Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 14 (11.6.2018)

Graphenalgorithmen III

UNI FREIBURG

Fabian Kuhn

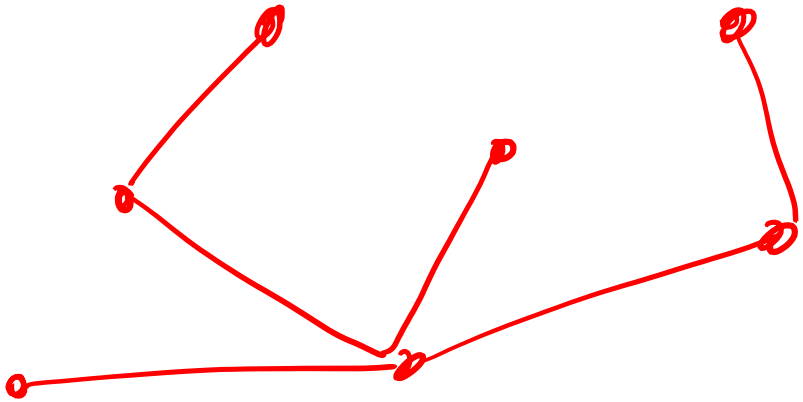
Algorithmen und Komplexität

Bäume

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

Baum:

- Zusammenhängender, ungerichteter Graph, ohne Zyklen



Äquivalente Definitionen:

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

$n := |V|$

Bäume

Baum:

Zusammenhängender, ungerichteter Graph, ohne Zyklen

Anzahl Kanten in einem Baum: $n - 1$

IA.: $n=1$: • ✓ $n=2$: •—• ✓

IV.: Beh. gelte für alle Bäume mit $\leq n$ Knoten

IS.: Sei T ein Baum mit $n+1$ Knoten. Sei e Kante von T.
Entferne e $\Rightarrow$ T zerfällt in zwei Teilbäume T_1, T_2
mit $n_1, n_2 \leq n$ Knoten $\Rightarrow$ T_1, T_2 haben n_1-1 bzw. n_2-1
Kanten. Insgesamt $(n_1-1)+(n_2-1)+1 = (n+1)-1 = n$ ✓

Ein nicht zus.-hängender zyklenfreier (unger.) Graph heisst Wald

Anzahl Kanten eines Waldes: $n - k$

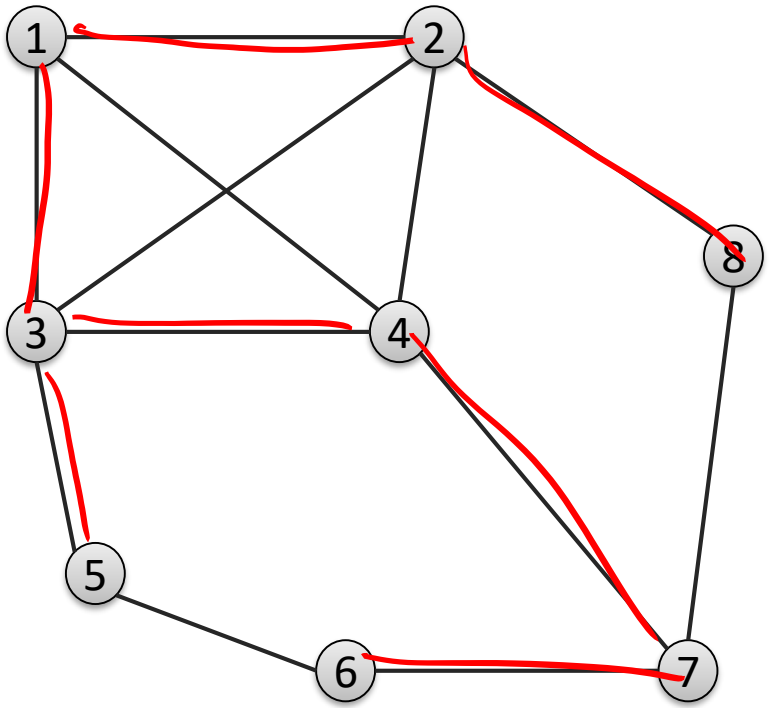
- k = Anzahl Zusammenhangskomponenten

Seien $T_1, \dots, T_k$ die Zsh.-komp. von Wald T, mit je
$n_1, \dots, n_k$ Knoten. Dann hat der Wald
$(n_1-1) + \dots + (n_k-1) = n-k$.

