Datenstrukturen und effiziente Algorithmen

Markus Vieth, David Klopp, Christian Stricker

12. Dezember 2015

Inhaltsverzeichnis

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,...,n-i enthalten die korrekt sortierten Feldelemente

Beweis durch Induktion nach i $\overset{i=n-1}{\Longrightarrow}$ 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
$1+2+3+\ldots+(n+1)$		

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

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 Indes 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 11

3.1. AVL-Bäume von Adelson-Velskii and Landis

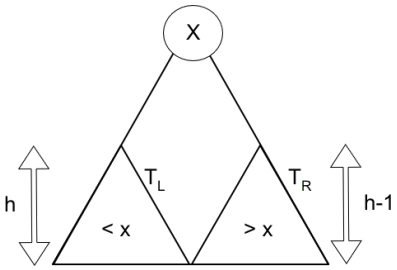


Abbildung 3.1.

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

3.1.1. 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.

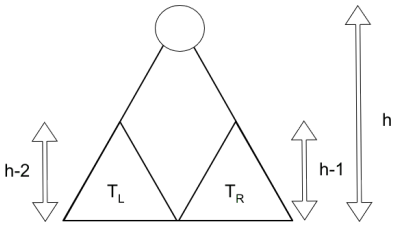


Abbildung 3.2.

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

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

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

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

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

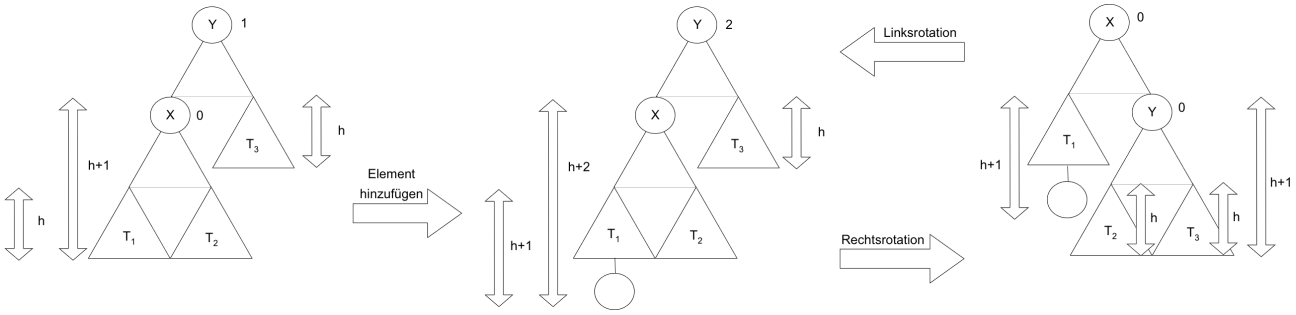
$$\Leftrightarrow h \leq \log(\frac{n}{c})$$

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

q.e.d.

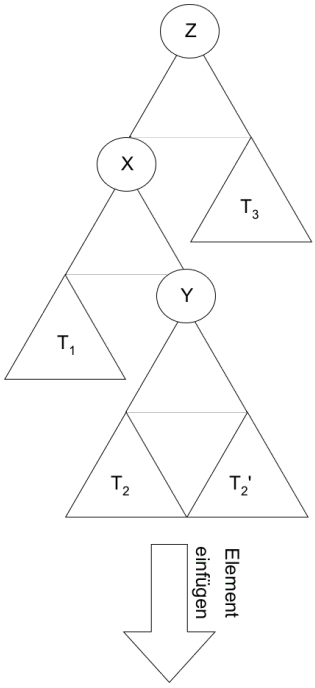
[I] $f(h)$ meint hierbei die h-te Fibonacci-Zahl

