

# Informatik II - SS 2016

## (Algorithmen & Datenstrukturen)

Vorlesung 15 (15.6.2016)

### Graphenalgorithmen III (MST)



**UNI  
FREIBURG**

Fabian Kuhn

Algorithmen und Komplexität

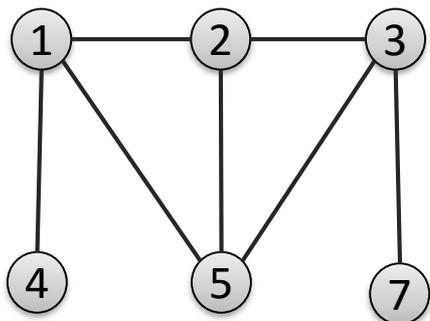
# DFS- “Klammer”-Theorem

Wir definieren für jeden Knoten  $v$  die folgenden zwei Zeitpunkte

- $t_{v,1}$ : Zeitpunkt, wenn  $v$  in der DFS-Suche grau gefärbt wird
- $t_{v,2}$ : Zeitpunkt, wenn  $v$  in der DFS-Suche schwarz gefärbt wird

**Theorem:** Im DFS-Baum ist ein Knoten  $v$  genau dann im Teilbaum eines Knoten  $u$ , falls das Intervall  $[t_{v,1}, t_{v,2}]$  vollständig im Intervall  $[t_{u,1}, t_{u,2}]$  enthalten ist. Zudem sind zwei Intervalle entweder disjunkt, oder das eine ist komplett im anderen enthalten.

**Beispiel:**



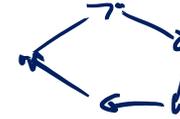


**Theorem:** Bei einer DFS-Suche in ungerichteten Graphen ist jede Kante entweder eine Baumkante oder eine Rückwärtskante.

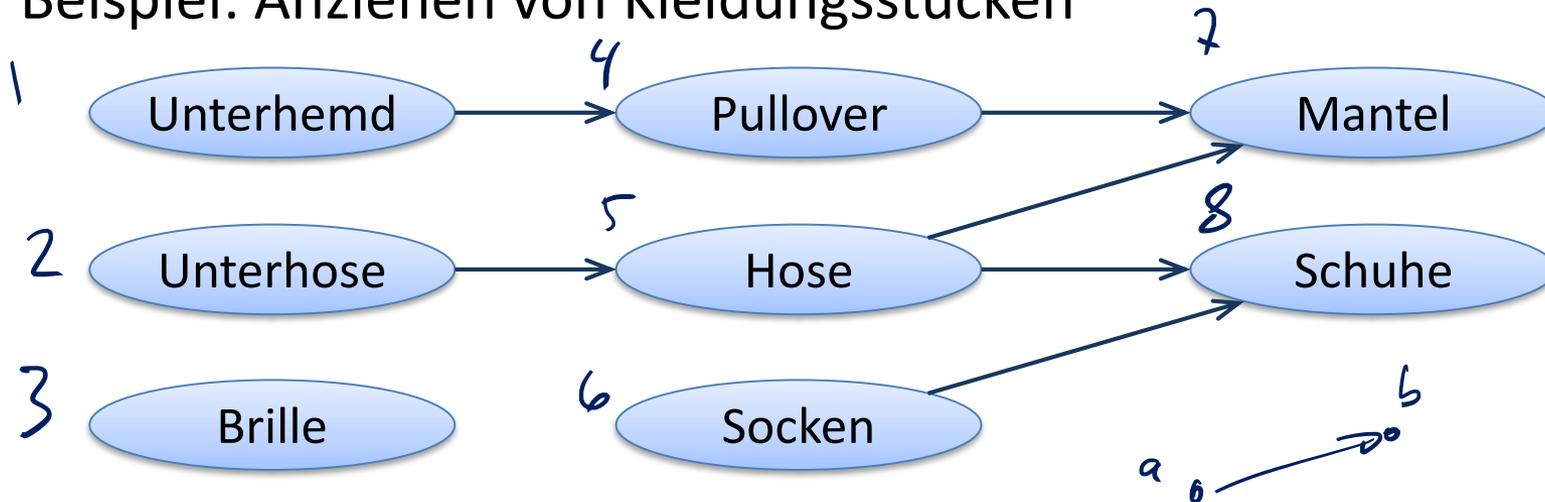
**Theorem:** Ein gerichteter Graph hat genau dann keine Zyklen, falls es bei der DFS-Suche keine Rückwärtskanten gibt.

# Anwendung: Topologische Sortierung

## Zyklenfreie, gerichtete Graphen:



- **DAG**: directed acyclic graph
- Modellieren z.B. zeitliche Abhängigkeiten von Aufgaben
- Beispiel: Anziehen von Kleidungsstücken



## Topologische Sortierung:

- Sortiere die Knoten eines DAGs so, dass  $u$  vor  $v$  erscheint, falls ein gerichteter Pfad von  $u$  nach  $v$  existiert
- Im Beispiel: Finde eine mögliche Anziehreihenfolge

