12 Union Find

Union-Find Datenstruktur $\mathcal{P}$: Verwaltet die Partionierung einer Menge M, d.h., die Zerlegung der Menge in disjunkte Teilmengen.

Operationen

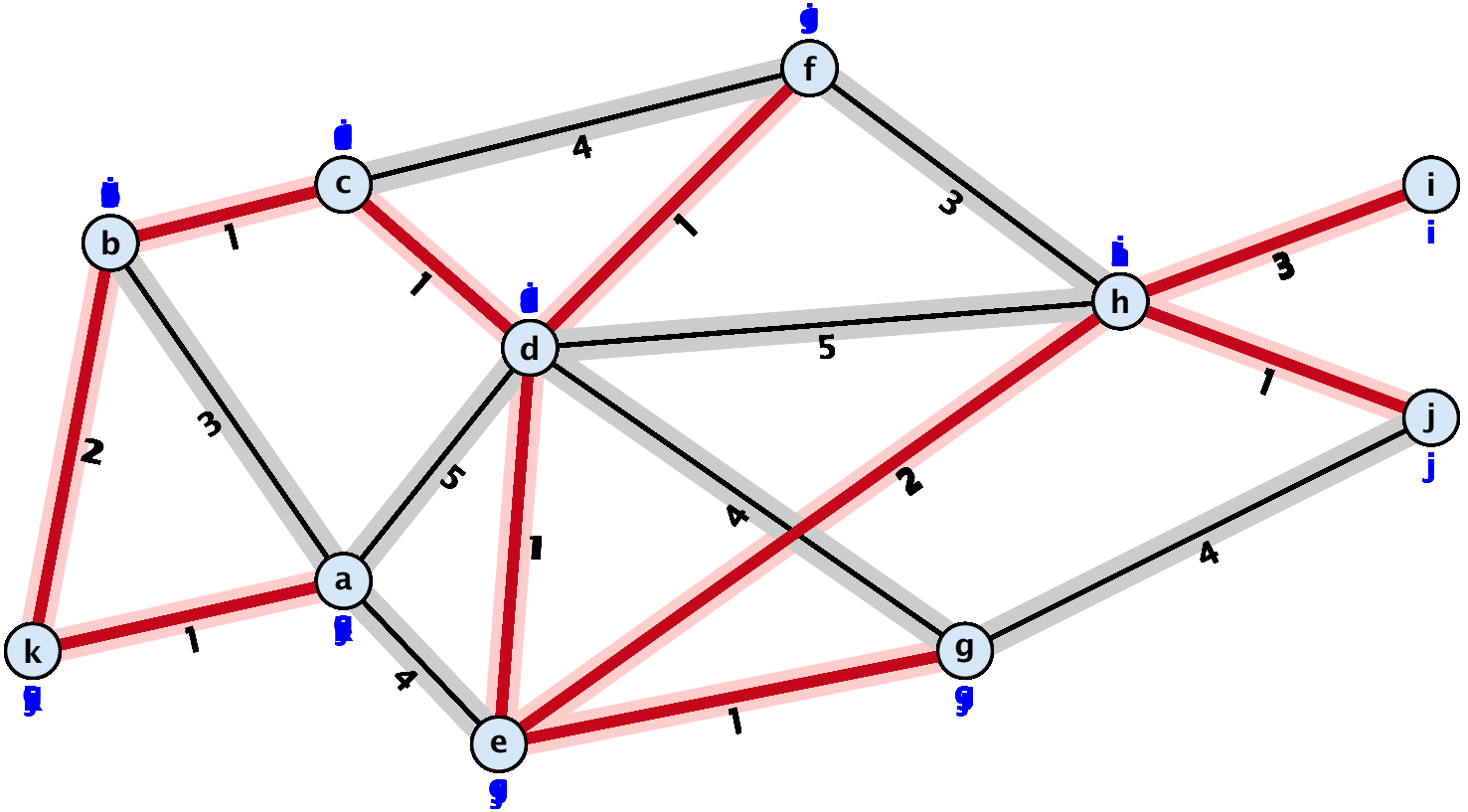
- $\mathcal{P}.\mathbf{makeset}(x)$**:** Fügt ein Element x zu M hinzu und fügt es in eine Teilmenge $\{x\}$ ein, d.h., erzeugt eine Teilmenge, die nur dieses Element enthält. Gibt ein handle für x zurück.
- $\mathcal{P}.\mathbf{find}(x)$**:** Input: handle für ein Element x; findet die Menge, die x enthält; gibt einen Repräsentanten/Identifier für die Menge zurück.
- $\mathcal{P}.\mathbf{union}(x,y)$**:** Input: handles von zwei Elementen x und y, die momentan in Mengen S_x bzw. S_y sind. Die Funktion ersetzt S_x und S_y durch $S_x \cup S_y$ und gibt den Repräsentanten/Identifier der neuen Menge zurück.

