

Algorithmen I

Tutorium 32

Eine Lehrveranstaltung im SS 2017 (mit Folien von Christopher Hommel)

Daniel Jungkind (ufesa@kit.edu) | 14. Juli 2017

INSTITUT FÜR THEORETISCHE INFORMATIK



- \forall , die nicht in die VL gehen: \exists Aufzeichnung vom **letzten Jahr!**
 \Rightarrow <https://www.youtube.com/playlist?list=PLfk0Dfh13pBPqVarZXE8VZxsyBIW8taxe>
- **Gesamtpunkte** der **ÜBs** stehen fest:
Es wird **11 Blätter** geben \Rightarrow **198 P** insgesamt!
 \Rightarrow für 25 % mind. notwendig: 49.5 P
 \Rightarrow für 50 % mind. notwendig: 99 P
 \Rightarrow für 75 % mind. notwendig: 148.5 P
- **Klausuranmeldung** freigeschaltet (bis **spätestens** zum 28.08.!)
Klausur selbst am **04.09.2017** von **11–13 Uhr**.

SPANNBÄUME

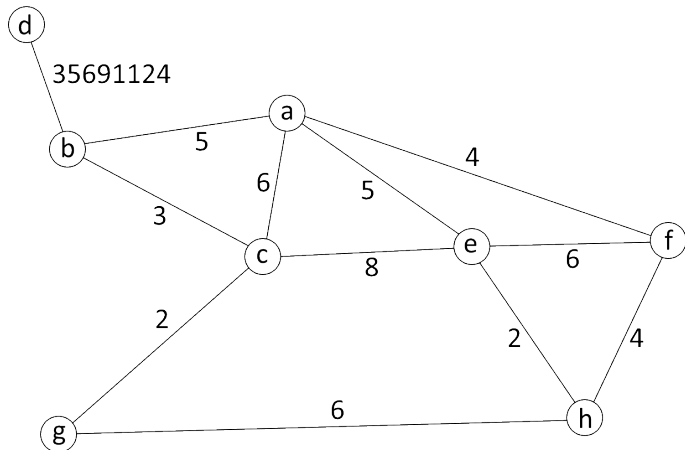
Spannung pur!

Ein paar Definitionen

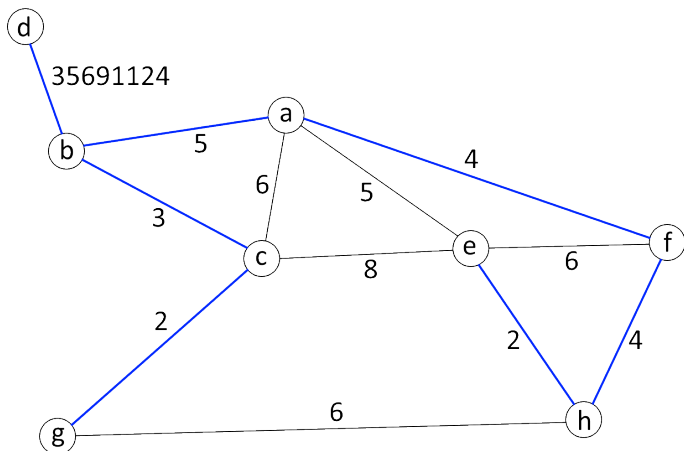
- Für heute: Alle Graphen $G = (V, E)$
ungerichtet, zusammenhängend, mit positiven Kantengewichten
- **Spannbaum:** Baum $(V, T \subseteq E)$ von G ,
„spannt G auf“ (= zusammenhängend)
- Bekannte Algorithmen, die Spannbäume bestimmen:
 - Tiefensuche
 - Breitensuche
 - Dijkstra
- Spannbaum hat **Gewicht** $\sum_{t \in T} c(t)$
- **Minimaler** Spannbaum (MST = Minimum Spanning Tree):
Spannbaum mit **minimalem Gewicht**
- Minimales Gewicht ist **eindeutig**,
minimaler Spannbaum jedoch i. A. **nicht**

Aufgabe 1: Scharf hinsehen

Gebt von folgendem Graphen einen MST an:



Lösung zu Aufgabe 1



Die Schnitteigenschaft („Cut Property“)

- Sei $U \dot{\cup} W = V$ irgendeine „Aufteilung“ von V und $C = \{\{u, w\} \in E \mid u \in U, w \in W\}$ alle „Brücken dazwischen“ (genannt **Schnitt**)

⇒ **Schnitteigenschaft:**

Die **leichteste** Kante $e \in C$ kann in einem MST verwendet werden.

- **Warum?**

⇒ Betrachte MST T . U und W in T durch e verbunden?

⇒ **Ja:** ✓

⇒ **Nein:** Dann U und W durch andere Kante $e' \in C$ verbunden, und e' darf **nicht schwerer** als e sein (sonst ⚡ T minimal) ⇒ e und e' **austauschbar**. □

Die Kreiseigenschaft („Cycle Property“)

- Sei $C \subseteq E$ ein (beliebiger) Kreis in G

⇒ **Kreiseigenschaft:**

Für einen MST T von G wird die **schwerste** Kante $e \in C$ **nicht** benötigt.

- **Warum?**

⇒ Angenommen, $e \in T$ und dafür **leichtere Kreiskante** $e' \notin T$:
Durch **Austausch** von e und e' wird Gewicht von T **kleiner**

⇒ ⚡ T minimal ⇒ Müssen e rausschmeißen.



Ein Algorithmus für MSTs: Jarník-Prim

⇒ **Idee**: Schnitteigenschaft irgendwie ausnutzen!

1. Starte ab beliebigem $s \in V$, setze $S := \{s\}$

2. **Erweitere** Knotenmenge S und Baum T **schrittweise** um die **minimale** Verbindungskante zu $V \setminus S$

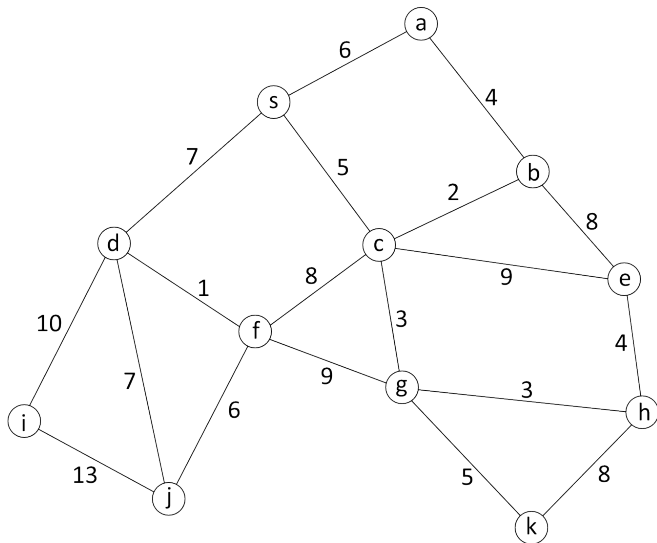
■ Schnittkantenmenge C zwischen S und $V \setminus S$