## Zyklenfreie, gerichtete Graphen:

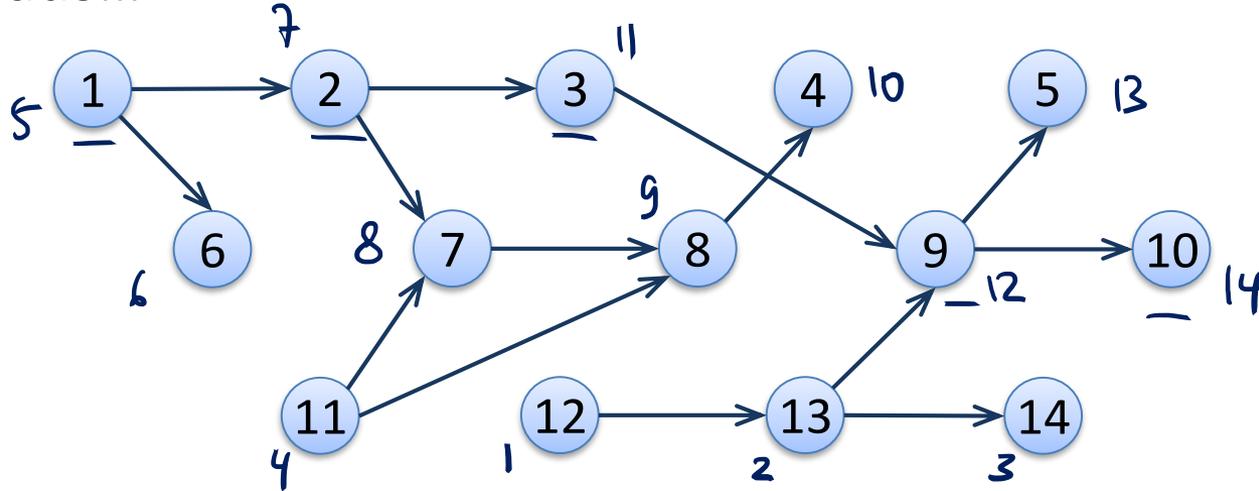
- repräsentieren partielle Ordnungsrelationen
  - asymmetrisch:  $a < b \Rightarrow \neg(b < a)$
  - transitiv:  $a < b \wedge b < c \Rightarrow a < c$
  - partielle Ordnung: nicht alle Paare müssen vergleichbar sein
- Beispiel: Teilmengenrelation bei Mengen

## Topologische Sortierung:

- Sortiere die Knoten eines DAGs so, dass  $u$  vor  $v$  erscheint, falls ein gerichteter Pfad von  $u$  nach  $v$  existiert
- Erweitere eine partielle Ordnung zu einer totalen Ordnung

# Topologische Sortierung: Algorithmus

Führe DFS aus...



$t_{1,1}, t_{2,1}, t_{3,1}, t_{9,1}, t_{10,1}, t_{10,2}, t_{5,1}, t_{5,2}, t_{9,2}, t_{3,2}, t_{7,1}, t_{8,1}, t_{4,1}, t_{4,2}, t_{8,2},$   
 $t_{7,2}, t_{2,2}, t_{6,1}, t_{6,2}, t_{1,2}, t_{11,1}, t_{11,2}, t_{12,1}, t_{13,1}, t_{14,1}, t_{14,2}, t_{13,2}, t_{12,2}$

## Beobachtung:

- Knoten ohne Nachfolger werden als erstes besucht (schwarz gef.)
- Besuchreihenfolge ist umgekehrte topologische Sortierung

# Topologische Sortierung: Algorithmus

**Theorem:** Umgekehrte "Visit"-Reihenfolge (schwarz färben) der Knoten bei DFS-Traversierung ergibt topologische Sortierung



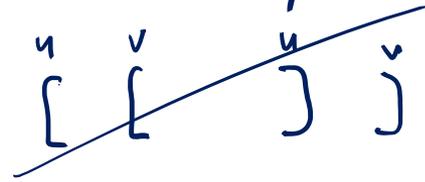
DFS betrachtet Kante  $u \rightarrow v$ .  $v$  ist weiss oder schwarz (zyklenfr. Graph)

v weiss:

$(u,v)$  wird Baumkante



$$t_{u,1} < t_{v,1}$$



Klammerthm.  $\implies t_{v,2} < t_{u,2} \checkmark$

v schwarz:

$u$  ist grau  $\implies t_{u,2} > t_{v,2} \checkmark$



## Stark zusammenhängende Komponenten

- Stark zus.-hängende Komponente eines gerichteten Graphen:  
“Maximale Knoten-Teilmenge, so dass jeder jeden erreichen kann”

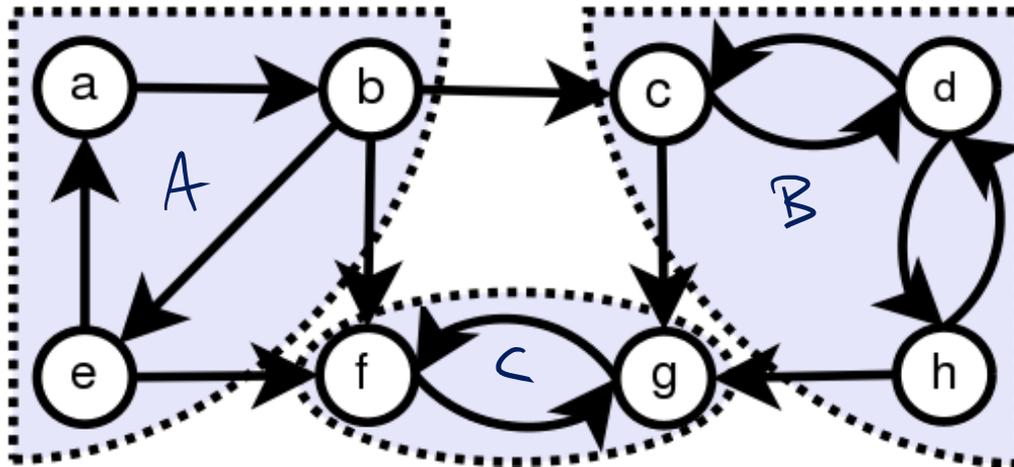
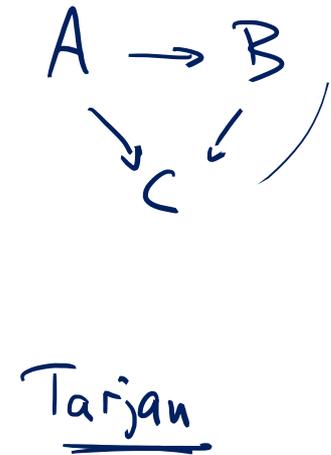