3.2. Rotationen

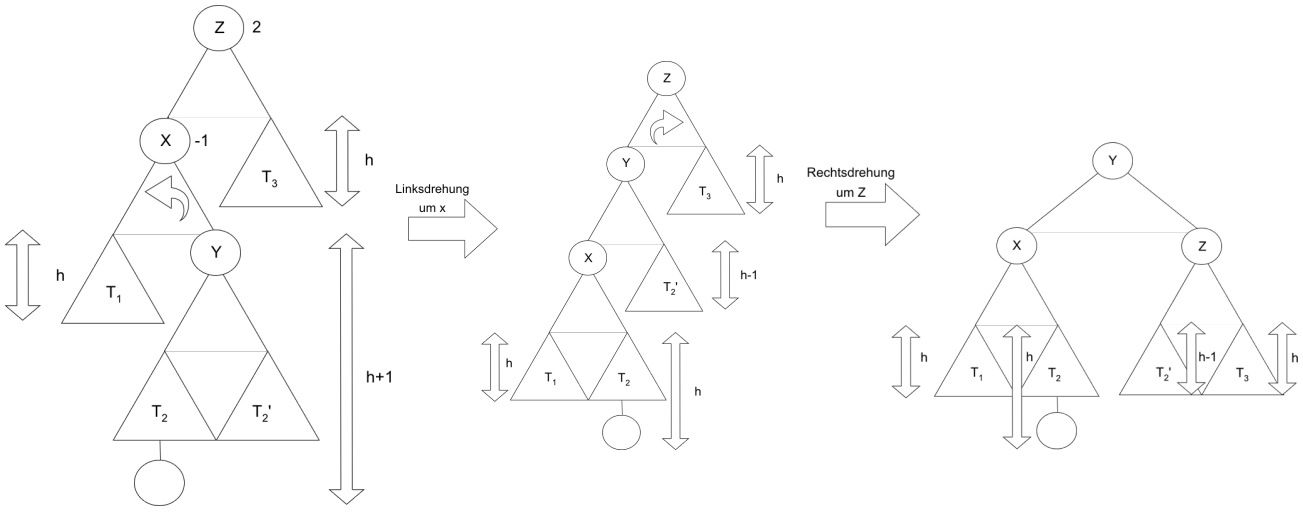


$Keys(T_1) < Key(X) < Keys(T_2) < Key(Y) < Keys(T_3)$
balance(Y) = height(Y.left)−height(Y.right)

$Keys(T_1) < Key(X) < Keys(T_2) < Key(Y) < Keys(T_2^{'}) < Key(Z) < Keys(T_3)$



Element einfügen



3.3. 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.

4. Vorlesung 12

4.1. (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:

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

4.1.1. Aufspaltung bei Einfügen

4.1.2. 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 $= O(\log n)$

4.2. Amortisierte Analyse

Beispiel: Binärzähler

000		
001	Kosten(1)	$= 1$
010		$= 2$
011		$= 1$
100		$= 3$
101		$= 1$
110		$= 2$
111		$= 1$
		$\overline{11}$

Kosten der Inkrement-Operation $\hat{=}$ Zahl der Bit-Flips

Naive Analyse $2^k = n$

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

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 $2n$

4.2.1. 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 $1_{€} \to 0$ kann dann mit dem entsprechenden Euro Betrag auf dieser 1 bezahlt werden.
Es gibt pro Inkrement Operation nur einen $0 \to 1$ Übergang
$2€$ Einzahlung für jede Inc-Operation reichen aus um:

1. diesen $0 \to 1$ Übergang zu bezahlen
2. die neu entstandene $1_{€}$ mit einem Euro zu besparen.

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

[I] Zählerstand(000)

[II] Zählerstand($\overbrace{111\ldots1}^{k}$)

5. 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 Löschoperationen. Dann gilt: $C \in O(m)$
d.h. Amortisierte Kosten der Split- und Fusionsopeartionen sind konstant.
! Dies bezieht sich nicht auf die Suchkosten, die in $O(\log n)$ liegen.

