

Datenstrukturen und effiziente Algorithmen

Markus Vieth

David Klopp

Christian Stricker

22. Dezember 2015

Inhaltsverzeichnis

I. Sortieren	5
1. Vorlesung 1	6
1.1. Bubblesort	6
1.1.1. Pseudocode	6
1.1.2. Laufzeitanalyse	6
1.2. Heapsort	7
1.2.1. Heap-Eigenschaft	7
2. Vorlesung 2	8
2.0.1. Pseudocode	8
2.0.2. Korrektheitsbetrachtung	8
2.0.3. Laufzeitanalyse	8
3. Vorlesung 9	9
3.1. Vergleichbasierte Sortieralgorithmen	9
3.2. Radix-Sort	9
3.2.1. Beispiel:	9
3.2.2. Pseudo-Code	9
3.3. Binäre Suchbäume	10
4. Vorlesung 10	11
4.1. Binärer Suchbaum	11
4.2. Pseudo-Code	11
4.3. AVL-Bäume	12
4.4. Laufzeitanalyse	12
5. Vorlesung 11	14
5.1. AVL-Bäume von Adelson-Velskii and Landis	14
5.1.1. AVL-Eigenschaft:	14
5.2. Rotationen	15
5.3. Pseudo-Code	16
6. Vorlesung 12	17
6.1. (a,b)-Suchbäume	17
6.1.1. Aufspaltung bei Einfügen	17
6.1.2. Verschmelzen von Knoten beim Löschen	17
6.2. Amortisierte Analyse	17
6.2.1. Bankkonto-Methode	17
7. Vorlesung 13	19
7.1. Hashing	19
7.1.1. Universelles Hashing	20

8. Vorlesung 14	22
8.0.1. Definition	22
8.0.2. Beispiel	22
8.0.3. Abschätzung nach oben	23
8.1. Perfektes Hashing	23
8.1.1. Definition	23
8.1.2. Nachteil	25
9. Vorlesung 15	26
II. Graphen-Algorithmen	27
9.0.1. Einführung	28
9.0.2. BFS (Breadth-First Search) Breitensuche	30
10. Vorlesung 16	32
10.1. Kürzeste Wege Algorithmen	37
10.1.1. Dijkstra-Algorithmus	37
11. Vorlesung 17	39
11.0.1. Vorläufige Laufzeitanalyse von Dijkstra	40
11.1. Bellman-Ford-Algorithmus	40
11.1.1. Pseudocode	41
11.1.2. Laufzeit: Bellman-Ford	41
11.1.3. Korrektheitsbeweis: Bellman-Ford	41
11.1.4. Induktionsschritt: $i \rightarrow i + 1$	41
12. Vorlesung 18	42
12.1. All-Pairs-Shortest Path Algorithmen	42
12.1.1. Laufzeit zur Berechnung von $D^{(n)}$	43
12.2. Floyd-Warshall-Algorithmus	43
12.2.1. Korrektheitsbeweis:	43
12.2.2. Beweis der Invariante durch Induktion nach k	44
12.3. Naive lösung	44
12.4. Johnson-Algorithmus	44
12.4.1. Laufzeit des Johnson-Algorithmus	45

Teil I.

Sortieren

1. Vorlesung 1

1.1. Bubblesort

1.1.1. Pseudocode

```
void bubblesort (int [] a) {  
    int n = a.length;  
    for (int i = 1; i < n; i++) {  
        for (int j = 0; j < n-i; j++) {  
            if ( a[j] > a[j+1])  
                swap (a, j, j+1);  
        }  
    }  
}
```

Schleifen-Invariante: Nach dem Ablauf der i-ten Phase gilt:

Die Feldpositionen $n-i, \dots, n-1$ enthalten die korrekt sortierten Feldelemente

Beweis durch Induktion nach $i \xrightarrow{i=n-1} 1$ Sortierung am Ende korrekt.

1.1.2. Laufzeitanalyse

1.	Phase	$n-1$
2.	Phase	$n-1$
3.	Phase	$n-1$
	\vdots	
i.	Phase	$n-1$
	\vdots	
(n-1).	Phase	$n-1$
<hr/>		
$1 + 2 + 3 + \dots + (n-1)$		

$$T(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in O(n^2)$$

n	T_{real}
2^{10}	8ms
2^{11}	11ms
2^{12}	26ms
\vdots	
2^{16}	5,819s
2^{17}	23,381s
\vdots	
2^{20}	16min
\vdots	
2^{26}	52d

$$T_{real}(n) \approx cn^2 \quad c \approx 10^{-6}$$

1.2. Heapsort

z.B. 21 6 4 7 12 5 3 11 14 17 19 8 9 10 42

Skizze

1.2.1. Heap-Eigenschaft

2. Vorlesung 2

Heapsort (Fortsetzung)

2.0.1. Pseudocode

```
heapify ( int[] a, int i, int n) {  
    while (2i + 1 < n) {           //linkes Kind von i existiert  
        int j = 2i + 1;  
        if ( 2i + 2 < n)           //rechtes Kind von i existiert  
            if ( a[j] < a[j+1])  
                j = j + 1;         //j steht für Index des größten Kindes  
        if ( a[i] > a[j])           //Vater größer als Kind  
            break;                 //Abbruch, weil heap bereits erfüllt  
        swap(a,i,j);               //Tausch zwischen Vater und Kind  
        i = j;  
    }  
}
```

1. Phase: Bottom-up Strategie zum Heapaufbau

```
for ( int i = n/2; i ≥ 0; i--)  
    heapify(a,i,n);
```

2. Phase: Sortierphase

```
for ( int i = n-1; i ≥ 0; i--) {  
    swap(a,0,i);  
    heapify(a,0,i);  
}
```

2.0.2. Korrektheitsbetrachtung

Invariante beim Heapaufbau: Beim Durchlauf der for-Schleife wird die Heapeigenschaft vom unteren Baumlevel bis zur Wurzel hergestellt.

Invariante für Sortierphase: Nach jedem weiteren Durchlauf der for-Schleife findet ein weiteres Element am Feldende seinen „richtigen Platz“.

2.0.3. Laufzeitanalyse

$T(n)$ = Zahl der Elementvergleiche.

Analyse Heapaufbau:

3. Vorlesung 9

3.1. Vergleichbasierte Sortialgorithmen

3.2. Radix-Sort

3.2.1. Beispiel:

10	1	0	1	0	1	00	001
01	0	1	0	0	1	01	010
00	1	1	1	0	0	01	011
11	1	1	0	1	0	10	100
10	0	0	0	1	1	10	101
01	1	1	1	1	1	11	110
11	0	0	1	1	0	11	111

Wichtig Beginne die Sortierung mit dem niedrigsten Bit

3.2.2. Pseudo-Code

```
void radixsort(int[] a) { // positives Element
    int n = a.length;
    int[] b0 = new int[n];
    int[] b1 = new int[n];
    int n0, n1;

    for (int i=0; i<32; i++) {
        n0 = n1 = 0;
        for (int j=0; j<n; j++) {
            if (a[j] & (1<<i)) { // i-tes Bit von a[j]
                b1[n1] = a[j];
                n1 = n1+1;
            } else {
                b0[n0] = a[j];
                n0 = n0+1;
            }
        }
    }
    for (int j=0; j<n0; j++)
        a[j] = b0[j];
    for (int j=0; j<n1; j++)
        a[n0+j] = b1[j];
}
```

3.3. Binäre Suchbäume

Zahlen 12, 8, 3, 16, 24, 17, 10, 21, 14, 9

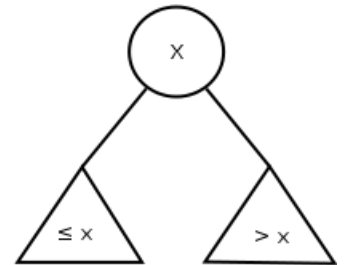
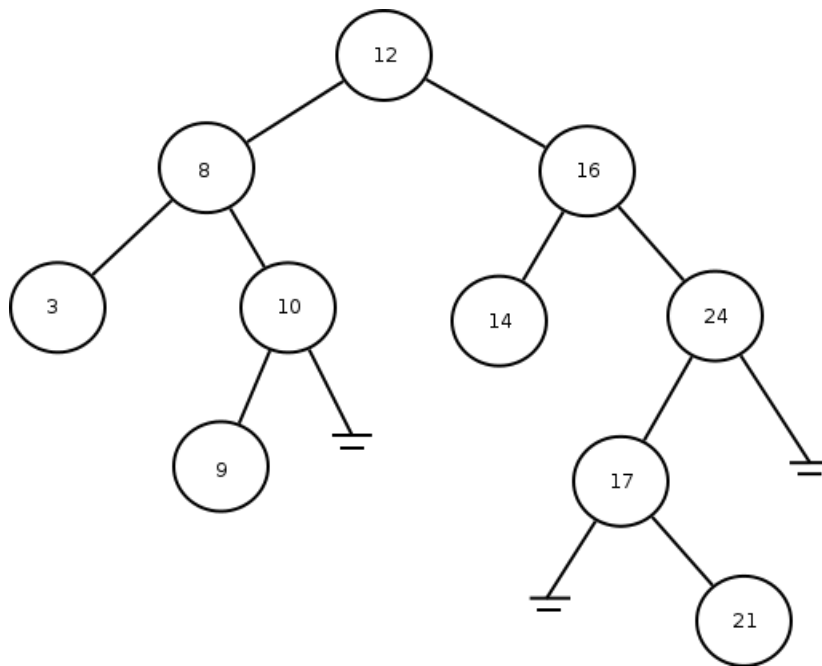


Abbildung 3.1.: Knotenorientierte Speicherung

4. Vorlesung 10

4.1. Binärer Suchbaum

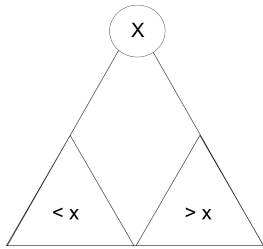


Abbildung 4.1.: Binärer Suchbaum

4.2. Pseudo-Code

```
class Node {  
    int key, info;           // info ist optional  
    Node left, right, parent; // parent ist optional  
}  
  
int height(Node node) {  
    if (node == NULL) return 0;  
    return height;  
}  
  
Node insert(Node node, int x) {  
    if (node == NULL)  
        return new Node(x, NULL, NULL);  
  
    if (node.key > x)  
        node.left = insert(node.left, x);  
    else  
        node.right = insert(node.right, x);  
    return node;  
}  
  
void inorder(Node node) {  
    if (node == NULL) return;  
    inorder(node.left) // linke Hälfte  
    print(node)  
    inorder(node.right) // rechte Hälfte  
}
```

4.3. AVL-Bäume

Ziel Binärer Suchbaum mit garantierter Such-, Einfüge- und Löschozeit $O(\log n)$

Idee Definiere eine Balancebedingung, die dafür sorgt, dass die Baumstruktur möglichst nahe an der Idealstruktur eines vollständigen binären Baumes liegt.