12 Union Find

Anwendungen:

- Verwaltung von Zusammenhangskomponenten in einem dynamischen Graphen, der sich durch das Einfügen von Knoten und Kanten ändert.
- Kruskals Algorithmus für minimale Spannbäume.

Minimaler Spannbaum (Kruskal)



Beobachtung:

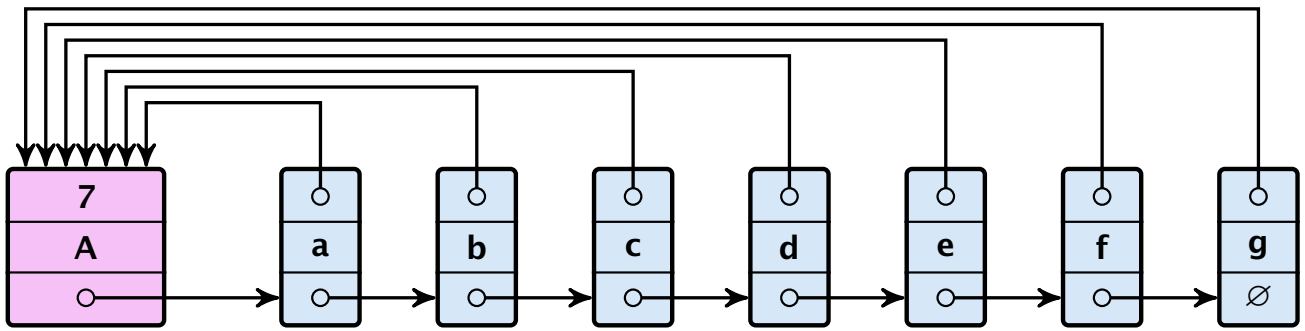
Sei E' eine Teilmenge der Kanten, und sei $e \in E'$ eine billigste Kante aus E', die mit Kanten aus $E - E'$ keinen Kreis schließt. Dann gibt es einen MST, der e enthält.

12 Union Find

```
1  Input: ungerichteter Graph G=(V,E,w)
2  Output: Minimaler Spannbaum von G
3
4  Kruskal(G)
5      A = ∅
6      foreach v ∈ V
7          v->set = P->makeset(v->label)
8      sort edges according to weight w;
9      foreach (u,v) ∈ E in increasing order
10         if (P->find(u->set) != P->find(v->set))
11             A = A ∪ {(u,v)}
12             P->union(u->set,v->set);
```

Listenimplementierung

- Die Elemente einer Teilmenge werden in verketteter Liste gespeichert; jedes Listenelement bekommt zusätzlich einen Zeiger auf den Listenkopf.
- Der Listenkopf enthält den Identifier der Teilmenge und die Größe der Menge



