

Kap. 6.5: Minimale Spannbäume

Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

20. VO

20. Juni 2006

Überblick

- Wiederholung (Folien s. letztes Mal):
 - Minimale Spann bäume (Einführung)

- Algorithmus von Kruskal

- ADT Union Find
- extrem langsam wachsende Funktion $\alpha(n)$

Motivation

„Was gibt es heute Besonderes?“

einfacher und schöner Greedy-Algorithmus

„Was gibt es heute Besonderes?“

ADT UNION-FIND: „So kam ich zur Informatik“

„Und was noch?“

extrem langsam wachsende Funktion

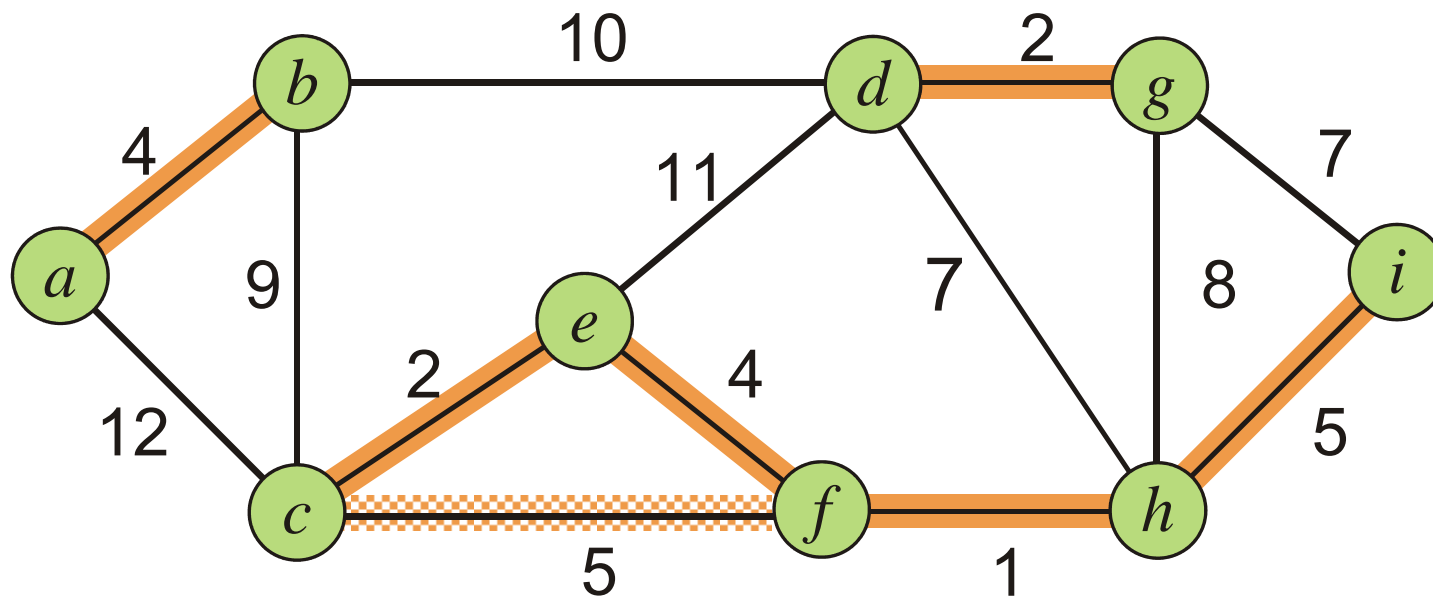
6.5.1 Algorithmus von Kruskal

Idee:

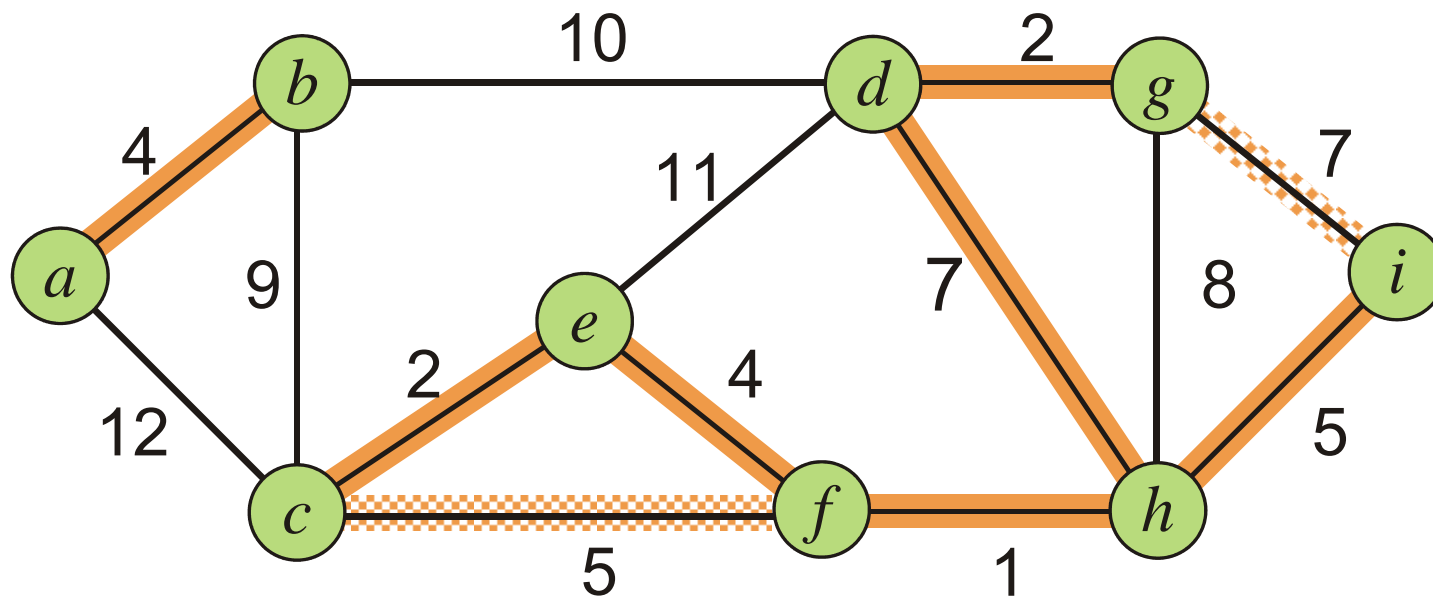
- Sortiere die Kanten nach aufsteigendem Gewicht: $w(e_1) \leq w(e_2) \leq \dots \leq w(e_m)$
- $T := \emptyset$
- Für $i := 1, \dots, m$:
- Falls $T \cup \{e_i\}$ kreisfrei, dann: $T := T \cup \{e_i\}$

Greedy-Algorithmus („gefräßig“): iterative Konstruktion einer Lösung, die immer um die momentan besten Kandidaten erweitert wird.

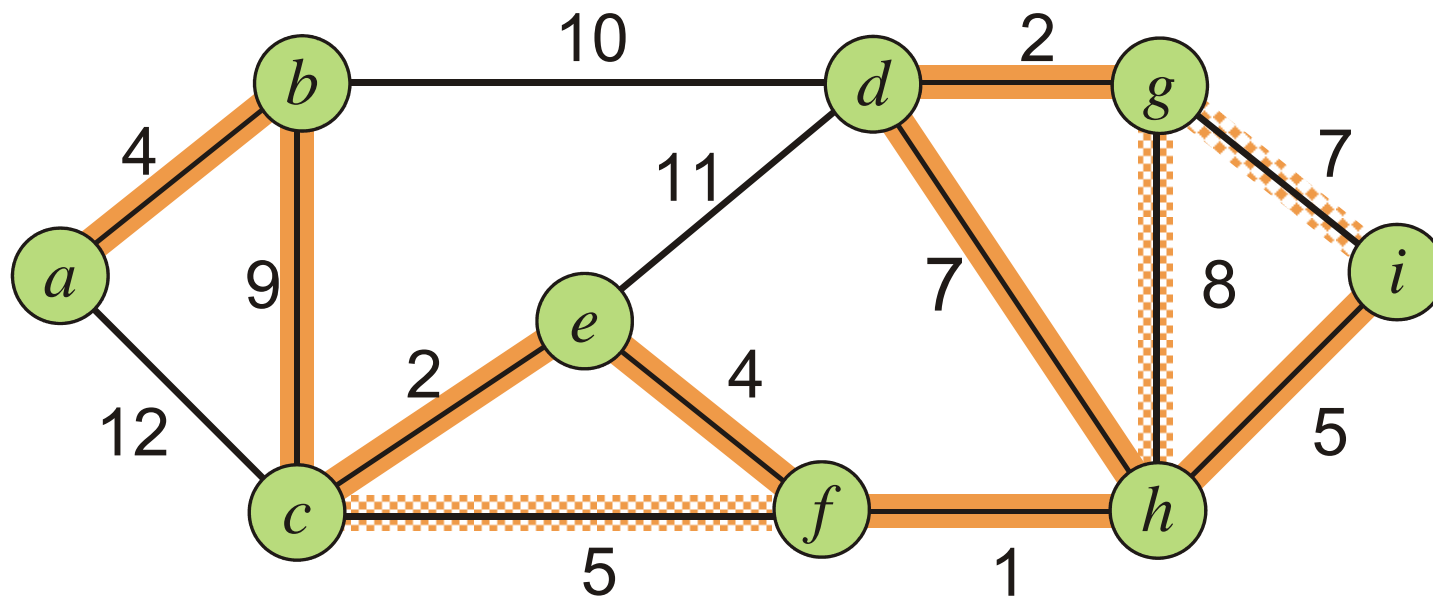
Beispiel für Kruskal



Beispiel für Kruskal



Beispiel für Kruskal



Korrektheit von Kruskal

Theorem:

Sei $G=(V,E)$ zshgd. mit Gewichtsfunktion $w : E \rightarrow \mathbb{R}$.
Dann berechnet der Algorithmus von Kruskal einen
MST von G .