Aber gleichzeitig soll es möglich sein "schnell" Strukturänderungen beim Einfügen und Löschen vorzunehmen.

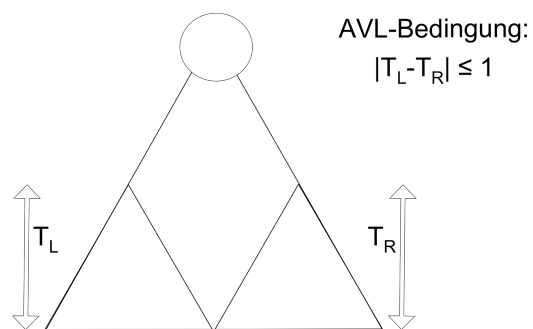
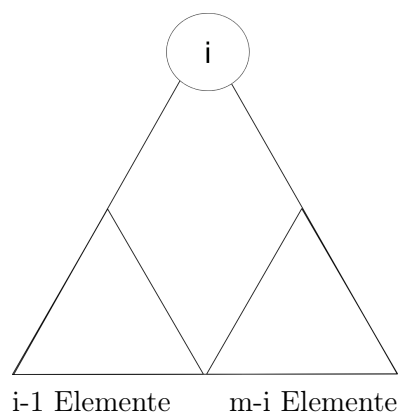


Abbildung 4.2.: AVL-Baum

4.4. Laufzeitanalyse

Ziel Analyse der erwarteten maximalen Tiefe randomisierter binärer Suchbäume

Sei der Schlüssel der Wurzel das i -kleinste Element



$T_n \hat{=}$ maximale Tiefe eines randomisierten Suchbaums mit $\{1, \dots, n\}$ Elementen

Abbildung 4.3.

Für den Fall, dass i als Wurzelknoten gewählt wird gilt:

$$T_n = \max\{T_{i-1}, T_{n-i}\} + 1$$

$$X_n = 2^{T_n} \text{ exponentielle Tiefe}$$

$$2^{T_n} = 2^{1+\max\{T_{i-1}, T_{n-1}\}} = 2 \cdot 2^{\max\{T_{i-1}, T_{n-1}\}} = 2 \cdot \max\{2^{T_{i-1}}, 2^{T_{n-1}}\}$$

$$\Rightarrow X_n = 2 \cdot \max\{X_{i-1}, X_{n-1}\}$$

Mit der Abschätzung: $\max\{2^{T_1}, 2^{T_2}\} \leq 2^{T_1} + 2^{T_2}$ folgt:

$$E(X_n) = E\left(\sum_{i=1}^n \frac{1}{n} \cdot 2 \cdot \max\{X_{i-1}, X_{n-1}\}\right)$$

$$= \frac{2}{n} \sum_{i=1}^n E(\max\{X_{i-1}, X_{n-1}\}) \leq \frac{2}{n} \sum_{i=1}^n E(X_{i-1} + X_{n-1}) = \frac{2}{n} \sum_{i=1}^n [E(X_{i-1}) + E(X_{n-i})] \leq \frac{4}{n} \sum_{i=0}^{n-1} E(X_i)$$

$$n \cdot E(X_n) = 4 \cdot \sum_{i=0}^{n-1} E(X_i) \quad (1)$$

$$(n-1) \cdot E(X_{n-1}) = 4 \cdot \sum_{i=0}^{n-2} E(X_i) \quad (2)$$

$$nE(X_n) - (n-1)E(X_{n-1}) = 4E(X_n) \quad (1) - (2)$$

$$\Leftrightarrow nE(X_n) = (n+3)E(X_{n-1})$$

$$E(X_n) = \frac{n+3}{n} E(X_{n-1}) = \frac{n+3}{n} \cdot \frac{n+2}{n-1} E(X_{n-2}) = \prod_{i=0}^{n-1} \frac{n+3-i}{n-i} = \frac{n+3}{n} \cdot \frac{n+2}{n-1} \cdot \frac{n+1}{n-2} \cdot \frac{n}{n-3} \cdot \dots \cdot \frac{6}{3} \cdot \frac{8}{2} \cdot \frac{4}{1}$$

Mit der "Jensenschen Ungleichung" folgt:

$$\left(\sum_i \Pr(T = t_i) \cdot f(t_i)\right) \geq f\left(\sum_i \Pr(T = t_i) \cdot t_i\right) = \frac{(n+3)(n+2)(n+1)}{3!} \cdot \Rightarrow E(X_n) \in O(n^3)$$

$$X_n = 2^{T_n}, E(X_n) = E(2^{T_n})$$

$$E(f(T)) \geq f(E(T)) \Leftrightarrow f \text{ konvex}$$

$$c \cdot n^3 \geq 2^{E(T_n)}, E(T_n) \leq \log_2(c \cdot n^3) \in O(\log n)$$

5. Vorlesung 11

5.1. AVL-Bäume von Adelson-Velskii and Landis

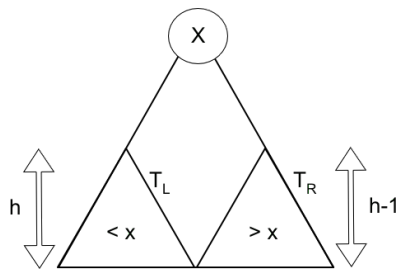


Abbildung 5.1.

Ziel: Zeige, dass die maximale Tiefe eines AVL-Baums mit n Knoten ($\hat{=}$ n gespeicherten Schlüsseln) $O(\log(n))$ beträgt.

5.1.1. AVL-Eigenschaft:

$|h(T_L) - h(T_R)| \leq 1$ muss für jeden Knoten des Baums gelten. \Rightarrow Suchzeit $O(\log(n))$ im worst case.

$n(h)$ = minimale Anzahl von Knoten in AVL-Baum der Tiefe h

$n(h) \geq 1 + n(h-2) + n(h-1)$ mit $n(0) = 0$ und $n(1) = 1$

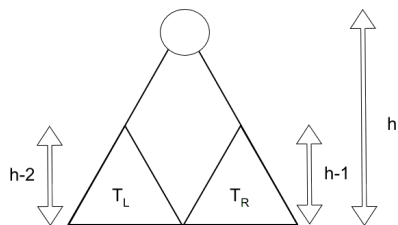


Abbildung 5.2.

$$n \geq f(h)^I = \frac{1}{\sqrt{5}} \cdot (\phi^h - \phi^{-h}) \text{ mit}$$

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1,61 \dots$$

$$\Rightarrow n \geq c \cdot \phi^h$$

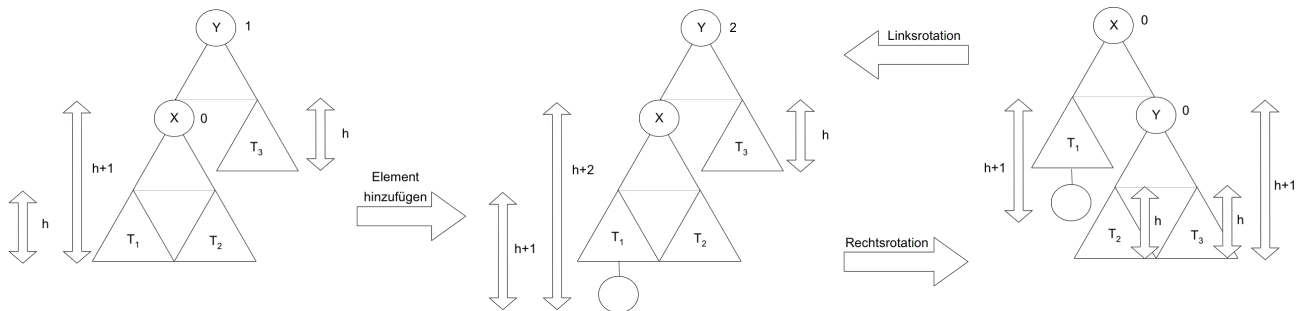
$$\Leftrightarrow h \leq \log\left(\frac{n}{c}\right)$$

$$\Rightarrow h \in O(\log n)$$

q.e.d.

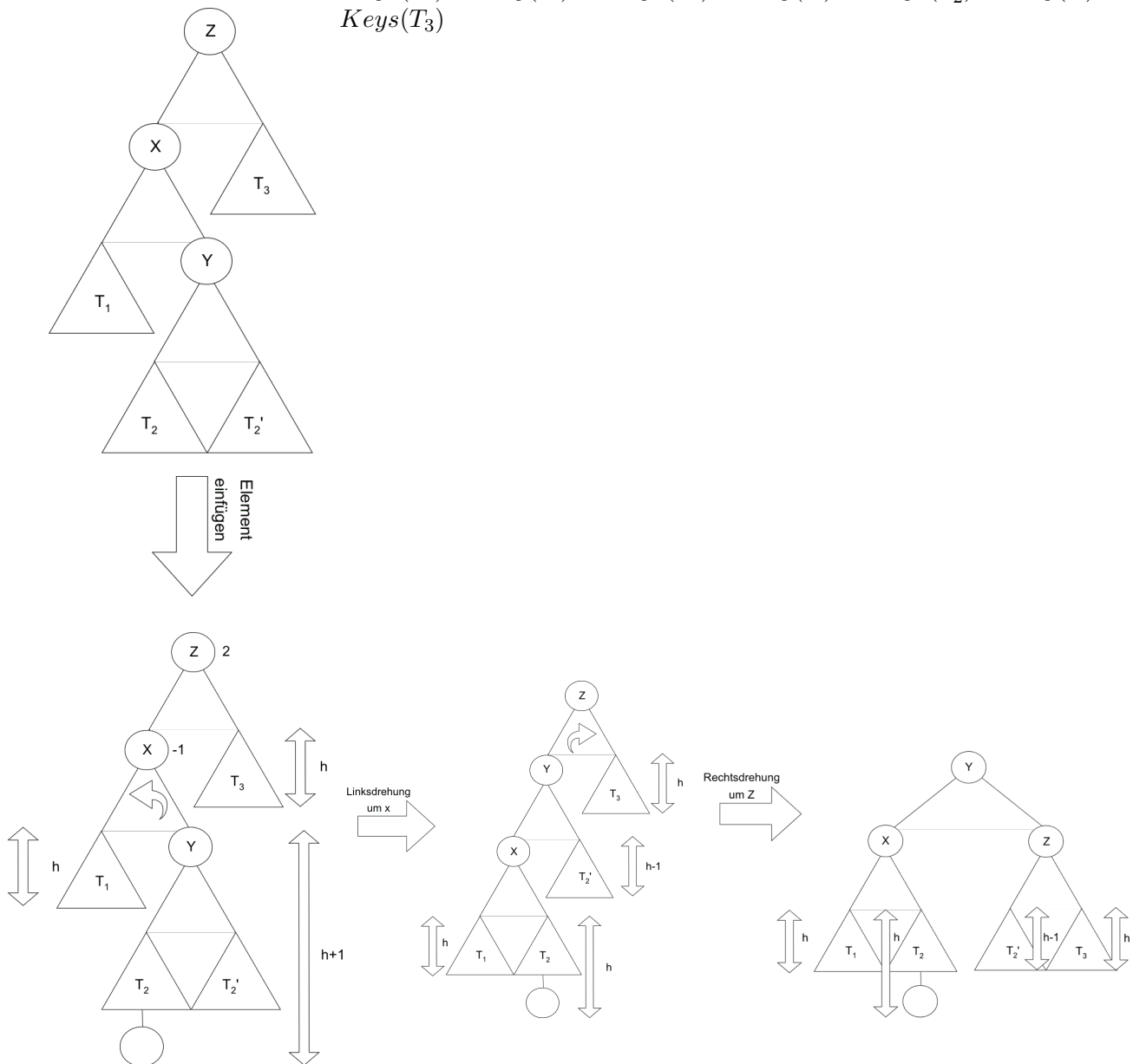
^I $f(h)$ meint hierbei die h -te Fibonacci-Zahl

