann ist x Wurzel, der Vater von x rechtes Kind, und der Großvater von x rechtes Kind des rechten Kindes von x .

4. Fall: x ist rechtes Kind, Vater von x ist auch rechtes Kind (Zack Zack). Analog zu Zick Zick.

5. Fall: x ist linkes Kind, Vater von x ist rechtes Kind (Zick Zack). Dann wird zunächst eine Rechtsrotation am Vater von x durchgeführt und dann eine Linksrotation am Großvater von x . Dann ist x die Wurzel, der Großvater von x das linke Kind von x und der Vater von x das rechte Kind von x .

6. Fall: x ist rechtes Kind, Vater von x ist linkes Kind (Zack Zick). Analog zu Zick Zack.

Beachte: Splay-Bäume können unbalanciert werden. Wird dann auf ein Element zugegriffen, das tief im Baum liegt, ist die entsprechende Splay-Operation teuer. Aber: Wir können Splay-Operationen amortisiert analysieren, d.h. zeigen dass eine Splay-Operation im Durchschnitt nur $O(\log n)$ Rotationen ausführt. Wir nutzen die Analyse mittels einer Potenzialfunktion. Sei dazu

$W(Z)$ die Anzahl der Knoten im Teilbaum von Z , auch das „Gewicht“ des Teilbaumes genannt

Anmerkung: Später definieren wir für jeden Knoten ein individuelles Gewicht $w(x)$. Das Gewicht eines Teilbaumes ist dann gerade die Summe der Gewichte seiner Knoten. Der Beweis, den wir gleich herleiten, geht für diese Gewichtsdefinition Wort für Wort durch.

$r(z) = \log W(Z)$, auch „Rang“ von z genannt

$$\Phi(T) = \sum_{z \in T} r(z)$$

Satz: Die amortisierten Kosten K_x einer Operation $splay(x, T)$ sind höchstens $3(r(t) - r(x)) + 1 \in O(1 + \log \frac{w(t)}{w(x)}) \subset O(\log n)$, wobei t die Wurzel des Baumes T ist.

Hilfsbehauptung: Ein „Zick“ oder „Zack“ hat amortisierte Kosten $1 + 3(r'(x) - r(x))$. Ein „Zick Zack“, „Zack Zick“, „Zick Zick“ oder „Zack Zack“ hat höchstens $3(r'(x) - r(x))$ Operationen, wobei $r'(x)$ der Rang von x nach der Operation ist.

Mit der Hilfsbehauptung erhalten wir für die amortisierten Kosten einer Splay-Operation eine Teleskopsumme: $1+3(\text{Endrang von } x - \text{Anfangsrank von } x)$. Der Rang der Wurzel ist unabhängig von der Höhe des Baumes stets $\log n$.

Fall „Zick“ Amortisierte Kosten:

$$1 + \Phi(T')\Phi(T) = 1 + r'(x) + r'(z) - r(x) - r(z) = 1 + r'(z) - r(x) \stackrel{r'(x) \geq r(x)}{\leq} 1 + 3(r'(x) - r(x))$$

Fall „Zick Zick“ Amortisierte Kosten:

$$2 + r'(z) + r'(y) + r'(x) - r(z) - r(y) - r(x) \stackrel{r(y) \geq r(x), r'(x) \geq r'(y), r'(x) = r(z)}{\leq} 2 + r'(y) + r'(z) - 2r(x) \stackrel{(1)}{\leq} 3(r'(x) - r(x))$$

(1) ist äquivalent z:

$$\begin{aligned} -2 + 2r'(x) - r'(z) - r(x) &\geq 0 \\ \Leftrightarrow \log W'(z) \log W(x) - 2 \log W'(x) &\leq -2 \\ \Leftrightarrow \underbrace{\log \frac{W'(z)}{W'(x)}}_{=:a} + \underbrace{\log \frac{W(x)}{W'(z)}}_{=:b} &\stackrel{(2)}{\leq} -2 \end{aligned}$$

(2) gilt, da für $0 \leq a, b \leq 1$ mit $a + b \leq 1$ (da $W'(z) + W(x) \leq W'(x)$) der Ausdruck $\log a + \log b$ für $a + b = \frac{1}{2}$ maximiert wird. Dann ist $\log a + \log b = -2$.

Fall „Zick Zack“ Amortisierte Kosten:

$$2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \stackrel{r'(x) = r(z), r(x) \leq r(y)}{\leq} 2 + r'(y) + r'(z) - 2r(x) \stackrel{\text{wie oben}}{\leq} 3(r'(x) - r(x))$$

Die anderen Fälle verlaufen analog.

3.3.2 Statische Optimalität von Splay-Bäumen

Gegeben:

Menge X von Elementen

C Folge von Zugriffen auf Elemente von X Splay-Bäume sind reihenfolgeabhängig!

B der optimale Suchbaum für C auf X

k_B Anzahl Vergleiche, die ausgeführt werden, wenn C auf B bearbeitet wird

k_{Splay} sind die Kosten wenn C auf einem beliebigen Anfangssuchbaum mit Hilfe der Splay-Methode bearbeitet wird

Satz:

$$k_{Splay} \in O(k_B + n^2)$$

Anmerkung: Es ist ein offenes Problem, ob es möglich ist, die Kosten k_{Splay} mit den Kosten des optimalen dynamischen Suchbaumes in Verbindung zu setzen.

Zum Beweis müssen wir die Definition der Potenzialfunktion abändern. Sei t die Tiefe von B und $t(x)$ die Tiefe von x in B .

$$w(x) = 3^{t-t(x)}$$

$$W(x) = \sum_{z \text{ im Teilbaum von } x} w(z)$$

$$\Phi(T) = \sum_{x \in T} \log W(x)$$

Im Vergleich zur vorherigen Potenzialfunktion geben wir jedem Knoten ein individuelles Gewicht.

Mit dem vorherigen Beweis (Wort für Wort) zeigen wir, dass die amortisierten Kosten von $\text{splay}(x, T)$ in $O(1 \log \frac{W(t)}{W(x)})$ sind, wobei t die Wurzel von T ist.

Als nächstes analysieren wir $W(t)$ und $\Phi(T)$. Behauptung: $W(t) \leq 3^{t+1}$.

Im binären Baum gibt es höchstens 2^i Knoten in Tiefe i . Also gilt:

$$W(t) \leq \sum_{i=0}^t 2^i 3^{t-i} = 3^t \cdot \sum_{i=0}^t \left(\frac{2}{3}\right)^i \leq 3^t \cdot 3 = 3^{t+1}$$

Weiter gilt:

$$\Phi(T) = \sum_{x \in T} \log W(x) \leq \sum_{x \in T} \log W(t) \leq n \cdot \log 3^{t+1} \in O(n^2)$$

Aus $W(x) \geq w(x) 3^{t-t(x)}$ und $W(t) \leq 3^{t+1}$ folgt, dass die amortisierten Kosten k_x der Anfrage an x beschränkt sind durch

$$k_x \leq O\left(1 + \log \frac{W(t)}{W(x)}\right) \subseteq O\left(1 + \log \frac{3^{t+1}}{3^{t-t(x)}}\right) = O(1 + \log 3^{t(x)})$$

$$= O\left(1 + \underbrace{k(x)}\right)$$

Anzahl Vergleiche, die in B bei der Anfrage an x ausgeführt werden

Also gilt:

$$k_{\text{splay}} \leq \sum_{x \in C} k_x + \Phi(T_{\text{Start}}) - \Phi(T_{\text{End}}) \in O(k_B + n^2)$$

3.4 Priority Queue

Wir repräsentieren einen perfekten binären Baum in einem Array. $A[1]$ ist die Wurzel des Baumes, Die Kinder des Knotens, die in $A[i]$ gespeichert sind, werden in $A[2i]$ bzw. $A[2i+1]$ gespeichert. Dementsprechend findet man den Vater von $A[i]$ in $A[\lfloor \frac{i}{2} \rfloor]$.

Bei einem perfekten Beispiel sind gerade die Arrayeinträge $A[1]$ bis $A[n]$ belegt.

Realisierung der Befehle einer Priority-Queue im binären Heap:

$\text{Min}(H)$ liefert einfach $A[1]$, da das kleinste Element an der Wurzel gespeichert sein muss.

$\text{Insert}(x, H)$. Wir speichern x im ersten freien Platz des Arrays. Natürlich kann die Heapeigenschaft an x verletzt sein. Ist sie verletzt, so vertauschen wir x mit seinem Vater und beobachten, dass die Heap-Eigenschaft nur an der neuen Position von x verletzt sein kann. Wir führen das Vertauschen mit dem Vater so lange fort, bis die Heap-Eigenschaft erfüllt ist.

Im Pseudocode:

$\text{Insert}(x, H)$: Sei n die Anzahl der Elemente im Heap vor der Operation

$A[n+1] \leftarrow x$

while $A[i] < A[\frac{i}{2}]$ do

vertausche($A[i], A[\frac{i}{2}]$)

$i \leftarrow \frac{i}{2}$

end

$\text{Extract_Min}(H)$: Wir schieben das letzte Element im Array an die Wurzel. Dadurch kann die Heapeigenschaft an den Kindern verletzt sein. Um die Heapeigenschaft wiederherzustellen, vertauschen wir das Element mit dem Kind mit dem kleineren Schlüssel. Jetzt kann die Heapeigenschaft nur an den Kindern der neuen Position des Elementes verletzt sein.

Pseudocode:

procedure $\text{EXTRACT_MIN}(H)$

Verschiebe $A[n]$ nach $A[1]$

$i \leftarrow 1$

while $A[i] > A[2i]$ oder $A[i] > A[2i+1]$ **do**

if $A[2i] < A[2i+1]$ **then**

$i \leftarrow 2i$

else

$i \leftarrow 2i+1$

Vertausche $A[i]$ und $A[\frac{i}{2}]$

Decrease_key(id, k, H): Wir werden gleich sehen, wie man einen Identifier realisieren kann, so dass man in konstanter Zeit die Position im Array finden kann, an der das zugehörige Element gespeichert ist. Haben wir die Position gefunden, ändern wir den Schlüssel und stellen die Heapeigenschaft wie bei „Insert“ wieder her.

Delete(id, H): decrease_key(id, $-\infty$, H), extract_min(H)

Alle Operationen (außer min) haben eine Laufzeit, die proportional zur Tiefe des Baumes ist, also $O(\log n)$.

Realisierung des Identifiers: Bei jedem Insert(x) legen wir ein Objekt an, das x und die aktuelle Position des Elementes im Array speichert und liefern einen Zeiger auf das Objekt zurück. Im Array speichern wir ebenfalls einen Zeiger auf das Objekt. Verschieben wir ein Element im Array, aktualisieren wir die Positionsinformation im Objekt. Die asymptotische Laufzeit ändert sich dadurch nicht.

3.5 Binomial Queue

Beispiel: Queue für 20 Elemente besteht aus einem B_4 und einem B_2 .

Hilfsoperation:

procedure ADDTREE(Heap B , Zeiger i)

▷ i ist Zeiger in die Wurzelliste mit Vorgänger von i kleineren Wurzelgrad als B , Nachfolger größeren

if In der Wurzelliste ist ein Heap B' der gleichen Größe enthalten **then**

Verbinde B und B' zu B''

ADDTREE(B'' , next(i))

else

Füge B in Wurzelliste ein

Min: Wir speichern einen Zeiger auf das kleinste Element. Bei den anderen Operationen achten wir darauf, dass wir den Zeiger aktualisieren. (Dazu braucht nur einmal die Wurzelliste durchlaufen zu werden)

Insert(x): Erzeuge B_0 , der nur x enthält. Füge diesen mit addTree in die Wurzelliste ein, wobei der Zeiger genau der Anfangszeiger der Wurzelliste ist. Die Laufzeit von insert hat asymptotisch die gleiche Laufzeit von addTree. Da alle Heaps der Wurzelliste eine unterschiedliche Größe haben, und der größte Heap ein $B_{\lfloor \log n \rfloor}$ ist, gibt es in der Wurzelliste höchstens $\log n$ Heaps. Wir rufen addTree höchstens $\log n$ mal rekursiv auf. Also ist die Laufzeit $O(\log n)$.

DecreaseKey(id, k): wie beim binären Heap (Zeiger auf Knoten als Identifier)

ExtractMin: Das kleinste Element ist die Wurzel eines Heaps B_i . Wir löschen B_i aus der Wurzelliste und fügen die Kinder des Knotens (die selber wiederum Binomiale Heaps B_0 bis B_{i-1} sind) in die Wurzelliste ein. Dazu iterieren wir parallel über die Wurzelliste und die Kinder (ähnlich wie beim Addieren von Binärzahlen). Haben wir einen Heap einer bestimmten Größe, so fügen wir ihn in die Wurzelliste ein. Dabei gehen wir wie folgt vor:

- Haben wir einen Heap einer bestimmten Größe, so fügen wir ihn in die Wurzelliste ein.
- Haben wir zwei Heaps einer bestimmten Größe, so verbinden wir diese.
- Haben wir drei Heaps einer bestimmten Größe, so verbinden wir zwei davon und fügen den dritten in die Wurzelliste ein.

Delete(id): Wieder decreaseKey und extractMin.

3.5.1 Amortisierte Analyse von insert

Das Einfügen der Kinder eines Knotens verlief ähnlich dem Addieren zweier Binärzahlen. So kommt man auf die Idee, dass das Einfügen eines B_0 ähnlich dem Hochzählen eines Binärzählers erfolgt. Wir definieren deshalb

$$\Phi(H) = c_1 \cdot (\text{Anzahl binomialer Heaps})$$

Wobei c_1 die Worst-Case-Laufzeit von addTree ohne rekursive Aufrufe ist.

Wird bei insert(x) addTree k -mal rekursiv aufgerufen, so werden k Heaps gelöscht, und einer kommt hinzu. Daher ist die Amortisierte Laufzeit

$$kc_1 - (k-1)c_1 = c_1 \in O(1)$$

3.6 Fibonacci-Heap

Behauptung: Der maximale Grad eines Knotens im Fibonacci-Heap ist logarithmisch beschränkt.

Wir zeigen: Die maximale Anzahl Knoten in einem Heap B_i mit Wurzelgrad i ist mindestens c^i für eine Konstante $c > 1$. Damit kann es keine Wurzel vom Grad $> \log_c n$ geben.

Offensichtlich gilt $s_0 = 1, s_1 = 2$.