Beweis:

- T ist aufspannender Baum: offensichtlich, denn G ist zshgd., und wir fügen Kanten nur dann nicht hinzu, wenn sie einen Kreis schließen würden.
- Minimalität: mittels obigem Lemma (MST Eigenschaft):

MST Eigenschaft

Lemma:

Sei $G=(V,E)$ zshgd. mit Gewichtsfunktion $w : E \rightarrow \mathbb{R}$.

Seien:

- $S \subset V$
- $T \subset E$ aussichtsreich und *keine* Kante aus T verlässt S
- $e \in E$ eine Kante mit minimalem Gewicht, die S verlässt

Dann ist $T \cup \{e\}$ aussichtsreich.

Beweis ff: Minimalität

Seien t_1, t_2, \dots, t_k die Kanten, die Kruskal in dieser Reihenfolge zu T hinzufügt. Wir zeigen:

$T_i := \{t_1, t_2, \dots, t_i\}$ mit $0 \leq i \leq k$ ist aussichtsreich.

Induktion: $i=0$: $T_0 = \emptyset \rightarrow$ o.k.

Sei nun $1 \leq i \leq k$: Ind. Ann.: T_{i-1} ist aussichtsreich.

Sei $t_i = (u, v)$ und $C_u = (U, E_u)$ die Komponente des Graphen $G_{i-1} = (V, T_{i-1})$ die u enthält.

- t_i verläßt U (da $T_{i-1} \cup \{t_i\}$ kreisfrei bleibt)
 - keine Kante aus T_{i-1} verläßt U
 - keine billigere Kante als t_i verläßt U
 - t_i ist eine Kante minimalen Gewichts, die U verläßt
- $\rightarrow T_i := T_{i-1} \cup \{t_i\}$ ist aussichtsreich.

Realisierung von Kruskal

Idee:

- Sortiere die Kanten nach aufsteigendem Gewicht: $w(e_1) \leq w(e_2) \leq \dots \leq w(e_m)$
- $T := \emptyset$
- Für $i := 1, \dots, m$:
- Falls $T \cup \{e_i\}$ kreisfrei, dann: $T := T \cup \{e_i\}$

Problem: Teste, ob $T \cup \{e_i\}$ kreisfrei

Teste, ob $T \cup \{e_i\}$ kreisfrei

Idee:

- Aufruf der Funktion $\text{ISACYCLIC}(V, T \cup \{e_i\})$
- Gesamtlaufzeit für alle Kreis-Tests:
 $O(|V| |E|)$, da $|E|$ Aufrufe nötig sind, und T pro Aufruf bis zu $|V|-1$ Kanten besitzen kann.