5.2. Rotationen



$Keys(T_1) < Key(X) < Keys(T_2) < Key(Y) < Keys(T_3)$
 $balance(Y) = height(Y.left) - height(Y.right)$

$Keys(T_1) < Key(X) < Keys(T_2) < Key(Y) < Keys(T'_2) < Key(Z) < Keys(T_3)$



5.3. Pseudo-Code

```
class Node {
    int key;
    Node left , right;
    int height;
}

int height(Node node) {
    if (node == null) return 0;
    return height;
}

Node rotateRight(Node y) {
    Node x = y.left;
    Node T2 = x.right;
    y.left = T2;
    T2.right = y;
    y.height = 1+max(height(y.left), height(y.right));
    x.height = 1+max(height(x.left), height(x.right));
    return x;
}

Node rotateLeft(Node y) { //analog }

Node insert(Node node, int key) {
    if (node == null) return new Node(key);
    if (key < node.key)
        node.left = insert(node.left , key);
    else
        node.right = insert(node.right , key);

    if (balance(node)>1 && key < node.left.key)
        return rotateRight(node);
    if (balance(node)<-1 && key > node.right.key)
        return rotateLeft(node);
    if (balance(node)>1 && key > node.left.key) {
        node.left = rotateLeft(node.left);
        return rotateRight(node);
    }
    if (balance(node)<-1 && key < node.right.key) {
        node.right = rotateRight(node.right);
        return rotateLeft(node);
    }
    return node;
}
```

Anmerkung: Die Laufzeit des Einfügens bleibt in $O(\text{Baumtiefe}) = O(\log n)$. Nur einer der vier Fälle ist notwendig, um die Balance herzustellen.

6. Vorlesung 12

6.1. (a,b)-Suchbäume

Blattorientierte Speicherung der Elemente

Innere Knoten haben mindestens a und höchstens b Kinder und tragen entsprechende Schlüsselwerte, um die Suche zu leiten.

Beispiel:

$$h \hat{=} \text{Tiefe} \Rightarrow a^h \leq n \leq b^h \Rightarrow \log_b n \leq h \leq \log_a n$$

6.1.1. Aufspaltung bei Einfügen

6.1.2. Verschmelzen von Knoten beim Löschen

Aufspalte- und Verschmelze-Operationen können sich von der Blattebene bis zur Wurzel kaskadenartig fortpflanzen. Sie bleiben aber auf den Suchpfad begrenzt.

\Rightarrow Umbaukosten sind beschränkt durch die Baumtiefe $= O(\log n)$

6.2. Amortisierte Analyse

	000	
	001	Kosten(1) = 1
	010	= 2
	011	= 1
Beispiel: Binärzähler	100	= 3 Kosten der Inkrement-Operation $\hat{=}$ Zahl der Bit-Flips
	101	= 1
	110	= 2
	111	= 1
		<u>11</u>

Naive Analyse $2^k = n$

$$1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + 3 \cdot \frac{n}{8} + \dots + k \cdot \frac{n}{2^k} = \frac{n}{2} \sum_{i=1}^k i \left(\frac{1}{2}\right)^{i-1} = 2^{k+1} - k - 2 = 2n - k - 2$$

Von 0 bis n im Binärsystem zu zählen kostet $\leq 2n$ Bit-Flips

Sprechweise: amortisierte Kosten einer Inkrement-Operation sind 2

Folge von n -Ops kostet $2n$

6.2.1. Bankkonto-Methode

$$\text{Konto}(i+1) = \text{Konto}(i) - \text{Kosten}(i) + \text{Einzahlung}(i)$$

$$\sum_{i=1}^n \text{Kosten}(i) = \text{tatsächliche Gesamtkosten} = \sum_{i=1}^n (\text{Einzahlung}(i) + \text{Konto}(i) - \text{Konto}(i+1))$$

$$= \sum_{i=1}^n \text{Einzahlung}(i) + \text{Konto}(1) - \text{Konto}(n+1)$$

000	
001€	Kosten(1) = 1
01€0	= 2
01€1€	= 1
1€00	= 3
1€01€	= 1
1€1€0	= 2
1€1€1€	= 1
	$\overline{11}$

Kontoführungsschema: für Binärzähler

1€ pro 1 in der Binärdarstellung

Jeder Übergang $1€ \rightarrow 0$ kann dann mit dem entsprechenden Euro Betrag auf dieser 1 bezahlt werden.

Es gibt pro Inkrement Operation nur einen $0 \rightarrow 1$ Übergang

2€ Einzahlung für jede Inc-Operation reichen aus um:

1. diesen $0 \rightarrow 1$ Übergang zu bezahlen
2. die neu entstandene 1€ mit einem Euro zu besparen.

$$\text{GK} = 2(2^k - 1) + 0^{\text{I}} - k^{\text{II}} = 2n - k - 2$$

^IZählerstand(000)

^{II}Zählerstand($\overbrace{111 \dots 1}^k$)

7. Vorlesung 13

Satz: Ausgehend von einem leeren 2-5-Baum betrachten wir die Rebalancierungskosten C (Split- und Fusionsoperationen) für eine Folge von m Einfüge- oder Löschooperationen. Dann gilt: $C \in O(m)$
d.h. Amortisierte Kosten der Split- und Fusionsoperationen sind konstant.
! Dies bezieht sich nicht auf die Suchkosten, die in $O(\log n)$ liegen.

Beweisidee:

Kontoführung:

1	2	3	4	5	6
2€	1€	0€	0€	1€	2€

regelmäßige Einzahlung: 1€

Durch eine Einfüge- oder Löschooperation steigt oder fällt der Knotengrad des direkt betroffenen Knotens um höchstens 1. \Rightarrow 1€ Einzahlung reicht zur Aufrechterhaltung dieses Sparplanes.

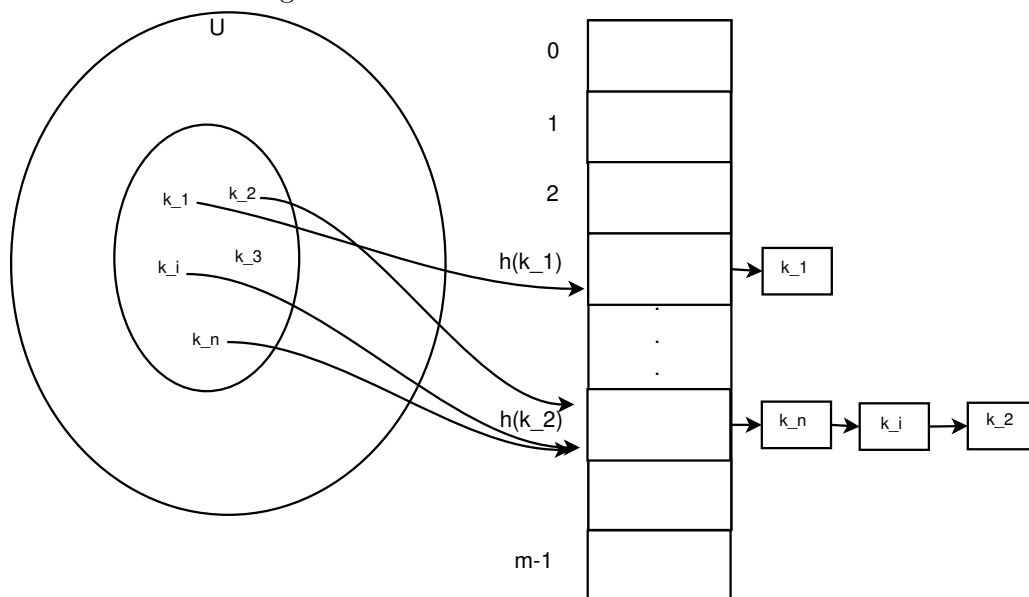
Jetzt Beseitigung der temporären 1- und 6-Knoten:

Ein 6-Knoten nutzt 1€ um seinen Split zu bezahlen. Die beiden neu entstehenden 3-Knoten benötigen kein Kapital. Der Vaterknoten des gesplitteten 6-Knotens benötigt ggf. den zweiten verfügbaren €.

Analoge Betrachtung für Fusion eines temp. 1-Knotens.