Spannbaum

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

Spannbaum $T = (V, E_T)$ von G : Teilgraph ($E_T \subseteq E$) der Baum ist

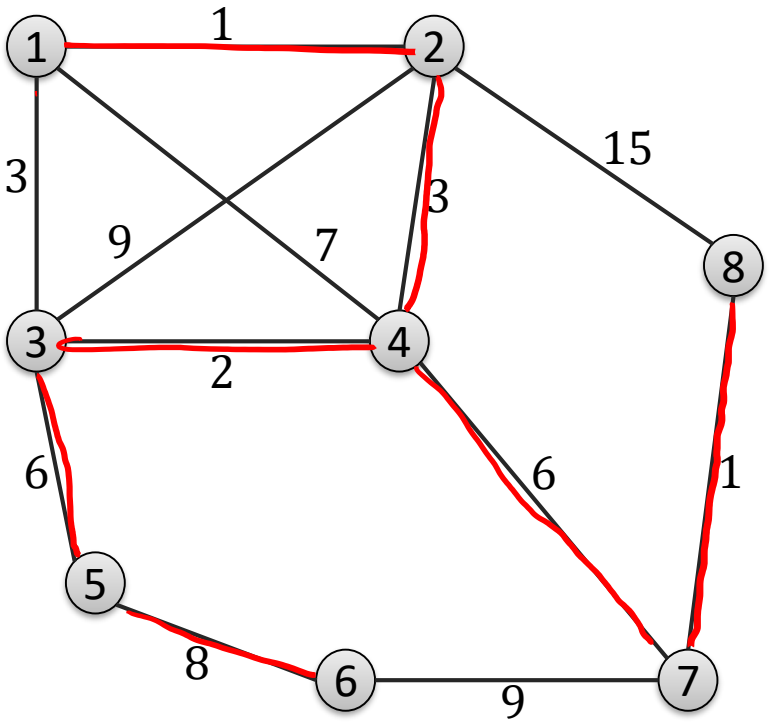


Minimaler Spannbaum

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

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

- Spannbaum mit kleinstem Gesamtgewicht

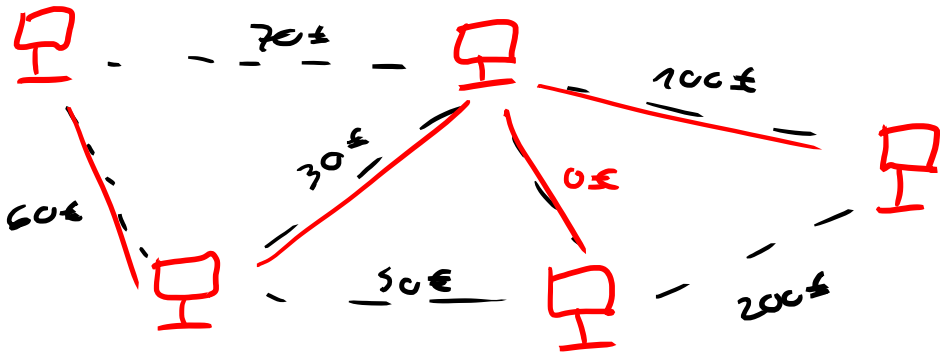


Minimale Spannbäume

Gegeben: gewichteter, ungerichteter Graph $G = (V, E, w)$

Ziel: Finde einen Spannbaum mit minimalem Gesamtgewicht.

- Abkürzung: MST = Minimum Spanning Tree
- Ein grundlegendes Optimierungsproblem auf Graphen
  - eines von sehr vielen Optimierungsproblemen auf Graphen
- kommt oft als Teilproblem anderer Problemstellungen vor
- mögliches Echtweltbeispiel:



Basis-MST-Algorithmus

UNI FREIBURG

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 $A = \emptyset$
- Füge jeweils eine Kante hinzu, ohne die Invariante zu verletzen
- Wir nennen eine Kante $e \in E$, für welche wir sicher sein können, dass wir sie zu A hinzufügen können eine **sichere Kante für A**
- Es gibt also einen MST der die sichere Kante e enthält
- Wie man sichere Kanten findet, werden wir sehen…