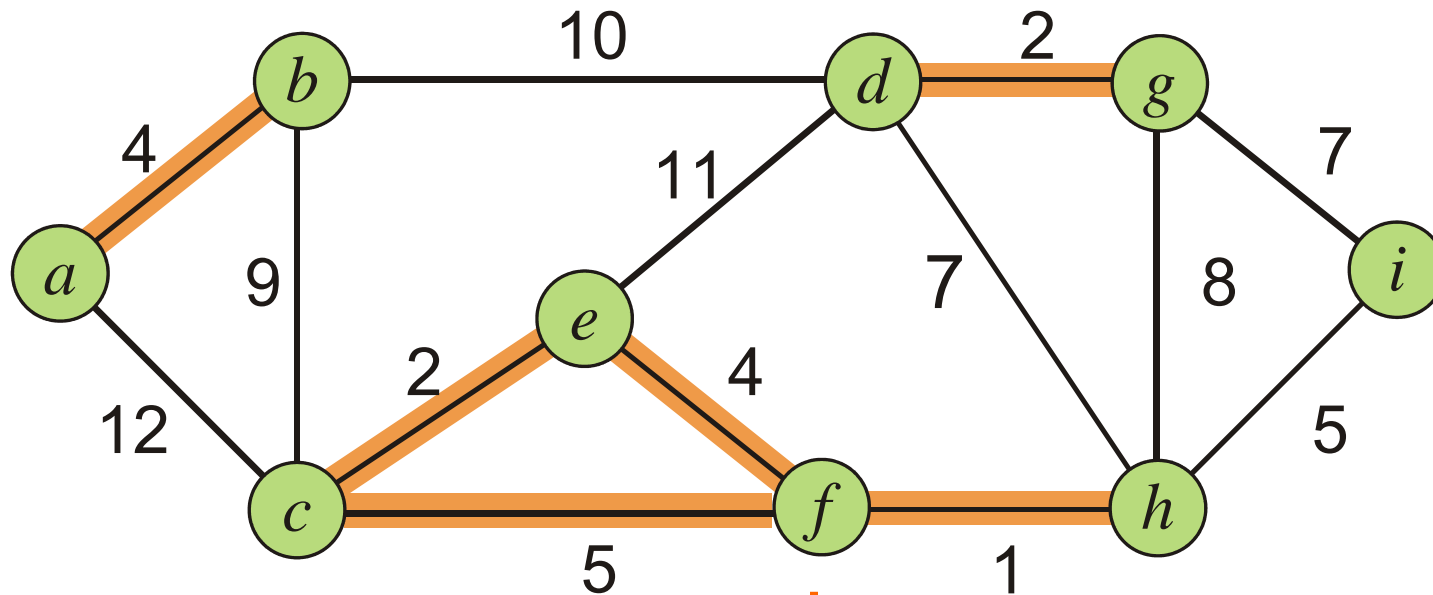
Besser: Datenstruktur Union-Find

Idee: „Lasse Wälder wachsen“ bzw.

Partitionierung von Mengen

→ fast „Linearzeit“ erreichbar

Kruskal mit Mengenpartitionierung



{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}

(f,h)

{a}, {b}, {c}, {d}, {e}, {f,h}, {g}, {i}

(c,e)

{a}, {b}, {c,e}, {d}, {f,h}, {g}, {i}

(d,g)

{a}, {b}, {c,e}, {d,g}, {f,h}, {i}

(e,f)

{a}, {b}, {c,e,f,h}, {d,g}, {i}

(a,b)

{a,b}, {c,e,f,h}, {d,g}, {i}

(c,f)

Partitionierung von Mengen

Idee:

- Zu Beginn ist jeder Knoten in einer eigenen Menge
- Wird eine Kante $e=(u,v)$ zu T hinzugenommen, dann werden die beiden Mengen, in denen u und v jeweils liegen, vereinigt.
- Test auf Kreisfreiheit von $T \cup \{(u,v)\}$: Kreis entsteht genau dann wenn u und v in der gleichen Menge liegen

Operationen zur Partitionierung

- Erzeuge eine einelementige Menge der Partition: **MAKESET**
- Finde die Partitionsmenge, in der sich ein gegebenes Element befindet: **FIND**
- Vereinige zwei Partitions Mengen: **UNION**

→ hierfür: ADT UNION-FIND

6.5.3 Der ADT UNION-FIND

- Zweck: Verwaltung einer Partition einer endlichen Menge, deren Elemente aus einem endlichen Universum U sind.
- Wertebereich: Familie $P = \{S_1, S_2, \dots, S_k\}$ paarweiser disjunkter Teilmengen einer endlichen Menge U . Für jede Teilmenge gibt es einen eindeutigen Repräsentanten $s_i \in S_i$.
- Sei $P = \{S_1, S_2, \dots, S_k\}$ mit Repräsentanten s_i die Instanz vor Anwendung der Operation.

Operationen des ADT UNION-FIND

MAKESET(U x)

- Voraussetzung: $x \notin S_1 \cup S_2 \cup \dots \cup S_k$
- Erzeugt eine neue Menge $S_{k+1} := \{x\}$ mit Repräsentant $s_{k+1} := x$ und fügt sie zu P hinzu.

FIND(U x): U

- Voraussetzung: x ist in einer der Mengen S_1, S_2, \dots, S_k enthalten
- Gibt den Repräsentanten der Menge zurück, die x enthält.

Operationen des ADT UNION-FIND

UNION(U_x, U_y)

- Voraussetzung: x und y sind jeweils Repräsentant einer der Mengen S_1, S_2, \dots, S_k
- Sei x Repräsentant von S_x , und y Repräsentant von S_y . Dann werden S_x und S_y in P durch $S' := S_x \cup S_y$ ersetzt, d.h. die Instanz P' nach der Operation ist