Seien v_1, \dots, v_i die Kinder eines Knotens vom Grad i in der Reihenfolge in der sie Kinder von v wurden. Als v_j zum Kind von v wurde, hatte v mindestens $j-1$ Kinder (nämlich v_1, \dots, v_{j-1} , eventuell zusätzlich noch andere Kinder, die v später

verloren hat). Zu diesem Zeitpunkt hatte v_j also mindestens $j - 1$ Kinder. Also hat v_j jetzt mindestens $j - 2$ Kinder (da er nur ein Kind verlieren kann). Also:

$$s_i \geq \underbrace{1}_v + \underbrace{1}_{v_1} + \underbrace{s_0}_{v_2} + \underbrace{s_1}_{v_3} + \dots + \underbrace{s_{i-2}}_{v_i}$$

Per Induktion kann man leicht zeigen, dass $s_i \geq (\frac{3}{2})^i$.

Analysiert man s_i genauer, so sieht man $s_i = F_{i+2}$, wobei F_i die i -te Fibonacci-Zahl ist.

Amortisierte Analyse:

Benutze Potenzialfunktion:

$$\Phi(H) = c_1(\text{Anzahl Heaps in Wurzelliste}) + (c_1 + c_2)(\text{Anzahl Markierungen})$$

wobei c_2 geeignet gewählt wurde.

Insert verändert die Anzahl der Markierungen nicht \Rightarrow gleiche Analyse wie vorher.

ExtractMin: Es gibt nicht mehr Markierungen nach der Operation (und wie vorher höchstens $\log n$ Heaps) $\Rightarrow O(\log n)$

DecreaseKey: Wenn wir k neue Heaps in die Wurzelliste einfügen, löschen wir $k - 1$ Markierungen und markieren höchstens einen Knoten. \Rightarrow Amortisierte Laufzeit:

$$\underbrace{k(c_2 + c_1)}_{\text{Amortisierte Kosten von AddTree + Zeiger umbiegen}} - (k - 2)(c_2 + 1)$$

4 Datenstrukturen für ungeordnete Mengen

4.1 Hashing

4.1.1 Mittlere Laufzeit von Hashing

Wir machen die (leider meist unrealistische) Annahme, dass jede Teilmenge gleich wahrscheinlich ist. Also ist der Wahrscheinlichkeitsraum

$$W = \{S \subseteq U \mid |S| = n\}$$

$$P_S = \frac{1}{|W|} = \frac{1}{\binom{N}{n}}$$

Für ein festes $x \in U$ ist C_X eine Zufallsvariable. Wir berechnen ihren Erwartungswert:

$$E[C_X] = \frac{1}{|W|} \sum_{\substack{S \subseteq U \\ |S|=n}} C_X(S) = \frac{1}{|W|} \sum_{\substack{S \subseteq U \\ |S|=n}} \sum_{\substack{y \in S \\ h(x)=h(y)}} 1$$

Wir wollen die Summationsreihenfolge vertauschen. Leider hängt die 2. Summe von der 1. ab. Wir nutzen folgenden Trick: Wir definieren

$$i_y(S) = \begin{cases} 1 & \text{falls } y \in S \\ 0 & \text{sonst} \end{cases}$$

Damit ist

$$\sum_{\substack{y \in S \\ h(x)=h(y)}} 1 = \sum_{\substack{y \in U \\ h(x)=h(y)}} i_y(S)$$

Also folgt

$$E[C_X] = \sum_{\substack{y \in U \\ h(x)=h(y)}} \frac{1}{|W|} \underbrace{\sum_{\substack{S \subseteq U \\ |S|=n}} i_y(S)}_{\binom{N-1}{n-1}} = \sum_{\substack{y \in U \\ h(x)=h(y)}} \frac{\binom{N-1}{n-1}}{\binom{N}{n}} = \sum_{\substack{y \in U \\ h(x)=h(y)}} \frac{n}{N} = h^{-1}(h(x)) \frac{n}{N}$$

Definition (Faire Hashfunktion) Eine Hashfunktion $h : U \mapsto \{0, \dots, m-1\}$ heißt *fair*, falls

$$|h^{-1}(i)| \leq \left\lceil \frac{N}{m} \right\rceil \text{ für alle } i, 0 \leq i \leq m-1$$

Satz: Wenn h eine faire Hashfunktion ist, dann gilt für alle $x \in U$:

$$E[C_x] = \frac{n}{m} + \frac{n}{N}$$

Beweis: Nach obiger Rechnung gilt:

$$E[C_X] = \frac{n}{N} h^{-1}(h(x)) \leq \frac{n}{N} \left\lceil \frac{N}{m} \right\rceil \leq \frac{n}{N} \cdot \left(\frac{N}{m} + 1 \right) = \frac{n}{m} + \frac{n}{N}$$

Korollar: Die mittlerer Laufzeit von Hashing mit fairer Hashfunktion pro Operation ist $O(1 + \frac{n}{m})$.

Beweis: Da $N > m$, gilt $E[C_X] \leq \frac{n}{m} + 1$.

4.1.2 Universelles Hashing

Ziel: Entferne die Annahme, dass S gleichverteilt ist. Trick: Algorithmus wählt Hashfunktion zufällig. Definition: Sei $c \in \mathbb{R}$ und $U = \{0, \dots, N-1\}$. Dann heißt eine Menge

$$H \subseteq \{h : U \mapsto \{0, \dots, m-1\}\}$$

c -universell, wenn für alle $x, y \in U$ mit $x \neq y$ gilt

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

Satz: Wenn H c -universell ist, dann gilt:

$$E[C_X] \leq c \frac{n}{m} \text{ für alle } x \in U, \text{ wenn } h \text{ gleichverteilt zufällig gewählt wird}$$

Beweis:

$$E[C_X] = \frac{1}{|H|} \sum_{h \in H} \sum_{\substack{y \in S \\ h(x)=h(y)}} 1$$

Sei

$$\delta_{XY}(h) = \begin{cases} 1 & \text{falls } h(x) = h(y) \\ 0 & \text{sonst} \end{cases}$$

Dann gilt:

$$\begin{aligned} E[C_X] &= \frac{1}{|H|} \sum_{h \in H} \sum_{y \in S} \delta_{XY}(h) = \sum_{y \in S} \frac{|H|}{\delta_{XY}}(h) \\ &\leq 1 + \sum_{\substack{y \in S \\ y \neq x}} \frac{1}{|H|} c \frac{|H|}{m} = 1 + c \frac{n}{m} \end{aligned}$$

Lemma: Sei N eine Primzahl. Für $a \in \{1, \dots, N-1\}$ sei

$$h_a : U \mapsto \{0, \dots, m-1\}, h_a(x) = ((ax) \bmod N) \bmod m$$

Dann ist $H = \{h_a, 1 \leq a, \leq N-1\}$ 2-universell.

Beweis: Seien $x, y \in U$ mit $x \neq y$. Für wie viele $a \in \{1, \dots, N-1\}$ ist $h_a(x) = h_a(y)$?

Wenn $h_a(x) = h_a(y)$, dann $((ax) \bmod N) \bmod m = ((ay) \bmod N) \bmod m$. Also:

$$m \mid \underbrace{(ax) \bmod N - (ay) \bmod N}_{\substack{0, \dots, N-1 \\ 0, \dots, N-1 \\ -N+1, \dots, N-1}}$$

Also ist $(ax) \bmod N - (ay) \bmod N = im$ für i mit $0 \leq |i| \leq \frac{N-1}{m}$. Also:

$$a(x-y) \equiv im \pmod{N}$$

Da N eine Primzahl ist, gibt es für jedes $i \neq 0$ nur eine Lösung von $a(x - y) \equiv im \pmod{N}$. Für $i = 0$ hat die Gleichung keine Lösung. Also:

$$|\{a \mid h_a(x) = h_a(y)\}| \leq 2 \frac{N-1}{m}$$

Da $|H| = N - 1$ ist, gilt

$$|\{h \in H \mid h(x) = h(y)\}| \leq 2 \frac{|H|}{m}$$

Satz: Sei $S \subseteq U$, $|S| = n$ und H eine c -universelle Klasse von Hash-Funktionen. Wenn wir h gleichverteilt zufällig aus H wählen, dann brauchen die Operationen insert, delete, lookup *erwartete* Laufzeit $O(1 + c \frac{n}{m})$.

4.1.3 Idee von Hashing

Wir wollen Elemente eines Universums $U = \{0, \dots, N\}$ speichern, wobei N sehr groß ist. Da N sehr groß ist, ist ein Array nicht möglich.

Idee: Wähle eine Funktion (genannt Hashfunktion) $h : U \mapsto \{0, \dots, m-1\}$ mit $m \approx n$ (Anzahl der zu speichernden Elemente). Speichere $x \in U$ in einem Array an Position $h(x)$. Da nun mehrere Elemente am gleichen Index gespeichert werden müssen, erzeugen wir ein Array von Listen. Eine Hashfunktion ist „gut“, wenn die Elemente S gleichmäßig über das Array verteilt werden. Die Laufzeit zum Suchen von x ist nämlich $O(1 + C_x)$, wobei C_x die Größe der $h(x)$ -ten Liste ist.

Da der Algorithmus die Menge S nicht kennt, kann er keine solche Hashfunktion finden.

2 Ansätze zum Lösen dieses Problem.

1. Wir machen die (leider unrealistische) Annahme, dass jede Teilmenge gleich wahrscheinlich ist. Eine Rechnung zeigt, dass es sinnvoll ist, das Universum gleichmäßig über die Werte $\{0, \dots, m-1\}$ zu verteilen. Die erwartete Größe einer Liste ist höchstens $O(1 + \frac{n}{m})$.
2. Der Algorithmus wählt eine zufällige Hashfunktion. Da wir nur mit Hashfunktionen arbeiten wollen, die wir effizient auswerten und kompakt speichern können, wählen wir nicht eine beliebige Hashfunktion, sondern beschränken uns auf eine (geschickt gewählte) Menge von Funktionen. Eine Menge von Hashfunktionen ist „gut“, wenn die Wahrscheinlichkeit, dass zwei verschiedene Elemente auf den gleichen Hashwert abgebildet werden, „klein“ ist. Wir sagen, eine Menge von Hashfunktionen ist c -universell, wenn diese Wahrscheinlichkeit immer höchstens $\frac{c}{m}$ ist.
Zufallsexperiment: Wähle gleichverteilt zufällig $h \in H$ (Klasse der Hashfunktionen). c -universell heisst: Für $x, y \in U$ mit $x \neq y$ gilt

$$P(h(x) = h(y)) = \frac{1}{|H|} \sum_{h \in H} \delta_{xy}(h) \leq \frac{c}{m}, \text{ wobei } \delta_{xy}(h) = \begin{cases} 1 & \text{falls } h(x) = h(y) \\ 0 & \text{sonst} \end{cases}$$

Eine Rechnung zeigt, dass die erwartete Größe einer Liste, wenn man eine zufällige Hashfunktion einer c -universellen Klasse wählt, beschränkt ist durch $O(1 + \frac{n}{m})$.

Eine 2-universelle Klasse von Hashfunktionen ist

$$H := \{h_a : U \mapsto \{0, \dots, m-1\}, \text{ wobei } h_a(x) = ((ax) \bmod N) \bmod m\}$$

4.1.4 Feldgröße

Frage: Welche Tafelgröße ist ideal: Zugriffszeit $\approx \frac{n}{m}$, also wähle m möglichst groß. Speicherbedarf $\approx n + m$, also wähle m möglichst klein. Kompromiss: Wähle $m \approx n$.

Was ist zu tun, wenn man n nicht kennt?

Lösung: Wähle Tafel der Größe 2^i . Starte mit einem i_0 (z.B. $i_0 = 10$).

„Insert“: Füge Element wie gehabt ein. Wenn jetzt 2^{i+1} Elemente gespeichert sind, gehe zu Tafelgröße 2^{i+1} über. Dazu müssen für alle Elemente neue Hashwerte berechnet werden.

„Delete“: Lösche Element wie gehabt. Wenn jetzt nur noch 2^{i-1} Elemente gespeichert sind (und $i > i_0$), dann halbiere Tafelgröße.

Satz: Wenn die Kosten für das Umspeichern in eine neue Tafel linear in der Anzahl der ungespeicherten Elemente ist, dann haben alle Operationen erwartete amortisierte Laufzeit von $O(1)$.

Beobachtungen:

- Wird die Tafelgröße geändert müssen mindestens 2^i Elemente eingefügt werden, bzw. mindestens 2^{i-1} Elemente gelöscht werden, bis die Tafelgröße erneut geändert wird.
- Die Tafelgröße ist immer zwischen $\frac{n}{2}$ und $2n$, also $\Theta(n)$.

Eine insert- oder delete-Operation ist teuer, wenn die Elemente in eine neue Tafel umgespeichert werden. Wir müssen eine Potenzialfunktion wählen, die einen höheren Wert hat, wenn wir nahe am Rand der tolerierten Tafelgröße sind. Da jede Operation amortisiert erwartet $O(1)$ kosten soll, darf sich das Potenzial pro Operation nie um mehr als eine Konstante erhöhen.

Wähle deshalb

$$\Phi = c \cdot |n - 2^i|, \text{ wobei } 2^i \text{ die aktuelle Tafelgröße ist}$$

Wir nehmen an, dass k Elemente in $c' \cdot k$ Zeit umgespeichert werden können.

Zur Analyse der Kosten unterscheiden wir 2 Fälle.

1. Ohne Umspeichern.

insert, delete und lookup brauchen erwartete Zeit $O(1)$, das Potenzial erhöht sich um höchstens c . Also ist die erwartete amortisierte Laufzeit $O(1) + c \subseteq O(1)$.

2. Mit Umspeichern.

„insert“: Die Anzahl der gespeicherten Elemente erhöht sich also auf 2^{i+1} . Das Potenzial fällt von $c(2^{i+1} - 1 - 2^i)$ auf 0. Also sind die amortisierten Kosten

$$c_i + \Phi_i - \Phi_{i-1} \leq 2^{i+1} \cdot c' - (2^i - 1)c \leq c, \text{ falls } c \geq 2c'$$

„delete“: Die Anzahl der gespeicherten Elemente fällt auf 2^{i-1} . Das Potenzial fällt also von $c(2^i - (2^{i-1} + 1))$ auf 0. Amortisierte Kosten

$$c' \cdot 2^{i-1} - c(2^{i-1} - 1) \leq c \text{ falls } c \geq c'$$

„lookup“: Kein Umspeichern möglich.

Wähle also $c = 2c'$.

4.1.5 Dynamisches Hashing

Tafelgröße 2^i . Starte mit 2^{i_0} .

„insert“: wie gehabt. Wenn 2^{i+1} Elemente gespeichert, verdopple Tafelgröße.

„delete“: wie gehabt. Wenn 2^{i-1} Elemente gespeichert, halbiere Tafelgröße.

Amortisierte Analyse Wähle Potenzialfunktion

$$\Phi = c|n - 2^i|$$

Umspeichern von K Elementen kostet höchstens $c'j$.