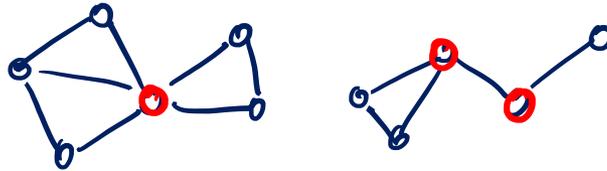
Bild: Wikipedia



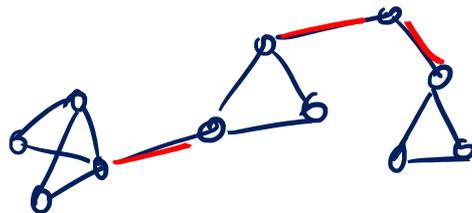
- Kann man mit 2 DFS-Traversierungen finden (Zeit  $\in O(\underline{\underline{m + n}})$ )
  - auf  $G$  und auf  $G^T$  (alle Kanten umgedreht)
  - $G$  und  $G^T$  haben die gleichen stark zus.-hängenden Komponenten
- Details z.B. in [CLRS]

## Artikulationsknoten, Brücken, Biconnected Components

- Annahme: ungerichteter Graph
- **Artikulationsknoten  $v$ :**  
 $v$  entfernen vergrößert die Anzahl Komponenten



- **Brücke  $e$ :**  
Kante  $e$  entfernen vergrößert die Anzahl Komponenten



## Biconnected Components

- Komponenten, welche übrig bleiben, wenn man alle Brücken entfernt

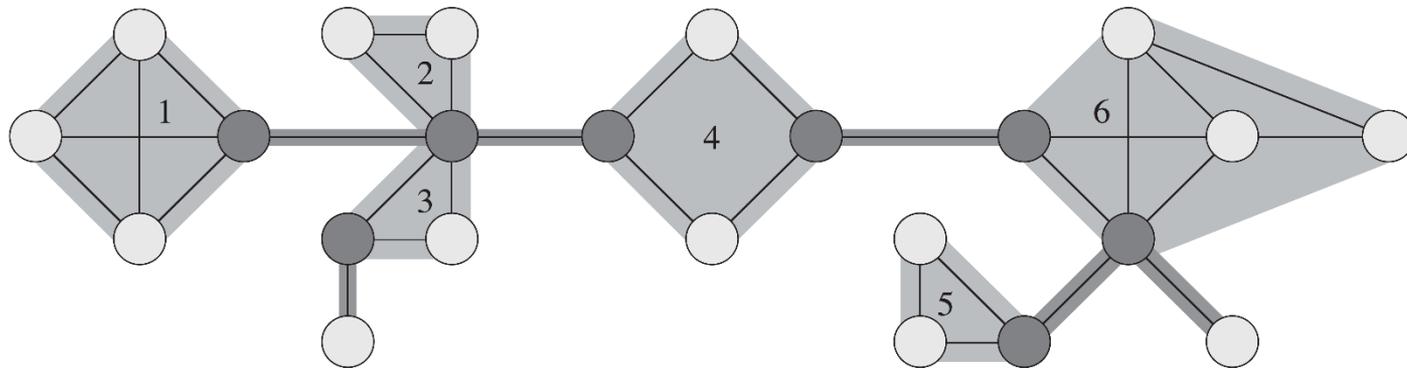


Bild: [CLRS]

- Artikulationsknoten und Brücken können mit einer DFS-Traversierung in  $O(m + n)$  Zeit gefunden werden
  - Algorithmus von Hopcroft, Tarjan (1973)
- Zerlegung in Biconnected Components daher in der gleichen Zeit

- Als ungerichtete Graphen (mit  $n$  Knoten) betrachtet...

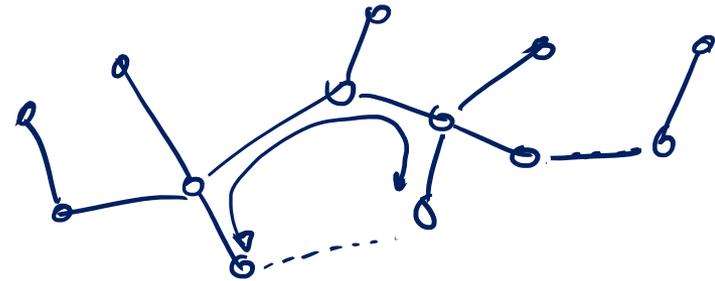
## Baum:



- Zusammenhängender ungerichteter Graph, ohne Zyklen
  - Ein nicht zus.-hängender zyklensfreier (unger.) Graph heisst Wald
  - Anzahl Kanten:  $n - 1$  (jede Kante reduziert die #Komponenten um 1)

minimal: eine Kante entfernen  
→ nicht mehr zus.-hgd.