$$P' := P \setminus \{S_x, S_y\} \cup \{S'\}.$$

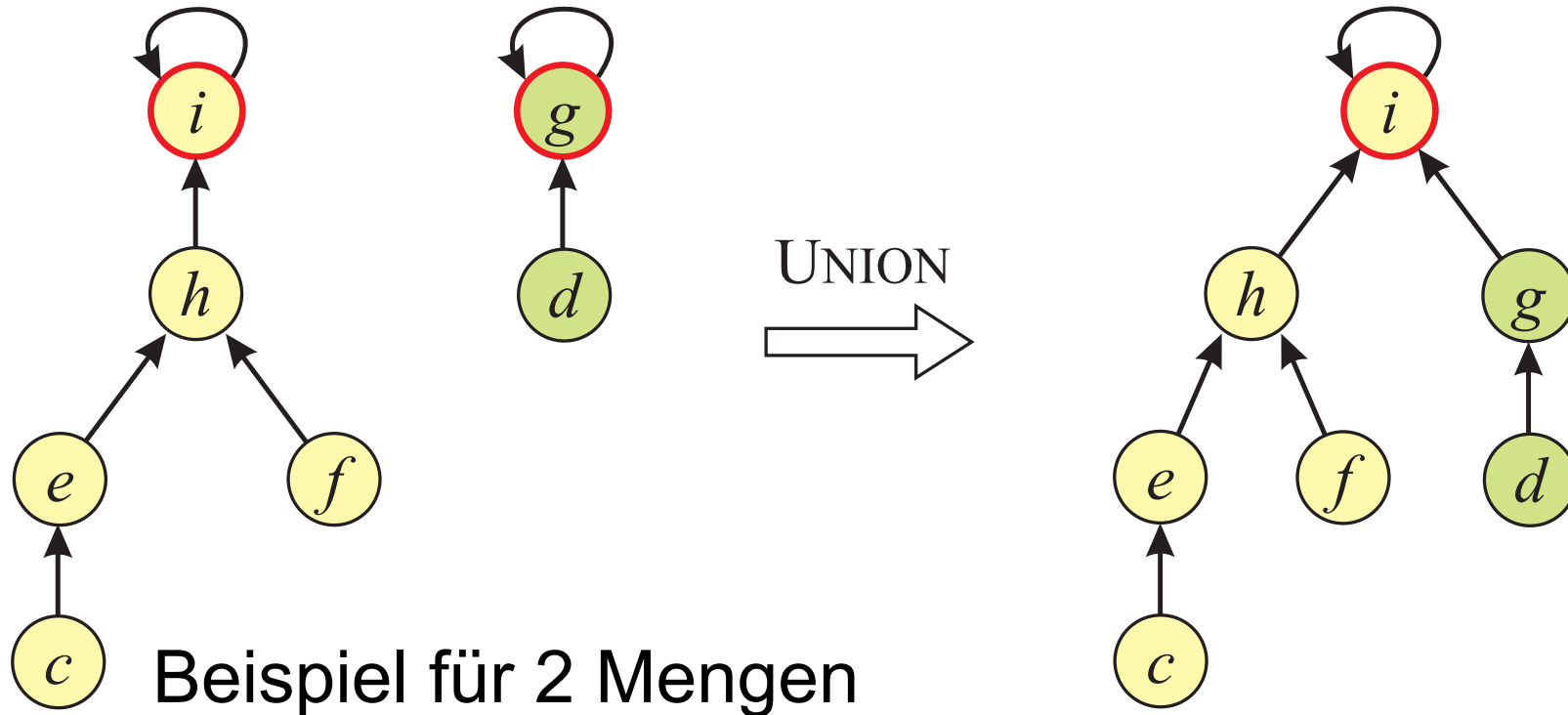
- Als Repräsentant für S' wird ein beliebiges Element in S' gewählt.

Realisierung des ADT UNION-FIND

Idee: Realisierung durch gewurzelte Bäume:

- Jede Menge S_k wird durch gewurzelten Baum dargestellt.
- Die Knoten des Baums sind die Elemente der Menge.
- Die Wurzel des Baumes ist der Repräsentant der Menge.
- In Implementierung hat jeder Knoten v einen Verweis auf seinen Elter

Realisierung durch gewurzelte Bäume



- Implementierung:
- Jeder Knoten v hat Verweis auf seinen Elter: $\pi(v)$
- Die Wurzel zeigt auf sich selbst.

Implementierung von Union-Find

```
(1)  procedure MAKESET(U x) {  
(2)     $\pi[x] := x$   
(3)  }  
(4)  procedure UNION(U x, U y) {  
(5)     $\pi[x] := y$   
(6)  }  
(7)  function FIND(U x):U {  
(8)    while  $\pi[x] \neq x$  do {  $x := \pi[x]$  }  
(9)    return x  
(10) }
```

Laufzeitanalyse des ADT

- MAKESET(x): $O(1)$
- UNION(x, y): $O(1)$
- FIND(x): $O(h(T_x))$, wobei T_x der Baum ist, der x enthält.

- Gesamtlaufzeit von n MAKESET Operationen und $n-1$ UNION und $2m$ FIND Operationen (für Kruskal): $O(n+mn)$, da die Höhe eines Baumes $\Theta(n)$ sein kann.