1.Fall: Tafelgröße wird nicht geändert. Postenzial ändert sich um höchstens c . Also ist die erwartete Laufzeit von insert, delete und lookup $O(1) + c = O(1)$.

2.Fall: Tafelgröße ändert sich.

„insert“: Die Anzahl der gespeicherten Elemente erhöht sich von $2^{i+1} - 1$ auf 2^{i+1} . Also fällt das Potenzial von $c \cdot |2^{i+1} - 2^i|$ auf $c \cdot |2^{i+1} - 2^{i+1}| = 0$. Die Kosten der Operation sind erwartete Kosten von $O(1)$ plus $c'2^{i+1}$ für das Umspeichern. Die erwarteten amortisierten Kosten sind also

$$O(1) + c'2^{i+1} - c \cdot |2^{i+1} - 1 - 2^i| = O(1) + 2'2^{i+1} - c2^i - c = O(1), \text{ falls } c > 2c'$$

„delete“: Die Anzahl gespeicherter Elemente erniedrigt sich von $2^{i-1} + 1$ auf 2^{i-1} . Also fällt das Potenzial von $c \cdot |2^{i-1} + 1 - 2^i|$ auf 0. Die Kosten für das Umspeichern sind $c'2^{i-1}$. Die erwarteten Kosten sind also

$$O(1)c'2^{i-1} - c(2^{i-1} - 1) = O(1), \text{ falls } c \geq c'$$

Wählen wir $c = 2c'$, so ergeben sich erwartete amortisierte Kosten von $O(1)$ für Operation.

4.1.6 Perfektes Hashing

Fragestellung: Angenommen wir kennen die Menge S der zu speichernden Elemente. Können wir eine Hashfunktion finden, so dass gilt:

- h ist konfliktfrei, d.h. $h(x) \neq h(y)$ für alle $x, y \in S; x \neq y$.
- Die Tafelgröße von H ist $O(n)$.
- Wir können die Hashfunktion mit $O(n)$ Platz im Speicher repräsentieren und in $O(1)$ auswerten.

Eine solche Hashfunktion nennen wir perfekt.

1. Ansatz: Wähle eine zufällige Hashfunktion einer c -universellen Klasse. Ist eine zufällige Hashfunktion konfliktfrei mit Wahrscheinlichkeit von mindestens $\frac{1}{2}$, so können wir durch wiederholtes zufälliges Auswählen eine solche Hashfunktion finden (insbesondere existiert dann eine solche Hashfunktion)

Frage: Wie groß muss man die Tafelgröße wählen, damit eine zufällige Hashfunktion mit Wahrscheinlichkeit von mindestens $\frac{1}{2}$ konfliktfrei ist?

Für feste $x, y \in S$ mit $x \neq y$ wissen wir: $P(h(x) = h(y)) \leq \frac{c}{m}$. Also ist

$$P(\text{Konflikt existiert}) \leq \sum_{\substack{x, y \in S \\ x \neq y}} P(h(x) = h(y)) \leq \binom{n}{2} \frac{c}{m}$$

Wählt man $m = 2c \binom{n}{2}$, ergibt sich $P(\text{Konflikt existiert}) \leq \frac{1}{2}$ und somit $P(\text{Konfliktfrei}) \geq \frac{1}{2}$

Als kann man eine konfliktfreie Hashfunktion mit Tafelgröße $O(n^2)$ finden.

2. Ansatz: Zweistufiges Hashing Suche Hashfunktion h mit Tafelgröße n_i , die S „relativ gut verteilt“. Sei b_i die Anzahl der Elemente von S mit $h(x) = i$.

Für jeden Tafelindex i wähle konfliktfreie Hashfunktion h_i mit Tafelgröße $2c \binom{b_i}{2}$ nach dem 1. Ansatz.

Für jeden Index i reservieren wir ein Array $A[i]$ der Größe $2c \binom{b_i}{2}$. Ein Element x wird in $A[h(x), h_{h(x)}(x)]$ gespeichert.

Beispiel Sei $S = \{3, 17, 63, 92, 113\}$, $N = 127$, $h(x) := (x \bmod N) \bmod 6$. Wähle $h_0 = h_1 = h_2 = h_4$ mit $h_i(x) = 0$. Wähle $h_3(x) = ((3x) \bmod N) \bmod 4$. Wähle $h_5(x) = ((2x) \bmod N) \bmod 4$.

Gesamtplatzbedarf und Laufzeit Platzbedarf für Hashtafeln:

$$O\left(n + 2c \sum_{i=1}^n \binom{b_i}{2}\right)$$

Platzbedarf für Hashfunktionen:

$O(n)$, denn wir speichern $n + 1$ Hasfunktionen

Laufzeit:

„Lookup“ worst-case-Laufzeit von $O(1)$.

Frage: Wie groß ist $\sum_{i=1}^n 2c \binom{b_i}{2}$ im Erwartungswert.

$$\begin{aligned} E\left[\sum_{i=1}^n \binom{b_i}{2}\right] &= \frac{1}{|H|} \sum_{h \in H} \sum_{i=1}^n \binom{\text{Größe von } b_i \text{ für } H}{2} \\ &= \frac{1}{|H|} \sum_{h \in H} \sum_{i=1}^n |\{\{x, y\} \mid x, y \in S, x \neq y, h(x) = h(y) = i\}| \end{aligned}$$

Hier reicht es, den Summanden mit $h(x) = i$ zu betrachten, alle anderen sind 0.

$$\begin{aligned} &= \frac{1}{|H|} \sum_{h \in H} |\{\{x, y\} \mid x, y \in S, x \neq y, h(x) = h(y)\}| \\ &\leq \frac{1}{|H|} \sum_{h \in H} \frac{C(n) |H|}{n} \end{aligned}$$

da man über alle Tupel von Elementen die Indikatorfunktion $h(x) = h(y)$ summiert.

$$\stackrel{c\text{-universell}}{\leq} \frac{1}{|H|} \sum_{\{x, y\} \mid x, y \in S, x \neq y} \sum_{h \in H} \frac{c}{n} = \binom{n}{2} \frac{2}{n} \leq \frac{1}{2} c(n+1)$$

Also ist die Wahrscheinlichkeit, dass für eine zufällige Hashfunktion $\sum \binom{b_i}{2} \geq c(n+1)$ gilt, höchstens $\frac{1}{2}$. (Markov-Ungleichung: $P(X \geq (E[X] \cdot \frac{1}{c})) \leq \frac{1}{c}$)

Zusammenfassung des Algorithmus zum Finden einer Perfekten Hashfunktion:

1. Wähle iteriert eine zufällige Hashfunktion h mit Tafelgröße n , bis $\sum_{i=1}^n \binom{b_i}{2} \leq c(n+1)$.
2. Für $i = 1$ bis n Wähle iteriert eine zufällige Hashfunktion h mit Tafelgröße $2c \binom{b_i}{2}$, bis h_i konfliktfrei.

Laufzeit: 1. Schritt: Jede Iteration braucht $O(n)$. Im Erwartungswert braucht man höchstens 2 Iterationen. 2. Schritt: Für jedes i braucht jede Iteration $O(n^2)$. Im Erwartungswert braucht man jeweils höchstens 2 Iterationen.

Satz: Perfektes Hashing garantiert konstante Zugriffszeit. Eine perfekte Hashfunktion kann in erwarteter Zeit $O(n)$ gefunden werden.

5 Union-Find-Datenstruktur

Sei $U = \{0, \dots, n\}$ Universum. Wir wollen Partitionen $P = \{P_0, \dots, P_{k-1}\}$ von U verwalten, d.h. $\bigcup_{i=0}^{k-1} P_i = U$, $P_i \cap P_j = \emptyset$ falls $i \neq j$. Jedes P_i nennen wir einen Block.

Wir starten mit der Partition $\{\{0\}, \dots, \{n-1\}\}$, also n Blöcke, die jeweils ein Element enthalten und unterstützen die Befehle:

Union(x, y): Vereinigt die Blöcke, die x und y enthalten.

SameBlock(x, y): Liefert „true“, falls x und y im gleichen Block sind.

Anwendung:

- Berechnung minimaler aufspannender Bäume
- Offline-Min-Probleme (Übung)

Wir realisieren dies leicht modifiziert. Wir geben jedem Block einen eindeutigen Namen, der gerade einem Element des Blocks entspricht, und unterstützen die Befehle:

Find(x) liefert den Namen des Blockes, der x enthält.

Union(A, B), wobei A und B Namen von Blöcken sind: Vereinigt die Blöcke A und B .

Die Befehle von oben kann man wie folgt simulieren:

Union(x, y) durch Union(Find(x), Find(y))

SameBlock(x, y) durch Find(x) = Find(y).

Übungsaufgabe: Realisieren sie die Union-Find-Datenstruktur über Listen bzw. Arrays und analysieren Sie die Laufzeit.

5.1 Realisierung durch Bäume

Wir verwalten die Partition als Menge von Bäumen. Jeder Block entspricht einem Baum. Der Name des Blockes ist das Element, das an der Wurzel gespeichert ist. Im Gegensatz zum Suchbaum speichern wir für jeden Knoten nur einen Zeiger, der auf den Vater zeigt.

Find verfolgt den Vater-Zeiger, bis die Wurzel erreicht ist.

Union: Macht die Wurzel des Baumes mit geringerer Tiefe zum Kind der Wurzel es anderen Baumes. Sind beide Bäume gleich tief, wähle einen beliebigen als Kind des anderen.

Beobachtung: Die Tiefe der Bäume kann leicht gespeichert werden: Werden zwei Bäume gleicher Tiefe vereinigt, so erhöht sich die Tiefe um 1, andernfalls ändert sich die Tiefe nicht.

Laufzeiten: Union braucht $O(1)$

Find braucht $O(\text{maximale Tiefe eines Baumes})$.

Frage: Wie tief kann ein Baum werden?

Behauptung: Ein Baum der Tiefe t enthält mindestens 2^t Elemente.

Beweis: Induktiv.

Induktionsanfang: Jeder Baum der Tiefe 0 enthält genau ein Element.

$t - 1 \rightarrow t$: Ein neuer Baum der Tiefe t entsteht nur, wenn wir 2 Bäume der Tiefe $t - 1$ vereinigen. Nach IV enthalten sie jeweils mindestens 2^{t-1} Elemente. Also enthält der neue Baum mindestens $2 \cdot 2^{t-1} = 2^t$ Elemente. Danach kann der Baum nur noch mehr Elemente hinzubekommen.

Satz: Eine Union-Find-Datenstruktur kann so realisiert werden, dass „Find“ eine Laufzeit von $O(\lg n)$ und „Union“ eine Laufzeit von $O(1)$ hat.

5.1.1 Vereinigung nach Rang mit Pfadkomprimierung

Idee von Pfadkomprimierung: Wir merken uns alle Knoten, die bei einer Find-Anweisung gesucht werden, und hängen sie direkt unter die Wurzel. Danach können weitere Find-Anweisungen billiger werden. Problem: Wir können die Tiefe eines Baumes nicht mehr so leicht speichern. Deshalb vereinigen wir nach Rang, wobei der Rang eines Knotens gerade der Tiefe entspricht, die sein Teilbaum hätte, wenn wir keine Pfadkomprimierung machen würden.

Am Anfang hat jeder Knoten Rang 0. Vereinigen wir 2 Wurzeln mit verschiedenem Rang ändern sich die Ränge nicht, andernfalls erhöht sich der Rang der Wurzel der Vereinigung von 1.

Beobachtungen zum Rang eines Knotens:

- Wird in Knoten zum Kind eines anderen, ändert sich sein Rang nicht mehr.
- Der Rang eines Knotens kann nur steigen
- Der Rang eines Knotens ist immer kleiner als der Rang seines Vaters (gilt sowohl bei Union, als auch bei der Pfadkomprimierung)
- Die Anzahl der Elemente mit Rang k ist höchstens $\frac{n}{2^k}$. Erhält ein Element Rang k , enthält der Baum mindestens 2^k Elemente. Diese 2^k Elemente steuern nicht mehr dazu bei, dass ein anderer Knoten Rang k enthält.

5.1.2 Die \log^* -Funktion

$$\log^*(n) := \min\{i \in \mathbb{N} \mid \log^{(i)}(n) \leq 1\}$$

also die Anzahl, wie oft man den Logarithmus ziehen muss, bis man höchstens 1 erhält. Beispiel: $\log^*(2^{2^{16}}) = 5$, denn $\log(2^{2^{16}}) = 2^{16}$, $\log 2^{16} = 16$, $\log 2^{16} = 16$, $\log 16 = 4$, $\log 4 = 2$, $\log 2 = 1$.

Anmerkung : $2^{2^{16}} \approx 2 \cdot 10^{19728}$, die Anzahl der Atome im Universum wird auf höchstens 10^{80} geschätzt.

5.1.3 Amortisierte Analyse

Wir analysieren jetzt die Kosten von m find-Operationen (wobei zwischendurch Union-Operationen ausgeführt werden). Sei X_i die Menge der Knoten, die bei der i -ten Find-Anweisung besucht werden. Die Laufzeit der i -ten Find-Operation ist dann $O(|X_i|)$, die Gesamtlaufzeit $O(\sum_{i=1}^m |X_i|)$. Wir analysieren $\sum_{i=1}^m |X_i|$. Wir teilen die Kosten von X_i auf die Konten K_1, K_2, K_3^i für $0 \leq i \leq \log^*(n)$ und zählen anschließend, wieviele Knoten auf den einzelnen Konten liegen. Betrachte eine Menge X_i .

1. Die Wurzel und das Kind der Wurzel kommen auf das Konto K_1 .
2. Für einen Knoten v in X_i sei $p(v)$ der Vater von v bei dieser find-Operation. Ist $\log^*(rang(v)) = \log^*(rang(p(v)))$, dann kommt v auf das Konto K_2 , andernfalls auf $K_3^{\log^*(rang(v))}$.

Wir analysieren die Anzahl der Knoten auf den einzelnen Konten:

K_1 : Jede find-Anweisung legt höchstens 2 Knoten auf K_1 , also enthält K_1 höchstens $2m$ Knoten.

K_2 : Da die Ränge zur Wurzel monoton wachsen, können bei einer find(v) Anweisung höchstens $\log^*(rang(v)) \leq \log^*(\log(n))$ Knoten auf Konto K_2 gelegt werden, also insgesamt höchstens $m \log^*(\log(n))$.