maximal: eine Kante hinzufügen  
→ Zyklus



## Äquivalente Definitionen:

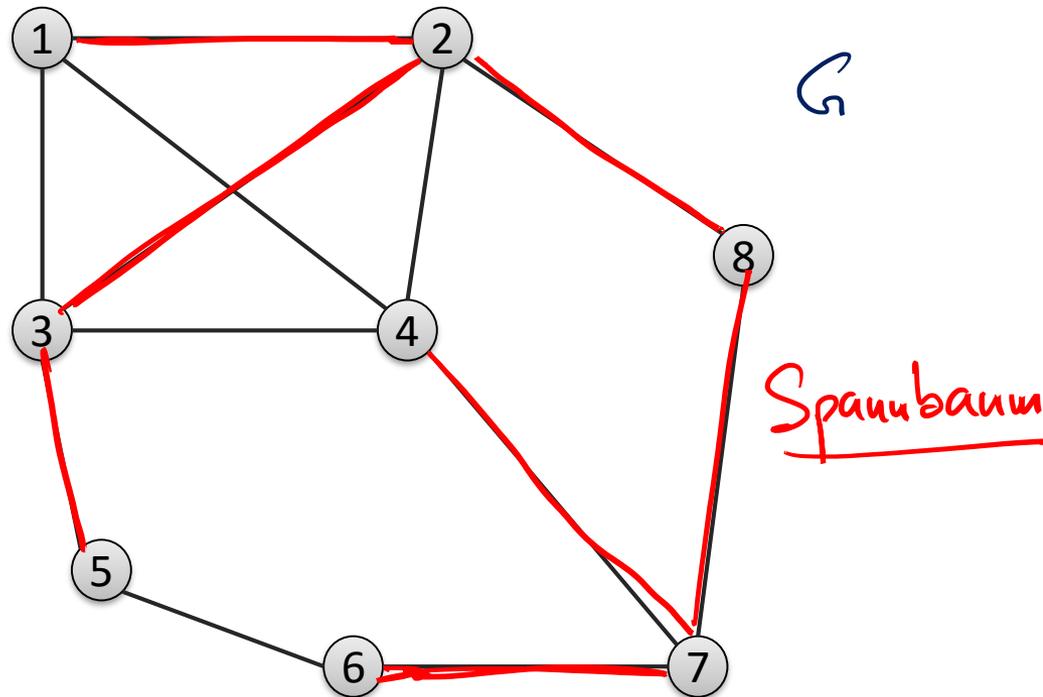
- Minimaler zusammenhängender Graph
- Maximaler zyklensfreier Graph
- Eindeutiger Pfad zwischen jedem Knotenpaar
- Zusammenhängender Graph mit  $n - 1$  Kanten

# Spannbaum

**Gegeben:** Zusammenhängender, ungerichteter Graph  $G = (V, E)$

**Spannbaum  $T = (V, E_T)$ :** Teilgraph ( $E_T \subseteq E$ )

- $T$  ist ein Baum, welcher alle Knoten von  $G$  enthält
- Alternativ:  $T$  ist ein Baum mit  $n - 1$  Kanten aus  $E$

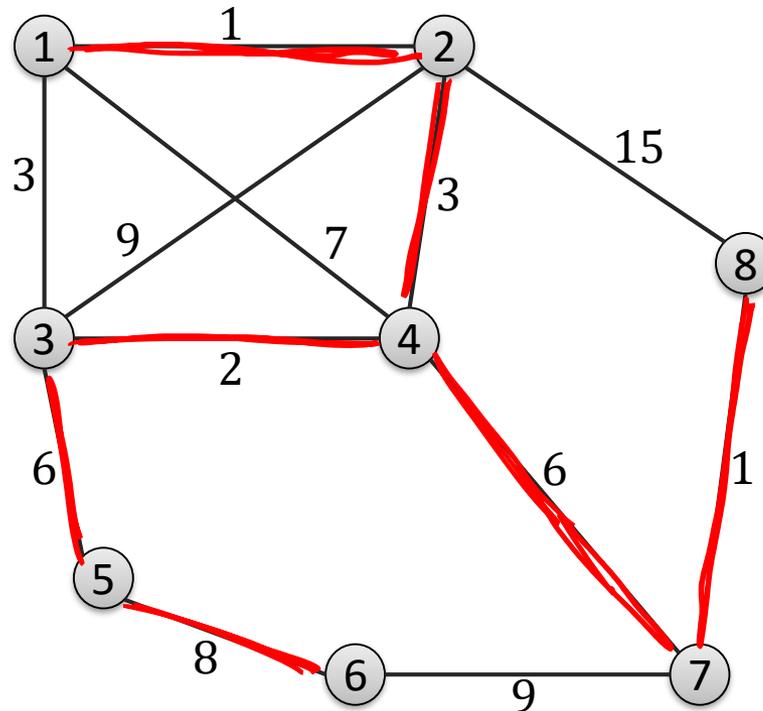


# Minimaler Spannbaum $w(e)$

**Gegeben:** Zus.-hängender, ungerichteter Graph  $G = (V, E, w)$  mit Kantengewichten  $w : E \rightarrow \mathbb{R}$

**Minimaler Spannbaum  $T = (V, E_T)$ :**

- Spannbaum mit kleinstem Gesamtgewicht



minimaler  
Spannbaum

# Minimale Spannbäume

**Ziel:** Gegeben ein gewichteter, ungerichteter Graph  $G$ , finde einen Spannbaum mit minimalem Gesamtgewicht.

- **Minimaler Spannbaum = Minimum Spanning Tree = MST**
- Ein grundlegendes Optimierungsproblem auf Graphen
  - eines von sehr vielen Optimierungsproblemen auf Graphen
- kommt oft als Teilproblem vor
- ist aber auch interessant an sich

**Idee:** Starte mit leerer Kantenmenge und füge die Kanten schrittweise hinzu, bis es ein Spannbaum ist

## **Invariante:**

Algorithmus hat zu jeder Zeit eine Kantenmenge  $A$ , so dass  $A$  Teilmenge eines minimalen Spannbaums ist.

- Am Anfang ist  $A = \emptyset$
- Danach wird jeweils eine Kante hinzugefügt, ohne die Invariante zu verletzen
- Wir nennen eine Kante, für welche wir sicher sein können, dass wir sie zu  $A$  hinzufügen können eine sichere Kante für  $A$
- Wie man sichere Kanten findet, werden wir sehen...