⇒ Verwalte sie in einer **PriorityQueue** PQ

✓ Funktioniert dank Schnitteigenschaft

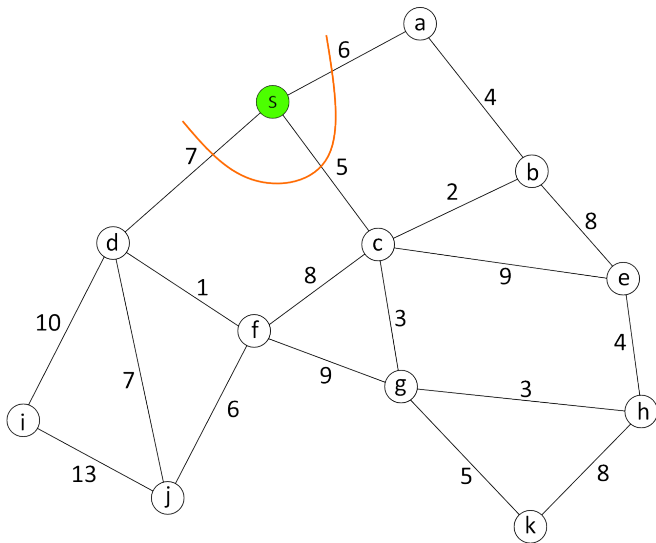
```
function Jarník-Prim( $G = (V, E)$ )    // sieht aus wie Dijkstra
  pick any  $s \in V$ 
   $d := (\infty, \dots, \infty) : \text{array}[1\dots n]$  of  $\mathbb{R}$     // Distanz zu Knotenmenge  $S$ , nicht
    zu  $s \in V!$ 
   $parent := (\perp, \dots, \perp) : \text{array}[1\dots n]$  of  $V$ 
   $PQ = \{s\} : \text{PriorityQueue}$ 
   $parent[s] := s$ 
  while  $PQ \neq \emptyset$  do
     $u := PQ.deleteMin(), \quad d[u] := 0$ 
    foreach  $e = \{u, v\} \in E$  do
      if  $c(e) < d[v]$  then
         $d[v] := c(e)$ 
         $parent[v] := u$ 
        if  $v \in PQ$  then  $PQ.decreaseKey(v)$ 
        else  $PQ.insert(v)$ 
  return  $\{ (parent[v], v) \mid s \neq v \in V \}$ 
```

Spannbäume – Beispiel Jarník-Prim

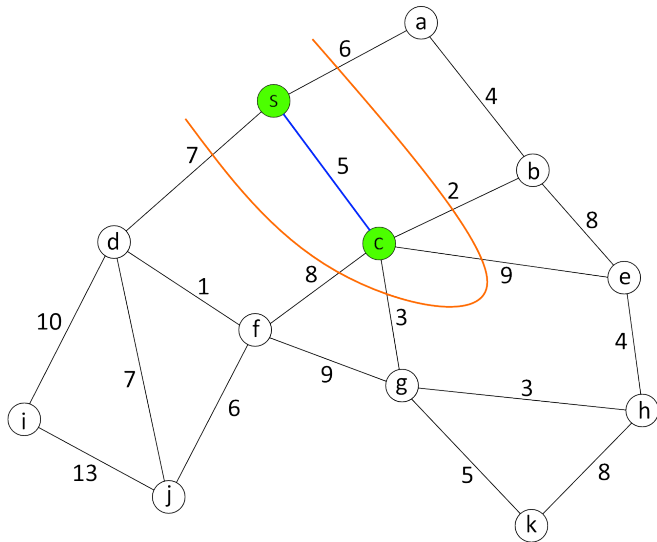


Hier klicken, um das Beispiel zu überspringen.