K_3^i : Jedes Mal, wenn v auf das Konto $K_3^{\log^*(rang(v))}$ gelegt wird, erhöht sich der Rang des Vaters von v . Gilt $\log^*(rang(v)) \neq \log^*(rang(p(v)))$, wird v nie mehr auf Konto $K_3^{\log^*(rang(v))}$ gelegt. Da $\log^*(k) \neq \log^*(2^k)$ gilt, wird v höchstens $2^{rang(v)}$ -mal auf das Konto $K_3^{\log^*(rang(v))}$ gelegt.

Sei k minimal mit $\log^*(k) = i$. Insgesamt liegen auf K_3^i höchstens

$$\underbrace{\frac{n}{2^{k-1}}}_{\geq \text{Anzahl Knoten } v \text{ mit } \log^*(rang(v)) = i} \cdot \underbrace{2^k}_{\geq \text{Wie oft ein Knoten auf das Konto } K_3^i \text{ kommen kann}} = 2n \text{ Knoten}$$

Insgesamt liegen auf den Knoten höchstens

$$2m + m \log^*(n) + \underbrace{(\log^*(\log(n)) + 1)}_{\log^*(n)} \cdot 2n \in O((n+m) \log^*(n)) \text{ Knoten}$$

Satz: Eine Union-Find-Datenstruktur kann so implementiert werden, dass m find-Anweisungen höchstens $O((n+m) \log^*(n))$ Laufzeit und jede Union-Anweisung Laufzeit $O(1)$ hat.

Bemerkungen: Es macht keinen Sinn, Elemente zu speichern, auf die nie zugegriffen wird. Auch in der Analyse kann man diese Elemente „wegdiskutieren“, so dass „m“ find Operationen $O(m \log^*(n))$ kosten.

Man kann genauer analysieren, dass die worst-case-Laufzeit von m find-Operationen $\Theta(m \cdot \alpha(n, m))$ ist, wobei α eine inverse Ackermannfunktion ist, die noch wesentlich langsamer wächst als $\log^*(n)$.

6 Graphenalgorithmen

Definition: Ein gerichteter Graph ist eine endliche Menge V von Knoten und eine endliche Menge $E \subset V \times V$ von Kanten. Wir bezeichnen einen Graphen mit $G = (V, E)$. Eine Kante e von v nach w bezeichnen wir mit $e = (v, w)$. Bei einer Kante $e = (v, w)$ bezeichnet man

- w als Nachbar von v
- w ist adjazent zu v
- v und w sind inzident zu e
- Die Anzahl eingehender Kanten u eines Knotens v nennt man den Eingangsgrad von v .

Beispiele für Graphen:

- Straßenkarten: Knoten \equiv Kreuzungen, Kanten \equiv Straßenstücke zwischen zwei Kreuzungen,

- Internet: Knoten \equiv Computer, Kanten \equiv Verbindungen (Links)
- Webseiten

Oft speichert man mit Knoten oder Kanten noch Zusatzinformationen, also eine Abbildung $c : E \mapsto \mathbb{R}$ (bzw. $V \mapsto \mathbb{R}$), z.B. Länge der Straßenstücke.

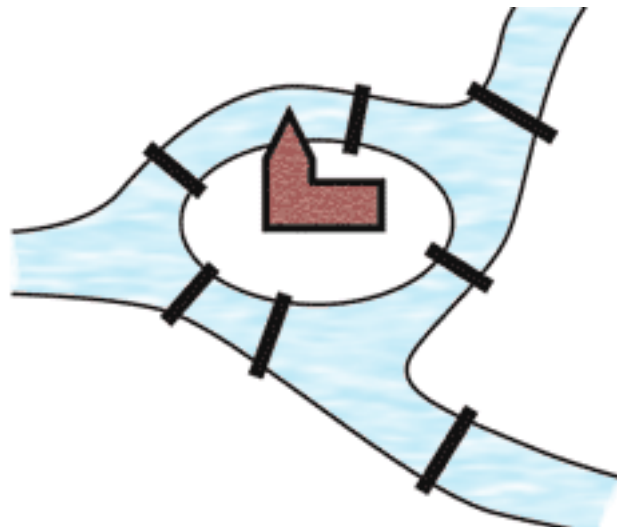
Weitere Graphen:

- bidirektionaler Graph: Gerichteter Graph, bei dem für jede Kante $(v, w) \in E$ die Kante (w, v) existiert.
- Ungerichteter Graph: Wir identifizieren (v, w) und (w, v) im bidirektionalen Graphen.
- Multigraph: Kantenmenge ist eine Multimenge.

Definitionen:

- Pfad: $P := (v_0, e_1, v_1, e_2, \dots, v_{k-1}, e_k, v_k)$, wobei $v_0, \dots, v_k \in V$, $e_1, \dots, e_k \in E$ und $e_i = (v_{i-1}, v_i)$.
Falls G kein Multigraph ist, so schreiben wir auch $P := (v_0, \dots, v_k)$
- Zyklus: Pfad mit $v_0 = v_k$.
- Einfacher Pfad: Pfad mit $v_i \neq v_j$ für alle $i \neq j$.
- Ein ungerichteter Graph heißt zusammenhängend, wenn es für je zwei Knoten v und w einen Pfad zwischen v und w gibt.
- Ein gerichteter Graph heißt azyklisch, wenn er keinen Zyklus enthält.

Beispiel: Königsberger Brückenproblem



Frage: Finde einen Rundweg, der jede Brücke einmal überquert.

Formulierung als Graphproblem: Wir haben einen Knoten für jedes „ohne Brücken erreichbare“ Gebiet. Kanten der Graphen entsprechen den Brücken (Ungerichteter Multigraph). Gesucht: Zyklus, der jede Kante genau einmal enthält. Einen solchen Zyklus nennt man einen Euler-Zyklus oder eine Eulertour.

Satz: Ein ungerichteter Multigraph G enthält eine Eulertour genau dann, wenn G zusammenhängend ist und jeer Knoten einen geraden Eingangsgrad hat.

Beweis: Eulertour \Rightarrow Zusammenhängend: Klar (außer falls isolierte Knoten gibt).

Eulertour \Rightarrow gerader Grad. Besucht der Zyklus einen Knoten v , besuchte er zwei Knoten, die Inzident zu v sind.

Zusammenhängend und gerader Grad \Rightarrow Eulertour. Wir finden einen Zyklus wie folgt:

- Starte bei einem beliebigen Knoten
- Solange möglich, gehe zu einem beliebigen adjazenten Knoten und lösche die Benutzte Kante.

Das Verfahren kann nur im Knoten v enden, da jeder Knoten einen geraden Grad hat. (Wenn ich über eine Kante zu einem Knoten gelange, gibt es noch mindestens eine Kante, die den Knoten verlässt)

Wiederholte Anwendung dieses Verfahrens zwelegt den Graphen in eine Menge von Zyklen. Da der Graph zusammenhängend ist, können diese Zyklen zu einem Zyklus zusammengesetzt werden (Zusammensetzen von 2 Zyklen: Nimm einen Zyklus, bis ein Knoten des anderen erreicht wird, dann komplett diesen Zyklus, dann Rest des 1. Zyklus)

6.1 Darstellung von Graphen im Speicher

Meistens Adjazenzlisten. Wir verwalten ein Array von $|V|$ Listen. Die Liste für Knoten v enthält alle Knoten, die v verlassen.

Alternativen/Erweiterungen

- Verwalte Listen sortiert
- Verwalte zusätzlich Listen von eingehenden Kanten
- Speichere zu jeder Kante (v, w) einen Zeiger auf die Kante (w, v) (bei bidirektionalen Graphen)
- Falls der Graph nicht verändert werden, kann man Arrays statt listen benutzen.

Alternative: Adjazenzmatrix: Speichere $|V| \times |V|$ -Matrix M . Gibt es eine Kante (i, j) , setze $M(i, j) = 1$, andernfalls setze $M(i, j) = 0$.

Vergleich: Adjazenzliste / -matrix. Sei $|V| = n, |E| = m$.

	Liste	Matrix
Iterieren über alle Kanten	$O(\sum_{v \in V} (1 + outdeg(v))) = O(n + m)$	$O(n^2)$
Speicherbedarf	$O(n + m)$	$O(n^2)$
Testen, ob Kante $(i, j) \in E$	$O(1 + outdeg(i)) \subseteq O(n)$, falls kein Multigraph	$O(1)$

6.2 Topologisches Sortieren

Sei $G = (V, E)$ ein gerichteter Graph. Eine bijektive Abbildung $num : V \mapsto \{1, \dots, n\}$ heißt topologische Sortierung, falls $num(i) < num(j)$, falls $(i, j) \in E$.

Satz: Ein gerichteter Graph $G = (V, E)$ besitzt eine topologische Sortierung genau dann, wenn er azyklisch ist.

Beweis: \Rightarrow : Sei num eine topologische Sortierung. Annahme G enthält einen Zyklus $(v_0, \dots, v_k = v_0)$. Dann gilt $num(v_0) < num(v_1) < \dots < num(v_k) = num(v_0)$ - Widerspruch.

\Leftarrow : Hilfsbehauptung: Ein azyklischer Graph enthält einen Knoten mit Eingangsgrad 0. Beweis: Angenommen jeder Knoten hat eingehende Kante $(p(v), v)$. Konstruiere $n + 1$ Knoten durch

$$p^i(v) = \begin{cases} v & \text{für } i = 0 \\ p(p^{i-1}(v)) & \text{sonst} \end{cases}$$

Da G nur n Knoten hat, gilt $i < j$ mit $p^i(v) = p^j(v)$. Dann ist $(p^j(v), p^{j-1}(v))$ ein Zyklus.

Mit der Hilfsbehauptung ergibt sich folgendes Verfahren zur topologischen Sortierung:

- Finde Knoten mit Eingangsgrad 0
- Nummeriere ihn, lösche ihn und die ausgehenden Kanten
- Nummeriere den Restgraphen

```

1: procedure TOPSORT(Graph  $G$ )
2:    $A \leftarrow$  Feld der Größe  $n$ , initialisiert mit Nullen
3:   for all  $v \in V$  do
4:     for all  $(v, w) \in Adj[v]$  do
5:        $A[w] \leftarrow A[w] + 1$ 
6:    $ZERO \leftarrow \emptyset$ 
7:   for all  $v \in V$  do
8:     if  $A[v] = 0$  then
9:        $ZERO \leftarrow ZERO \cup \{v\}$ 
10:  while  $ZERO \neq \emptyset$  do
11:    Wähle und entferne einen Knoten  $v \in ZERO$ 
12:     $A[v] = i$ 
13:     $i \leftarrow i + 1$ 
14:    for all  $(v, w) \in Adj[v]$  do
15:       $A[w] \leftarrow A[w] - 1$ 
16:      if  $A[w] = 0$  then
17:         $ZERO \leftarrow ZERO \cup \{w\}$ 
18:  if  $i \leq |V|$  then
19:    ERROR(Es existiert ein Zyklus)

```

▷ Dies sind die zu löschenden Kanten
▷ Aktualisiere Eingangsgrad von w

Laufzeit: Zeile 2: $O(n)$, Zeile 2-5: $O(n + m)$, Zeile 6: $O(1)$, Zeile 7-9: $O(n)$, Zeile 10-17: Jeder Knoten v kommt höchstens einmal in die Queue, der Aufwand ist dann $O(1 + outdeg(v))$, insgesamt also $O(n + m)$, Zeile 18-19: $O(1)$.

Insgesamt hat der Algorithmus also eine Laufzeit von $O(n + m)$.

6.3 Breitensuche (Bredth First Search)

Gegeben ein gerichteter Graph $G = (V, E)$ und sien Startknoten $s \in V$. Berechne den kürzesten Pfad (in der Anzahl der Kanten) von s zu jedem Knoten $v \in V$ (∞ , falls es keinen Pfad gibt).

Kürzester Teilpfad-Regel: Wenn $P = (s = v_0, v_1, \dots, v_k)$ ein kürzester Pfad von s zu v_k ist, dann ist $(s = v_0, \dots, v_{k-1})$ ein kürzester Pfad von s zu v_{k-1} .

Beweis: Angenommen $(s = w_0, w_1, \dots, w_i = v_{k-1})$ ist kürzer als $(s = v_0, \dots, v_{k-1})$, dann ist $(s = w_0, \dots, w_i = v_{k-1}, v_k)$ kürzer als P - Widerspruch.

Daher können wir kürzeste Pfade von s zu allen Knoten $v \in V$ als Baum repräsentieren. Jeder Knoten v kennt den Vorgänger $p(v)$ auf einem kürzesten Pfad.

Ein kürzester Pfad kann damit wie folgt gefunden werden:

- Starte bei v .
- Gehe so lange zum Vorgängerknoten bis s erreicht ist.
- Die besuchten Knoten in umgekehrter Reihenfolge bilden einen kürzesten Pfad.

BFS-Algorithmus: s ist der einzige Knoten mit Abstand 0 zu s . Die Nachbarn von s sind die Knoten mit Abstand 1. Die Nachbarn der Nachbarn haben dann höchstens Abstand 2 usw.

```

1: procedure BFS(Knoten  $s$ )
2:    $parent[1, \dots, n] \leftarrow (nil, \dots, nil)$ 
3:    $depth[1, \dots, n] \leftarrow (\infty, \dots, \infty)$ 
4:    $parent[s] \leftarrow s$ 
5:    $depth[s] \leftarrow 0$ 
6:   Queue  $q \leftarrow \langle s \rangle$ 
7:   while  $q \neq \emptyset$  do
8:      $v \leftarrow q.popFront$ 
9:     for all  $(v, w) \in E$  do
10:      if  $parent[w] = nil$  then
11:         $parent[w] \leftarrow v$ 
12:         $depth[w] \leftarrow depth[v] + 1$ 
13:         $q.pushBack(w)$ 

```

Laufzeitanalyse: Jeder Knoten wird höchstens einmal in die Queue eingefügt (da dann der parent-Zeiger gesetzt wird), und somit höchstens einmal aus der Queue gelöscht. Der Aufwand für Knoten v ist dann $O(1 + outdeg(v))$. Insgesamt also

$$O\left(\sum_{v \in V} (1 + outdeg(v))\right) = O(n + m)$$

Korrektheit: Entfernen wir einen Knoten mit Tiefe k aus der Queue, fügen wir nur Knoten mit Tiefe $k + 1$ in die Queue ein. Die Queue enthält also nur Knoten mit Tiefe K , gefolgt von Knoten der Tiefe $k + 1$ (einfacher Induktionsbeweis). Offenstichtlich weisen wir nie einem Knoten eine zu geringe Tiefe zu (da es immer einen Pfad zu s der entsprechenden Kantenzahl gibt).

Behauptung: Wurde der letzte Knoten mit Tiefe k bearbeitet, sind alle Knoten mit Tiefe $k + 1$ in der Queue.