## Invariante:

Algorithmus hat zu jeder Zeit eine Kantenmenge  $A$ , so dass  $A$  Teilmenge eines minimalen Spannbaums ist.

## Basis-MST-Algorithmus:

$A = \emptyset$

**while**  $A$  ist kein Spannbaum **do**

    Finde sichere Kante  $\{u, v\}$  für  $A$

$A = A \cup \{\{u, v\}\}$

**return**  $A$

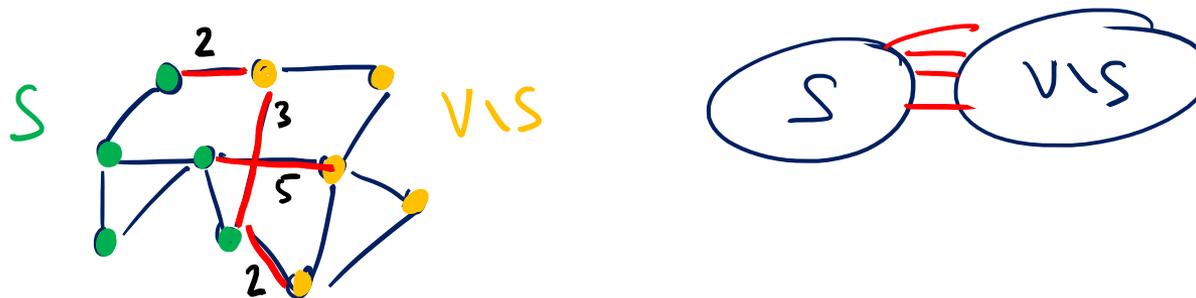
}  $n-1$  Wiederholungen

- Invariante ist eine gültige Schleifeninvariante
- **Invariante + Abbruchbedingung  $\Rightarrow A$  ist ein MST!**

# Wie findet man sichere Kanten?

- Invariante  $\rightarrow$  es gibt immer mindestens eine sicher Kante
  - $A$  ist Teilmenge eines MST und kann daher zu einem MST erweitert werden
- Zuerst benötigen wir ein paar Begriffe...

**Schnitt  $(S, V \setminus S)$ :**  $|S| \geq 1$   $|V \setminus S| \geq 1$

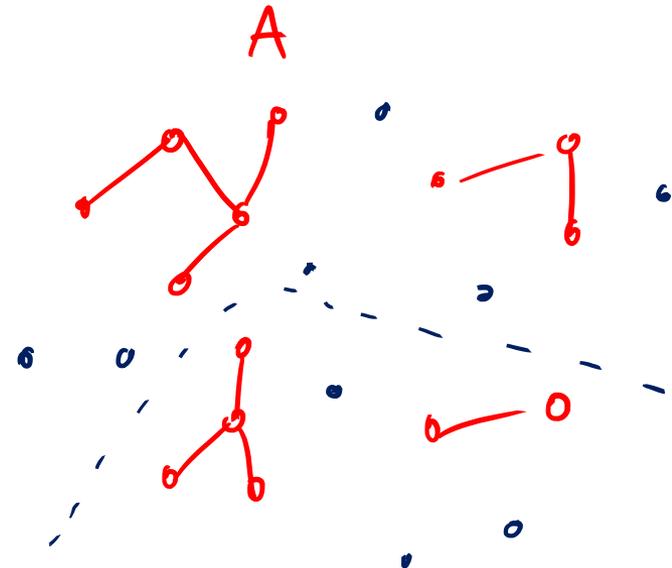
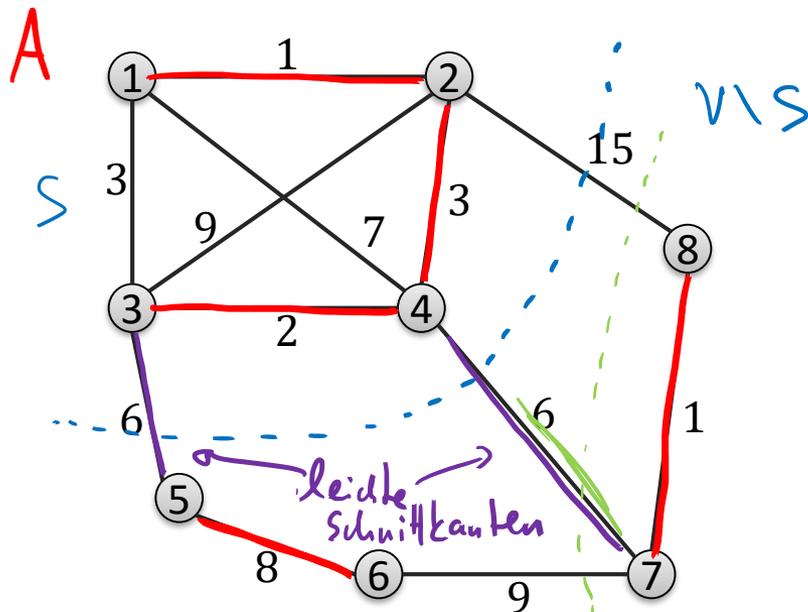


- Kante  $\{u, v\} \in E$  ist eine **Schnittkante** bezüglich  $(S, V \setminus S)$ , falls ein Ende in  $S$  und ein Ende in  $V \setminus S$  ist.
- Wir nennen Kante  $\{u, v\}$  eine **leichte Schnittkante** bez.  $(S, V \setminus S)$ , falls sie das **kleinste Gewicht** von allen Schnittkanten hat