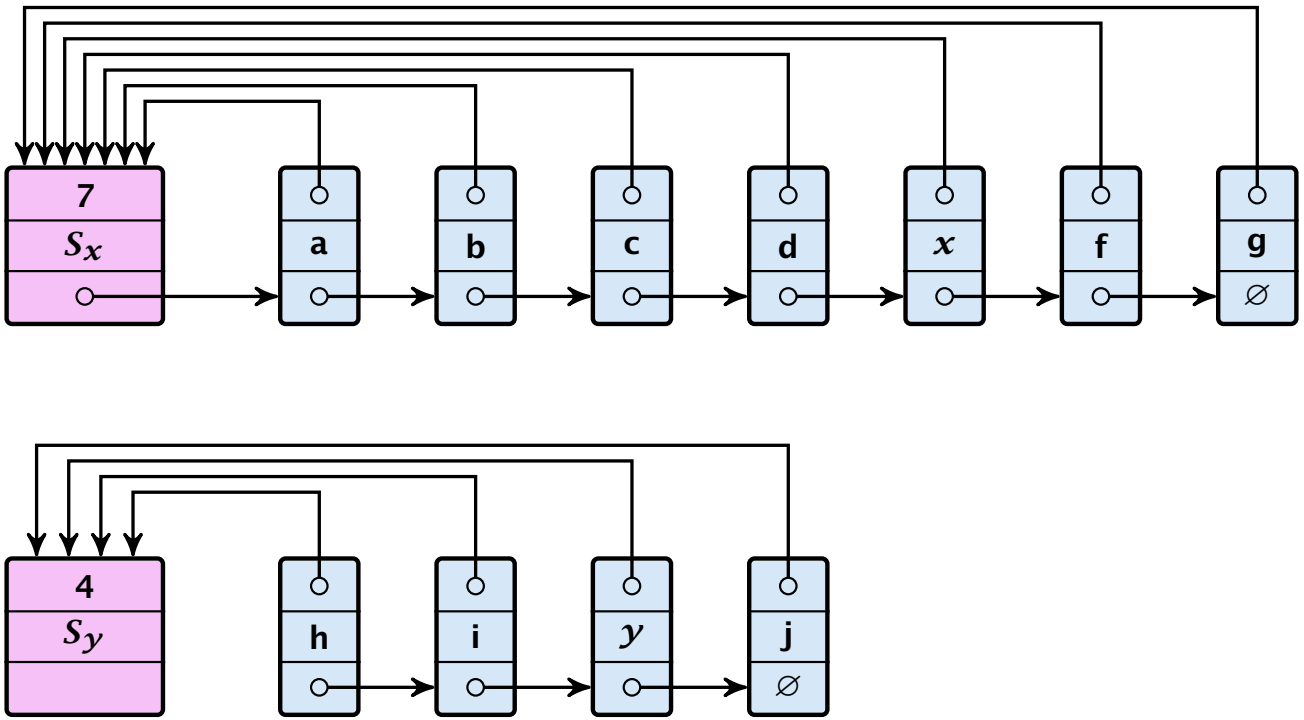
- $\mathrm{makeset}(x)$ kann in konstanter Zeit ausgeführt werden.
- $\mathrm{find}(x)$ kann in konstanter Zeit ausgeführt werden.

Listenimplementierung

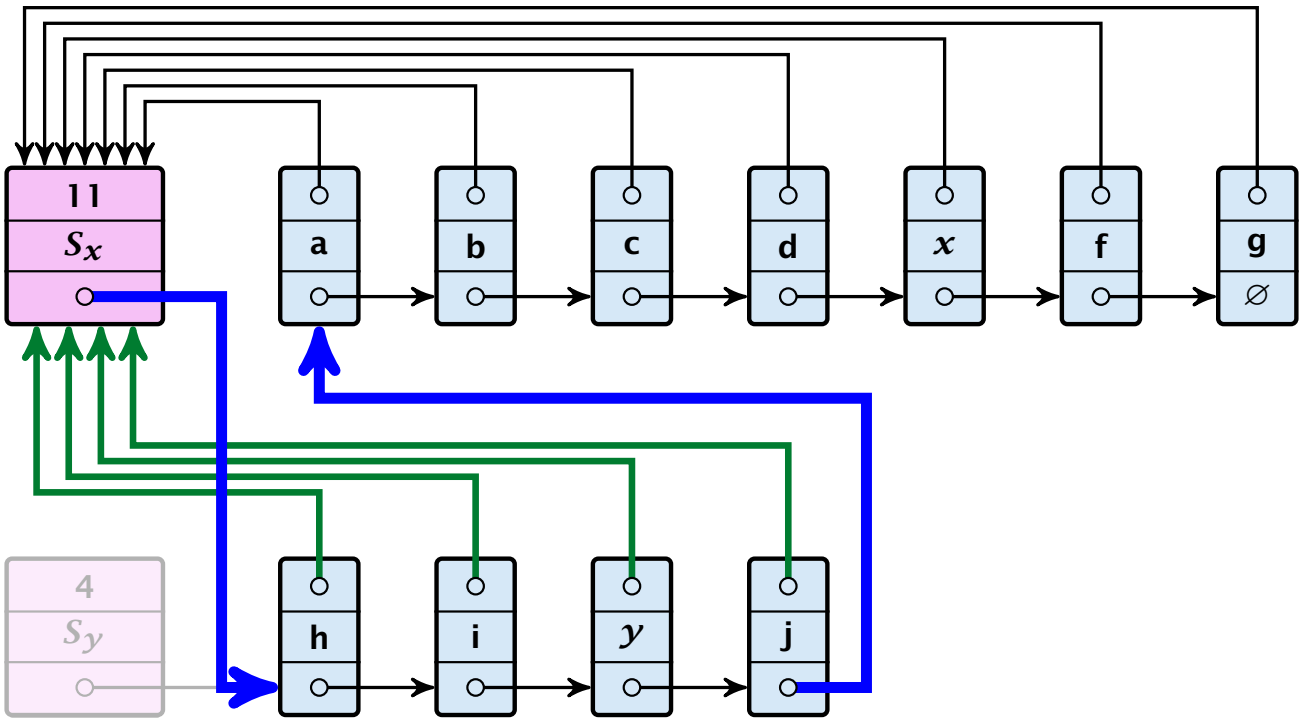
union(x, y)