7.1. Hashing

Abbildung 7.1.: Universum und Hashtabelle der Größe m



$U \subseteq \mathbb{N}$ z.B. 64-Bit-Integer

n = Zahl der zu verwaltenden Schlüssel

$$|U| \gg n$$

Hashfunktion h :

$$h : U \rightarrow [0, \dots, m-1]$$

$$\text{z.B. } k \mapsto k \bmod m$$

Einfache Annahme: (einfaches uniformes Hashing)

$$\forall k_i, k_j \in U : Pr(h(k_i) = h(k_j)) = \frac{1}{m}$$

Analyse der Laufzeit zum Einfügen eines neuen Elementes k

- $h(k)$ berechnen $\rightarrow O(1)$
- Einfügen am Listenanfang in Fach $h(k)$. $\rightarrow O(1)$

Analyse der Suchzeit für einen Schlüssel k

- $h(k) \rightarrow O(1)$
- Listenlänge zum Fach $h(k)$ sei $n_{h(k)}$ also beim Durchlauf der kompletten Liste $\rightarrow O(n_{h(k)})$

$$E(n_{h(k)}) = \frac{n}{m} = \alpha^I$$

$$\text{Suchzeit(Einfügen)} \in O(1 + \alpha)$$

Laufzeit beim Löschen von Schlüssel k

- $h(k) \rightarrow O(1)$
- Durchlaufen der Liste $\rightarrow O(n_{h(k)})$
- Löschen durch „Pointer-Umbiegen“ $\rightarrow O(1)$

7.1.1. Universelles Hashing

Idee Arbeite nicht mit einer festen Hashfunktion sondern wähle am Anfang eine zufällige Hashfunktion aus einer Klasse von Hashfunktionen aus.

z.B.

$$h_{a,b}(k) = ((a \cdot k + b) \bmod p) \bmod m$$

p sei eine hinreichend große Primzahl $0 < a < p, 0 \leq b < p$

$$\mathcal{H}_{p,m} = \{h_{a,b}(k) | 0 < a < p, 0 \leq b < p\}$$

$$|\mathcal{H}_{p,m}| = p(p-1)$$

Definition \mathcal{H} heißt universell $\Leftrightarrow \forall k, l \in U : Pr(h(k) = h(l)) \leq \frac{1}{m}$

¹Belegungsfaktor

Suchzeit

$$x_{k,l} = \begin{cases} 1 & \text{für } h(k) = h(l) \\ 0 & \text{sonst} \end{cases}$$

$$E(n_{h(k)}) = E\left(\sum_{l \in T, l \neq k}\right) = \sum_{l \in T, l \neq k} E(X_{k,l}) = \sum_{l \in T, l \neq k} Pr(h(k) = h(l)) = \sum_{l \in T, l \neq k} \frac{1}{m} = \frac{n-1}{m} = \alpha$$

8. Vorlesung 14

Universelles Hashing (Fortsetzung)

Könnte ein boshafter Mitspieler n Schlüssel bei gegebener fester Hashfunktion wählen, so würde er solche wählen, die auf den gleichen Slot unter gegebener Hashfunktion abgebildet werden. \rightsquigarrow Durchschnittliche Ablaufzeit von $O(n)$

Idee zufällige Wahl der Hashfunktion aus einer Familie von Funktionen derart, dass die Wahl unabhängig von den zu speichernden Schlüssel ist (universelles Hashing).

8.0.1. Definition

Sei \mathcal{H} eine endliche Menge von Hashfunktionen, welche ein gegebenes Universum U von Schlüssel auf $\{0, \dots, m-1\}$ abbildet. Sie heißt universell, wenn für jedes Paar von Schlüssel $k, l \in U$ $l \neq k$ die Anzahl der Hashfunktionen $h \in \mathcal{H}$ mit $h(l) = h(k)$ höchstens $\frac{|\mathcal{H}|}{m}$. Anders: Für ein zufälliges $h \in \mathcal{H}$ beträgt die Wahrscheinlichkeit, dass zwei unterschiedliche Schlüssel k, l kollidieren nicht mehr als $\frac{1}{m}$ ist.

8.0.2. Beispiel

p Primzahl, so groß, dass alle möglichen Schlüssel $k \in U$ im $0, \dots, p-1$ liegen. $\mathbb{Z}/p\mathbb{Z}$ bezeichnet den Restklassenring $\text{mod } p$ (weil p prim, ist $\mathbb{Z}/p\mathbb{Z}$ ein Körper). $\mathbb{Z}/p\mathbb{Z}^*$ ist die Einheitengruppe.

Annahme: Die Menge der Schlüssel im Universum U ist größer als die Anzahl der Slots in der Hashtabelle. Für $a \in \mathbb{Z}/p\mathbb{Z}^*$ und $b \in \mathbb{Z}/p\mathbb{Z}$ betrachte:

$$h_{a,b}(k) := (a \cdot k + b \text{ mod } p) \text{ mod } m \quad (*)$$

Damit ergibt sich die Familie

$$\mathbb{Z}/p\mathbb{Z}^* = \{1, \dots, p-1\} \quad \mathbb{Z}/p\mathbb{Z} = \{0, \dots, p-1\} \quad \mathcal{H}_{p,m} = \{h_{a,b} | a \in \mathbb{Z}/p\mathbb{Z}^*, b \in \mathbb{Z}/p\mathbb{Z} \quad |\mathcal{H}| = p(p-1)\}$$

Satz Die in $(*)$ eingeführte Klasse von Hashfunktionen ist universell.

Beweis Seien k, l Schlüssel auf $\mathbb{Z}/p\mathbb{Z}$ mit $k \neq l$

Für $h_{a,b} \in \mathcal{H}_{p,m}$ betrachten wir

$$r = (a \cdot k + b) \text{ mod } p$$

$$s = (a \cdot l + b) \text{ mod } p$$

Es ist $r \neq s$

Dazu:

$$r - s = a \cdot (k - l) \text{ mod } p \quad (*2)$$

Angenommen $r - s = 0$

$$0 = a \cdot (k - l) \pmod{p}, \text{ aber } a \in \mathbb{Z}/p\mathbb{Z}^* \Rightarrow a \neq 0 \text{ und } k \neq l \Rightarrow k - l \neq 0$$

Da p prim ist $\mathbb{Z}/p\mathbb{Z}$ ein Körper \Rightarrow kein Nullteiler $\Rightarrow a \cdot (k - l) \neq 0 \Rightarrow r \neq s$

Daher bilden $h_{a,b} \in \mathcal{H}_{p,m}$ unterschiedliche Schlüssel k, l auf unterschiedliche Elemente ab. („Auf dem level \pmod{p} gibt es keine Kollisionen).

Aus (*2) folgt:

$$(r - s)(k - l)^{-1} = a \pmod{p}$$

$$r - a \cdot k = b \pmod{p} \text{ Bijektion zwischen } (k, l) \text{ und } (a, b)$$

Daher ist die Wahrscheinlichkeit, dass zwei Schlüssel $h \neq l$ kollidieren, gerade die Wahrscheinlichkeit, dass $r \equiv s \pmod{m}$, falls $r \neq s$ zufällig gewählt (aus $\mathbb{Z}/p\mathbb{Z}$).

Für gegebenes r gibt es unter den übrigen $p-1$ Werten für s höchstens $\lceil \frac{p-1}{m} \rceil \leq \lceil \frac{p}{m} \rceil - 1$ Möglichkeiten, sodass $s \neq r \pmod{p}$ aber $r = s \pmod{m}$

8.0.3. Abschätzung nach oben

$$\lceil \frac{p}{m} \rceil - 1 \leq \frac{(p + m - 1)}{m} - 1 = \frac{p - 1}{m} \text{ Kollisionsmöglichkeiten}$$

Die Wahrscheinlichkeit, dass r und s kollidieren \pmod{m} Kollisionsmöglichkeiten / Gesamtzahl der Werte

$$= \frac{p - 1}{m} \cdot \frac{1}{p - 1} = \frac{1}{m}$$

\Rightarrow Für ein Paar von Schlüsseln $k, l \in \mathbb{Z}/p\mathbb{Z}$ mit $k \neq l$

$$P[h_{a,b}(k) = h_{a,b}(l)] \leq \frac{1}{m} \Rightarrow \mathcal{H}_{p,m} \text{ universell!}$$

8.1. Perfektes Hashing

Wichtig Menge der Schlüssel ist im Vorhinein bekannt und ändert sich nicht mehr.

Beispiele reserved words bei Programmiersprachen, Dateinamen auf einer CD

8.1.1. Definition

Eine Hashmethode heißt perfektes Hashing, falls $O(1)$ Speicherzugriffe benötigt werden, um die Suche nach einem Element durchzuführen.

Idee Zweistufiges Hashing mit universellen Hashfunktionen.

1. Schritt n Schlüssel, m Slots durch Verwendung der Hashfunktion h , welche aus einer Familie universeller Hashfunktionen stammt.
2. Schritt Statt einer Linkedlist im Slot anzulegen, benutzen wir eine kleine zweite Hashtabelle S_j mit Hashfunktion h_j

Bild Schlüssel $k = \{10, 22, 37, 49, 52, 60, 72, 75\}$

Äußere Hashfunktion $h(k) = ((a \cdot b) \bmod p) \bmod m$

$$a = 3, \quad b = 42, \quad p = 101, \quad m = 9$$

$$h(10) = \underbrace{(3 \cdot 10 + 42 \bmod 101)}_{=72} \bmod 9 = 0$$

Um zu garantieren, dass keine Kollision auf der zweiten Ebene auftreten, lassen wir die Größe von S_j

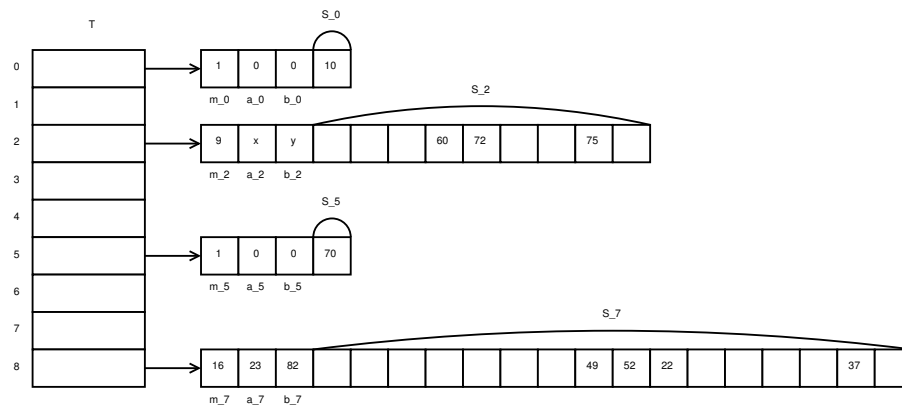


Abbildung 8.1.: Perfekte Hashtabelle

gerade n_j^2 sein ($n_j \neq \#\text{Schlüssel} \rightarrow j\text{Slot}$).