## Annahmen:

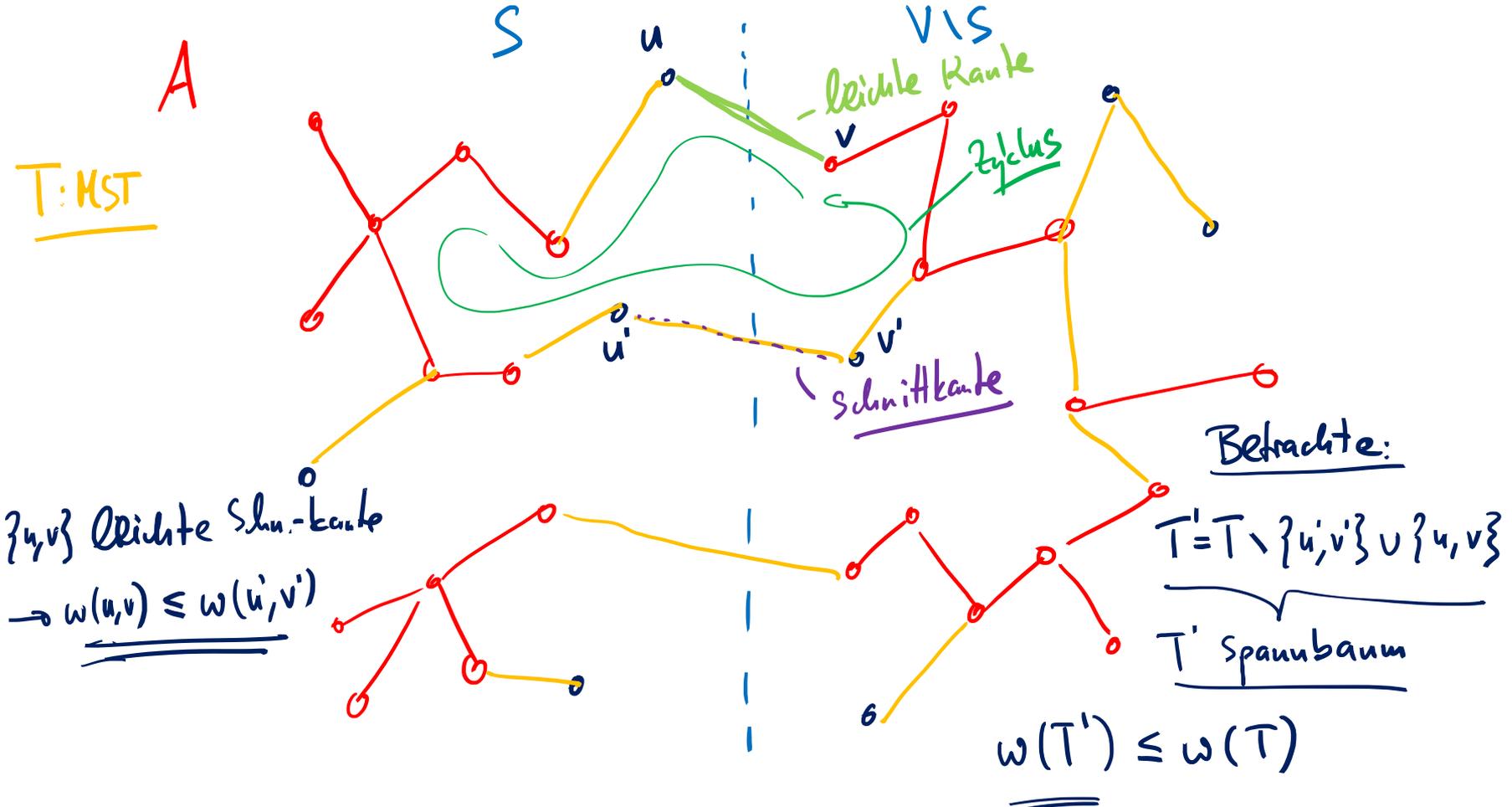
- $G = (V, E, w)$  ist zus.-h., unger. Graph mit Kantengewichten  $w(e)$
- $A$  ist Teilmenge (Teilgraph) eines MST

**Theorem:** Sei  $(S, V \setminus S)$  ein Schnitt, so dass  $A$  keine Schnittkanten enthält und sei  $\{u, v\}$ ,  $u \in S$ ,  $v \in V \setminus S$  eine leichte Schnittkante bezüglich  $(S, V \setminus S)$ . Dann ist  $\{u, v\}$  eine sichere Kante für  $A$ .



# Sichere Kanten

**Theorem:** Sei  $(S, V \setminus S)$  ein Schnitt, so dass  $A$  keine Schnittkanten enthält und sei  $\{u, v\}$ ,  $u \in S$ ,  $v \in V \setminus S$  eine leichte Schnittkante bezüglich  $(S, V \setminus S)$ . Dann ist  $\{u, v\}$  eine sichere Kante für  $A$ .



**Theorem:** Sei  $(S, V \setminus S)$  ein Schnitt, so dass  $A$  keine Schnittkanten enthält und sei  $\{u, v\}$ ,  $u \in S$ ,  $v \in V \setminus S$  eine leichte Schnittkante bezüglich  $(S, V \setminus S)$ . Dann ist  $\{u, v\}$  eine sichere Kante für  $A$ .