- Bestimme Mengen S_x und S_y.
- Durchlaufe die kleinere Liste (z.B. S_y), und ändere alle Zeiger auf den Listenkopf, so dass sie auf den Listenkopf von S_x zeigen.
- Füge S_y am Anfang von S_x ein.
- Passe den Größeneintrag von S_x an.
- Laufzeit: $\min\{|S_x|, |S_y|\}$.

List Implementation



List Implementation



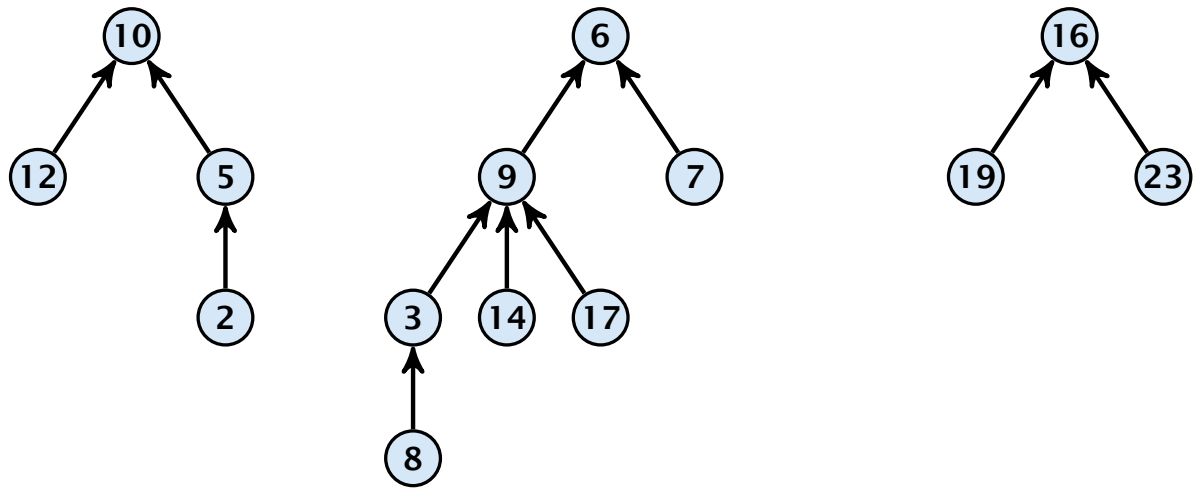
List Implementation

Laufzeiten:

- $\text{find}(x)$: konstant
- $\text{makeset}(x)$: konstant
- $\text{union}(x,y)$: $\mathcal{O}(n)$.

Baumimplementierung

- Verwalte Elemente in Bäumen.
- Die Wurzel eines Baumes dient als Identifier der jeweiligen Teilmenge.
- Nur Elternzeiger existieren; wir können den Baum nicht traversieren und z.B. all Elemente einer Teilmenge ausgeben.
- Beispiel:



Mengensystem $\{2,5,10,12\}$, $\{3,6,7,8,9,14,17\}$, $\{16,19,23\}$.