Basis-MST-Algorithmus

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

$n-1$

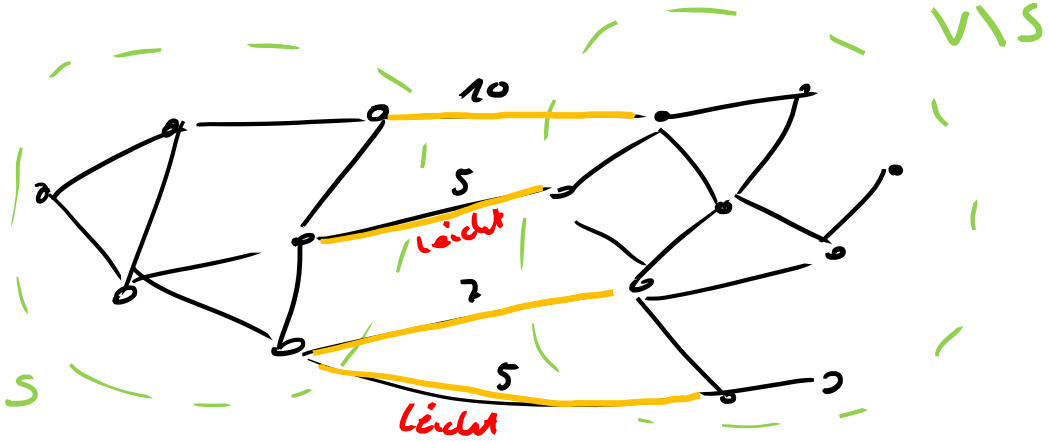
- Invariante ist eine gültige Schleifeninvariante
- Invariante + Abbruchbedingung $\Longrightarrow$ A ist ein MST!

Wie findet man sichere Kanten?

- Invariante → 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 $(S, V \setminus S)$ für $S \subseteq V$:



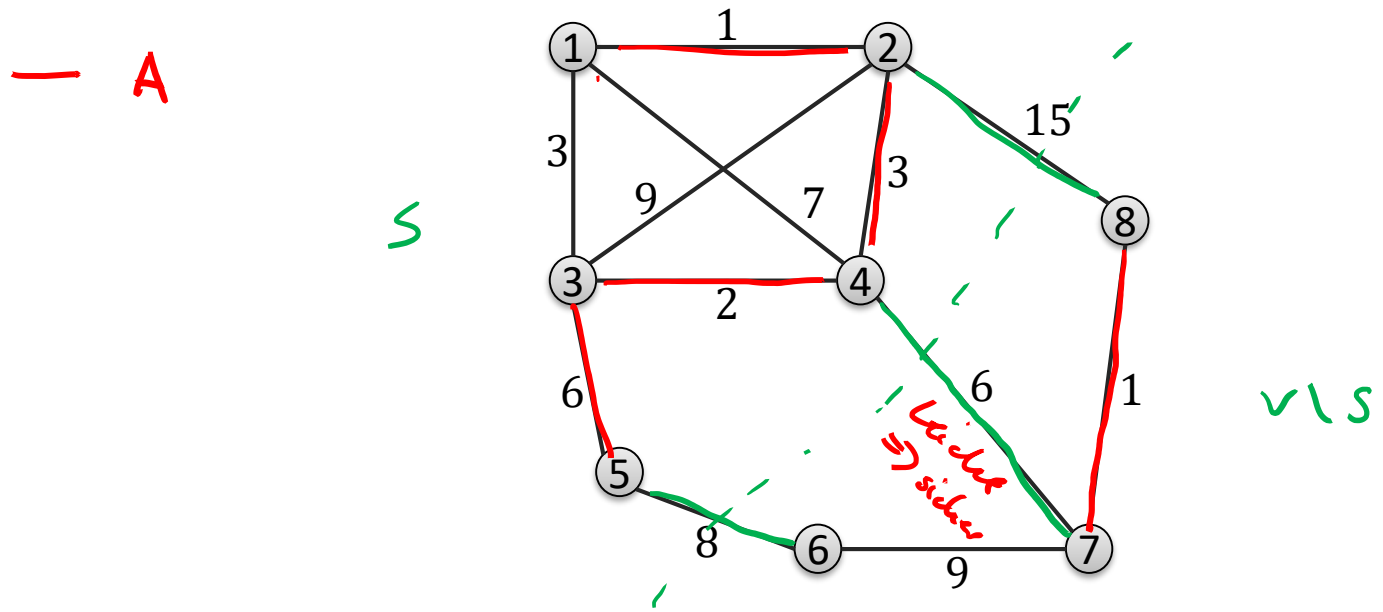
- Kante $\{u, v\} \in E$ ist eine Schnittkante bezüglich $(S, V \setminus S)$, falls ein Ende in S und ein Ende in $V \setminus S$ ist.
- Kante $\{u, v\}$ ist eine **leichte Schnittkante** bez. $(S, V \setminus S)$, falls sie das kleinste Gewicht von allen Schnittkanten in $(S, V \setminus S)$ hat