Beweisidee:

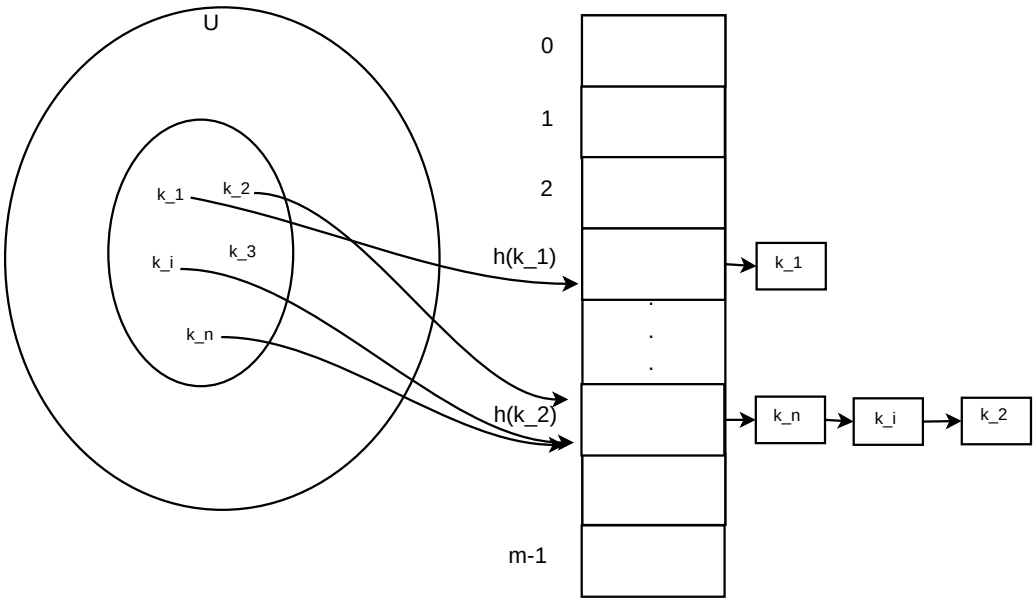
Kontoführung:

1	2	3	4	5	6
2€	1€	0€	0€	1€	2€

regelmäßige Einzahlung: 1€
Durch eine Einfüge- oder Löschoperation 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.

5.1. Hashing

Abbildung 5.1.: Universum und Hashtabelle der Größe m



$U \subseteq \mathbb{N}$ z.B. 64-Bit-Integer
n = Zahl dr zu verwaltenden Schlüssel

$$|U| >> n$$

Hashfunktion h:

$$h : U \to [0, \ldots, m-1]$$

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

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

Analyse der Suchzeit für einen Schlüssel k

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

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

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

Laufzeit beim Löschen von Schlüssel k

- $h(k) \longrightarrow O(1)$
- Durchlaufen der Liste $\longrightarrow 0(n_{h(k)})$
- Löschen durch „Pointer-Umbiegen“ $\longrightarrow O(1)$

5.1.1. Universelles Hashing

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

z.B.

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

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

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

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

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

[I] Belegungsfaktor

Suchzeit

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

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

6. 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).

6.0.1. Definition

Sei $\mathcal{H}$ eine endliche Menge von Hashfunktionen, welche ein gegebenes Universum U von Schlüsseln auf $\{0, \ldots, m-1\}$ abbildet. Sie heißt universell, wenn für jedes Paar von Schlüsseln $k, l \in U \quad l \neq k$ 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.

6.0.2. Beispiel

p Primzahl, so groß, dass alle möglichen Schlüssel $k \in U$ im $0, \ldots, p-1$ liegen. $\mathbb{Z}/p\mathbb{Z}$ bezeichnet den Restklassenring $\mod p$ (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:

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

Damit ergibt sich die Familie

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

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

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

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

Es ist $r \neq s$
Dazu:

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

Angenommen $r - s = 0$

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

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 $\mod p$“ gibt es keine Kollisionen).
Aus $(*2)$ folgt:

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

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

Daher ist die Wahrscheinlichkeit, dass zwei Schlüssel $h \neq l$ kollidieren, gerade die Wahrscheinlichkeit, dass $r \equiv s \mod m$, 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 $s \neq r \mod p$ aber $r = s \mod m$

6.0.3. 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 $\mod m$ 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 $k \neq l$

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

6.1. Perfektes Hashing

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

Beispiele reserved words bei Programmiersprachen, Dateinamen auf einer CD

6.1.1. 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 $h(k) = ((a \cdot b) \mod p) \mod m$