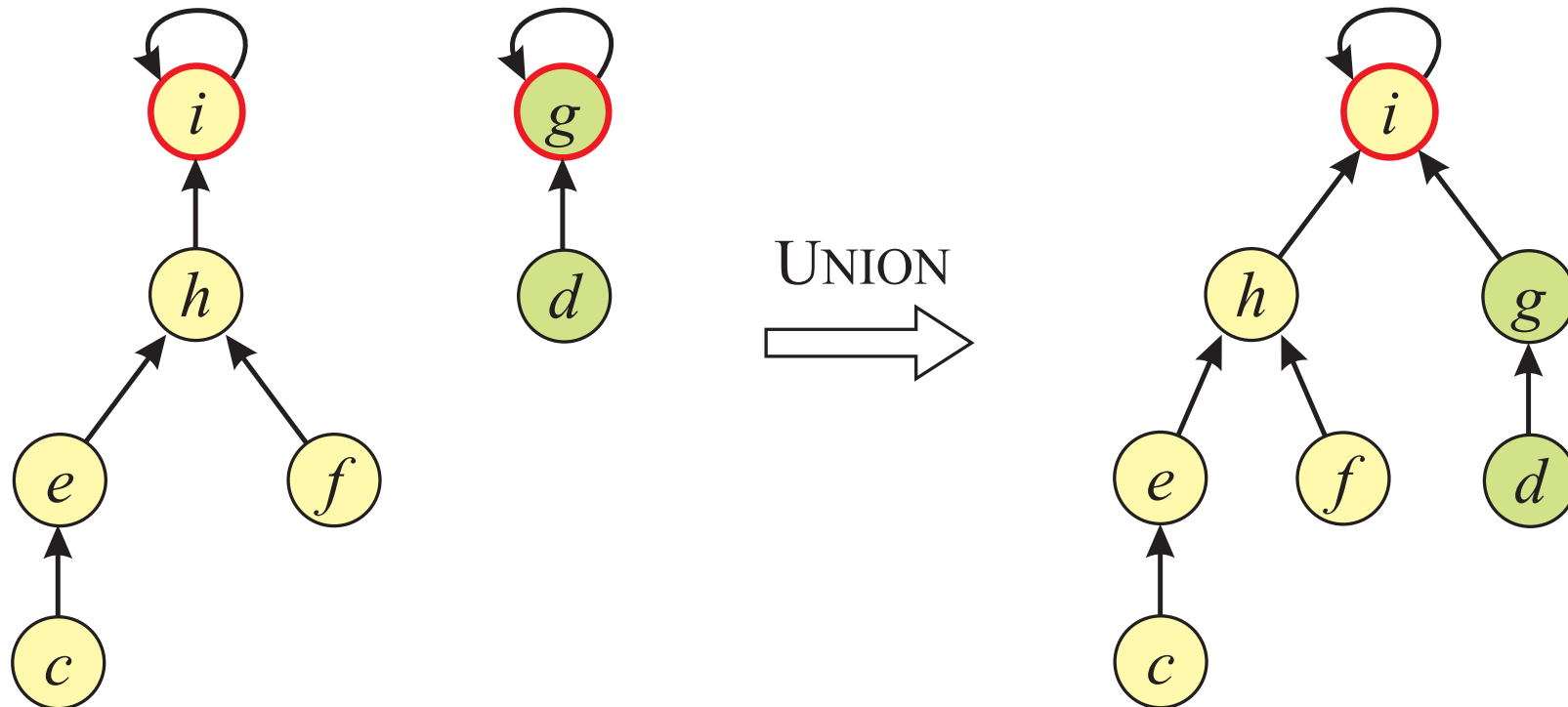
jetzt: Verbesserung der Laufzeiten

Verbesserung 1: Gewichtete Vereinigungsregel

Idee:

Bei UNION: Hänge den Baum mit der kleineren Höhe an den Baum mit der größeren Höhe.

Realisierung durch gewurzelte Bäume



- Wenn $h(T_x) > h(T_y)$, dann ist die neue Höhe $h(T_x) \rightarrow$ sie nimmt nicht zu
- Wenn beide Bäume gleiche Höhe haben, dann ist die neue Höhe: $h(T_x) + 1$

Verbesserung 1: Gewichtete Vereinigungsregel

Lemma: UNION mit gewichteter Vereinigungsregel liefert immer Bäume, deren Höhe $h(T)$ durch $\log s(T)$ nach oben beschränkt ist.

Anzahl der Knoten im Baum

Beweis: Wir zeigen, dass jeder erzeugte Baum mit Höhe d mindestens 2^d Knoten besitzt. Induktion:

$d=0$: Baum besitzt 1 Knoten: $1 \geq 2^0$

$d \geq 1$: Ind.-Ann.: Beh. gilt für alle Bäume mit $d' < d$

Beweis Lemma ff. z.z.: Jeder erzeugte Baum mit Höhe d hat $\geq 2^d$ Knoten

Sei T Baum mit Höhe d mit minimaler Knotenanzahl.