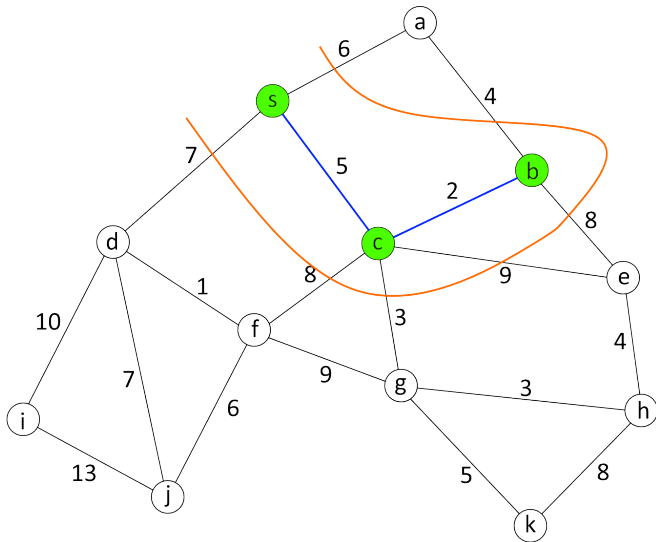
Spannbäume – Beispiel Jarník-Prim



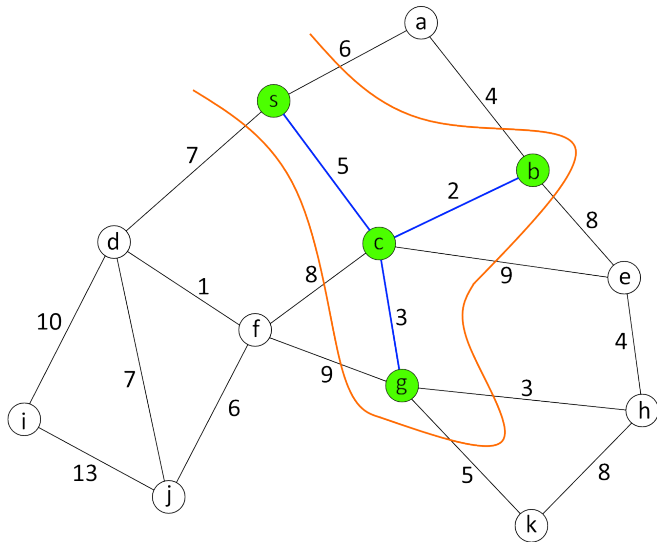
Spannbäume – Beispiel Jarník-Prim



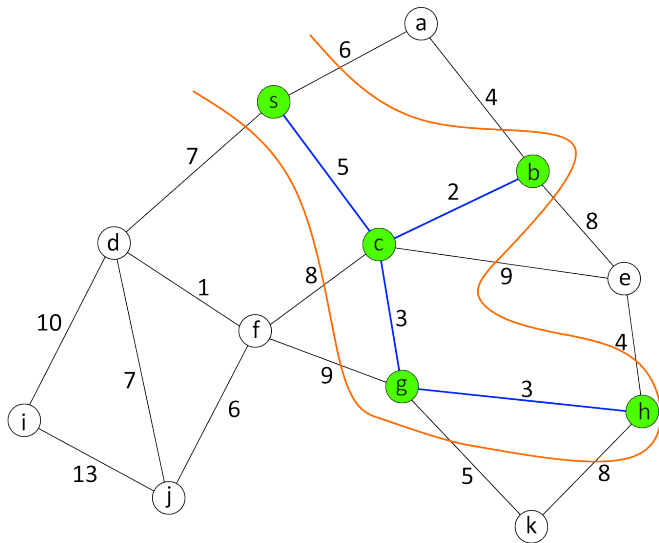
Spannbäume – Beispiel Jarník-Prim



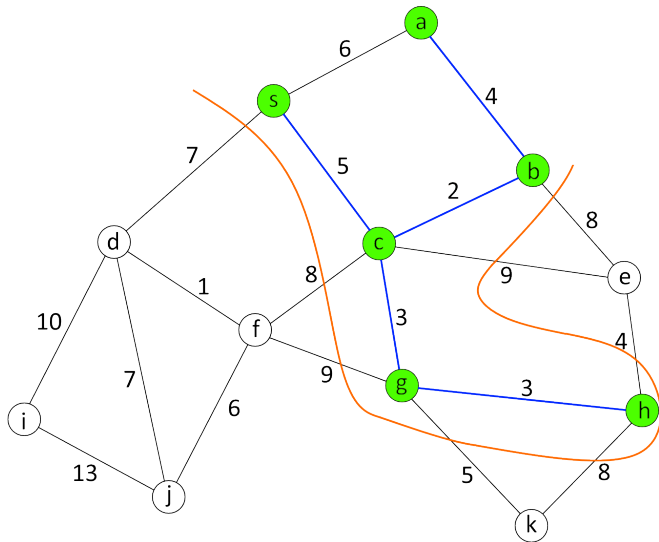
Spannbäume – Beispiel Jarník-Prim



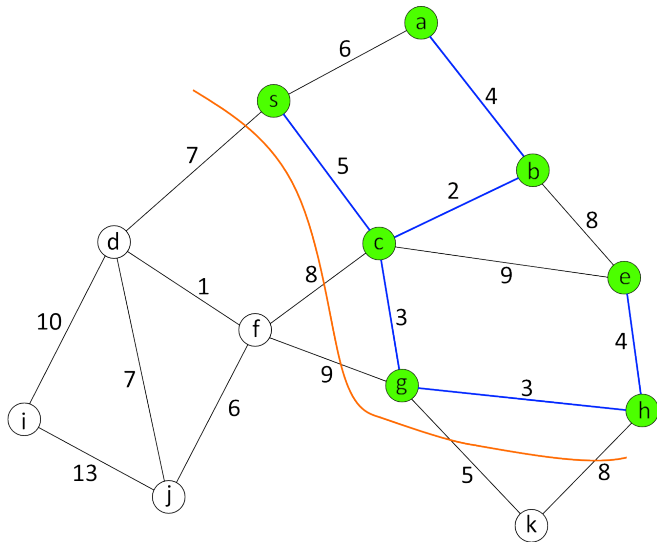
Spannbäume – Beispiel Jarník-Prim



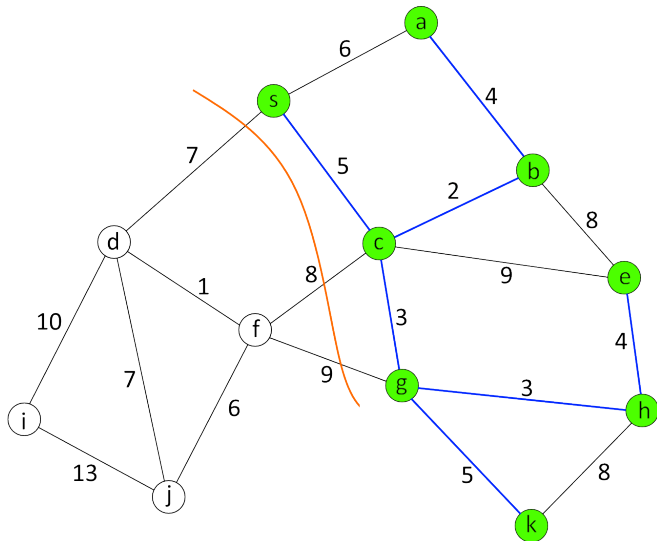
Spannbäume – Beispiel Jarník-Prim



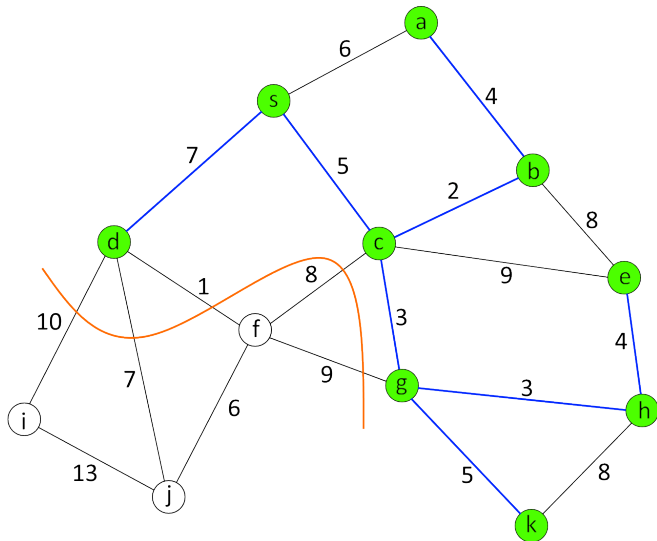
Spannbäume – Beispiel Jarník-Prim



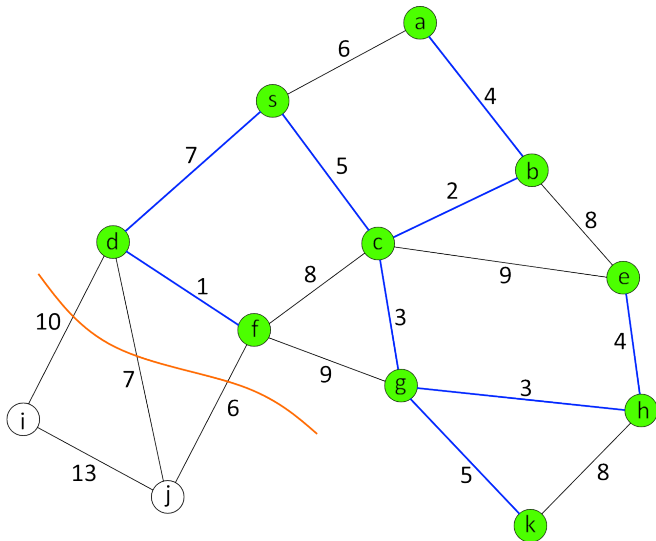
Spannbäume – Beispiel Jarník-Prim



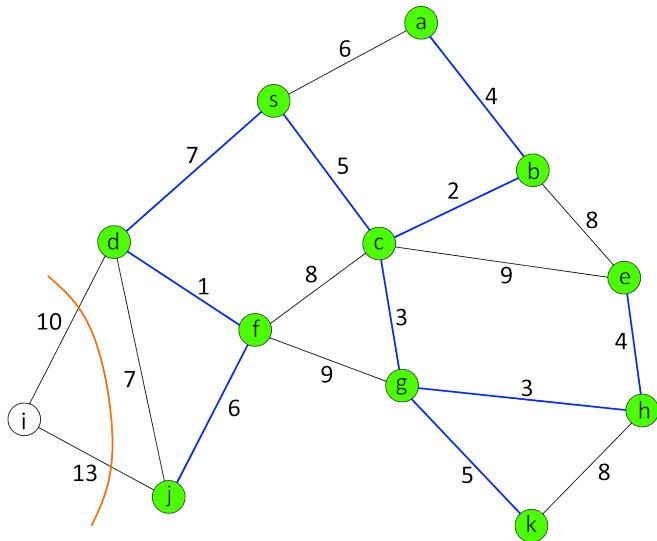
Spannbäume – Beispiel Jarník-Prim



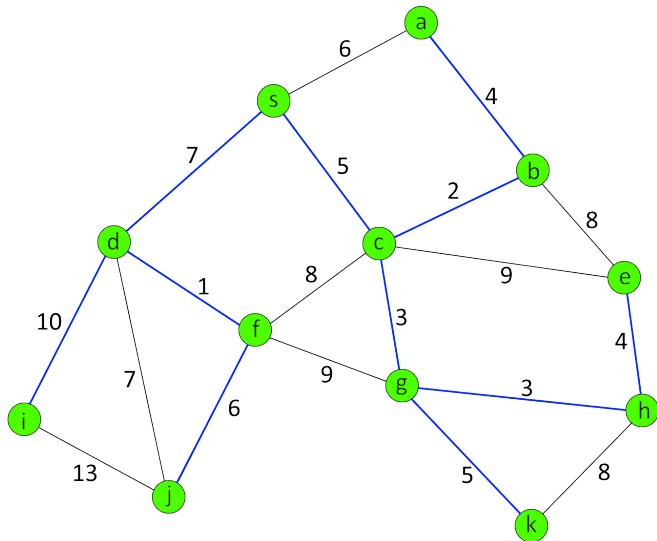
Spannbäume – Beispiel Jarník-Prim



Spannbäume – Beispiel Jarník-Prim



Spannbäume – Beispiel Jarník-Prim



Laufzeit von Jarník-Prim: same as Dijkstra

Im Worst-Case m -mal *decreaseKey*

+ Genau n -mal *deleteMin* und *insert*

= Mit binärem Heap: $O((m + n) \log n)$

= Mit Fibonacci-Heap: $O(m + n \log n)$ (amortisiert und mit höheren konstanten Faktoren)

Rosinen rauspicken mit Kruskal

- **Idee:** Baum (bzw. Wald) **schrittweise wachsen** lassen
- ⇒ Durchlaufe **Kanten** nach **aufsteigendem** Gewicht:
Nehme Kante zum Wald **dazu**, falls dadurch **kein Kreis** entsteht
(also wenn die Kante **zwei separate** Bäume vereinigt)
- Am **Ende:** Alle ausgewählten Kanten ergeben einen MST
- ✓ Funktioniert dank Schnitt- und Kreiseigenschaft

Wollen dafür effizient...

- ...heraus**finden**, zu welcher Menge ein Element gehört (*find*)
- ...die Mengen zweier Elemente **vereinigen** (*union*)

⇒ **Eine neue Datenstruktur!**

Fürs Grobe...

- Repräsentiere Menge $M \subseteq \{1 \dots n\}$ durch **beliebiges** Element $w \in M$