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

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

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

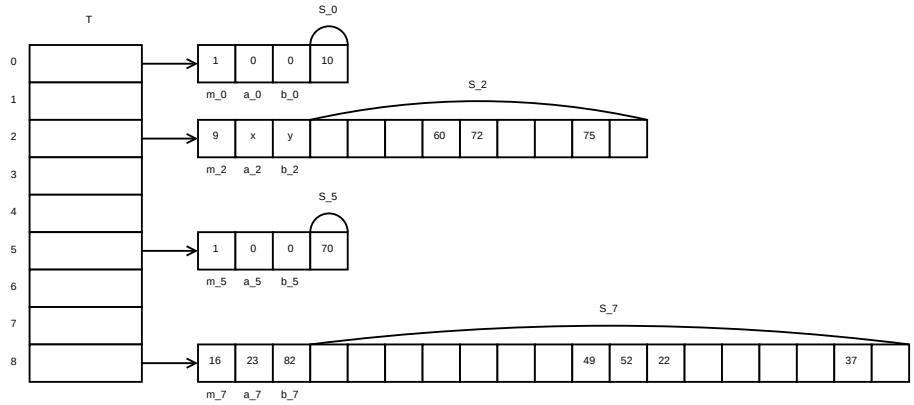


Abbildung 6.1.: Perfekte Hashtabelle

gerade n_j^2 sein ($n_j \neq$ #Schlüssel$\mapsto j$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

6.1.2. 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 $O(n)$ $\curvearrowright$ 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

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

Betrachte

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

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

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.

7. Vorlesung 15

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

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

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

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

Idee Zweistufiges Verfahren:

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

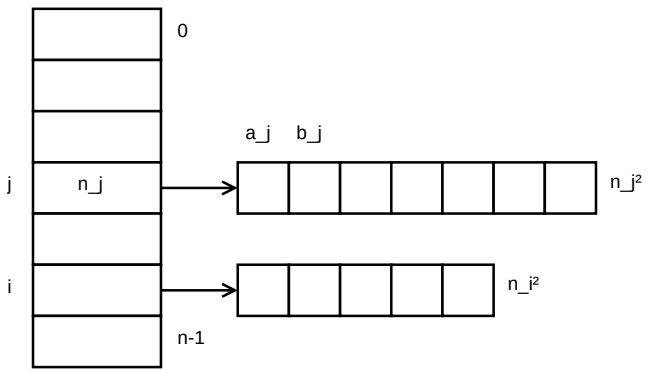


Abbildung 7.1.: Perfektes Hashing

7.1. Graphen-Algorithmen

7.1.1. Einführung

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

[I] Universalität von $\mathcal{H}$

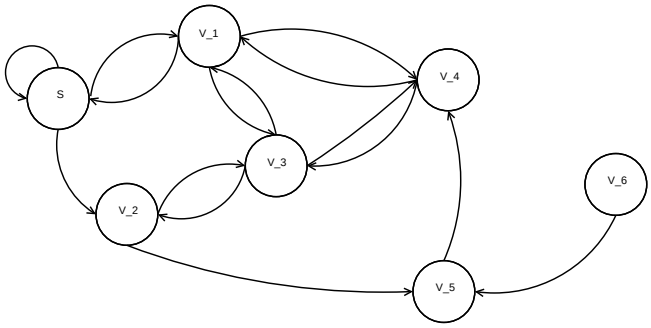


Abbildung 7.2.: Gerichteter Graph

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

Eulerische Polyederformel

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

$$8 + 6 = 12 + 2$$

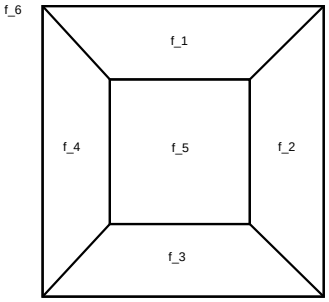


Abbildung 7.3.: Würfel

Es gilt:

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

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

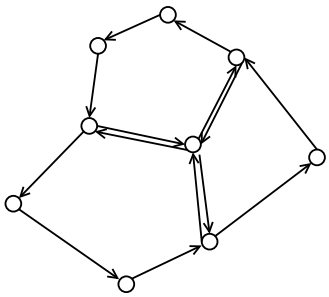


Abbildung 7.4.: Placeholder

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

$$\boxed{\Rightarrow |E| \leq 3 \cdot |V| - 6}$$

[II] Jedes f_i hat mindestens 3 Kanten

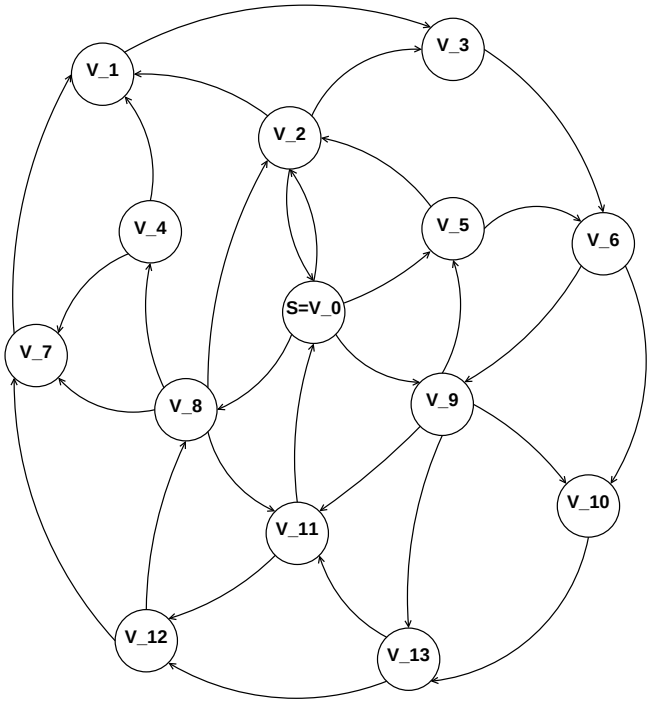


Abbildung 7.5.: Beispiel

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

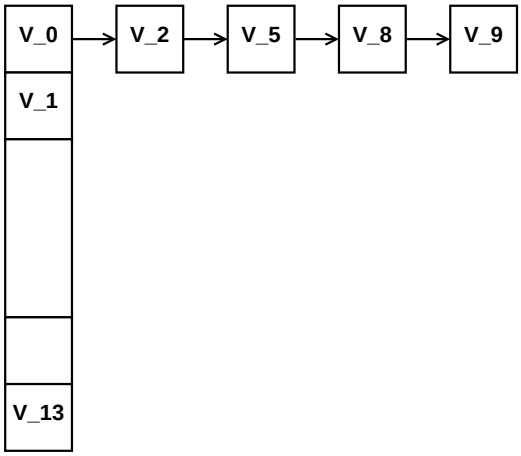


Abbildung 7.6.: Adjazenzliste

Platzbedarf

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

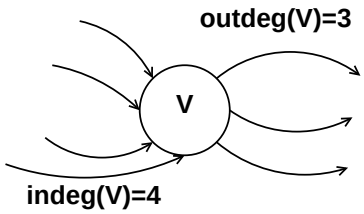


Abbildung 7.7.: indeg und outdeg

7.1.2. BFS (Breadth-First Search) Breitensuche

```
forall ( v in V\{S}) {
  col[v]=white;     // Farbe  weiß = unbekannt, grau = bekannt, schwarz = vollkommen b
  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;
}
```

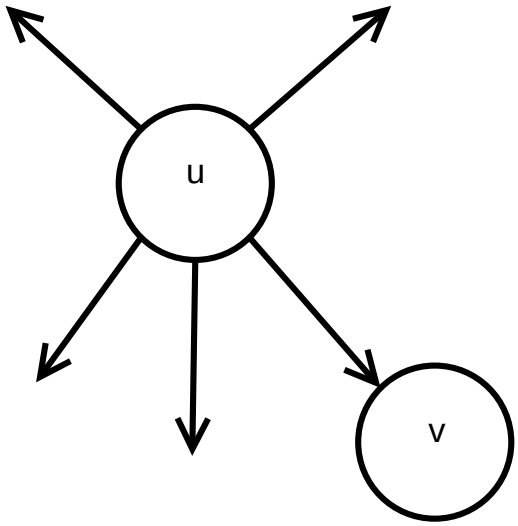


Abbildung 7.8.: Grafik zum Beispielcode

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.