Beweis (Induktiv): Sei v ein Knoten mit Tiefe $k + 1$ und $P = (s, \dots, v', v)$ ein beliebiger kürzester Pfad von s nach v .

Die kürzeste-Teilpfad-Regel besagt, dass (s, \dots, v') ein kürzester Pfad von s nach v ist. Nach IV war der Knoten v' (der Tiefe k) hat, in der Queue. Als dieser bearbeitet wurde war entweder v bereits in der Queue oder v wurde dann eingefügt (da es die Kante (v', v) gibt). Also ist v in der Queue.

6.4 Tiefensuche (depth first search, DFS)

Verwende einen Stack oder Rekursion statt einer Queue. Also:

```

1: procedure DFS-VISIT( $v$ )
2:    $visited[v] \leftarrow true$ 
3:   for all  $(v, w) \in E$  do
4:     if  $visited[w] = false$  then DFS-VISIT( $w$ )

```

DFS hat viele schöne Eigenschaften, wenn DFS-Visit wie folgt initialisiert wird:

```

1: procedure DFS
2:    $visited[1, \dots, n] \leftarrow (false, \dots, false)$ 
3:   for all  $v \in V$  do
4:     if  $visited[v] = false$  then DFS-VISIT( $v$ )

```

Wir betrachten also nicht einen vorgegebenen Startknoten sondern starten DFS an mehreren Knoten, bis alle Knoten besucht wurden.

Laufzeitanalyse: $O(n + m)$, da es genau einen Aufruf von DFS-Visit pro Knoten gibt, DFS inspiziert jede Kante genau einmal. Wir unterteilen die Kanten in 4 Gruppen:

- Baumkanten T : Wenn $(v, w) \in E$ inspiziert wird, wurde w noch nicht besucht. Also gibt es dann einen rekursiven Aufruf.
- Vorwärtskanten F : Wenn $(v, w) \in E$ inspiziert wird, gibt es einen Pfad aus Baumkanten von v nach w .
- Rückwärtskanten B : Wenn $(v, w) \in E$ inspiziert wird, gibt es einen Pfad aus Baumkanten von w nach v .
- Querkanten C : Wenn $(v, w) \in E$ inspiziert wird, wurde w bereits besucht, es gibt aber weder einen Pfad aus Baumkanten von v nach w noch von w nach v , also der Rest.

Die Kanten T entsprechen gerade den rekursiven Aufrufen. Offensichtlich ist (T, F, B, C) eine Partition von E . Diese wollen wir nun berechnen. Dazu speichern wir die Reihenfolge, in der wir die Knoten besuchen und verlassen und speichern, welche Kanten Baumkanten sind.

procedure DFS

$DFSStart[1, \dots, n] \leftarrow DFSEnd[1, \dots, n] \leftarrow (\infty, \dots, \infty)$

$counter \leftarrow 0$

for all $v \in V$ **do**

if $DFSStart[v] = \infty$ **then**

DFS-VISIT(v)

procedure DFS-VISIT(v)

$DFSStart[v] \leftarrow counter$

$counter \leftarrow counter + 1$

for all $(v, w) \in E$ **do**

if $DFSStart[w] = \infty$ **then**

DFS-VISIT(w)

$DFSEnd[v] \leftarrow counter$

Satz: (Charakterisierung der Kanten mittels $DFSStart$ und $DFSEnd$)

- 1) $(v, w) \in T \cup F \Leftrightarrow DFSStart[v] \leq DFSStart[w]$
- 2) $(v, w) \in B \Leftrightarrow DFSStart[v] > DFSStart[w] \wedge DFSEnd[v] < DFSEnd[w]$
- 3) $(v, w) \in C \Leftrightarrow DFSStart[v] > DFSStart[w] \wedge DFSEnd[v] > DFSEnd[w]$

Beweis: 1. „ \Rightarrow “: Ist $(v, w) \in F$, dann gibt es einen Pfad von Baumkanten von v nach w . Für Baumkanten (x, y) gilt offensichtlich $DFSStart[x] < DFSStart[y]$. Für einen Pfad aus Baumkanten kann man induktiv argumentieren.

„ \Leftarrow “: Da v vor w angefangen wurde und es die Kante (v, w) gibt, muss w angefangen werden, bevor v beendet wird. Wenn wir mit w anfangen, liegt also v noch auf dem Rekursionsstack, d.h. es gibt einen Pfad aus Baumkanten von v nach w .

2. Aus 1. folgt $(v, w) \in B \cup C \Leftrightarrow DFSStart[v] > DFSStart[w]$.

„ \Rightarrow “: Ist $(v, w) \in B$, dann gibt es einen Pfad aus Baumkanten von w nach v . Also liegt w auf dem Stapel, wenn v bearbeitet wird. Also wird v vom Stapel gelöscht, bevor w gelöscht wird.

„ \Leftarrow “: Ist $DFSEnd[v] < DFSEnd[w]$, wird v bearbeitet, während w auf dem Stapel liegt. Also liegt w auf dem Stapel, wenn v auf den Stapel kommt. Somit gibt es einen Pfad aus Baumkanten von w nach v .

3. Folgt aus 1. und 2.

Beobachtungen

1. Sei $I(v) = \{DFSStart[v], \dots, DFSEnd[v]\}$. Für zwei Knoten $v, w \in V$ gilt entweder

- (1) $I(v) \subseteq I(w)$,
- (2) $I(w) \subseteq I(v)$ oder
- (3) $I(v) \cap I(w) = \emptyset$.

2. Bei ungerichteten Graphen inspizieren wir jede Kante genau zweimal, einmal aus jeder Richtung. Also ordnen wir jeder Kante ein Tupel aus $(T, F, B, C) \times (T, F, B, C)$ zu. Das erste Element gibt den Typ der Kante beim ersten Besuch an, das zweite Element den Typ beim zweiten Besuch. Von diesen Tupeln kommen aber bei einem DFS-Aufruf nur zwei Arten vor, nämlich (T, B) und (B, F) .

6.5 Anwendungen von DFS (und BFS)

- Zusammenhangskomponenten (ZHK) für ungerichtete Graphen $G = (V, E)$. ZHK sind maximale Teilmengen $U \subseteq V$, so dass es für je zwei Knoten $v, w \in U$ einen Pfad zwischen v und w gibt.

ZHK bilden eine Äquivalenzrelation, d.h. wir definieren $v \sim w \Leftrightarrow v$ und w liegen in einer ZHK und zeigen das für $u, v, w \in V$ gilt:

- $v \sim v$
- $v \sim w \Leftrightarrow w \sim v$
- $u \sim v \wedge v \sim w \Rightarrow u \sim w$

Berechnung mittels DFS: Ein Aufruf von DFS-Visit aus DFS heraus besucht alle Knoten einer ZHK. (Wir zeigen dies später etwas allgemeiner)

Im Prinzip auch mit BFS möglich, wenn wir BFS an mehreren Knoten starten.

- Ein ungerichteter Graph $G = (V, E)$ heißt bipartite genau dann, wenn wir eine Partition $A \dot{\cup} B$ von V finden können, so dass alle Kanten einen Endpunkt in A haben und einen in B .
- Topologisches Sortieren: Seien v_1, \dots, v_n die Knoten in der Reihenfolge fallender $DFSEnd$ -Nummern für einen beliebigen DFS-Aufruf. Dann ist entweder $num : V \mapsto \mathbb{N}, num(v_i) = i$ eine topologische Sortierung, oder es gibt keine. Korrektheitsbeweis: Sei $(v_i, v_j) \in E$. Wir zeigen $i < j$ oder es gibt einen Zyklus.

- Ist $(v_i, v_j) \in T \cup F$. Also ist v_i auf dem Rekursionsstapel, wenn v_j bearbeitet wird, d.h. $DFSEnd[v_j] < DFSEnd[v_i]$. Da wir nach fallender $DFSEnd$ -Nummer sortiert haben $i < j$.
- Ist $(v_i, v_j) \in B$: Also gibt es einen Pfad aus Baumkanten von v_j nach v_i . Zusammen mit der Kante (v_i, v_j) bildet dieser einen Zyklus.
- Ist $(v_i, v_j) \in C$: Dann ist $DFSEnd[v_i] > DFSEnd[v_j]$. Somit $i < j$.

- Starke Zusammenhangskomponenten (strongly connected components, SCC) für gerichtete Graphen. Eine SCC eines gerichteten Graphen ist eine maximale Teilmenge $U \subseteq V$, so dass für je zwei Knoten $v, w \in U$ sowohl einen Pfad von v nach w als auch einen Pfad von w nach v gibt.

Beobachtungen:

- 1) Für ungerichtete Graphen entsprechen die SCC (des Graphen der alle Kanten des Ungerichteten Graphen in den beiden Richtungen enthält) gerade den ZHK.
- 2) Die SCC bilden eine Äquivalenzrelation:
 - Reflexiv \checkmark , symmetrisch \checkmark .
 - Transitiv: $u \sim v$ und $v \sim w \Rightarrow$ Es gibt Pfad von u nach v und von v nach w . Also gibt es einen Pfad von u nach w . Analog zeigt man, dass es einen Pfad von w nach u gibt.
- 3) Seien C_1, \dots, C_k die SCC von G . Der Graph $G' = (\{1, \dots, k\}, E')$, wobei $E' = \{(i, j) \mid \exists v \in C_i, w \in C_j, (v, w) \in E\}$ ist azyklisch.

6.5.1 Algorithmus zum Berechnen der SCC

1. Variante (sehr elegant) Für einen Graphen $G = (V, E)$ sei $\overleftarrow{G} = (V, E')$ mit $E' = \{(v, w) \mid (w, v) \in E\}$, also erhält man \overleftarrow{G} aus G indem man alle Kanten umdreht.

DFS(\overleftarrow{G}), seien \overleftarrow{DFSEnd} die $DFSEnd$ -Nummern dieses DFS-Aufrufes

DFS(G), wobei die for all $v \in V$ -Schleife von DFS in der Reihenfolge abnehmender \overleftarrow{DFSEnd} -Nummern erfolgt.

Die SCC sind die Mengen von Knoten, die bei einem DFS-Visit, das aus DFS heraus aufgerufen wird, gefunden werden.

Laufzeit: $2O(n+m)$ für DFS-Aufrufe, $O(n+m)$ um \overleftarrow{G} zu konstruieren, $O(n)$ um die Kanten nach \overleftarrow{DFSEnd} zu sortieren. Also insgesamt $O(n+m)$.

Korrektheit:

Behauptung 1: Es gibt einen Pfad aus Baumkanten von v nach w genau dann, wenn $DFSEnd[v] < DFSEnd[w] < DFSEnd[v]$. Beweis: Übung.

Behauptung 2 (Lemma vom weißen Pfad): Gibt es unmittelbar, bevor der Knoten v besucht wird, einen Pfad von v nach w , der nur nicht besuchte Knoten benutzt, dann wird w gefunden, während v bearbeitet wird.

Beweis: Sei $P = (v = v_0, \dots, v_k = w)$ der Pfad, der nur aus nicht besuchten Knoten besteht. Angenommen w wird nicht besucht während v aktiv ist. Dann gibt es einen Knoten v_i der nicht besucht wird, aber v_{i-1} wurde besucht. Da v_i besucht wird, wird die Kante (v_{i-1}, v_i) betrachtet. Dann wird aber v_i besucht, da es vorher noch nicht besucht wurde.

Behauptung 3: Sind v, w in der selben SCC, dann werden v und w beim gleichen Aufruf gefunden.

Beweis: Sei r der Knoten dieser SCC, der als erstes besucht wird. Unmittelbar, bevor dieser Knoten besucht wird, gibt es also einen Pfad aus unbesuchten Knoten von r nach v und auch von v nach w . Also kommen v und w in die gleiche SCC wie r .

Bemerkung: Liegen v und w in der gleichen SCC und liegt u auf einem Pfad von v nach w , dann liegt u auch in dieser SCC.

Behauptung 4: Alle Knoten $U \subseteq V$, die bei einem Aufruf von DFS-Visit gefunden werden, liegen in einer SCC.

Beweis: Sei r der erste besuchte Knoten von U . Aus der Wahl der Reihenfolge für die forall $v \in V$ -Schleife folgt, dass r die größte \overleftarrow{DFSEnd} -Nummer von allen Knoten, die zu diesem Zeitpunkt noch nicht besucht sind, hat. Insbesondere also die größte \overleftarrow{DFSEnd} -Nummer aller Knoten in U . Offensichtlich gibt es einen Pfad von r nach v für alle $v \in U$. Wir müssen zeigen, dass es einen Pfad von v nach r gibt. Sei w der Knoten aus U , der vom ersten DFS-Aufruf als erstes gefunden wurde. Beim ersten DFS-Aufruf gab es dann einen Pfad von w nach r (da es in G einen Pfad von r nach w gibt). Daraus folgt, dass entweder $w = r$ oder r wird während der Bearbeitung von w gefunden (Behauptung 2). Das zweite ist aber nicht möglich, da dann $\overleftarrow{DFSEnd}[w] > \overleftarrow{DFSEnd}[v]$ wäre. Also ist $w = r$. Also hat r die kleinste $\overleftarrow{DFSStart}$ -Nummer und die größte \overleftarrow{DFSEnd} -Nummer. Nach Behauptung 1 werden also alle Knoten in U während der Bearbeitung von r gefunden (beim ersten DFS-Aufruf). Also gibt es \overleftarrow{G} einen Pfad von v nach r für alle $v \in V$. Damit gibt es in G einen Pfad von v nach r für alle $v \in V$.

2.Version (Fleiß-Version) Idee: Wir speichern immer die SCC des bisher inspizierten Teilgraphen.

- Eine Baumkante (v, w) erzeugt eine neue SCC $\{w\}$ (da wir noch keinen Pfad von w zu einem anderen Knoten kennen).
- Eine Vorwärtskante (v, w) ändert nichts (Wir kennen schon einen Pfad von v nach w).
- Bei einer Rückwärtskante (v, w) vereinigen wir alle SCC, für die ein Knoten auf dem Pfad aus Baumkanten von w nach v liegt (wir kennen während der Bearbeitung von v nur eingehende Kanten zu v die entweder Baumkante oder Rückwärtskanten sind).
- Bei einer Querkante müssen wir zwei Fälle unterscheiden.
 1. w liegt in einer SCC mit einem Vorgänger von v . Dann gibt es einen Pfad von w nach v . Wir vereinigen deshalb alle SCC mit einem Knoten auf einem solchen Pfad.
 2. w liegt nicht in einer SCC mit einem Vorgänger von v . Dann kennen wir keinen Pfad von w nach v .