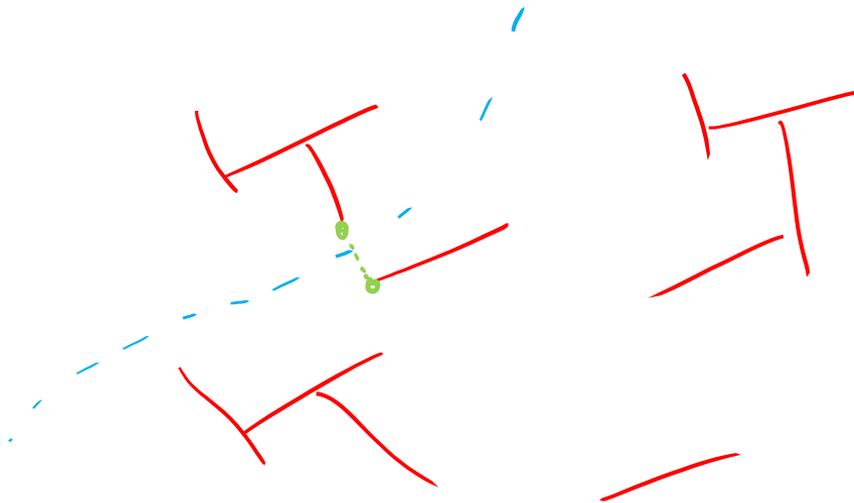
$A = \emptyset$

**while**  $A$  ist kein Spannbaum **do**

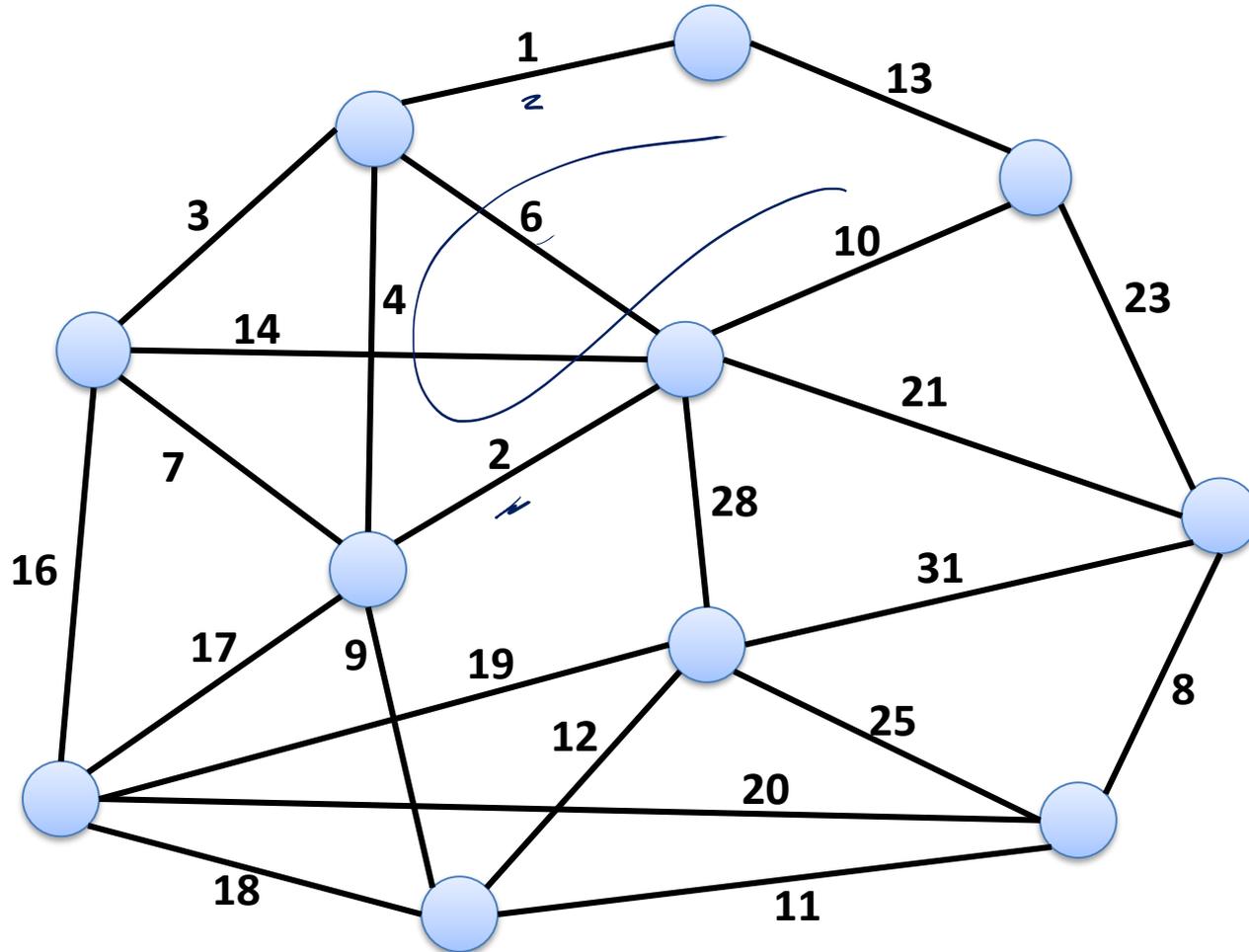
$e = \{u, v\}$  ist Kante mit kleinstem Gewicht,  
so dass  $A \cup \{\{u, v\}\}$  keinen Zyklus enthält

$A = A \cup \{\{u, v\}\}$

- Wir müssen zeigen, dass  $e$  eine sichere Kante für  $A$  ist



# Kruskal's MST-Algorithmus: Beispiel



Kruskal's Algorithmus berechnet minimalen Spannbaum

- Die hinzugefügte Kante ist in jedem Schritt sicher
- Wir haben gesehen, dass der Basis-Algorithmus korrekt ist