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

8. Vorlesung 16

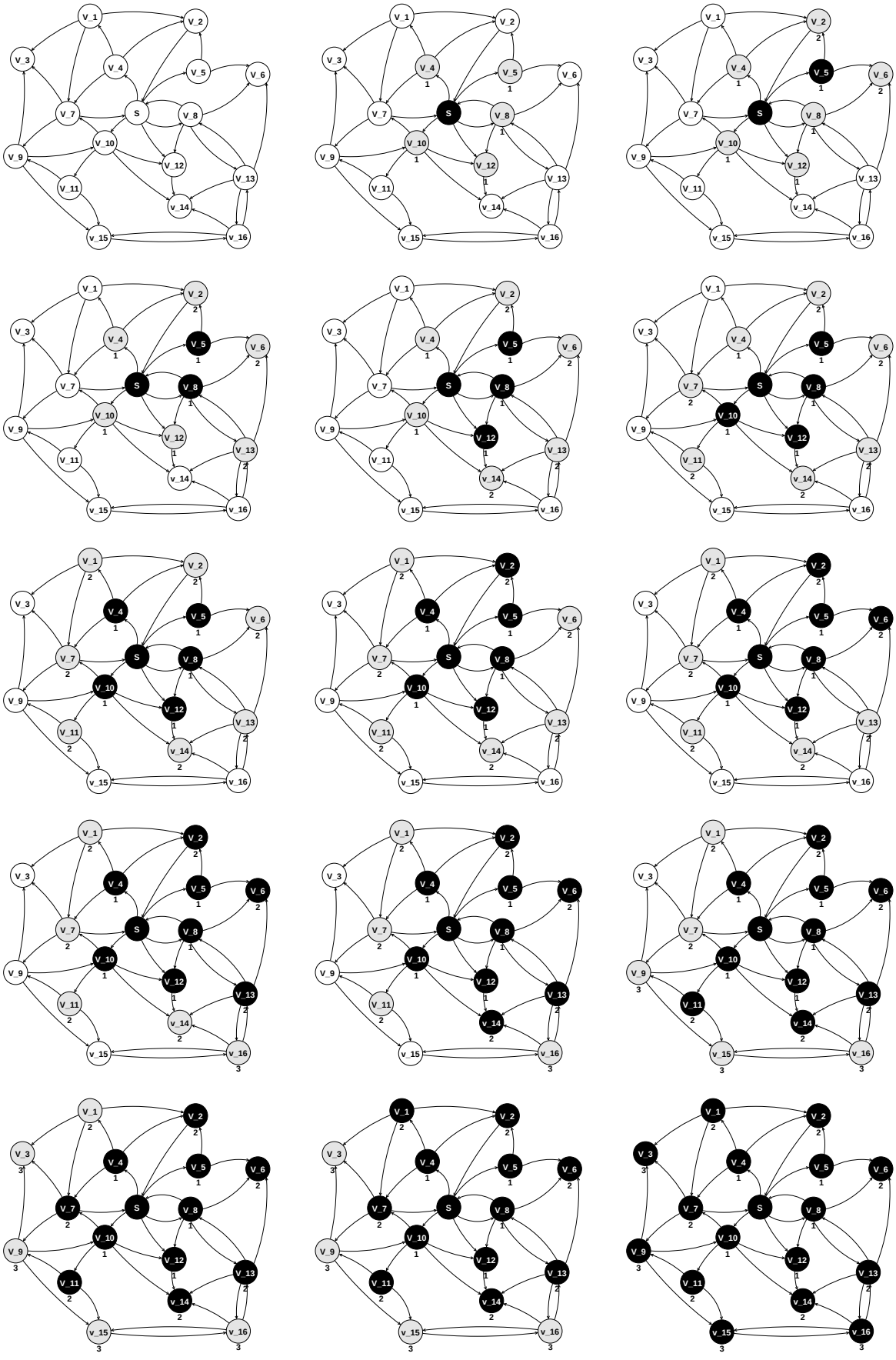


Abbildung 8.1.: Beispiel

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

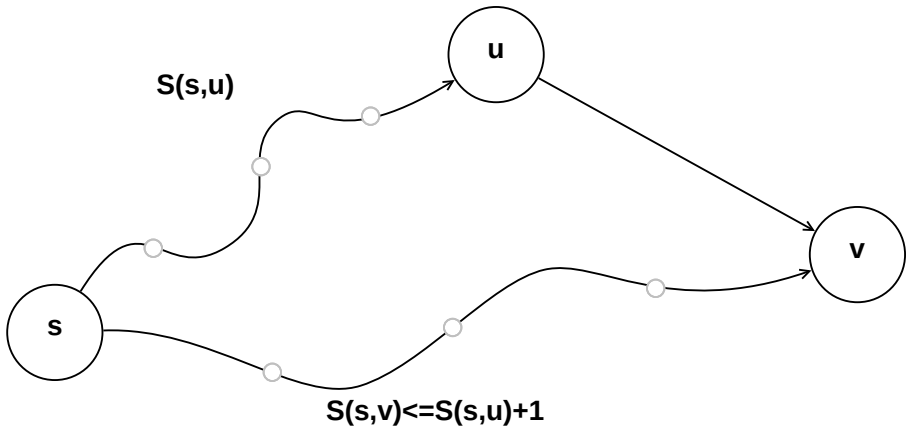


Abbildung 8.2.

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

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

Induktions-Schritt Knoten v wird von u aus neu entdeckt

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

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

Lemma 3

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

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

[I] Breitensuche

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

Induktions-Anfang

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

Induktions-Schritt

pop

$$\cancel{d[v_1]} \leq d[v_2] \leq \ldots \leq d[v_k] \leq d[v_1] + 1 \overset{!}{\leq} d[v_2] + 1$$

push

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

Beachte Kante (u, v) v ist weiß

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

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

Zustand von Q nach push

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

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 $\overset{Lemma2}{\Longrightarrow} d[v] > \delta(s, V)$