- T entstand dadurch, dass bei UNION der Baum T_1 an T_2 gehängt wurde.
- Wir wissen also: $h(T_1) \leq h(T_2)$ und
- $h(T_2) \leq d-1$, denn sonst hätte T_2 Höhe d aber weniger Knoten als T , was nach Wahl von T nicht möglich ist.
- $h(T)$ ist also höher als $h(T_1)$ und $h(T_2)$, deswegen müssen beide gleiche Höhe gehabt haben \rightarrow
 $h(T_1) = h(T_2) = d-1$
- aus Ind.Vorr. folgt: $s(T_1) \geq 2^{d-1}$ und $s(T_2) \geq 2^{d-1}$
- daraus folgt: $s(T) \geq 2^{d-1} + 2^{d-1} = 2^d$

Analyse der gewichteten Vereinigungsregel

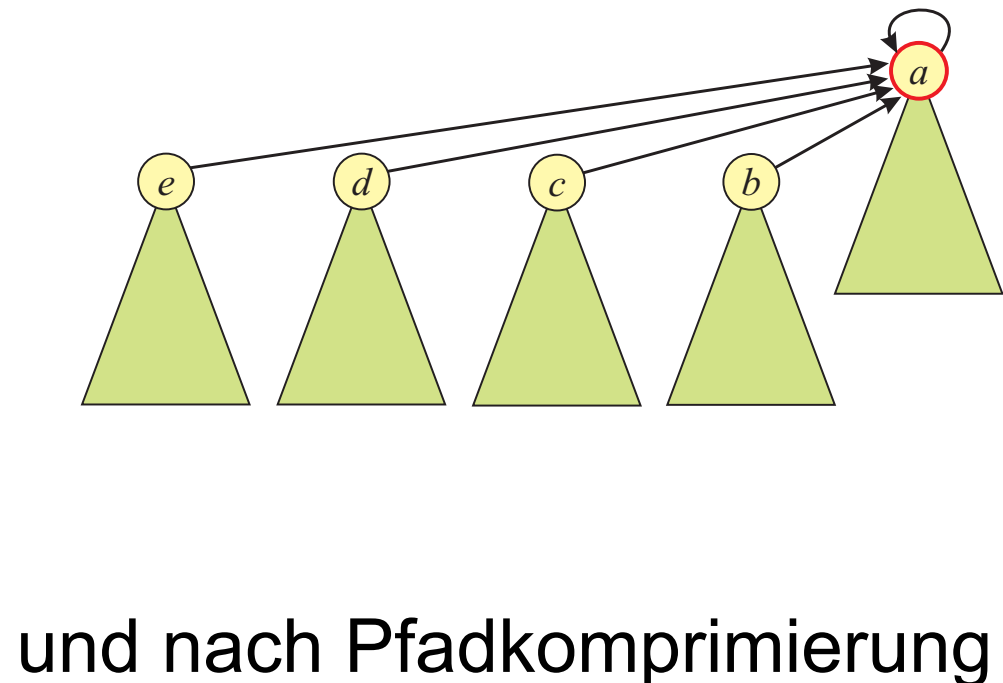
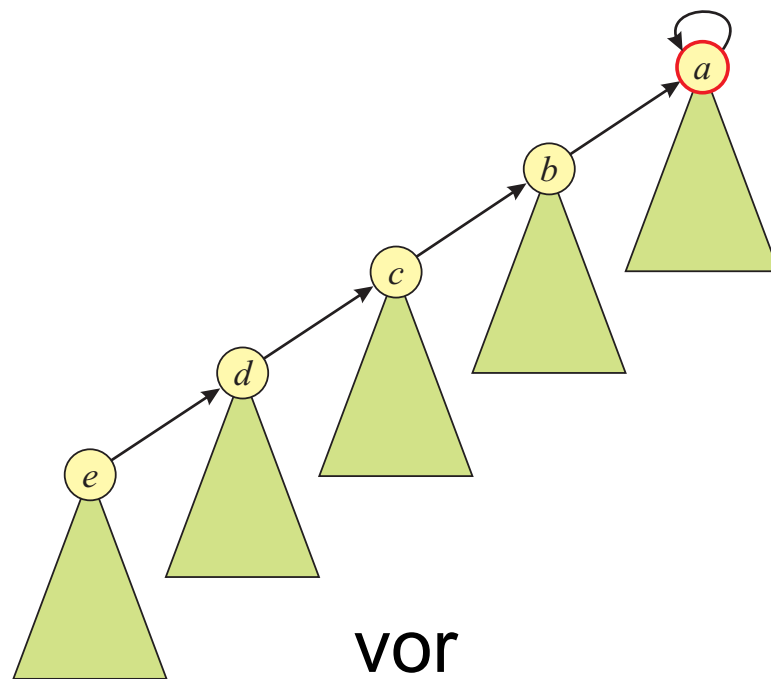
Lemma: Worst-Case Laufzeit einer FIND
Operation ist durch $O(\log n)$ beschränkt.

Gesamtlaufzeit von n MAKESET
Operationen und $n-1$ UNION und $2m$ FIND
Operationen (für Kruskal): $O(n+m \log n)$.