Beobachtung: Wird der DFS-Visit-Aufruf an einem Knoten v beendet, und liegt v nicht in einer SCC mit einem Vorgänger von v , dann ändert sich die SCC von v nicht mehr. Die Knoten der SCC von v sind dann alle bis dahin erreichten Knoten, denen wir noch keine endgültige SCC zugewiesen haben.

Um mit DFS die SCC zu berechnen, speichern wir zusätzlich die folgenden Informationen:

- Einen Stapel *unfinished* aller Knoten, die bisher besucht wurden, denen aber noch keine endgültige SCC zugewiesen wurde (aufsteigend nach $\overleftarrow{DFSStart}$ -Nummer)
- Einen Stapel *roots* mit den Knoten des Rekursionsstacks, die noch nicht mit einem Vorgänger in einer SCC liegen (d.h. von jeder aktuellen SCC haben wir den zuerst gefundenen Knoten auf diesem Stapel).
- Ein Array *compnum*, das die Nummer der Komponente von v enthalten sollm initialisiert mit $(-1, \dots, -1)$.
- Einen Komponentenzähler *compcounter*, initialisiert mit 0.

procedure DFSVISIT(v)

unfinished.push(v)

roots.push(v)

$\overleftarrow{DFSStart}[v] \leftarrow \text{counter}$

$\text{counter} \leftarrow \text{counter} + 1$

for all $(v, w) \in E$ **do**

if $\overleftarrow{DFSStart}[w] = \infty$ **then**

DFSVISIT(w)

▷ Baumkante

else

if $\text{compnum}[w] = -1$ **then** ▷ Rückwärtskante oder Querkante, und w noch nicht in endgültiger SCC

```

while  $DFSSstart[roots.peek()] > DFSSstart[w]$  do
     $roots.pop()$ 
if  $v = roots.peek()$  then
     $\triangleright$  Wurde  $v$  noch nicht mit einem Vorgänger in eine SCC aufgenommen
     $\triangleright$ , so liegt  $v$  noch auf dem  $roots$ -Stapel
    repeat
         $w = unfinished.pop()$ 
         $compnum[w] = compcounter$ 
    until  $w = v$ 

```

6.6 Artikulationspunkte

Definition: Sei $U \subseteq V$. Der von U induzierte Teilgraph des Graphen $G = (V, E)$ ist der Graph $G' = (U, E')$, wobei $E' := \{(v, w) \in E \mid v, w \in U\}$.

Definition: Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph. Ein Knoten $v \in V$ heißt Artikulationspunkt, wenn der von $V \setminus \{v\}$ induzierte Graph nicht zusammenhängt.

Berechnung der Artikulationspunkte:

1. Variante:

```

for all  $v \in V$  do
    Konstruiere den von  $V \setminus \{v\}$  induzierten Teilgraphen.
    Überprüfe, ob dieser Graph zusammenhängt ist mittels DFS

```

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

2. Variante mittels einem DFS-Aufruf.

Beobachtungen:

- Da G ungerichtet und zusammenhängend ist, bildet T einen gerichteten Baum
- Die Wurzel des Baumes ist Artikulationspunkt genau dann, wenn zwei oder mehr Kanten von T adjazent zu r sind. Ein Blatt von T ist niemals Artikulationspunkt. Entfernt man die zum Blatt adjazente Kante aus T , so bilden die restlichen Kanten einen zusammenhängenden Teilgraphen des von $V \setminus \{Blatt\}$ induzierten Teilgraphen.
- Ein sonstiger Knoten v ist Artikulationspunkt genau dann, wenn es einen Teilbaum unterhalb von v gibt, der keine Rückwärtskante zu einem Vorgängerknoten von v hat.

„ \Rightarrow “: Sei T' ein solcher Teilbaum. Entfernt man v , so gibt es keine Kante die T' verlässt. (Keine Rückwärtskante nach Annahme, Querkanten existieren nicht im ungerichteten Graphen)

„ \Leftarrow “: Entferne v und die adjazenten Kanten aus T und sei w ein Knoten, der nicht von r aus erreichbar ist. Dann gibt es keine Kante, die die THK von w verlässt und nicht adjazent zu v ist. Insbesondere gibt es keine Rückwärtskante wie oben beschrieben.

Algorithmisch: Während des DFS-Aufrufes berechnen wir für jeden Knoten v den Knoten $H[v]$ mit der kleinsten DFS-Startnummer, den man von einem Nachkommen von v über eine Rückwärtskante erreichen kann.

Ein Knoten v ist Artikulationspunkt genau dann, wenn kein Kind von v einen $H[v]$ -Knoten hat, so dass $DFSSstart[H[w]] \geq DFSSstart[v]$.

6.7 Kürzeste Pfade

Definition: Sei $G = (V, E)$ ein gerichteter Graph und $c : E \mapsto \mathbb{R}$ eine Kosten-/Längen-/Gewichtsfunktion. Das Kosten-/Länge/Gewicht eines Pfades $P = (v_0, \dots, v_k)$ ist definiert als:

$$c(P) := \sum_{i=1}^k c((v_{i-1}, v_i))$$

Für $c((v, w))$ schreiben wir kurz $c(v, w)$. Die Distanz $\delta(v, w)$ von Knoten v zu Knoten w wird definiert durch:

$$\delta(v, w) = \begin{cases} \min\{c(P) \mid P \text{ ist Pfad von } v \text{ nach } w\} & \text{falls ein solcher Pfad existiert} \\ \infty & \text{sonst} \end{cases}$$

Wichtig: Da $c((v_i, v_{i+1})) < 0$ sein kann, kann $\delta(v, w) = -\infty$ sein.

Wir unterscheiden die folgenden Aufgaben:

- Single-pair shortest path: Gegeben $v, w \in V$, finde kürzesten Pfad von v nach w .
- Single-source shortest path: Gegeben $v \in V$, finde kürzesten Pfad von v zu allen $w \in V$.

(3) All-pairs shortest path: Finde kürzesten Pfad für je zwei Knoten $v, w \in V$.

Offensichtlich sollte die Laufzeit für Aufgabe (1) nicht größer sein als die für Aufgabe (2), und diese nicht größer als die für Aufgabe (3). Es gibt keinen Algorithmus, der Aufgabe 1 asymptotisch schneller löst als Aufgabe 2.

Beobachtungen:

1. Wenn $P = (v_1, \dots, v_k)$ ein kürzester Pfad von v_1 nach v_k ist, dann ist (v_i, \dots, v_j) ein kürzester Pfad von v_i nach v_j für $1 \leq i < j \leq k$.
2. Für alle Kanten $(v, w) \in E$ gilt $\delta(s, w) \leq \delta(s, v) + c(v, w)$.
3. Für alle Kanten $(v, w) \in E$, die auf einem kürzesten Pfad von s nach w liegen, gilt $\delta(s, w) = \delta(s, v) + c(v, w)$.

6.7.1 Das Single-Source shortest path Problem

Wie bei BFS wollen wir einen kürzesten-Pfade-Baum speichern, d.h. jeder Knoten $v \in V$ kennt den Vorgänger $p(v)$ eines kürzesten Pfades von s nach v . Im Fall, dass es einen negativen Zyklus gibt, genügt es uns, wenn der Algorithmus dies erkennt.

Idee: Wir berechnen temporäre Distanzwerte $d[v]$ für alle Knoten $v \in V$ des bisher kürzesten gefundenen Pfades und speichern den Vorgänger auf diesem Pfad. Wir terminieren, wenn wir wissen, dass wir die kürzesten Pfade gefunden haben.

Initialisierung: $d[s] = 0$; $d[v] = \infty \forall v \in V \setminus \{s\}$; $p[v] = \text{nil} \forall v \in V$.

Die Aktualisierung der Distanzwerte erfolgt über

```

procedure RELAX( $v, w$ )
  if  $d[w] > d[v] + c(v, w)$  then
     $d[w] \leftarrow d[v] + c(v, w)$ 
     $p[w] = v$ 

```

Satz: Folgende Invariante (die offensichtlich nach der Initialisierung gilt), wird von Relax aufrecht erhalten:

$$d[v] \geq \delta(s, v) \quad \forall v \in V$$

Satz: Sei $(v_i, w_i)_{i=1, \dots, N}$ die Folge von Relax-Operationen, die der Algorithmus A ausführt. Sei $P = (s = u_0, \dots, u_k)$ ein kürzester Pfad von s nach u_k . Gibt es eine Teilfolge j_1, \dots, j_k von $1, \dots, N$, so dass $(v_{j_i}, w_{j_i}) = (u_{i-1}, u_i)$ ist, dann ist nach der Ausführung von A $d[u_k] = \delta(s, u_k)$.

In Worten: Relaxiert der Algorithmus neben anderen Kanten alle Kanten des kürzesten Pfades in der richtigen Reihenfolge, so ist der Distanzwert am Ende des Algorithmus korrekt bestimmt.

Beweis: Induktion über k . Ist der Distanzwert $\delta(s, s) = -\infty$, so gibt es keinen kürzesten Pfad und wir haben nichts zu zeigen. Andernfalls gilt $d[s] = \delta(s, s) = 0$.

Gilt zu einem Zeitpunkt $\delta(s, u_{i-1}) = d[u_{i-1}]$ und wird die Kante (u_{i-1}, u_i) relaxiert, so gilt danach:

$$\delta(s, u_i) \leq d[u_i] \leq d[u_{i-1}] + c(u_{i-1}, u_i) \stackrel{IV}{=} \delta(s, u_{i-1}) + c(u_{i-1}, u_i) \stackrel{\text{Beh.1+3}}{\leq} \delta(s, u_i)$$

Da es immer einen kürzesten Pfad von s nach v gibt, der keinen Zyklus enthält (ist das Gewicht eines Zyklus nicht negativ, so kann der Zyklus gestrichen werden, ohne die Kosten des Pfades zu erhöhen), können wir die kürzesten Pfade wie folgt berechnen (jeder Pfad hat damit höchstens $|V| - 1$ Kanten):

```

procedure BELLMAN-FORD( $G, s$ )
  Initialisiere  $d[]$  und  $p[]$ .
  for  $i$  from 1 to  $|V| - 1$  do
    for all  $(v, w) \in E$  do
      RELAX( $v, w$ )

```

Korrektheit: Direkt aus obigem Satz.

Laufzeit: $O(|V| \cdot |E|)$

Wie erkennt man nun die Existenz eines negativen Zyklus? Wenn es keinen negativen Zyklus gibt, so sind nach Ausführung des Algorithmus alle Distanzwerte korrekt bestimmt, d.h. es gilt $d[w] \leq d[v] + c(v, w)$ für alle $(v, w) \in E$. Gibt es einen negativen Zyklus v_0, \dots, v_k , so gilt $d[v_0] > d[v_0] + c(v_k, v_0)$ für mindestens eine Kante dieses Zyklus. Andernfalls gilt

$$d[v_0] = d[v_k] \leq d[v_{k-1}] + c(v_{k-1}, v_k) \leq \dots \leq d[v_0] + \sum_i c(v_{i-1}, v_i) \Rightarrow \sum_i c(v_{i-1}, v_i) \geq 0$$

Wenn es einen kürzesten Pfad gibt, so gibt es einen kürzesten Pfad, der keinen Zyklus enthält.

```

procedure BELLMAN-FORD( $G, s$ )
  Initialisiere  $d[]$  und  $p[]$ .
  for  $i$  from 1 to  $|V| - 1$  do

```

```

for all  $(v, w) \in E$  do
  RELAX( $v, w$ )
for all  $(v, w) \in E$  do
  if  $d[w] > d[v] + c(v, w)$  then
    return „Negativer Zyklus“

```

Korrektheit:

- Gibt es keinen negativen Zyklus, so haben wir vor dem „Negativen Zyklus Test“ die korrekten Distanzen in $d[]$ gespeichert. Insbesondere kann es keine Kante geben, über die wir einen Distanzwert verkürzen können. Also gibt es keine Kante $(v, w) \in E$ mit $d[w] > d[v] + c(v, w)$.
- Gibt es einen negativen Zyklus $P = (v_0, \dots, v_k = v_0)$, dann gibt es mindestens eine Kante (v, w) mit $d[v] + c(v, w) < d[w]$, da ansonsten gilt:

$$d[v_0] \leq d[v_{k-1}] + c(v_{k-1}, v_k) \leq d[v_{k-2}] + c(v_{k-2}, v_{k-1}) + c(v_{k-1}, v_k) \leq \dots \leq d[v_0] + \sum_{i=1}^k c(v_{i-1}, v_i)$$

$$\text{Also } c(P) = \sum_{i=1}^k c(v_{i-1}, v_i) \geq 0$$

Also kann P kein negativer Zyklus sein.

Laufzeit: $O(n + m)$ (beste bekannte Laufzeit für den Fall allgemeiner Kantenkosten).

Nicht-Negative Kantenkosten Idee: Haben wir alle ausgehenden Kanten eines Knotens v relaxiert, so müssen diese „nur“ relaxiert werden, wenn sich der temporäre Distanzwert von v ändert. Relaxieren wir nur Kanten (v, w) , so dass v schon den richtigen Distanzwert hat, müssen wir jede Kante nur einmal relaxieren.

Satz: Sei $U \subseteq V$ die Menge der Knoten, von denen wir wissen, dass wir den endgültigen Distanzwert berechnet haben. Wir nehmen weiter an, dass alle Kanten (v, w) mit $v \in U$ relaxiert wurden, nachdem v den endgültigen Distanzwert erhalten hat. Sei $w \in V \setminus U$ der Knoten in $V \setminus U$ mit geringstem temporärem Distanzwert, dann hat w seinen endgültigen Distanzwert.

Beweis: Sei $P = (s = v_0, \dots, v_k = w)$ ein kürzester Pfad von s nach w und sei v_i der erste Knoten auf P der nicht in U liegt. Dann gilt $d[v_i] \geq d[w]$ (w ist Knoten mit kleinstem Distanzwert der nicht in U liegt) und $d[v_i]$ ist der endgültige Distanzwert (da (v_{i-1}, v_i) relaxiert wurde, nachdem v_{i-1} den endgültigen Distanzwert erhalten hat. Also:

$$d[w] \leq d[v_i] = \delta(s, v_i) \stackrel{v_i \text{ liegt auf kürzestem Pfad und alle Kanten haben nicht-negative Kosten}}{\leq} \delta(s, w) \leq d[w]$$

$$\Rightarrow d[w] = \delta(s, w)$$

Algorithmus:

```

procedure DIJKSTRA( $G, s$ )
  Initialisiere  $d[]$  und  $p[]$ 
   $PQ \leftarrow$  Prioritätswarteschlange
  for all  $v \in V$  do
     $PQ.add(v, d[v])$  ▷ Füge alle Knoten mit aktuellem Distanzwert als Priorität in PQ ein
  while  $PQ \neq \emptyset$  do
     $v = PQ.extractMin$ 
    for all  $(v, w) \in E$  do
      RELAX( $v, w$ )

```

Laufzeit:

$$O(n + n(\text{Einfügen in Prioritätswarteschlange}) + n(\text{ExtractMin}) + m(\text{DecreaseKey}) + m)$$

PQ mittels binärem Heap: $O((n + m) \log n)$

PQ mittels Fibonacci-Heap: $O(n \log n + m)$ (Der Fibonacci-Heap wurde genau für diese Anwendung erfunden)

6.7.2 All-pairs shortest path Problem

Berechne kürzesten Pfad zwischen je zwei Knoten.