Baumimplementierung

makeset(x)

- Erzeuge Baum mit einzelnem Element. Gib Zeiger auf Wurzel zurück.
- Zeit: $\mathcal{O}(1)$.

find(x)

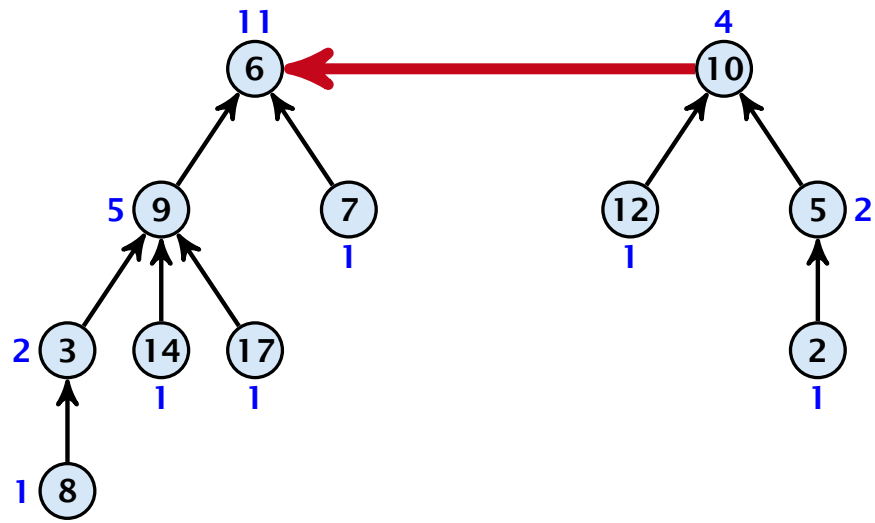
- Handle ist Zeiger auf Element;
- Starte beim Element x im Baum. Gehe aufwärts bis zu Wurzel.
- Gib Identifier der Wurzel zurück.
- Zeit: $\mathcal{O}(\text{level}(x))$, wobei $\text{level}(x)$ die Distanz von x zu Wurzel des jeweiligen Baumes ist. nicht konstant.

Baumimplementierung

Trick: wir speichern für jeden Baum zusätzlich die Anzahl der Elemente des Baumes.

union(x, y)

- Führe $a = \text{find}(x)$; $b = \text{find}(y)$ aus. Dann: $\text{link}(a, b)$.
- $\text{link}(a, b)$ fügt den kleineren Teilbaum als Kind an den größeren an.
- Zusätzlich wird dabei das Größenfeld der neuen Wurzel aktualisert.



- Laufzeit: konstant für $\text{link}(a, b)$ + zwei find-Operationen.

Baumimplementierung

Lemma 1

Laufzeit für $\mathrm{find}(x)$ *ist* $\mathcal{O}(\log n)$.

Beweis.

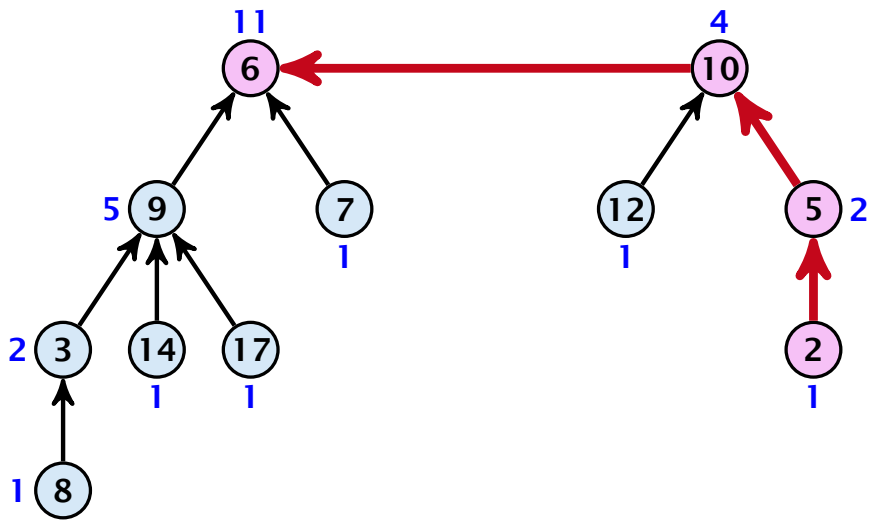
- Wenn wir einen Teilbaum mit Wurzel c an einen Teilbaum mit Wurzel p anfügen, gilt nachher $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$.
- Danach kann sich $\mathrm{size}(c)$ nicht mehr ändern, während $\mathrm{size}(p)$ noch weiter steigen kann (durch weitere union-Operationen).
- D.h. jeder Baum erfüllt immer $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$, für eine Kante (p, c), wobei p der Elternknoten von c ist.
- Deshalb kann die Höhe des Baumes nur $\mathcal{O}(\log n)$ sein.