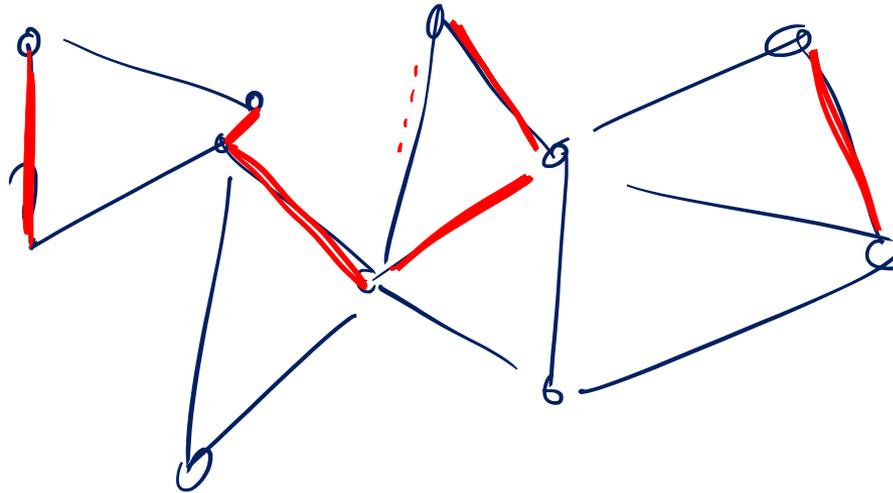
Kruskal's Algorithmus ist ein typisches Beispiel eines sogenannten

## Greedy Algorithmus

- Wir beginnen mit einer leeren Kantenmenge
- In jedem Schritt wird die im Moment beste Kante hinzugenommen
- Eine gewählte Kante wird nie wieder verworfen
  
- Wir werden noch kurz besprechen, wie man den Algorithmus effizient implementieren kann...

## Kruskals Algorithmus (Pseudo-Code)

1.  $A = \emptyset$
2. Sortiere Kanten aufsteigend nach Kantengewicht
3. **for**  $e = \{u, v\} \in E$  (in sorted order) **do**
4.     **if**  $u$  and  $v$  are in different components **then**
5.          $A = A \cup \{e\}$



## Kruskals Algorithmus (Pseudo-Code)

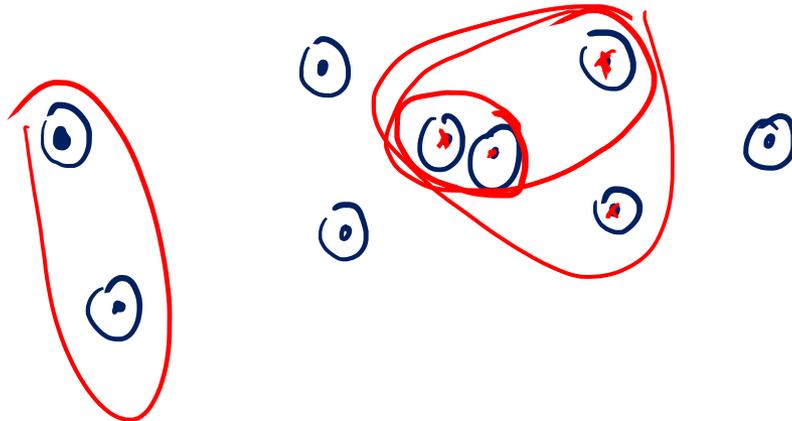
1.  $A = \emptyset$
  2. Sortiere Kanten aufsteigend nach Kantengewicht
  3. **for**  $e = \{u, v\} \in E$  (in sorted order) **do**
  4.     **if**  $u$  and  $v$  are in different components **then**
  5.          $A = A \cup \{e\}$
- Müssen Komponenten des durch  $A$  bestimmten Graphen effizient **verwalten** können
  - **Laufzeit:**  $O(m \log n)$  für's Sortieren, sowie die Gesamtzeit, um die Komponenten zu verwalten...

# Union-Find / Disjoint Sets Datentyp

Verwaltet eine Partition von Elementen

## Operationen:

- *create* : erzeugt eine leere Union-Find-DS
- *U.makeSet(x)* : fügt Menge  $\{x\}$  zur Partition hinzu
- *U.find(x)* : gibt Menge mit Element  $x$  zurück
- *U.union(S1, S2)* : vereinigt die Mengen  $S1$  und  $S2$



- Details dazu werden in der Algorithmentheorie besprochen...

## Kruskals Algorithmus

1.  $A = \emptyset$
2.  $U =$  create new *Union-Find D.S.*
3. **for all**  $u \in V$  **do**
4.      $U.\text{makeSet}(u)$
5.     Sortiere Kanten aufsteigend nach Kantengewicht
6. **for all**  $e = \{u, v\} \in E$  (in sorted order) **do**
7.      $S_u = U.\text{find}(u)$ ;  $S_v = U.\text{find}(v)$
8.     **if**  $S_u \neq S_v$  **then**
9.          $A = A \cup \{e\}$      *u makeset*
10.          $U.\text{union}(S_u, S_v)$      *u-1 union*     *2m find*      *$O(m)$*

## Beste Union-Find Datenstruktur

- Laufzeit für  $m$  Union-Find-Operationen auf  $n$  Elementen ( $n$  makeSet-Operationen):

$$O(m \cdot \alpha(m, n))$$

$10^{80}$

- $\alpha(m, n)$  ist die Inverse der Ackermannfunktion und wächst extrem langsam (für alle halbwegs vernünftigen  $m, n$ ,  $\alpha(m, n) \leq 5$ )

## Laufzeit Kruskal

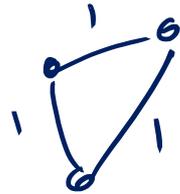
$$\alpha(m, n) = o(\log n)$$

- Kanten sortieren:  $O(m \cdot \log n)$
- Union-Find-Operationen:  $O(m \cdot \alpha(m, n))$
- Insgesamt:  $O(m \cdot \log n)$ 
  - besser, falls Kantengewichte schneller sortiert werden können

Gewichte  
Int.  $\in \{1, \dots, n^{10}\}$

# MST Eindeutigkeit

- Im Allgemeinen ist der MST nicht eindeutig



**Satz:** Bei paarweise verschiedenen Kantengewichten ist der MST eindeutig.

Kruskal eindeutig