Wir verwenden für die Hashfunktion der ersten Ebene eine Funktion aus $\mathcal{H}_{p,m}$. Schlüssel die im j-ten Slot werden in der sekundären Hashtabelle S_j der Größe m_j mittels h_j gehasht. $h_j \in \mathcal{H}_{p,m}$

Wir zeigen: 2 Dinge:

1. Wie versichern wir, dass die zweite Hashfunktion keine Kollision hat.
2. Der erwartete Speicherbedarf ist $O(n)$

zu 1.

Satz Beim Speichern von n Schlüsseln in einer Hashtabelle der Größe $m = n^2$ ist die Wahrscheinlichkeit, dass eine Kollision auftritt $< \frac{1}{2}$

Beweis: Es gibt $\binom{n}{2}$ mögliche Paare, die kollidieren können. Jedes kollidiert mit der Wahrscheinlichkeit $\leq \frac{1}{m}$, falls $h \in \mathcal{H}$ zufällig gewählt wurde.

Sei X eine zufallsvariable(ZV), X zählt Kollisionen:

Für $m = n^2$ ist die erwartete Zahl der Kollisionen:

$$E[X] = \binom{n}{2} \cdot \frac{1}{m} = \binom{n}{2} \cdot \frac{1}{n^2} = \frac{n!}{2!(n-2)!n^2} = \frac{(n-1)}{2n} \leq \frac{1}{2}$$

Anwenden der Markow-Ungleichung ($a=1$):

$$P[X \geq 1] \leq \frac{E[X]}{1} = \frac{1}{2} \Rightarrow \text{Wahrscheinlichkeit für irgendeine Kollision ist } < \frac{1}{2}$$

q.e.d

8.1.2. Nachteil

Für große n ist $m = n^2$ nicht haltbar!

zu 2. Wenn die Größe der primären Hashtabelle $m = n$ ist, dann ist der Platzverbrauch in $O(n) \rightsquigarrow$ Betrachte Platzverbrauch der sekundären Hashtabellen.

Satz Angenommen wir wollen n Schlüssel in einer Hashtabelle der Größe $m = n$ mit Hashfunktion $h \in \mathcal{H}$. Dann gilt:

$$E \left[\sum_{j=0}^{m-1} n_j^2 \right] < 2n$$

Beweis

Betrachte

$$a^2 = a + 2 \cdot \binom{a}{2} = a + 2 \cdot \frac{a^2 - a}{2} \quad (*3)$$

Betrachte

$$E \left[\sum_{j=0}^{m-1} n_j^2 \right] \stackrel{(*3)}{=} E \left[\sum_{j=0}^{m-1} \left(n_j + 2 \binom{n_j}{2} \right) \right]$$

$$\stackrel{\text{lini. des EW}}{=} E \left[\underbrace{\sum_{j=0}^{m-1} n_j}_{=n} \right] + 2E \left[\sum_{j=0}^{m-1} \binom{n_j}{2} \right] = n + 2E \left[\sum_{j=0}^{m-1} \binom{n_j}{2} \right] \# \text{ der Kollisionen}$$

Da unsere Hashfunktion universell ist, ist die erwartete Zahl dieser Paare:

$$\binom{n}{2} \frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2}, \text{ da } m = n$$

Somit

$$E \left[\sum_{j=0}^{m-1} n_j^2 \right] \leq n + 2 \frac{n-1}{2} = 2n - 1 < 2n$$

Korollar Speichern wir n Schlüssel in einer Hashtabelle der Größe $m = n$ mit einer zufälligen universellen Hashfunktion und setzen die Größe der Hashtabellen der zweiten Ebene auf $m_j = n_j^2$ für $j = 0, m = 1$, so ist der Platzverbrauch des perfekten Hashings weniger als $2n$. Die Wahrscheinlichkeit, dass der Platzverbrauch der zweiten Hashtabellen $\geq 4n$ ist, ist $\leq \frac{1}{2}$ ohne Beweis.

9. Vorlesung 15

Bei n Elementen sollte die Hashtabelle $m = n^2$ groß sein.
Für die universellen Hashfunktionen

$$\mathcal{H}_{p,m} = \{h_{a,b}(k) = (a \cdot k + b) \bmod p \mid 0 < a < p, 0 \leq b < p\}$$

$\binom{n}{1}$ Schlüsselpaare (k, l) mit $k \neq l$

$$E(\# \text{Kollisionen}) \leq \binom{n}{2} \cdot \frac{1}{m} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} \leq \frac{1}{2}$$

Idee Zweistufiges Verfahren:

- primäre Hashfunktion für Tabelle der Größe $m = n$

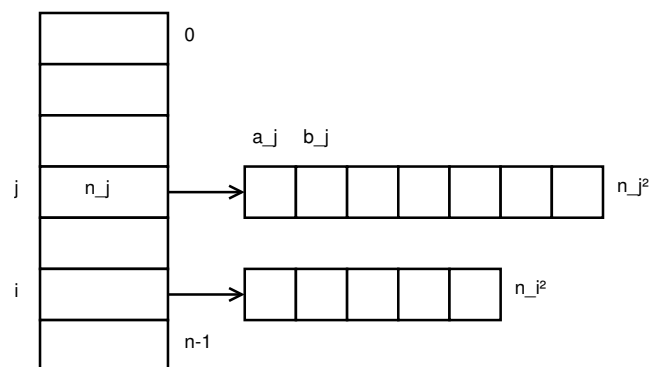


Abbildung 9.1.: Perfektes Hashing

¹Universalität von \mathcal{H}

Teil II.

Graphen-Algorithmen

9.0.1. Einführung

$$G = (V, E) \quad V \text{ vertices, } E \text{ edges} \quad E \subseteq V \times V$$

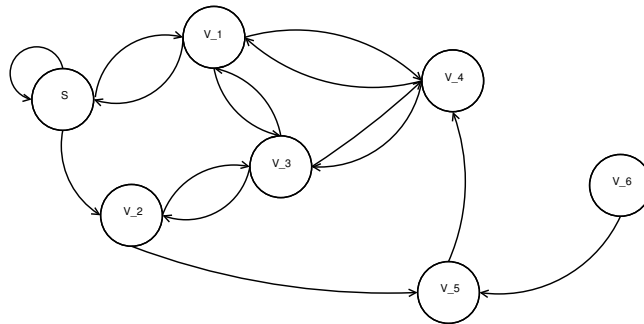


Abbildung 9.2.: Gerichteter Graph

Planare Graphen können ohne Überkreuzung der Kanten in die Ebene eingebettet werden.

Eulerische Polyederformel

$$|V| + |F| = |E| + 2$$

$$8 + 6 = 12 + 2$$

Es gilt:

$$2 \cdot |E| \geq 3 \cdot |F|$$

$$\# \text{gerichtete Kanten} = 2 \cdot |E| = \sum_{i=1}^{|F|} \# \text{Kanten}(f_i)^{\text{II}} \geq 3 \cdot |F|$$

$$|F| \leq \frac{2}{3}|E|, \quad |V| + |F| = |E| + 2 \leq |V| + \frac{2}{3}|E| \Rightarrow \frac{1}{3}|E| + 2 \leq |V|$$

$$\Rightarrow |E| \leq 3 \cdot |V| - 6$$

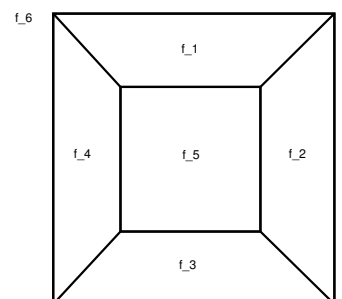


Abbildung 9.3.: Würfel

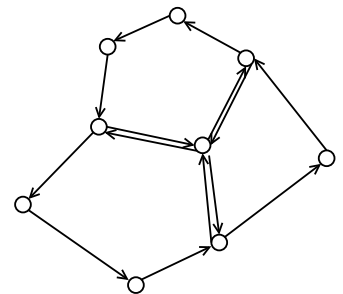


Abbildung 9.4.: Placeholder

^{II}Jedes f_i hat mindestens 3 Kanten

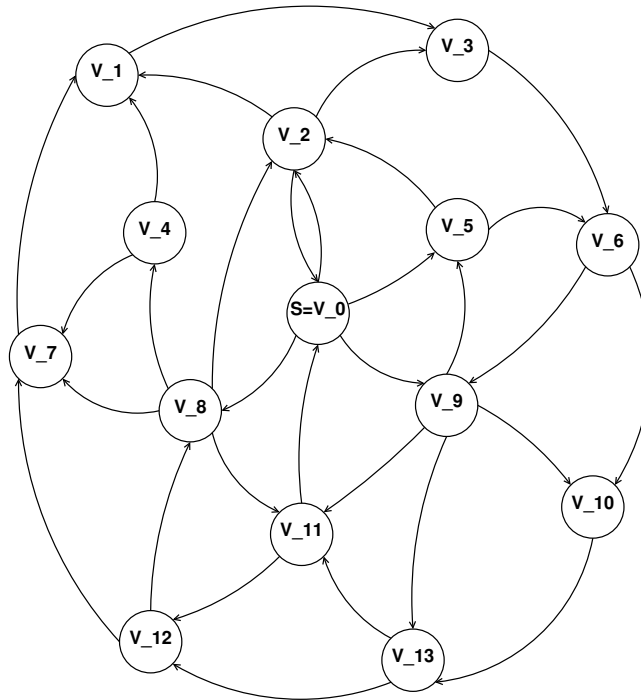


Abbildung 9.5.: Beispiel

Adjazenzmatrix

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	0	1	0	0	1	0	0	1	1	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	1	0	0	0	0	0	0	0
4	0	1	0	0	1	0	0	1	0	0	0	0	0	0
5	0	0	1	0	0	1	1	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	1	1	0	0	0
7	0	1	0	0	0	0	0	1	0	0	0	0	0	0
8	0	0	1	0	1	0	0	1	1	0	0	0	0	0
9	0	0	0	0	0	1	0	0	0	1	1	1	0	1
10	0	0	0	0	0	0	0	0	0	0	1	0	0	1
11	1	0	0	0	0	0	0	0	0	0	0	1	1	0
12	0	0	0	0	0	0	0	1	1	0	0	0	1	0
13	0	0	0	0	0	0	0	0	0	0	0	1	1	1