Sichere Kanten

Annahmen:

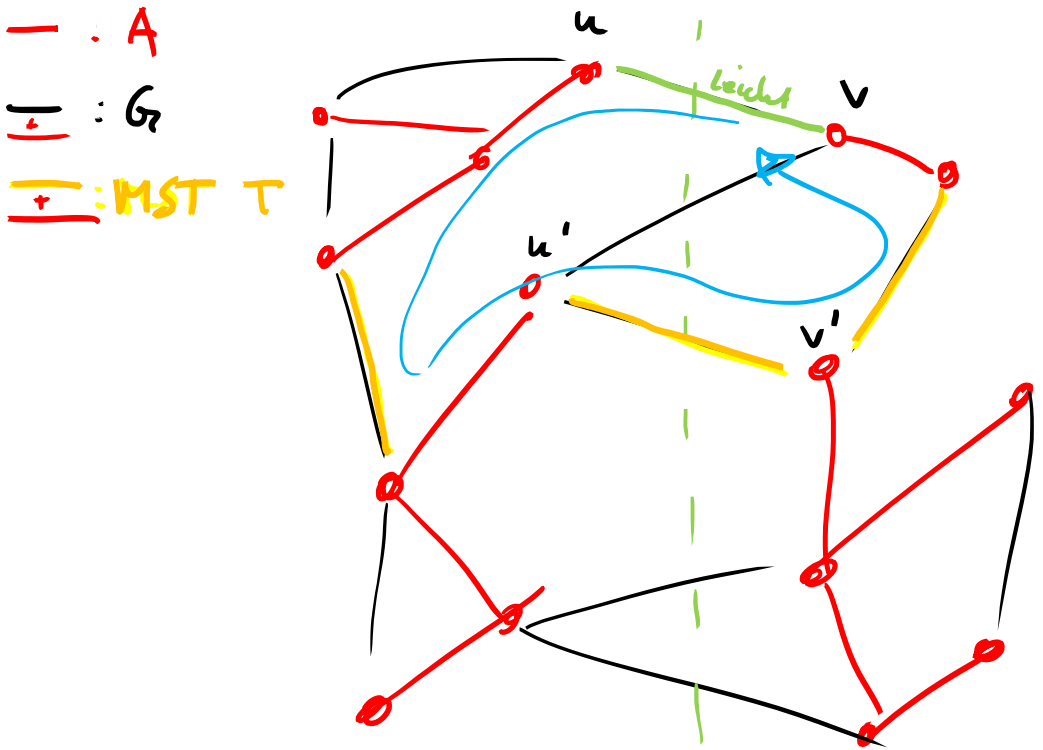
- $G = (V, E, w)$ ist zsh., unger. Graph mit Kantengewichten $w(e)$
- $A \subseteq E$ ist Teilmenge (Teilgraph) eines MST

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.