□

Pfadkomprimierung

find(x):

- Gehe aufwärts bis zur Wurzel.
- Hänge alle besuchten Knoten als Kinder unter die Wurzel.
- Beschleunigt nachfolgende find-Operationen.

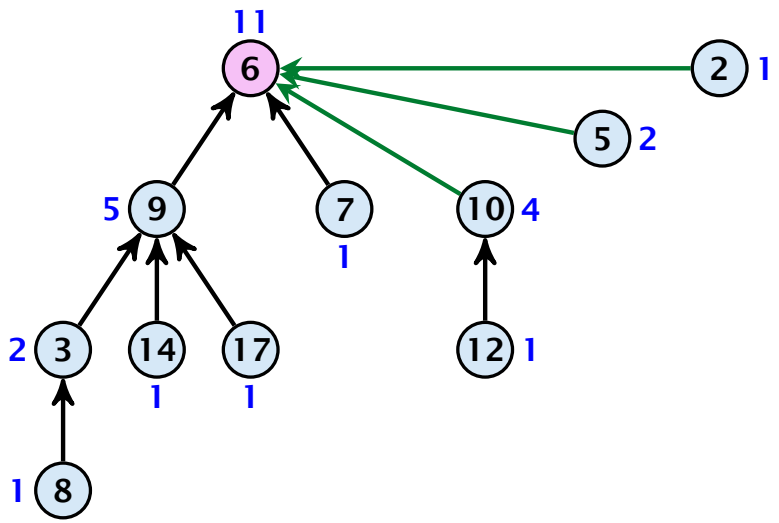


- Beachte, dass die Größenfelder jetzt nur an den Wurzeln der Teilbäume korrekt sind.

Pfadkomprimierung

find(x):

- Gehe aufwärts bis zur Wurzel.
- Hänge alle besuchten Knoten als Kinder unter die Wurzel.
- Beschleunigt nachfolgende find-Operationen.



- Beachte, dass die Größenfelder jetzt nur an den Wurzeln der Teilbäume korrekt sind.

Pfadkomprimierung

Die asymptotische Laufzeit ändert sich durch das Hinzufügen der Pfadkomprimierung nicht.

Die worst-case Laufzeit ist immer noch nur $\mathcal{O}(\log n)$.

Amortisierte Analyse

Bei der amortisierten Analyse betrachtet man nicht die Laufzeit für eine Operation, sondern die durchschnittliche Laufzeit über eine Folge von Operationen (auf höchstens n Elementen beginnend mit leerer Menge).

- Die Listenimplementierung hat eine amortisierte Laufzeit von $\mathcal{O}(\log n)$ für `union`, und $\mathcal{O}(1)$ für `makeset`, und `find`.
- Die Baumimplementierung hat eine amortisierte Laufzeit von $\mathcal{O}(\log^* n)$ für alle Operationen.

$\log^* n$ kann man sich folgendermaßen vorstellen. Man gibt n in den Taschenrechner ein, und zählt wie oft man die log-Taste drücken muß damit man eine Zahl ≤ 1 erhält. Falls n die Anzahl der Atome im (beobachtbaren) Universum ist dann ist $\log^* n = 5$.

Laufzeit Kruskal

- Der Algorithmus sortiert die Kanten gemäß Gewicht; Laufzeit $\mathcal{O}(m \log m)$
- Dann läuft er über alle Kanten; für jede Kante führt er 2 find-Operationen aus und ggf. eine union-Operation. $\mathcal{O}(m \log^* m)$

Insgesamt: $\mathcal{O}(m \log m)$ (dominiert durch das Sortieren der Kanten)