- Intern: M wird als **Baum**, w als **Wurzel** von M behandelt

⇒ Verwalte $parent : \text{array}[1 \dots n]$ of $1 \dots n$, wobei

$parent[v] = v \Leftrightarrow v$ ist Wurzel

Am **Anfang**: $parent[v] := v$ // Alle Elemente sind eigene Wurzel



...und für die Effizienz

⇒ Verwalte $rank = (0, \dots, 0) : \text{array}[1 \dots n]$ of $0 \dots \log n$, wobei

$$rank[v] = \begin{cases} \text{Höhe von Baum von } v \\ \text{(ohne Pfadkompression)} & , \quad v \text{ ist Wurzel} \\ \text{garbage}, & v \text{ ist } \mathbf{keine} \text{ Wurzel} \end{cases}$$

Finden

```
function find( $e : 1..n$ ) :  $1..n$   
  if  $\text{parent}[e] = e$  then  
    return  $e$   
  else  
     $w := \text{find}(\text{parent}[e])$   
    // Pfadkompression  
     $\text{parent}[e] := w$   
    return  $w$ 
```

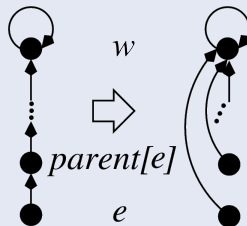


Abbildung: Pfadkompression

Vereinigen

procedure union($a', b' : 1 \dots n$)

$\begin{pmatrix} a \\ b \end{pmatrix} := \begin{pmatrix} \text{find}(a') \\ \text{find}(b') \end{pmatrix}$

if $a \neq b$ **then**

// union by rank

if $\text{rank}[a] < \text{rank}[b]$ **then**

$\text{parent}[a] := b$

else

$\text{parent}[b] := a$

if $\text{rank}[a] = \text{rank}[b]$ **then**

$\text{rank}[a] ++$

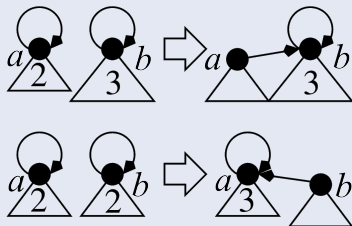
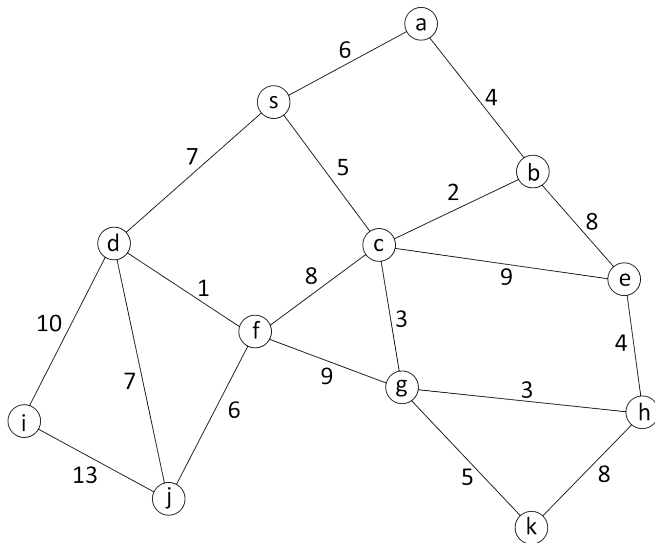


Abbildung: Union-by-rank

Kruskal – Pseudocode

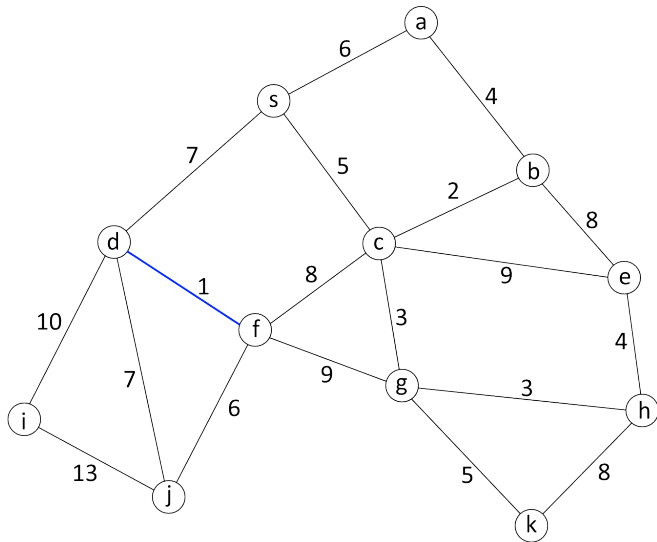
```
function Kruskal( $G = (V, E)$ )  
   $forest := \text{new UnionFind}(n)$   
   $T := \emptyset$   
  Sort( $E$ ) by  $c$   
  foreach  $e = \{u, v\} \in E$  do  
    if  $forest.find(u) \neq forest.find(v)$  then  
       $T := T \cup \{e\}$   
       $forest.union(u, v)$   
  return  $T$ 
```