$\{u,v\}$ leicht $\Rightarrow w(u,v) \leq w(u',v')$

Betrachte $T' := (T \cup \{u,v\}) \setminus \{u',v'\}$

T' ist Spannbaum

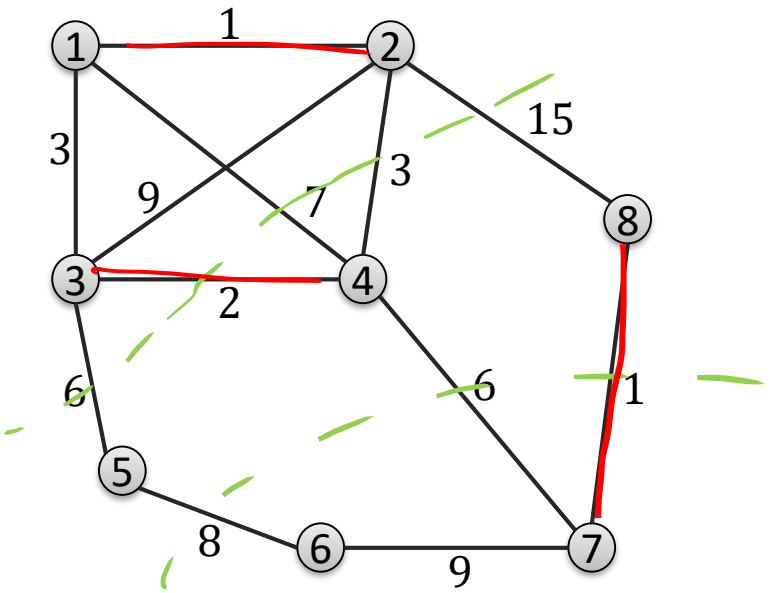
$w(T') \leq w(T)$

$\Rightarrow \{u,v\}$ ist sicher

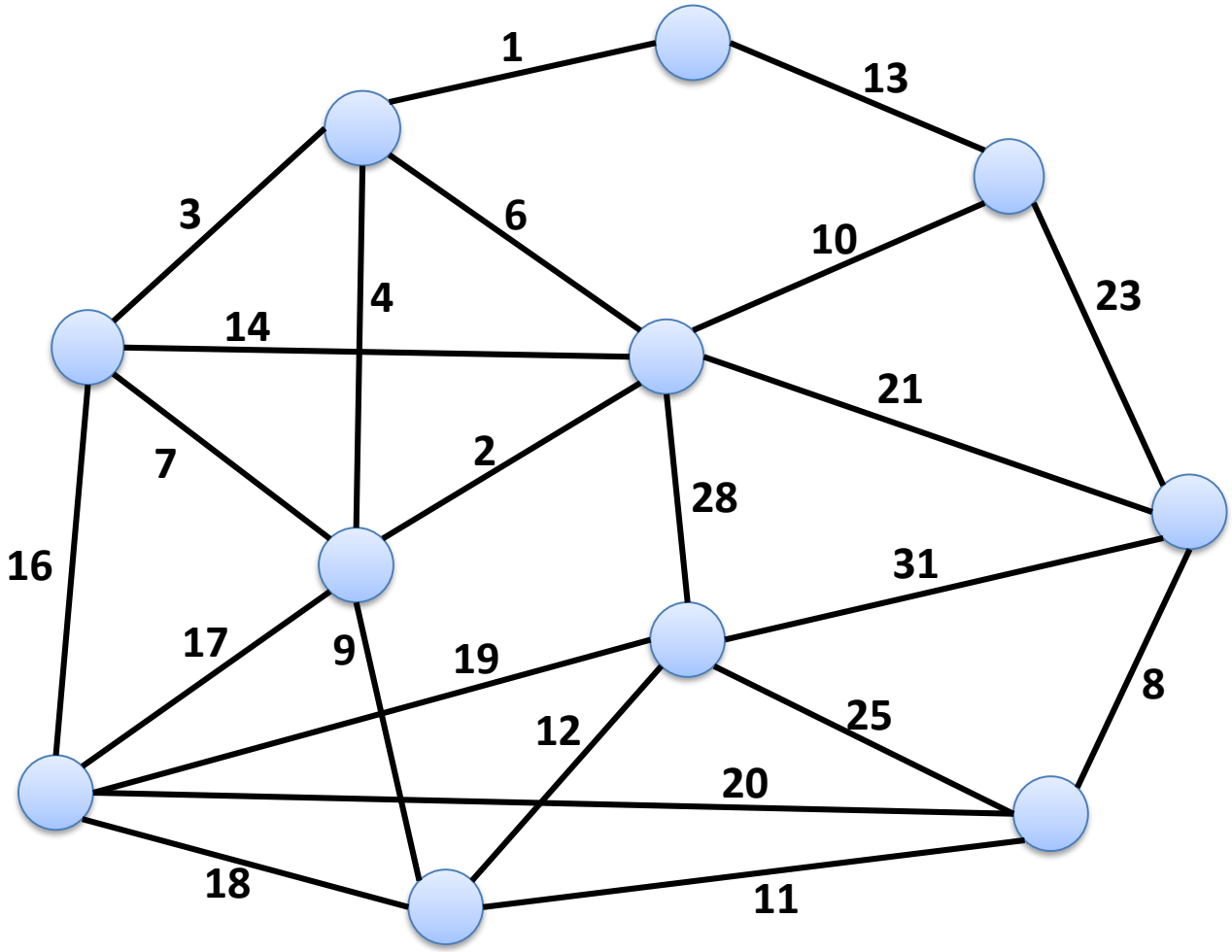
Kruskal's MST-Algorithmus

$A = \emptyset$
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 sicherstellen, dass e eine sichere Kante für A ist