Sei v so gewählt, dass es der erste knoten ist mit der Eigenschaft, dass sein d-Wert flasch gesetzt wird. d.h. Alle d-Werte bis zu diesem Zeitpunkt sind korrekt.
Sei $s \mapsto u' \to v$ ein kürzester Weg s ui v
Betrachte die Situation bei Bearbeitung von u':

1. Fall v ist in diesem Moment schwarz.

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

2. Fall v ist in diesem Moment weiß.

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

[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 $\lightning$

q.e.d.

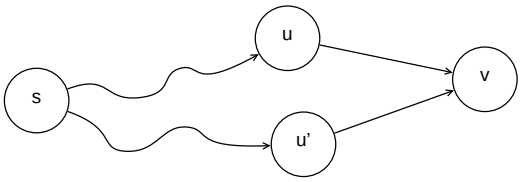


Abbildung 8.3.

8.1. Kürzeste Wege Algorithmen

8.1.1. Dijkstra-Algorithmus

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

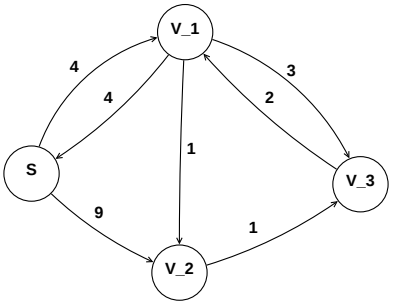


Abbildung 8.4.

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

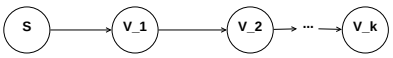


Abbildung 8.5.

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

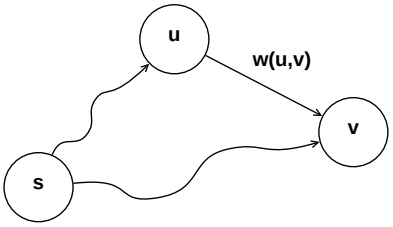


Abbildung 8.6.

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

```
relax(u,v,w) {
  if(d[v] > d[w]+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

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

Induktion über Zahl der reflex-Aufrufe