$= A$

$$a \in B^{|V| \times |V|}$$

falls G ungerichtet $\Rightarrow A = A^T$

Adjazenzlisten Repräsentation

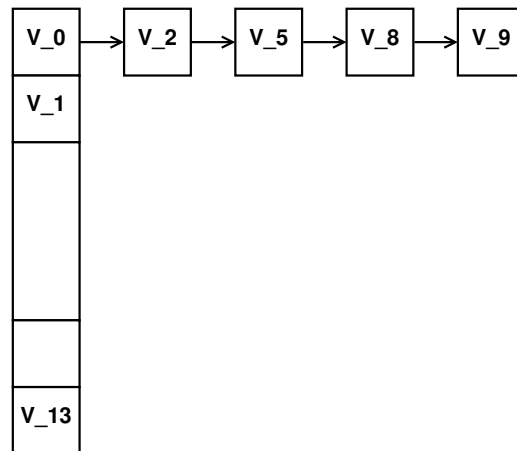


Abbildung 9.6.: Adjazenzliste

Platzbedarf

$$\mathcal{O}(|V| + |E|) = \mathcal{O} \left(|V| + \sum_{i=0}^{|V|-1} \text{outdeg}(v_i) \right)$$

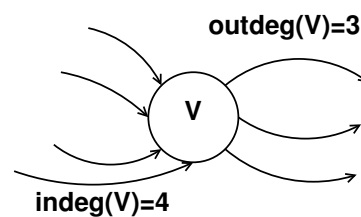


Abbildung 9.7.: indeg und outdeg

9.0.2. BFS (Breadth-First Search) Breitensuche

```

forall ( v in V \ {S} ) {
    col[v] = white;    // Farbe  weiß = unbekannt, grau = bekannt, schwarz = vollkomm
    d[v] = infinity;  // Distanz
    pi[v] = NULL;     // pi ist Vorgänger
}
col[s] = grey;       // s ist Startknoten
d[s] = 0;
pi[s] = null;

```

Queue	vs	Stack
Schlange		Stapel
empty()		"
push()		"
pop()		
FIFO		FILO
First-In-First-Out		First-In-First-Out

```

Queue Q;
Q.push(s);
while (!Q.empty()) {
    u = Q.pop();
    forall( (u,v) in E) {
        if (col[v] == white) {
            col[v] = grey;
            d[v] = d[u]+1;
            pi[v] = u;
            Q.push(v);
        }
    }
    col[u] = black;
}

```

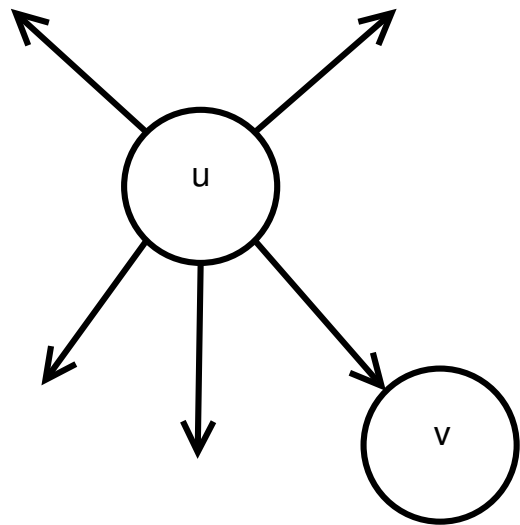


Abbildung 9.8.: Grafik zum Beispielcode

Laufzeit

$$\mathcal{O}(|V| + |E|)$$

Begründung: Jeder von s aus erreichbare Knoten wird nur einmal in die Queue aufgenommen und auch ihr entfernt. Für jeden Knoten muss nur einmal seine Adjazenzliste durchlaufen werden.

$$\Rightarrow \mathcal{O} \left(|V| + \sum_{v \in V} \text{outdeg}(v) \right)$$

10. Vorlesung 16

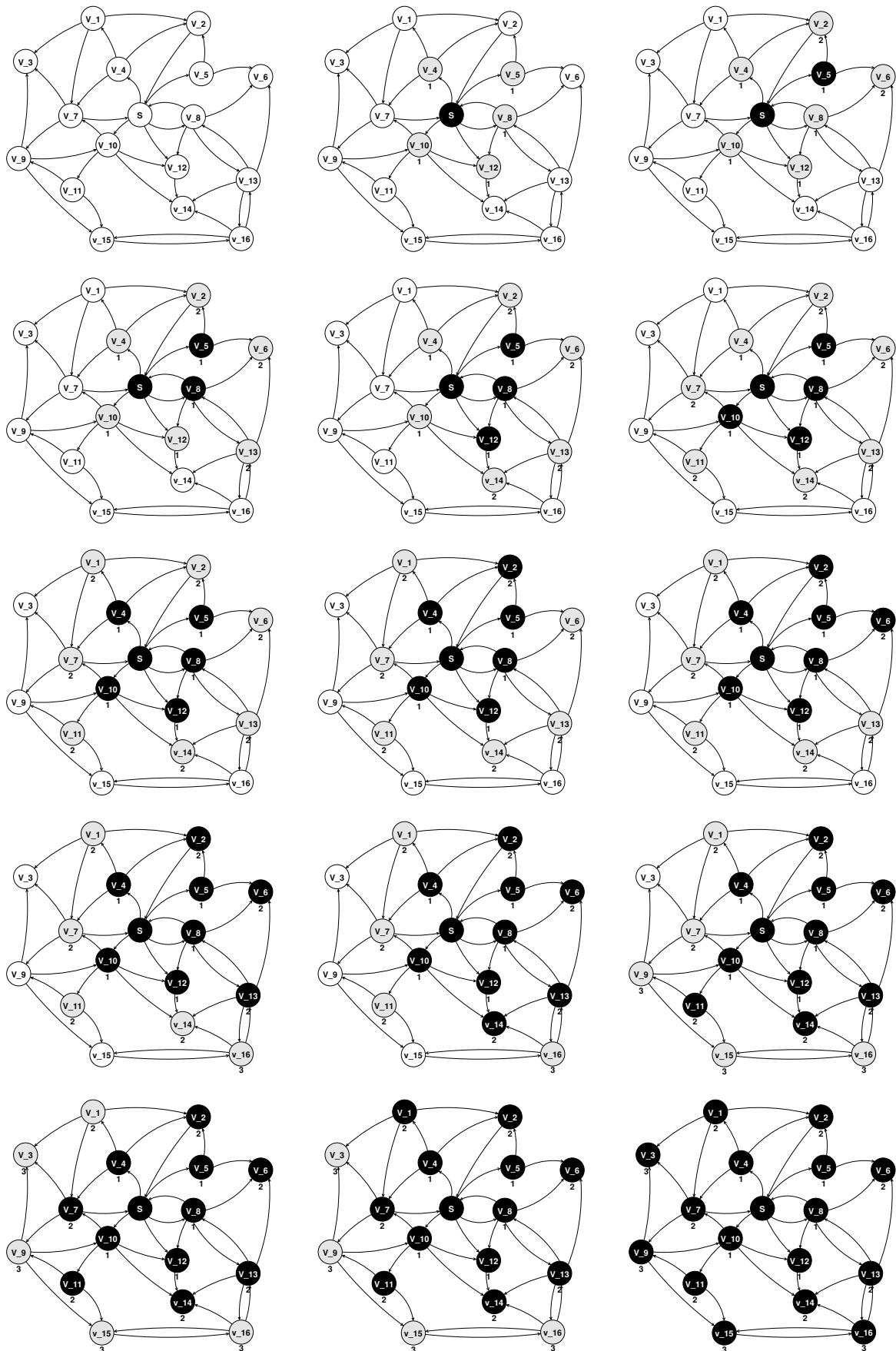


Abbildung 10.1.: Beispiel

Definition: Länge kürzesten Weges

$\delta(s, v)$ = Länge eines kürzesten Weges vom Startknoten s zum Knoten v .
Setze $\delta(s, v) = \infty$, falls v nicht erreichbar von s aus.

Satz: Richtigkeit des Algorithmus

Nach Ablauf von BFS^I gilt

$$\forall v \in V : d[v] = \delta(s, v)$$

Lemma 1: Dreiecksungleichung für kürzeste Wege

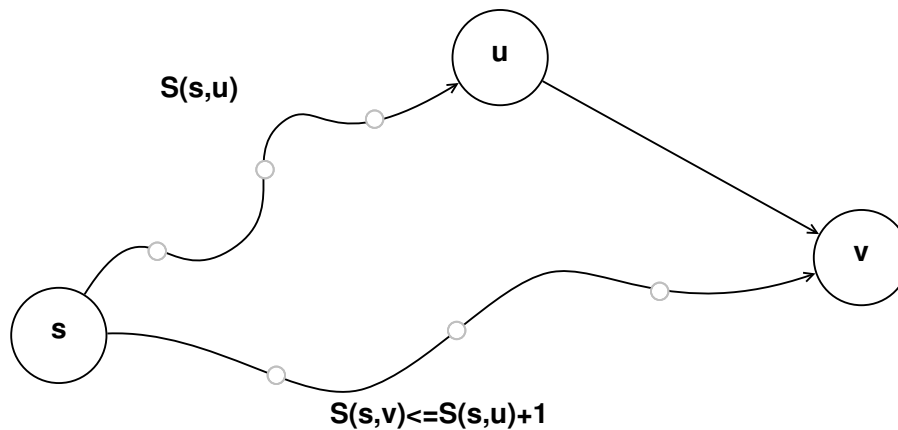


Abbildung 10.2.

Lemma 2

Zu jedem Zeitpunkt im Verlauf von BFS gilt:

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

Beweis (induktiv über Zahl der Operationen, die d-Wert verändern)

Induktions-Anfang

$$d[s] = 0$$

Induktions-Schritt Knoten v wird von u aus neu entdeckt

$$d[u] \geq \delta(s, u)$$

$$d[v] = d[u] + 1 \geq \delta(s, u) + 1 \stackrel{D.U.}{\geq} \delta(s, v)$$

Lemma 3

Sei $Q = (v_1, v_2, \dots, v_k)$ eine Queue, dann gilt stets:

$$d[v_1] \leq d[v_2] \leq \dots \leq d[v_k] \leq d[v_1] + 1$$

^IBreitensuche

Beweis (induktiv über die Zahl der push- und pop-Operationen)

Induktions-Anfang

$$d[s] = 0 \checkmark$$

Induktions-Schritt

pop

$$d[v_1] \leq d[v_2] \leq \dots \leq d[v_k] \leq d[v_1] + 1 \stackrel{!}{\leq} d[v_2] + 1$$

push

$$d[u] = d[v_1] \leq d[v_2] \leq \dots \leq d[v_k] \leq d[u] + 1$$

Beachte Kante (u, v) v ist weiß

$v = v_{k+1}$ wird gepushed

$$d[v_{k+1}] = d[v_1] + 1$$

Zustand von Q nach push

$$d[v_2] \leq d[v_3] \leq \dots \leq d[v_k] \leq d[v_1] + 1 = d[v_{k+1}] \quad \checkmark$$

Satz: Richtigkeit des Algorithmus

Nach Ablauf von BFS^{II} gilt

$$\forall v \in V : d[v] = \delta(s, v)$$

Beweis durch Widerspruch

Sei $v \in V$, so dass $d[v] \neq \delta(s, v)$ am Ende des Algorithmus $\stackrel{\text{Lemma 2}}{\implies} d[v] > \delta(s, v)$

Sei v so gewählt, dass es der erste Knoten ist mit der Eigenschaft, dass sein d-Wert falsch gesetzt wird.

d.h. Alle d-Werte bis zu diesem Zeitpunkt sind korrekt.