Spannbäume – Beispiel Kruskal

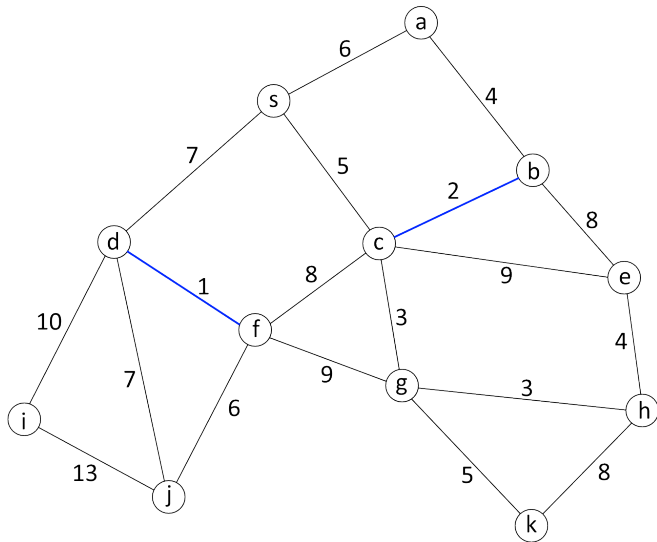


Hier klicken, um das Beispiel zu überspringen.