Statt als Rang einer Wurzel die Höhe des
Baumes zu benutzen, kann man alternativ auch
die Anzahl der Knoten im Baum verwenden →
auch Höhe $\log s(T)$.

Verbesserung 2: Pfadkomprimierung

Idee: Bei einem Aufruf von $\text{FIND}(x)$ werden alle Knoten auf dem Pfad von x bis zur Wurzel zu direkten Kindern der Wurzel gemacht.



Analyse der Pfadkomprimierung

Theorem: Wird die gewichtete Vereinigungsregel und Pfadkomprimierung benutzt, so kosten $n-1$ UNION und $2m$ FIND Operationen im Worst Case $O((n+m)\alpha(n))$.

Dabei ist $\alpha(n)$ eine extrem langsam wachsende Funktion.

Tarjan machte die ursprüngliche Analyse mit der Inversen der Ackermann-Funktion, die sehr ähnlich zu $\alpha(n)$ ist.

Definition

Für $k \geq 0$ und $j \geq 1$ definieren wir:

$$A_k(j) := \begin{cases} j + 1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1 \end{cases}$$

(j+1)-fache Anwendung der Funktion

$$A_0(1) = 1+1=2$$

Es gilt: $A_2(j) = 2^{j+1}(j+1) - 1$

$$A_1(1) = A_0^{(2)}(1) = A_0(A_0(1)) = A_0(2) = 2+1=3$$

$$A_2(1) = A_1^{(2)}(1) = A_1(3) = A_0^{(4)}(3) = A_0^{(3)}(A_0(3)) = 7$$

$$A_3(1) = A_2^{(2)}(1) = A_2(7) = 2^8 \cdot 8 - 1 = 2047$$

$$A_4(1) = A_3^{(2)}(1) = A_3(2047) \gg A_2(2047) > 16^{512} \gg 10^{80}$$

Definition

Die Inverse der Funktion A_k ist definiert als

$$\alpha(n) := \min \{ k \mid A_k(1) \geq n \}$$

$$\alpha(n) = \begin{cases} 0 & \text{für } 0 \leq n \leq 2 \\ 1 & \text{für } n = 3 \\ 2 & \text{für } 4 \leq n \leq 7 \\ 3 & \text{für } 8 \leq n \leq 2047 \\ 4 & \text{für } 2048 \leq n \leq A_4(1) \end{cases}$$

Wir können also annehmen, dass $\alpha(n) \leq 4$ ist für alle praktisch relevanten Werte von n .

Implementierung von UnionFind

```
(1)  procedure MAKESET(U x) {  
(2)     $\pi[x] := x$   
(3)     $\text{rank}[x] := 0$   
(4)  }  
(5)  procedure UNION(U x, U y) {  
(6)    if  $\text{rank}[x] > \text{rank}[y]$  then  
(7)       $\pi[y] := x$   
(8)    else  $\pi[x] := y$   
(9)    if  $\text{rank}[x] == \text{rank}[y]$  then  $\text{rank}[y] := \text{rank}[y] + 1$   
(10)  }  
(11)
```

Implementierung von UnionFind

```
(1) function FIND( $U$   $x$ ): $U$  {  
(2)   if  $\pi[x] \neq x$  then  
(3)      $\pi[x] := \text{FIND}(\pi[x])$   
(4)   return  $\pi[x]$   
(5) }
```

Realisierung von Kruskal mit Union-Find

```
(1)   Sortiere Kanten, so dass  $w(e_1) \leq w(e_2) \leq \dots \leq w(e_m)$ 
(2)    $T := \emptyset; i := 1$ 
(3)   while  $|T| < |V| - 1$  do           // let  $e_i = (u, v)$ 
(4)        $x := P.FIND(u)$ 
(5)        $y := P.FIND(v)$ 
(6)       if  $x \neq y$  then {
(7)            $T := T \cup \{e_i\}$ 
(8)            $P.UNION(x, y)$ 
(9)       } // while
(10)    $i := i + 1$ 
(11) }
```

Analyse von Kruskal mit Union-Find

- Sortieren der Kanten: $O(m \log m)$
- Initialisieren von MAKESET: $O(n)$
- while-Schleife: wird im schlechtesten Fall für alle Kanten durchlaufen: m Mal
- Dabei insgesamt $2m$ FIND-Operationen und $n-1$ UNION-Operationen
- Dies ist also: $O((n+m)\alpha(n))$ und falls G zusammenhängend ist $O(m \alpha(n))=O(m)$
- Gesamtlaufzeit wird von Sortieren dominiert: $O(m \log m)$

Analyse von Kruskal mit Union-Find

- Der Algorithmus von Kruskal berechnet einen minimalen Spannbaum eines ungerichteten, zusammenhängenden Graphen $G=(V,E)$ in Zeit $O(|E| \log |E|)$.