Sei $s \mapsto u' \rightarrow v$ ein kürzester Weg s zu v

Betrachte die Situation bei Bearbeitung von u' :

1. Fall v ist in diesem Moment schwarz.

$$d[v] > \delta(s, v) = \delta(s, u') + 1 \geq \text{III} d[v] \quad \text{!}$$

2. Fall v ist in diesem Moment weiß.

$$d[v] > \delta(s, u') + 1 = d[u'] + 1 = \text{IV} d[v] \quad \text{!}$$

^{II}Breitensuche

^{III} v vor u' aus Q entfernt und Lemma 3.

^{IV}wegen Wahl von v ; d-Wert von u' muss also korrekt sein

3. Fall v ist grau.

$$d[v] > \delta(s, u') + 1 = d[u'] + 1 \geq d[u] + 1 = d[v]$$

$d[u] \leq d[u']$, weil u vor u' aus Q entfernt \nmid

q.e.d.

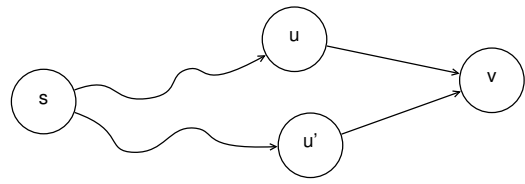


Abbildung 10.3.

10.1. Kürzeste Wege Algorithmen

10.1.1. Dijkstra-Algorithmus

$$G = (V, E) \quad w : E \rightarrow \mathbb{R}_0^+$$

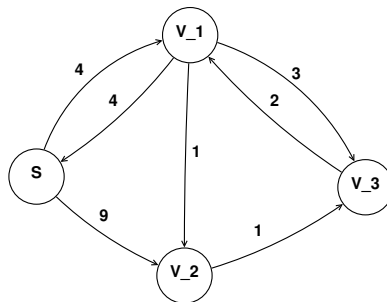


Abbildung 10.4.

Sei $p = (s = v_0, v_1, v_2, \dots, v_k)$



Abbildung 10.5.

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}) = \delta(s, v_k)$$

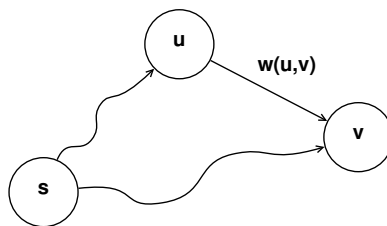


Abbildung 10.6.

$$\delta(s, v) \leq \delta(s, u) + w(u, v)$$

```

relax(u, v, w) {
  if (d[v] > d[u] + w(u, v)) {
    d[v] = d[u] + w(u, v);
    Π[v] = u;
  }
}

```

Betrachte Algorithmen zur kürzesten Wege Berechnung, die Distanzwerte nur mit Hilfe dieser relax-Funktion verändern, dann gilt:

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

Beweis

$$d[v] = d[u] + w(u, v) \stackrel{I.A.}{\geq} \delta(s, u) + w(u, v) \geq \delta(s, v)$$

Induktion über Zahl der reflex-Aufrufe

11. Vorlesung 17

Dijkstra Algorithmus (Fortsetzung)

$$G = (V, E) \quad w : E \rightarrow \mathbb{R}^{\geq 0}$$

```
forall (v ∈ V) {  
    d[v] = ∞;  
    Π[v] = NULL;  
}  
d[s] = 0;  
S = ∅;  
PriorityQueue PQ;  
forall (v ∈ V)  
    PQ.insert((d[v], v));  
while (!PQ.empty()) {  
    u = PQ.deleteMin();  
    forall ( (u, v) ∈ E ) {  
        if ( d[v] > d[u] + w(u, v) ) {  
            d[v] = d[u] + w(u, v);  
            Π[v] = u;  
            PQ.decreaseKey((d[v], v));  
        }  
    }  
    S = S ∪ {u};  
}
```

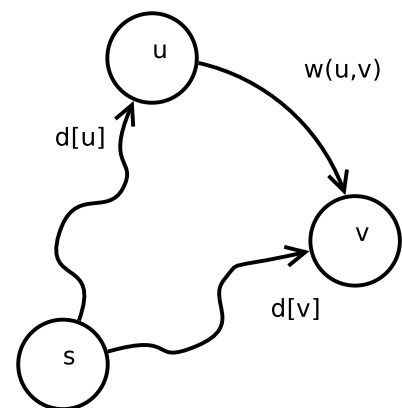


Abbildung 11.1.

Satz: Der Dijkstra Algorithmus berechnet alle d-Werte, so dass nach Ablauf des Algo $\forall v \in V$ gilt: $d[v] = \delta(s, v)$.

Beweis:

Annahme:

$$\exists v \in V : d[v] \neq \delta(s, v)$$

$$\xRightarrow{\text{LemmaRelax}} d[v] > \delta(s, v)$$

Sei v so gewählt, dass v der erste Knoten mit der Eigenschaft ist, der mit deleteMin der PQ entnommen wird und nach Relaxation aller von ihm ausgehenden Kanten der Menge S hinzugefügt wird.

Betrachte einen kürzesten Weg $s \rightsquigarrow v$

$$d[v] > \delta(s, v) \geq \text{I} \delta(s, y) = d[y] = \text{II} d[x] + w(x, y) = d[y] \geq \text{III} d[v] \quad \nexists$$

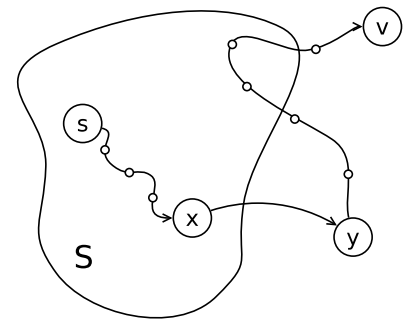


Abbildung 11.2.: Skizze

11.0.1. Vorläufige Laufzeitanalyse von Dijkstra

PQ.insert	x	V
PQ.empty	x	V
PQ.deleteMin	x	V
PQ.decreaseKey	x	E

Mit balanciertem Suchbaum oder mit binärem Heap (siehe Heapsort)

können diese Operationen alle in Zeit $\mathcal{O}(\log |V|)$ realisiert werden.

\Rightarrow Gesamtlaufzeit: $\mathcal{O}((|V| + |E|) \log |V|)$

Wir werden später zeigen, dass Laufzeit $\mathcal{O}(|V| \log |V| + |E|)$ möglich ist.

11.1. Bellman-Ford-Algorithmus

$$G = (V, E) \quad w : E \rightarrow \mathbb{R}$$

Voraussetzung G enthält keine negativen Zyklen

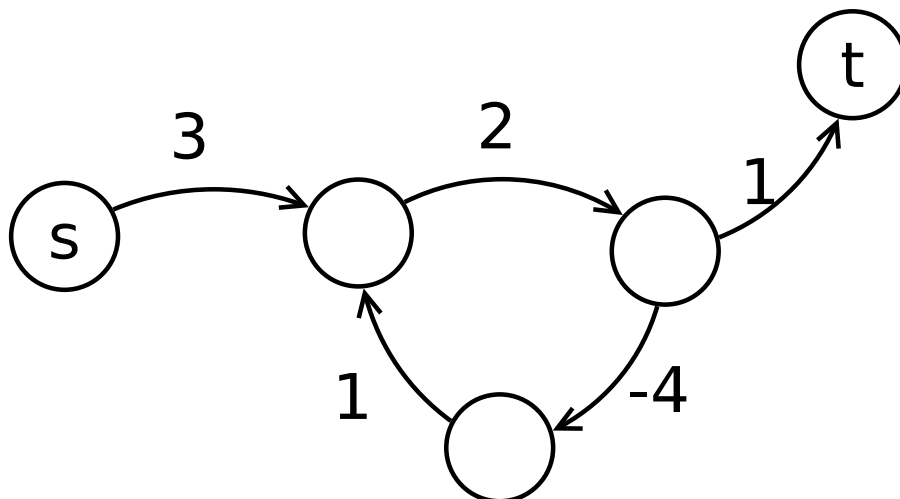


Abbildung 11.3.: Ein verbotener, negativer Zyklus

^Iweil Kantengewichte nicht negativ sein dürfen

^{II}x wurde schon zu S hinzugefügt, hat also Korrekten d-Wert $d[x] = \delta(s, x)$

^{III}weil v vor y aus der PQ entnommen wird.

11.1.1. Pseudocode

```
forall (v ∈ V) {
    d[v] = ∞;
    Π[v] = NULL;
}
d[s] = 0;
for (i = 1; i < |V|; i++)
    forall ((u, v) ∈ E)
        if (d[v] > d[u] + w(u, v)) {
            d[v] = d[u] + w(u, v);
            Π[v] = u;
        }
```

11.1.2. Laufzeit: Bellman-Ford

$$\mathcal{O}(|V| \cdot |E|)$$

11.1.3. Korrektheitsbeweis: Bellman-Ford

Invariante: Nach den i -ten Schleifendurchlauf sind alle Kürzesten Wege korrekt berechnet, die $\leq i$ Kanten benutzen.

Beweis: Induktion über i

Induktionsanfang

$i = 0$ $d[s] = 0 = \delta(s, s)$, da keine negativen Zyklen vorliegen.

11.1.4. Induktionsschritt: $i \rightarrow i + 1$

Betrachte kürzesten Weg mit $i + 1$ Kanten:

$$s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_i \rightarrow v_{i+1}$$

Aufgrund der Induktionsannahme^{IV} gilt: $d[v_i] = \delta(s, v_i)$, weil $s = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_i$ ein kürzester Weg $s \rightsquigarrow v_i$ mit i Kanten ist. Da alle Kanten in der inneren Schleife einmal relaxiert werden, trifft dies insbesondere auf die Kante (v_i, v_{i+1}) zu:

$$d[v_{i+1}] = d[v_i] + w(v_i, v_{i+1}) = \delta(s, v_i) + w(v_i, v_{i+1}) = \delta(s, v_{i+1})$$

Frage: Warum folgt aus der Gültigkeit dieser Invariante die Korrektheit des Algo?