1. Variante: Löse für jeden Knoten $v \in V$ das single-source-Problem. Laufzeit $O(n^2 m)$ für allgemeinen Fall, $O(n^2 \log n + nm)$ für nicht-negative Kantenkosten.

2. Variante: Algorithmus mit Laufzeit $O(n^3)$ für den allgemeinen Fall.

Sei $V = \{1, \dots, n\}$. Wir nehmen an, dass es keinen negativen Zyklus gibt (dies können wir nach dem Aufruf des Algorithmus testen, in dem wir überprüfen, ob eine Kante relaxiert werden kann, Laufzeit $O(n \cdot m)$)

Sei $d_{ij}^{(k)}$ die Länge des kürzesten Pfades von i nach j , dessen innere Knoten höchstens k sind. Dann gilt: $d_{ij}^{(n)}$ ist die Länge des kürzesten Pfades (jeder Knoten ist als innerer Knoten erlaubt.) Weiterhin:

$$d_{ij}^{(0)} = \begin{cases} c(i, j) & \text{falls } (i, j) \in E \\ \infty & \text{sonst} \end{cases}$$

$d_{ij}^{(k)}$: Ist k nicht auf einem kürzesten Pfad, dann ist $d_{ij}^{(k)} = d_{ij}^{(k-1)}$, andernfalls verläuft der kürzeste Pfad erst von i nach K und alle inneren Knoten dieses Teilstücks sind höchstens $k-1$ und dann von j nach j und die inneren Knoten sind wieder höchstens $k-1$. Also:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$

Algorithmus:

procedure FLOYD-WARSHALL(G)

$$d_{ij}^{(0)} = \begin{cases} c(i, j) & \text{falls } (i, j) \in E \\ \infty & \text{sonst} \end{cases}$$

for K from 1 to n **do**

for i from 1 to n **do**

for j from 1 to n **do**

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$

Frage: Wie berechnet man den Vorgänger $p_j^{(k)}$ auf einem kürzesten Pfad von i nach j mit inneren Knoten höchstens k .

$$p_i^{(0)}(j) = \begin{cases} i & \text{falls } (i, j) \in E \\ \text{nil} & \text{sonst} \end{cases}$$

$$p_i^{(k)}(j) = \begin{cases} p_i^{(k-1)}(j) & \text{falls } d_{i,j}^{(k)} = d_{i,j}^{(k-1)} \\ p_k^{(k-1)}(j) & \text{sonst} \end{cases}$$

Speicherbedarf: So wie der Algorithmus hier beschrieben ist, hat er einen Speicherbedarf von $O(n^3)$. Da wir, nachdem $d_{ij}^{(k)}$ berechnet wird, $d_{ij}^{(l)}$ für $l \leq k-1$ nicht mehr benötigen, können wir die entsprechenden Matrizen wieder aus dem Speicher löschen. Damit sinkt der Speicherbedarf auf $O(n^2)$.

7 Fehlende Vorlesungen

Diese Vorlesungen habe ich leider nur „analog“ mitgeschrieben.

8 Algorithmen für NP-Vollständige Probleme

Wir unterscheiden drei Techniken für Algorithmen für NP-schwere Probleme:

- Approximationsalgorithmen: Finde einen Algorithmus, der in polynomieller Zeit eine Lösung findet mit einer Garantie an die Qualität der Lösung.
- Heuristik: Finde einen Algorithmus, der in der Praxis gute Lösungen findet und „effizient“ ist. Oft ohne Garantie an die Laufzeit und/oder die Qualität der Lösung.
- exakte Algorithmen: Finde die optimale Lösung, aber nicht in polynomieller Zeit (vgl. die Fragestellung P=NP?)

8.1 Approximationsalgorithmen

8.1.1 Relative Approximationsalgorithmen

Definition Ein (relativer) c -Approximationsalgorithmus für ein Minimierungsproblem P ist ein Algorithmus polynomieller Laufzeit, der eine Lösung mit Kosten von höchstens c mal der optimalen Lösung findet.

Für Maximierungsprobleme gilt eine analoge Definition für Kosten mindestens c -mal den Kosten der optimalen Lösung. Bemerkungen:

- Für Minimierungsprobleme ist $c \geq 1$, für Maximierungsprobleme $c \leq 1$.

- Oft ist c eine Konstante, es gibt aber auch $\log n$ -Approximationsalgorithmen.

Exemplarisch betrachten wir das Traveling-Salesman-Path-Problem (TSPP):

Gegeben einen (un)gerichteten Graphen $G = (V, E)$, eine Kostenfunktion $c : E \mapsto \mathbb{R}$ und einen Startknoten $s \in V$, finde einen einfachen Pfad, der von s aus jeden Knoten besucht mit minimalen Kosten.

Satz: Man kann keinen relativen Approximationsalgorithmus für TSPP finden, falls $P \neq NP$.

Beweis: Es ist bekannt, dass das Hamilton-Path-Problem (HPP) NP-vollständig ist. Gegeben eine Instanz $G = (V, E)$ des HPP, konstruieren wir eine Instanz des TSPP wie folgt:

$G' = (V, E')$ sei der vollständige Graph über V und $c' : E' \mapsto \mathbb{R}$ sei definiert durch

$$c'(v, w) = \begin{cases} 0 & \forall (v, w) \in E \\ 1 & \text{sonst} \end{cases}$$

Ein Hamiltonscher Pfad in G wird zu einem TSP der Kosten 0 in G' . Gibt es in G keinen Hamiltonschen Pfad, dann kostet der TSP mindestens 1.

Da jeder relative Approximationsalgorithmus eine Lösung mit Kosten 0 finden muss, falls es eine gibt, müsste ein Approximationsalgorithmus für P das HPP lösen.

Definition Das metrische TSPP (mTSPP) ist das TSPP auf einem vollständigen, ungerichteten Graphen $G = (V, E)$ mit einer Kostenfunktion $c : E \mapsto \mathbb{R}_{\geq 0}$, die die Dreiecksungleichung erfüllt, also $c(u, v) \leq c(u, w) + c(w, v)$.

Satz: Es gibt einen 2-Approximationsalgorithmus für mTSPP.

Beweis: Der Algorithmus sieht wie folgt aus:

- Berechne den MST T für (G, c) .
- Betrachte den Multigraphen $(V, T \cup T)$ (Den Graphen, in dem jede Kante aus V doppelt vorkommt).
- Da dieser Graph ein Eulergraph ist, können wir eine Eulertour P' berechnen.
- Konstruiere einen TSP, indem man von s aus die Eulertour P' entlangläuft und jeden neu besuchten Knoten an den Pfad P anhängt.

Beobachtungen:

1. Sei OPT ein TSP minimaler Kosten. Dann gilt $c(OPT) \geq c(T)$, da OPT ein aufspannender Baum ist und die Kosten von OPT mindestens so groß sind wie die Kosten des MST.
2. $c(O) \leq c(P')$: Konstruieren wir P aus P' , so gehen wir „Abkürzungen“, d.h. wir ersetzen Teilpfade (v_0, \dots, v_k) von P' durch die Kante (v_0, v_k) . Aus der Dreiecksungleichung folgt $c(v_0, \dots, v_k) \leq c(v_0, v_k)$.

Aus den Beobachtungen folgt, dass $c(P) \leq c(P') = 2c(T) \leq 2c(OPT)$.

Anmerkung: Es gibt einen $\frac{3}{2}$ -Approximationsalgorithmus (Christofidus-Algorithmus), bei dem nur jeweils Knoten, die in T einen ungeraden Grad haben, miteinander verbunden werden.

8.1.2 Polynomial Time Approximation Scheme (PTAS)

Definition Ein PTAS ist ein Algorithmus, der für jede Problem Instanz I und für jedes konstante $\epsilon > 0$ in polynomieller Zeit eine Lösung findet, die höchstens bzw. mindestens $1 \pm \epsilon$ mal so groß ist wie die optimale Lösung-

Ein fully PTAS ist ein PTAS mit einer Laufzeit, die polynomiell in $\frac{1}{\epsilon}$ ist.

Beispiele: Ein Algorithmus mit Laufzeit $O(n^\epsilon)$ ist ein PTAS, aber kein fully PTAS.

Ein Algorithmus mit Laufzeit $O\left(\left(\frac{n}{\epsilon}\right)^{100}\right)$ ist ein fully PTAS.

Beispiel: Knapsack-Problem Gegeben einen Rucksack mit maximalem Füllgewicht W und Waren $T = \{1, \dots, n\}$ mit Gewichten $w_i \in \mathbb{N}$, $0 \leq w_i \leq W$ und Profiten $p_i \geq 0$, finde eine Teilmenge $I \subseteq \{1, \dots, n\}$ der Waren mit Gesamtgewicht $\sum_{i \in I} w_i \leq W$ und maximalem Profit.

Bemerkung: Das Problem ist NP-vollständig.

Ein exakter Algorithmus für Knapsack, falls $p_i \in \mathbb{N}$ Idee: Für jede Teilmenge $\{1, \dots, i\}$ und möglichen Profit p berechne die Teilmenge von $\{1, \dots, i\}$, die mit minimalem Gewicht einen Profit p erreicht.

Algorithmus:

1: **procedure** KNAPSACK

2: $p_{max} \leftarrow \max\{p_i \mid 1 \leq i \leq n\}$

3: $P \leftarrow n \cdot p_{max}$

▷ Offensichtlich ist der optimale Profit $\leq P$.

```

4:    $M[j] \leftarrow \begin{cases} 0 & \text{für } j = 0 \\ \infty & \text{für } j = 1, \dots, P \end{cases}$ 
5:   for  $i$  from 1 to  $n$  do
6:     for  $j$  from 0 to  $P$  do
7:        $M[j] \leftarrow \min\{M[j], M[j - p_i] + w_i\}$ 

```

Laufzeit: $O(n \cdot P) = O(n^2 p_{max})$ (Nicht polynomiell, da p_{max} sehr groß sein kann).

Beobachtungen: Für eine Instanz I des Knapsack-Problemes sei $OPT(I)$ die optimale Lösung von I und $p_{max}(I)$ der maximale Profit einer Ware aus I . Dann gilt:

1. Sei I_α die Instanz, die man erhält, wenn man die Profite aller Waen der Instanz I mit α multipliziert. Dann gilt:

$$p(OPT(I_\alpha)) = \alpha p(OPT(I))$$

2. Sei I'_α die Instanz, die man erhält, wenn man alle Profite der Instanz I_α auf die nächste ganze Zahl abrundet. Dann gilt:

$$p(OPT(I'_\alpha)) \leq p(OPT(I_\alpha)) \leq p(OPT(I'_\alpha)) + n$$

denn jede Lösung von I_α hat bzgl. I'_α einen Profit von höchstens n weniger als $p(I_\alpha)$ und jede Lösung von I'_α hat bzgl. I_α einen mindestens gleich großen Profit wie $p(I_\alpha)$.

PTAS für Knapsack:

- Wähle $\alpha = \frac{n}{\varepsilon \cdot p_{max}}$.
- Berechne $\frac{1}{\alpha} \cdot (\text{opt. Lösung von } I'_\alpha)$.

Laufzeit:

$$O(n^2 \cdot p_{max}(I_\alpha)) \subseteq O(n^2 \cdot \alpha \cdot p_{max}) = O\left(n^2 \cdot \frac{n}{\varepsilon \cdot p_{max}} \cdot p_{max}\right) = O\left(\frac{n^3}{\varepsilon}\right)$$

Garantie: Sei L die gefundene Lösung. Dann gilt:

$$\begin{aligned}
p(OPT(I)) &= \frac{1}{\alpha} p(OPT(I_\alpha)) \leq \frac{1}{\alpha} (p(OPT(I'_\alpha)) + n) \\
&\leq p(L) + \frac{n}{\alpha} \leq p(L) + \varepsilon \cdot \underbrace{p_{max}}_{\leq p(OPT(I)), \text{ denn man könnte nur den wertvollsten Gegenstand mitnehmen}} \leq p(L) + \varepsilon OPT(I) \\
&\Leftrightarrow p(L) \geq (1 - \varepsilon) p(OPT(I))
\end{aligned}$$

Bemerkung: Dies ist ein fully PTAS.

8.2 Heuristiken

Wir unterscheiden zwei Arten von Heuristiken:

- Konstruktive Heuristiken: „Baue“ gute Lösungen - sehr problemspezifisch.
- Lokale Suche: „Verbessere“ gegebene Lösung iterativ.

8.2.1 Konstruktive Heuristiken

Konstruktive Heuristiken für das TSPP sind zum Beispiel

- Greedy Algorithmen: Baue Lösung iterativ auf und wähle immer die lokal beste Erweiterung (vgl. MST-Algorithmen). Für TSPP: Starte bei s , gehe immer zum am nächsten gelegenen, nicht besuchten Knoten.
- Alle Approximationsalgorithmen.

8.2.2 Lokale Suche

Verfahren bei der lokalen Suche:

- Definiere eine „Nachbarschaft“ für Lösungen (problemspezifisch)
- Durchsuche Nachbarschaft einer bekannten Lösung nach guten Lösungen.

Beispiele für Nachbarschaft für das TSPP:

- Besuche einen Knoten zu einem anderen Zeitpunkt (etwa n^2 Nachbarn, 3 Kanten werden verändert).
- Besuche k Knoten zu einem anderen Zeitpunkt (Sehr große Nachbarschaft).
- Besuche einen Teilpfad zu einem anderen Zeitpunkt (Etwa n^3 Nachbarn).
- Viele weitere Nachbarschaften sind denkbar.

Suche in der Nachbarschaft: Für eine LLösung x sei $f(x)$ der Zielfunktionswert der Lösung (für ein Maximierungsproblem) und $N(x)$ die Nachbarschaft der Lösung.

1.Variante: Hill-Climbing Gehe immer zu einem Nachbarn mit besserem Zielfunktionswert.

```

1: procedure HILLCLIMBING
2:    $x \leftarrow$  beliebige Lösung
3:   while not finished do
4:     Wähle  $\hat{x} \in N(x)$  mit  $f(\hat{x}) > f(x)$ , falls so ein  $\hat{x}$  existiert.
5:      $x \leftarrow \hat{x}$ 

```

Hill-Climbing ist oft sehr effizient, läuft aber in lokale Optima.