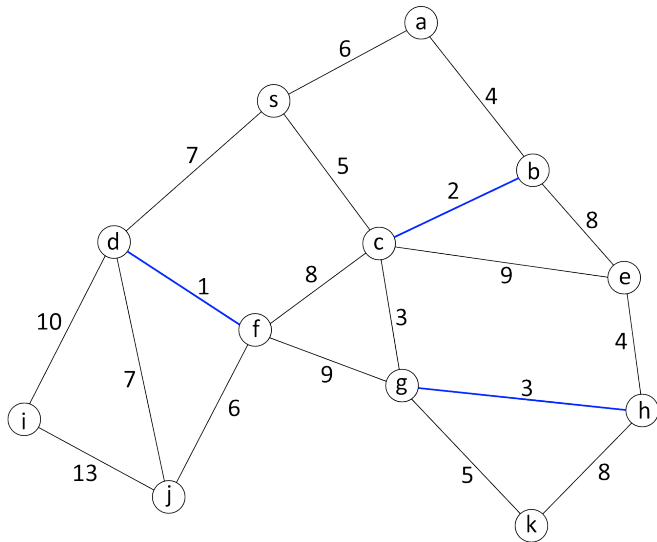
Spannbäume – Beispiel Kruskal



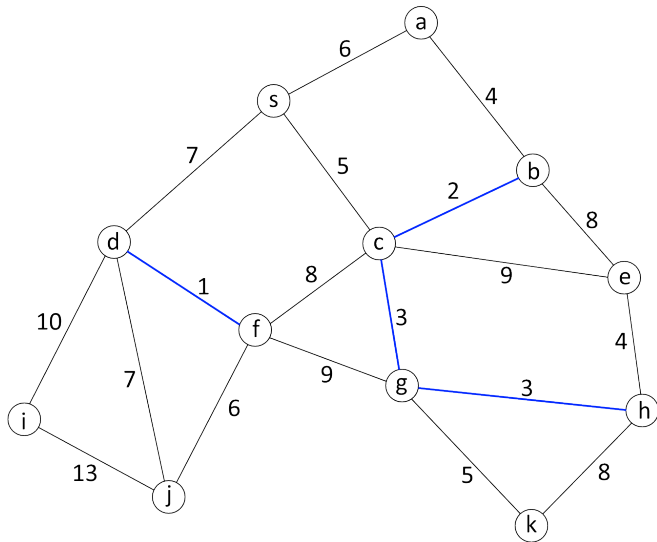
Spannbäume – Beispiel Kruskal



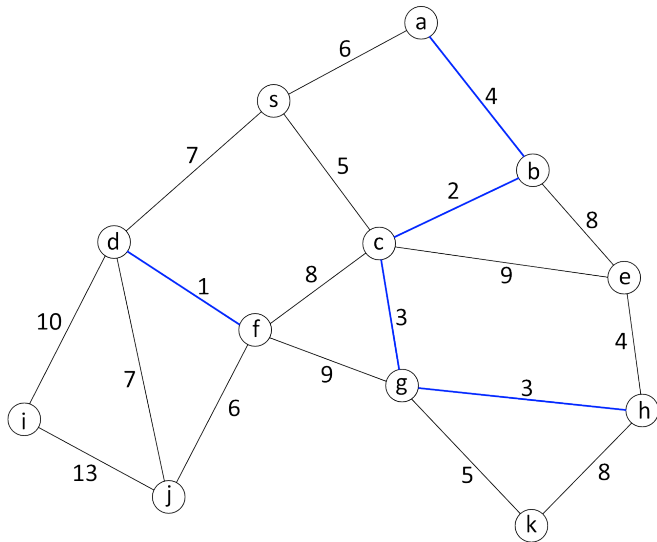
Spannbäume – Beispiel Kruskal



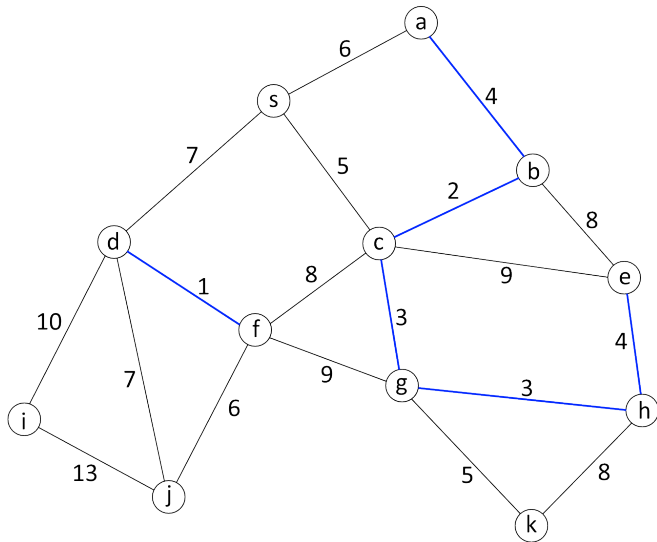
Spannbäume – Beispiel Kruskal



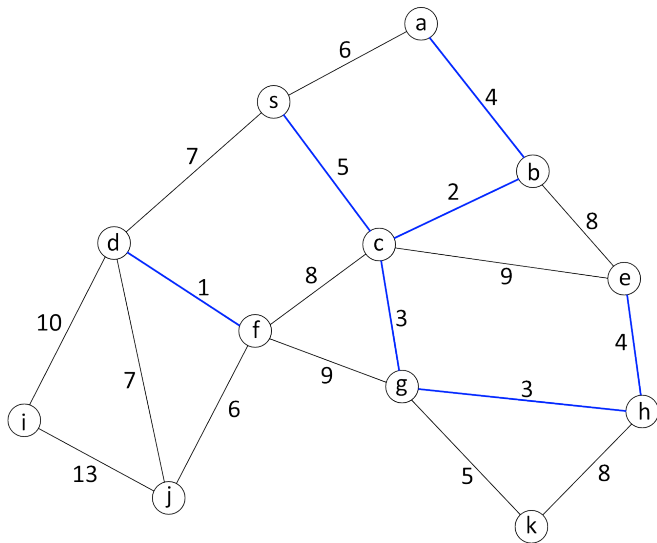
Spannbäume – Beispiel Kruskal



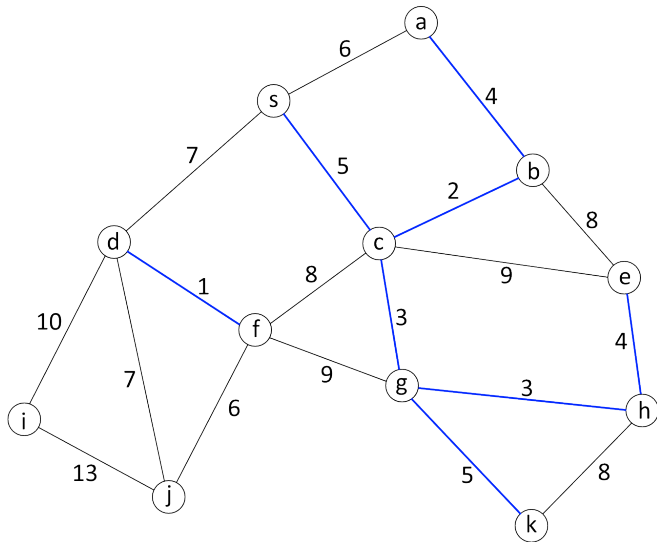
Spannbäume – Beispiel Kruskal



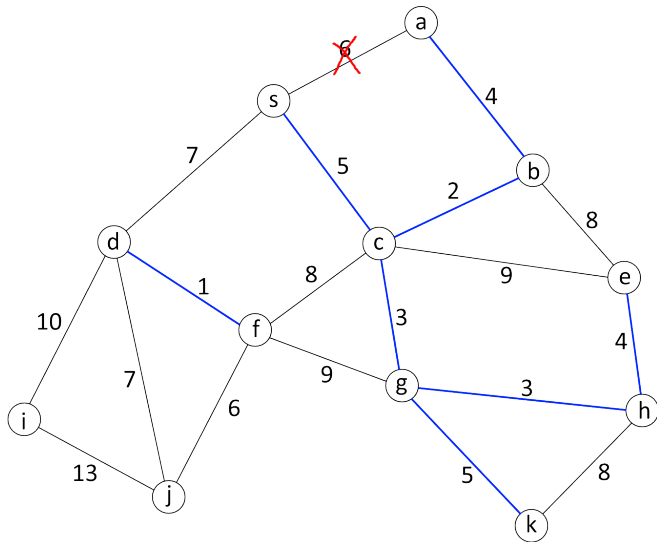
Spannbäume – Beispiel Kruskal



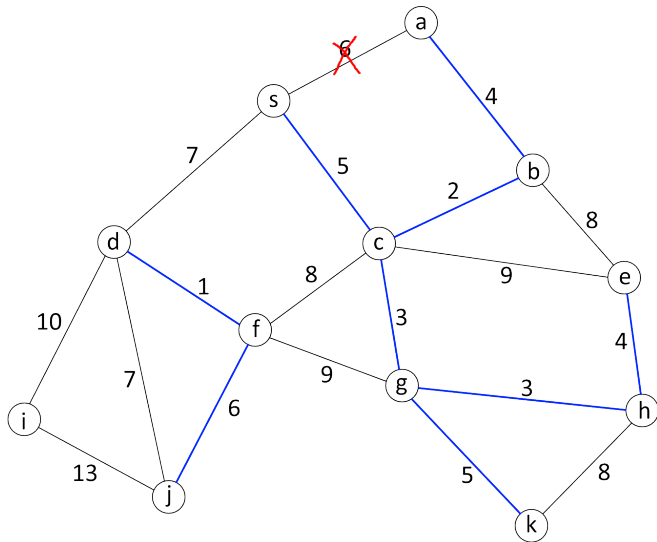
Spannbäume – Beispiel Kruskal



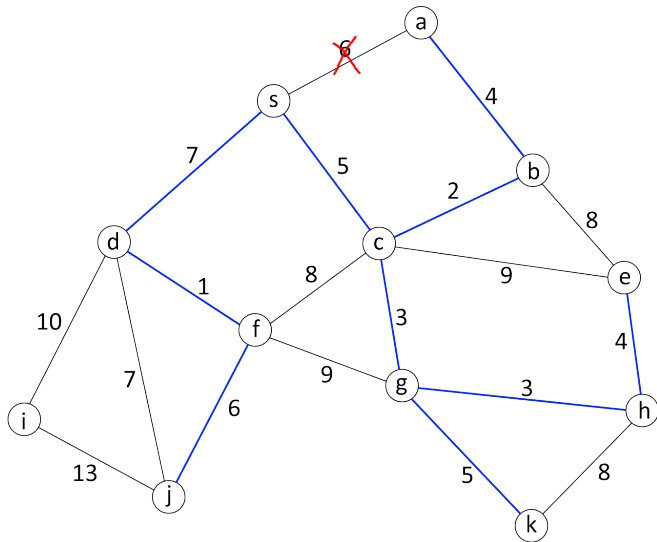
Spannbäume – Beispiel Kruskal



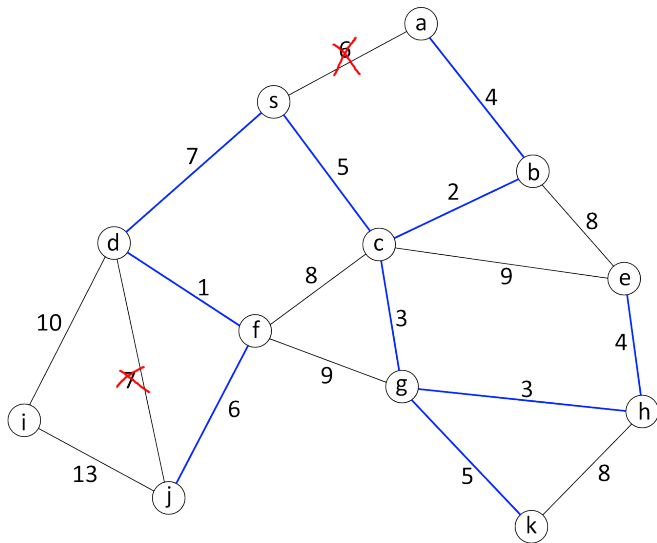
Spannbäume – Beispiel Kruskal



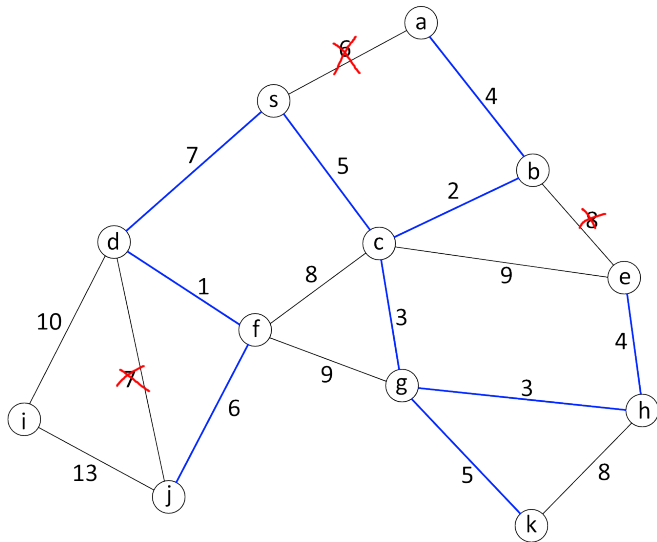
Spannbäume – Beispiel Kruskal



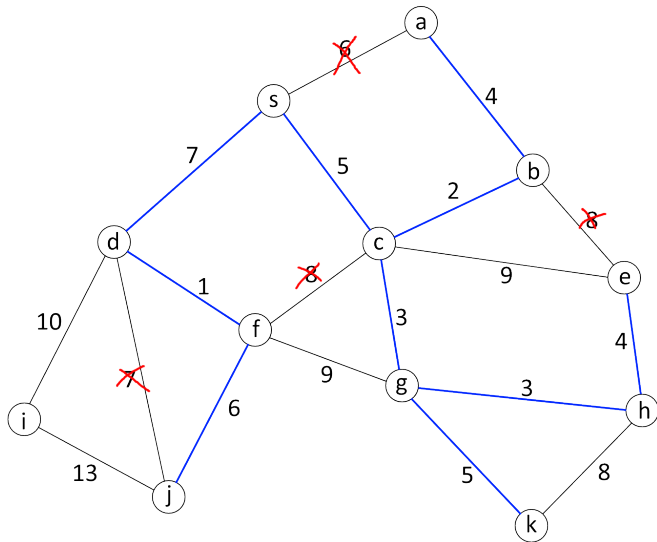
Spannbäume – Beispiel Kruskal



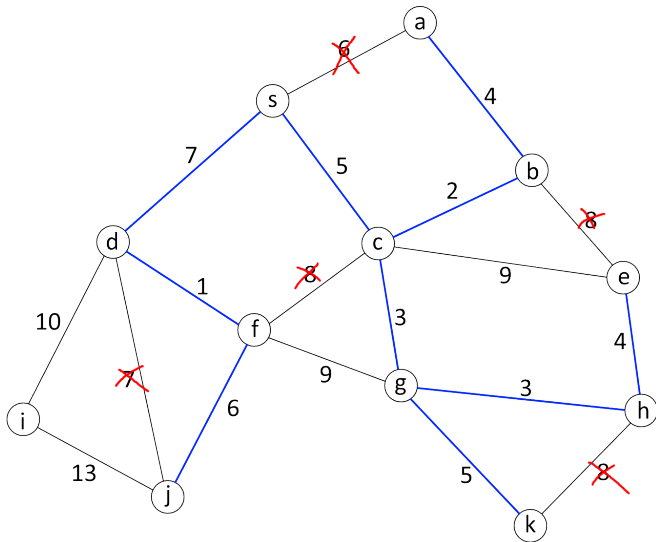
Spannbäume – Beispiel Kruskal



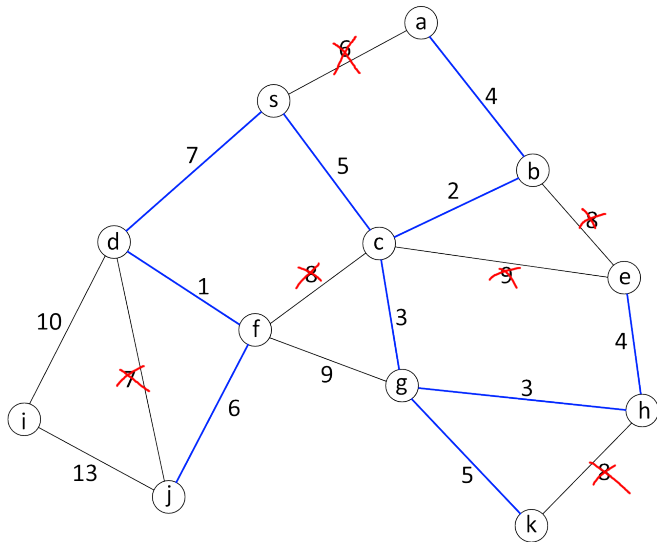
Spannbäume – Beispiel Kruskal



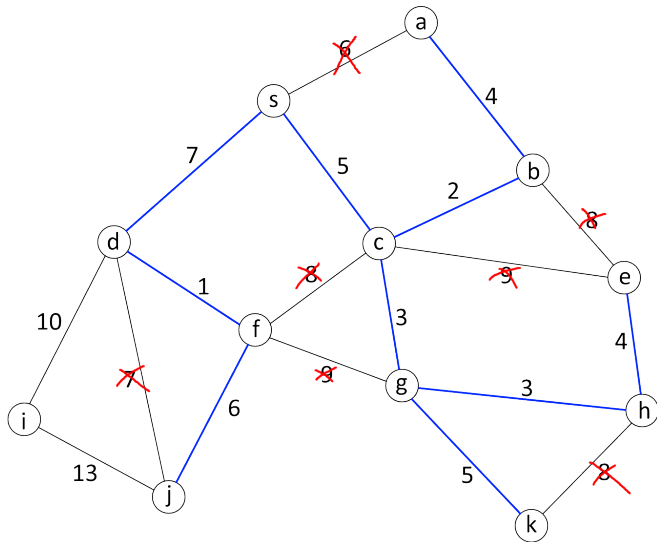
Spannbäume – Beispiel Kruskal



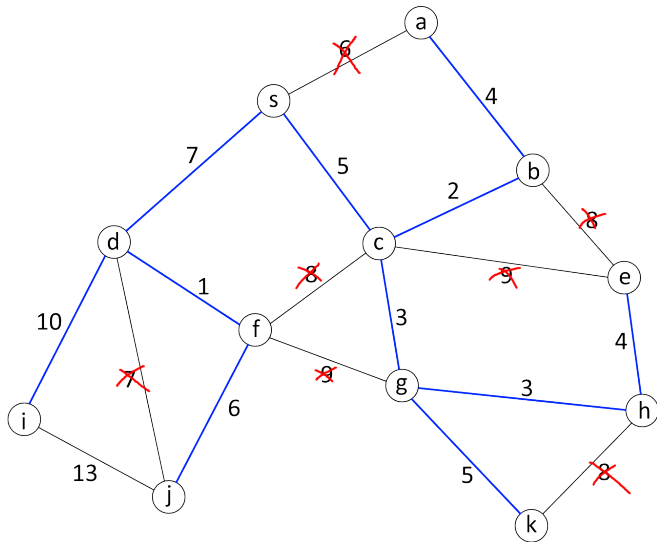
Spannbäume – Beispiel Kruskal



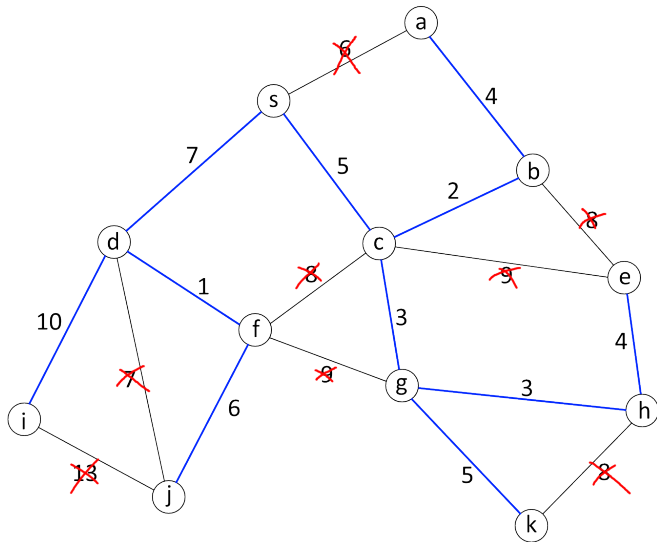
Spannbäume – Beispiel Kruskal



Spannbäume – Beispiel Kruskal



Spannbäume – Beispiel Kruskal



Laufzeit von Kruskal

$O(m \log m)$ fürs Sortieren von E

+ $O(m \cdot \alpha_T(m, n))$ für $m \times \text{find}() + n \times \text{union}()$ laut VL
 $\approx O(m \cdot 5)$ (**Inv. Ackermannfunktion** $\alpha_T(\cdot, \cdot) \leq 5$ for all sane inputs)

= $O(m \log m)$.

Bei Kantengewichten in \mathbb{Z}_+ sogar schneller möglich.

Die Union-Find-Datenstruktur bei Kruskals Algorithmus repräsentiert den bisher gefundenen MST.

Falsch.

Bloß interne Hierarchie für die Knoten. MST-Kanten tauchen da drin gar nicht auf.

Dijkstra ist zur Bestimmung eines MST bei gerichteten Graphen (mit nichtnegativen Kantengewichten) geeignet.