Antwort Alle kürzesten Wege benutzen höchstens $|V| - 1$ Kanten, ansonsten hätten sie einen Zyklus mit Gewicht ≥ 0 , den man auch weglassen kann.

```
//Erkennung der Existenz negativer Zyklen
forall ((u, v) ∈ E)
    if (d[v] > d[u] + w(u, v))
        negativer Zyklus
```

^{IV}Die Invariante

12. Vorlesung 18

12.1. All-Pairs-Shortest Path Algorithmen

Distanzmatrix D für einen Graphen $G = (V, E)$ $V = v_1, v_2, \dots, v_n$, $w : E \mapsto \mathbb{R}$

$$d_{ij} = \begin{cases} 0 & \text{für } i = j \\ w(v_i, v_j) & \text{für } (v_i, v_j) \in E \\ \infty & \text{sonst} \end{cases}$$

$$D = (d_{ij})_{\substack{i=1,\dots,n \\ j=1,\dots,n}} \in \mathbb{R}^{n \times n}$$

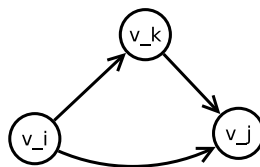


Abbildung 12.1.: Grafik

$$d_{ij}^{(2)} = \min(d_i^{(1)} j, \min_{k=1,\dots,n} (d_{ik}^{(1)} + d_k^{(1)} j))$$

$$D^{(2)} = D^{(1)} \circ D^{(1)} = \min(d_{ik}^{(1)} + d_k^{(1)} j)$$

Vergleich zu Matrixmultiplikation

$$C = A \circ B, \text{ mit } A, B \in \mathbb{R}^{n \times n}$$

$$C_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$$

im Ring $(\mathbb{R}, +, \cdot)$

$$C_{ij} = \left(\sum_{k=1,\dots,n} A_{ik} + B_{kj} \right)$$

Kommutativgesetz

$$\min(\min(a, b), c) = \min(a, b, c)$$

im „Ring“¹ $(\mathbb{R}, \min, +)$

Distributivgesetz

$$a + \min(b, c) = \min(a + b, a + c)$$

¹der keiner ist

Assoziativgesetz

$$A \circ (B \circ C) = (A \circ B) \circ C$$

Ziel: $D^{(n)\text{II}} = D^{(1)} \circ D^{(1)} \circ \dots \circ D^{(1)}$

Es gilt: $D^{(n)} = D^{(n+m)}$ für $m \geq 1$

12.1.1. Laufzeit zur Berechnung von $D^{(n)}$

Naiv: $\mathcal{O}(n^4)$

$$D^{(2)} = D^{(1)} \circ D^{(1)}$$

$$D^{(4)} = D^{(2)} \circ D^{(2)}$$

$$D^{(8)} = D^{(4)} \circ D^{(4)}$$

$$\vdots$$

$$D^{(2^i)} = D^{(2^{i-1})} \circ D^{(2^{i-1})}$$

Schrittzahl i so wählen, dass $2^i \geq n$

sukzessives Quadrieren: $\mathcal{O}(n^3 \log n)$

12.2. Floyd-Warshall-Algorithmus

```
for ( k = 1; k ≤ n; k++)  
  for ( i = 1; i ≤ n; i++)  
    for ( j = 1; j ≤ n; j++)  
      d[i][j] = min(d[i][j], d[i][k]+d[k][j])
```

Laufzeit $\mathcal{O}(n^3)$

12.2.1. Korrektheitsbeweis:

Invariante Nach dem k -ten Schleifendurchlauf entspricht d_{ij} der Weglänge eines kürzesten Weges p von v_i nach v_j , wobei nur Zwischenknoten erlaubt sind, mit Index $\leq k$

$$p: v_i \rightarrow v_{l_1} \rightarrow v_{l_2} \rightarrow \dots \rightarrow v_{l_m} \rightarrow v_j$$

d.h. $1 \leq l_1, l_2, \dots, l_m \leq k$

^{II}In der Potenz stehen die Anzahl der betrachteten Kanten. n entspricht allen Kanten

12.2.2. Beweis der Invariante durch Induktion nach k

$k = 0$: Nach der Initialisierung von D , also vor dem 1. Schleifendurchlauf, gilt obige Invariante.

$k - 1 \rightarrow k$:

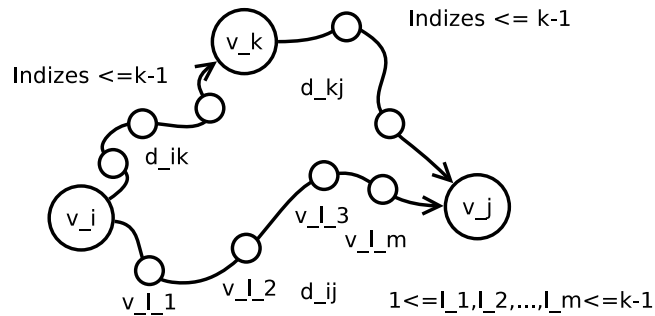


Abbildung 12.2.: Beweis der invariante

Durch die Operation $d_{ij} = \min(d_{ij}, d_{ik} + d_{kj})$ wird die Invariante sichergestellt.

12.3. Naive Lösung des All-Pairs Problems durch $|V|$ -malige Anwendung von Bellman-Ford oder Dijkstra-Algorithmus

Bellman-Ford $\mathcal{O}(|V| \cdot |V| \cdot |E|) = \mathcal{O}(|V|^2 \cdot |E|)$

Dijkstra $\mathcal{O}(|V| \cdot (|V| \cdot \log |V| + |E|)) = \mathcal{O}(|V| \cdot |E| + |V|^2 \cdot \log |V|)$

12.4. Johnson-Algorithmus

Idee: Neugewichtung der Kanten, so dass keine negativen Kantengewichte mehr vorhanden sind. Anschließend $|V|$ -mal Dijkstra-Algorithmus ausführen.

Naiver Ansatz

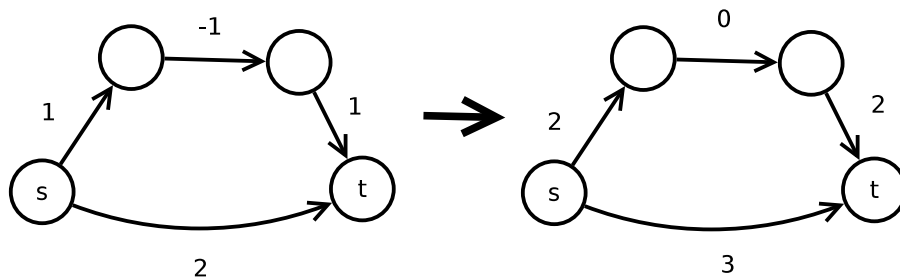


Abbildung 12.3.: Naiver Ansatz, kürzester Weg wird zerstört

Neuer Ansatz

$$w'(u, v) = \text{pot}^{\text{III}}(u) - \text{pot}(v) + w(u, v) \geq 0$$

Mit dieser Neugewichtung gilt, dass kürzeste Wege bzgl. w den kürzesten Wegen bzgl. w' entsprechen.

$$p : s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots v_i \rightarrow v_{i+1} \rightarrow \dots v_k = t$$

$$w'(p) = \sum_{i=0}^{k-1} w'(v_i, v_{i+1}) = \sum_{i=0}^{k-1} [\text{pot}(v_i) - \text{pot}(v_{i+1}) + w(v_i, v_{i+1})]$$

$$\stackrel{\text{Teleskopsumme}}{=} \text{pot}(v_0) - \text{pot}(v_k) + \sum_{i=1}^{k-1} w(v_i, v_{i+1}) = \text{pot}(s) - \text{pot}(t) + w(p)$$

d.h. Alle kürzesten Wege $s \rightsquigarrow t$ unterscheiden sich bzgl. w' im Vergleich zu w nur um eine feste additive Konstante $\text{pot}(s) - \text{pot}(t)$

$$\text{pot}(u) - \text{pot}(v) + w(u, v) \geq 0$$

$$\text{pot}(v) \leq \text{pot}(u) + w(u, v)^{\text{IV}}$$

$$\text{pot}(v) = \delta(z, v)$$

$$G' = (V', E') \quad V' = V \cup z, E' = E \cup (z, v) | v \in V \quad \text{mit } w'(z, v) = 0$$

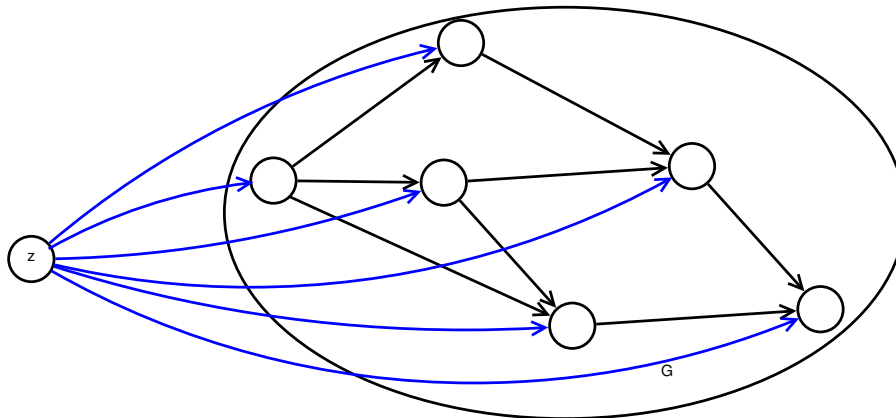


Abbildung 12.4.: Die blau markierten Kanten haben die Länge 0

- Löse single-source-shortest-Path Problem in G' mit z als Startknoten
- setze $\text{pot}(v) = \delta_{G'}(z, v)^{\text{V}}$
- Neugewichtung
- $|V|$ -mal Dijkstra

12.4.1. Laufzeit des Johnson-Algorithmus

$$\mathcal{O}(|V| \cdot |E| + |V| \cdot (|V| \cdot \log |V| + |E|)) = \mathcal{O}(|V| \cdot |E| + |V|^2 \cdot |V|)$$

^{III}Potentialfunktion

^{IV}Dreiecksungleichung

^Vberechnet mit Bellman-Ford