Es gibt Probleme, die keine lokalen Optima haben. Dann finden wir die optimale Lösung durch Hill-Climbing. Vgl. Maximale Flüsse: Nachbarschaft eines Flusses: Verändere denn Fluss maximal über einen s - t -Pfad. Für Hill-Climbing ist keine Laufzeitaussage möglich.

Um das Laufen in lokale Optima zu vermeiden, gibt es verschiedene Techniken.

Simulated Annealing „Annealing“ ist ein Verfahren in der Physik: Betrachte einen Behälter flüssigen Quarzes (SiO_2). Kühlt man den Behälter schnell ab, so entsteht Glas (Jedes Atom ist in einem lokalen Optimum). Kühlt man hingegen den Behälter sehr langsam ab, so entsteht Quarz (globales Optimum)

Diesen Prozess wollen wir im Computer simulieren:

Starte mit einer hohen Temperatur: Akzeptiere jeden Nachbarn. Kühle das System langsam ab: Akzeptiere schlechtere Nachbarn mit immer geringerer Wahrscheinlichkeit.

Algorithmus: Sei x eine Lösung und T eine Temperatur.

```

1: procedure SIMULATEDANNEALING
2:   while noch nicht abgekühlt do
3:     Wähle  $\hat{x} \in N(x)$  zufällig.
4:     Setze  $x \leftarrow \hat{x}$  mit Wahrscheinlichkeit  $\min \left\{ 1, e^{\frac{f(\hat{x})-f(x)}{T}} \right\}$ .
5:     Speichere beste bisher gefundene Lösung
6:     Kühle  $T$  langsam ab

```

Evolutionäre Algorithmen Idee:

- Speichere eine große Menge von Lösungen (\approx Individuen)
- Verändere Lösungen zufällig
- Kombiniere 2 (oder mehr) Lösungen zu neuer Lösung (Hoffnung: Das beste aus den beiden Lösungen kommt in die neue Lösung). Die Kombination ist problemspezifisch,
- Nur „starke“ = gute Individuen haben gute Chancen zu „überleben“.

Beispiel für das TSPP:

- Verändere Lösung zufällig: Gehe zu Nachbarn über.
- Kombiniere Pfade A und B zu neuem Pfad C :
 - Wähle C als Startpfad mit 10 bis $\frac{n}{2}$ Kanten von A . Sei v der zuletzt besuchte Knoten von C .

- * Ist der Nachfolger von v bzgl. B noch nicht besucht, gehe als nächstes zu diesem Knoten.
 - * Andernfalls gehe zu Nachfolger in A , falls möglich.
 - * Andernfalls gehe zum nächsten noch nicht besuchten Knoten.
- Verbessere C mit Hill-Climbing.

Der Algorithmus sieht in etwa so aus:

- 1: **procedure** EVOLUTIONARERALGORITHMUS
- 2: Konstruiere Population $pop = \{x_1, \dots, x_N\}$ (Menge von Lösungen)
- 3: **while** not finished **do**
- 4: Wähle $\frac{N}{2}$ Paare von Individuen (x_i, x_j) und berechne „kombinierte“ Lösung.
- 5: Mutiere jedes Individuum mit einer bestimmten Wahrscheinlichkeit.
- 6: Behalte jedes Individuum mit einer Wahrscheinlichkeit abhängig von der Qualität.

Weitere Techniken

- Tabu Search:
 - Merke „Eigenschaften“ von Lösungen aus Teilbereichen, die schon gut abgesucht sind.
 - Vermeide Lösungen mit dieser Eigenschaft bei der weiteren Suche.
- Restarts: Starte die Suche von Zeit zu Zeit mit einer neuen Startlösung (Insbesondere beim Hill-Climbing).
- Nutze Hill-Climbing als Teilstrategie:
 - Berechne Lösung mit SA oder EA.
 - Von Zeit zu Zeit optimiere aktuelle Lösung(en) mit Hill-Climbing.

8.3 Exakte Algorithmen für NP-Schwere Probleme \rightarrow Keine polynomiellen Algorithmen

1. Ansatz: Aufzählen von Lösungen mittels „branching“.

Abstrakt: Sei U eine endliche Menge (von Lösungen) und $f : U \mapsto \mathbb{R}$ eine Funktion. Wir wollen $\min_{x \in U} f(x)$ berechnen.

Branching-Ansatz:

- Partitioniere U in U_1, \dots, U_k
- Löse die Teilprobleme U_1, \dots, U_k rekursiv
- Das Minimum der Lösungen der Teilprobleme ist die Lösung unseres Problems.

Beispiel: TSPP mit Startknoten s auf einem vollst. Graphen:

- Starte mit dem Teilpfad $P = (s)$
- Teilprobleme: Fixiere einen weiteren Knoten auf P , wobei alle noch nicht besuchten Knoten ausprobiert werden.

- 1: **procedure** TSPP(G, P) ▷ Beim ersten Aufruf ist $P = (s)$
- 2: **if** P ist TSPfad **then**
- 3: **return** $c(P)$
- 4: **else**
- 5: $x \leftarrow \infty$
- 6: **for all** $v \in V \setminus P$ **do**
- 7: $x \leftarrow \min(x, TSPP(G, P \circ v))$
- 8: **return** x

Laufzeit: $O((n-1)!)$

2. Ansatz: Vermeide möglichst viele rekursive Aufrufe. Kann man (algorithmisch) nachweisen, dass die optimale Lösung von U nicht im aktuellen Teilproblem liegt, kann die Suche von diesem Teilproblem abgebrochen werden.

Eine Möglichkeit: „bounding“ (Der Gesamtalgorithmus heißt dann branch-and-bound). Angenommen, ie beste bekannte Lösung hat Kosten gub (Global upper Bound). Wir berechnen eine untere Schranke für die Kosten der Lösungen in einem Teilproblem llb (Local lower Bound). Ist $llb \geq gub$ können wir die Suche in diesem Teilproblem abbrechen.

Bsp für TSPP. Sind die Kosten des aktuellen Teilpfades schon größer als die Kosten des besten bekannten Pfades (und ist $c \geq 0$), kann die Suche abgebrochen werden.

Effizienz hängt ab von:

- Qualität der lokalen unteren Schranke

- Qualität der oberen Schranke (Starte eine Heuristik vor der Suche, betrachte Teilprobleme in denen man gute Lösungen vermutet zuerst)
- Laufzeit pro rekursiven Aufruf (zum Berechnen der unteren Schranke und zum Aufteilen in Teilprobleme)
- Geschickte Aufteilung
- Bearbeitungsreihenfolge der Teilprobleme: Beschreibung war „Tiefensuche im Rekursionsbaum“. Verbesserung: Bearbeite Teilprobleme mit kleiner unterer Schranke zuerst. Oder: Keine Teifensuche, sondern verwalte Teilprobleme in einer Prioritätswarteschlange und bearbeite sie in der Reihenfolge aufsteigender unterer Schranken.

8.3.1 Untere Schranke

1.Variante: Sei r der letzte Knoten auf dem Teilpfad P und $U = V \setminus P$. Die Kosten des TS-Pfades mit Startpfad P sind $c(P)$ plus die Kosten des TS-Pfades mit Startknoten r , der alle Knoten in $U \cup \{r\}$ besucht.

Untere Schranken hierfür sind

- 1) $c(p) + \sum_{v \in U} \min_{w \in U \cup \{r\}} c(w, v)$, da jeder Knoten in U genau eine eingehende Kante auf dem TS-Pfad besitzt.
- 2) $c(P) + \min_{\text{Branching}}((U \cup \{r\}, \underbrace{E|_{U \cup \{r\}}}_{\text{Kanten von } E \text{ mit Endknoten in } U \cup \{r\}}, r, c)$, da der TS-Pfad ein Branching ist, ist er teurer als das minimale Branching.

→ Liefert bessere Schranken, Branching ist aber teurer.

2.Variante: Lineare Programme.

Viele NP-schwere Probleme lassen sich wie folgt abstrahieren:

$$\min_{Ax \leq b, x \in \{0,1\}^n} c^T x, \text{ wobei } c \in \mathbb{R}^n, A \in 2^{n \times m} \text{ und } b \in 2^m$$

(Ganzzahliges Lineares Programm, ILP)

Beispiel: TS-Pfad. Benutze für jede Kante e eine Variable x_e , wobei die Variable den Wert 1 annehmen soll, falls e im TS-Pfad benutzt wird, ansonsten den Wert 0.

$$\begin{aligned} &\min \sum_{e \in E} c_e x_e (\text{Summiere die Kosten aller Kanten, die benutzt werden}) \\ &\sum_{e \in \delta^-(v)} x_e = 1 \text{ für alle } v \in V \setminus \{r\} \text{ Jeder Knoten } v \in V \setminus \{r\} \text{ hat genau eine eingehende Kante} \\ &\sum_{e \in \delta^+(v)} x_e \leq 1 \text{ für alle } v \in V \setminus \{r\} \text{ Jeder Knoten } v \in V \setminus \{r\} \text{ hat maximal eine ausgehende Kante} \\ &\sum_{u,v \in S} x_{(u,v)} \leq |S| - 1 \text{ für alle } \emptyset \neq S \subseteq V \text{ Verbietet Zyklen} \\ &x_e \in \{0, 1\} \forall e \in E \end{aligned}$$

Ersetzt man die Bedingung $x_e \in \{0, 1\}$ durch $0 \leq x_e \leq 1$ erhält man eine untere Schranke (Da die Menge der zulässigen Lösungen vergrößert wird) Diese untere Schranke (LP-Schranke) kann man in polynomieller Zeit berechnen, obwohl die Beschreibung exponentiell viele Ungleichungen enthält.

(Ein lineares Programm mit n Variablen kann in polynomieller Zeit gelöst werden, wenn man von jedem $x \in \mathbb{R}^n$ in polynomieller Zeit entscheiden kann, ob x alle Ungleichungen erfüllt und wenn nicht eine verletzte Ungleichungen berechnen)

3.Variante: Kombination der beiden Ansätze: Langrangean Relaxation. (Hoffnung: Finde Schranken mit der Qualität des 2.Ansatzes in Laufzeit vergleichbar der des 1. Ansatzes)

Ignoriert man die Ungleichungen $\sum_{e \in \delta^+(v)} x_e \leq 1$, erhält man die Beschreibung des Minimum-Branching-Problems.

Idee: Ziehe diese Ungleichungen in die Zielfunktion, d.h. „bestrafe“ Lösungen, die Ungleichungen verletzen.

Mathematisch: Wähle für jede Ungleichung $\sum_{e \in \delta^+(v)} x_e \leq 1$ einen Wert $\mu_v \geq 0$ und löse das Problem für die modifizierte Kosten Funktion für jeden TS-Pfad

$$\begin{aligned} &\min_{x \text{ branching}} c^T x + \underbrace{\sum_{v \in V} \mu_v \left(\sum_{e \in \delta^+(v)} x_e - 1 \right)}_{\leq 0} \\ &= \min_{(v,w) \in E} (c_{(v,w)} + \mu_v) x_e - \sum_{v \in V} \mu_v \end{aligned}$$

Fakt: Die Optimale Wahl von μ liefert die gleiche Schranke wie das LP. Algorithmus zum Berechnen einer guten Wahl für μ_i : Iteratives Verfahren:

- Starte mit bel. μ .
- Berechne minimales Branching bzgl. der modifizierten Kosten.
- Ist der Ausgangsgrad eines Knotens v größer als 1, erhöhe μ_v um $\lambda \cdot (\text{Ausgrad} - 1)$
- Ist der Ausgangsgrad eines Knotens v 0, erniedrige μ_v um $\min(\mu_v, \lambda)$
- Erniedrige λ langsam (z.B. $\lambda = 0.9\lambda$ alle 100 Iterationen)

Dieses Verfahren konvergiert bei geeigneter Wahl von μ gegen das Optimum.

9 Idee der amortisierten Analyse mittels Potenzialfunktion

Angenommen die i -te Operation auf einer Datenstruktur kostet c_i und c_i „schwankt stark“. Viele Operationen sind „billig“, aber einige haben „hohe“ Kosten. Also ist die worst-case-Laufzeit hoch.

Deshalb analysieren wir die worst-case Kosten von m Operationen insgesamt. Können wir zeigen, dass m Operationen höchstens $m \cdot f(n)$ kosten, sagen wir, jede Operation hat amortisierte Kosten $f(n)$ (Wobei n meist die maximale Größe der Datenstruktur repräsentiert).

Eine Potenzialfunktion ist ein Hilfsmittel, um die Kosten einer Folge von Operationen analysieren zu können:

- Identifiziere den Grad der Schwankungen
- Belege die Datenstruktur mit einem „hohen“ Potenzial, wenn es eine teure Operation geben kann. Wird diese dann ausgeführt, muss das Potenzial fallen.
- Wähle die Potenzialfunktion geschickt so, dass $c_i + \Phi_i - \Phi_{i-1}$ möglichst klein ist, wobei Φ_i das Potenzial der Datenstruktur nach der i -ten Operation ist. Sagen wir $c_i + \Phi_i - \Phi_{i-1} \leq f(n)$.

Dann kosten m Operationen

$$\sum_{i=1}^m c_i \leq \sum_{i=1}^m (f(n) + \Phi_{i-1} - \Phi_i) = mf(n) + \Phi_0 - \Phi_m$$

Kann man jetzt noch zeigen, dass $\Phi_0 \leq \Phi_m$ ist (z.B. $\Phi_0 = 0, \Phi_i \geq 0$), dann sind die amortisierten Kosten einer Operation $f(n)$.

Beispiel Binärzähler: Ein Hochzählen ist teuer, wenn viele Einsen zu Beginn. Beim Hochzählen verringert sich dann die Anzahl der Einsen. Wähle also

$$\Phi_i = c \cdot (\text{Anzahl der Einsen})$$

und rechne.

$$c_i \in O(k+1), \text{ wobei } k \text{ die Anzahl der Einsen am Ende der Bit-Repräsentation der aktuellen Zahl}$$

$$\Phi_i - \Phi_{i-1} = k - 1$$

$$c_i \in O(k+1) \text{ heißt: } \exists c' : c_i \leq c'(k+1)$$

$$a_i = c_i + \Phi_i - \Phi_{i-1} \leq c'(k+1) - c(k-1) = 2c \in O(1) \text{ für } c = c'$$

Da $\Phi_0 = 0$ und $\Phi_i \geq 0$, gilt, dass die amortisierten Kosten pro Operation $O(1)$ sind.