Falsch.

Dijkstra bestimmt im Allgemeinen **keinen** MST (und MST auf gerichteten Graphen nicht in dieser VL).

Sowohl der Algorithmus von Jarník-Prim als auch Kruskals Algorithmus funktionieren auch bei negativen Kantengewichten.

Wahr.

(JP nach kleiner Anpassung)

Dijkstra funktioniert nicht, wenn negative Kantengewichte vorhanden sind.

Falsch.

Dijkstra funktioniert nicht, wenn es negative **Zyklen** gibt
(\Rightarrow Endlosschleife).

Ansonsten bestimmt Dijkstra auch **mit** negativen Kantengewichten **korrekte kürzeste Pfade**, aber in deutlich **schlechterer Laufzeit**.

(Achtung, **nicht** in VL gezeigt!)

Bellman-Ford bestimmt stets einen **beliebigen** Spannbaum.

Falsch.

Nur wenn keine negativen Zyklen vorhanden sind!

Aufgabe 2: Streaming MST

Gegeben sei ein ungerichteter zusammenhängender Graph $G = (V, E)$ mit n Knoten, m Kanten und positiven Kantengewichten. Die Knoten sind lokal gespeichert, die Kanten sind hingegen zunächst **unbekannt** und können nur **stückweise** (und in zufälliger Reihenfolge) aus dem Netz angefordert und im Speicher gehalten werden, da **nur $O(n)$ Platz** zur Verfügung steht. Gebt einen Algorithmus an, der unter diesen Einschränkungen einen MST von G bestimmt.

Lösung zu Aufgabe 2

- Verwende eine **Union-Find-Datenstruktur** (wie bei Kruskal).
- Falls die neu angeforderte Kante **zwei Teilbäume verbindet**, füge sie hinzu.
- Falls sie zwei Knoten **im selben** Teilbaum verbindet, füge sie provisorisch hinzu, schmeiße auf dem Pfad zwischen den Knoten die **schwerste Kante** raus.
- **Laufzeit** in $O(m \cdot n)$, da bei der Bestimmung des Pfades maximal $n - 1$ Kanten abgelaufen werden

Aufgabe 3: Streaming MST mit Dünger

Wie bei Aufgabe 2:

Gegeben sei ein ungerichteter zusammenhängender Graph $G = (V, E)$ mit n Knoten, m Kanten und positiven Kantengewichten. Die Knoten sind lokal gespeichert, die Kanten sind hingegen zunächst **unbekannt** und können nur **stückweise** (und in zufälliger Reihenfolge) aus dem Netz angefordert und im Speicher gehalten werden, da **nur** $O(n)$ **Platz** zur Verfügung steht. Gebt einen Algorithmus an, der unter diesen Einschränkungen einen MST von G bestimmt.

Er darf **nur** $O(m \log n)$ **Rechenzeit benötigen**.

Lösung zu Aufgabe 3

Wir besorgen uns Kanten-Pakete mit jeweils n Kanten.

Mit dem **ersten** Paket bestimmen wir (mit Kruskal) einen Minimum Spanning Forest (MSF) und entfernen alle anderen Kanten.

Restliche Pakete behandeln wir so:

- Füge die n neuen Kanten zum Graphen hinzu
- Bestimme auf diesem neuen Graphen mit (maximal) $2n - 1$ Kanten einen MSF und entferne alle anderen Kanten

⇒ **Laufzeit** in $O(m \log n)$, denn:

Jeder dieser ca. $\frac{m}{n}$ Schritte benötigt Zeit in $O(n \log n)$.

Aufgabe 4: Ein Algorithmus mit Ecken und Kanten

Erneut betrachten wir einen ungerichteten zusammenhängenden Graphen $G = (V, E)$ mit $V = \{1 \dots n\}$ und Kantengewichten in $\{1, 3\}$. G sei in Form eines Adjazenzfeldes gegeben. Gebt einen Algorithmus an, der in $O(m)$ einen MST von G berechnet (und begründet das Laufzeitverhalten).

Lösung zu Aufgabe 4

Getweaktes **Jarník-Prim**:

- Komplette PriorityQueue wäre **overkill** \Rightarrow **stattdessen** zwei einfache Queues Q_1 und Q_3 (eine für jedes Kantengewicht)
 - Merke zu jedem Knoten Pointer in die Queue (oder \perp)
- \Rightarrow *insert*, *deleteMin* und *decreaseKey* in $O(1)$ durch einfaches Pointer-Umhängen
- \Rightarrow **Laufzeit**: $O(n + m) = O(m)$
(da für zusammenhängende Graphen $n \in O(m)$).

(Eine clevere Alternative wäre, die Kanten mit **Bucketsort** zu sortieren und dann **Kruskal** draufloszulassen \Rightarrow Laufzeit:

$O(m \cdot \alpha_T(m, n)) \overset{\text{realistisch}}{\approx} O(5 \cdot m) = O(m)$. Aber Achtung, α_T ist **nicht** konstant, deshalb „ \approx “!)