Kruskal's MST-Algorithmus: Beispiel



Kruskal's MST-Algorithmus: Bemerkungen

Kruskal's Algorithmus ist ein typisches Beispiel eines sogenannten

Greedy Algorithmus

- Wir beginnen mit einer leeren Lösungsmenge (Kantenmenge)
- In jedem Schritt wird die Teillösung erweitert sodas sie momentan am besten ist (leichteste mögliche Kante wird hinzugenommen)
- Bisher gewählte Teillösung wird nicht mehr geändert (Kante wird nie wieder verworfen)

Wir werden noch kurz besprechen, wie man den Algorithmus effizient implementieren kann...

Implementierung Kruskals Algorithmus

Kruskals Algorithmus (Pseudo-Code)

1. $A = \emptyset$
2. Sort edges by weight (ascending) $O(m \log m)$
3. **for** $e = \{u, v\} \in E$ (in sorted order) **do**
4. **if** u and v are in different components **then**
5. $A = A \cup \{e\}$

with respect to n

$m := |E|$
$n := |V|$

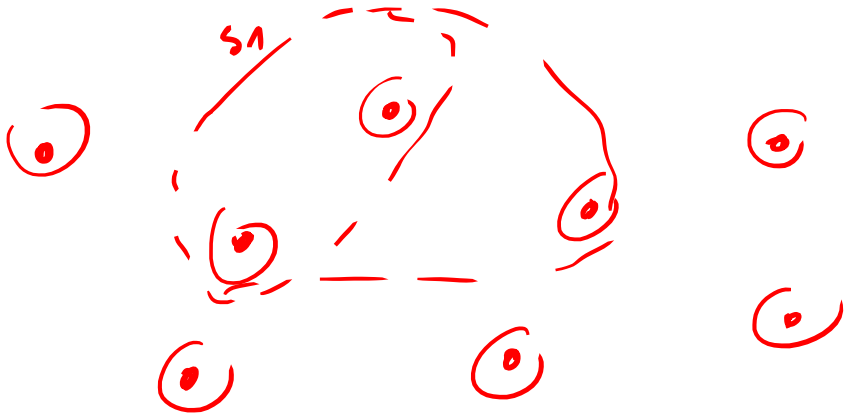
- Müssen Komponenten des durch A bestimmten Graphen effizient verwalten können n^2
- **Laufzeit:** $O(m \log m) = O(m \log n)$ für's Sortieren,
- zuzüglich zur Laufzeit für die Verwaltung der Komponenten
- Letzteres geht im Allgemeinen schneller als das Sortieren…

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…

Kruskal mit Union-Find

UNI FREIBURG

Kruskals Algorithmus

1. $A = \emptyset$
2. U = create new **Union Find** data structure
3. **for all** $u \in V$ **do**
4. U.makeSet(u)
5. Sort edges by weight (ascending)
6. **for all** $e = \{u, v\} \in E$ (in sorted order) **do**
7. S_u = U.find(u); S_v = U.find(v)
8. **if** $S_u \neq S_v$ **then**
9. $A = A \cup \{e\}$
10. U.union(S_u, S_v)

Kruskal: Laufzeitanalyse

UNI FREIBURG

Beste Union-Find Datenstruktur

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

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

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

$g = 10^{100}$

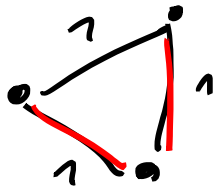
$\alpha(g,g) \leq 5$

Laufzeit Kruskal

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

MST Eindeutigkeit

- Im Allgemeinen ist der MST nicht eindeutig



Satz: Bei paarweise verschiedenen Kantengewichten ist der MST eindeutig.

Prims MST Algorithmus

- Sollte man wohl eigentlich Jarvíks Algorithmus nennen
  - wurde 1957 von Prim und bereits 1930 von Jarvík publiziert

- Eine zweite Implementierung des Basis-Algorithmus

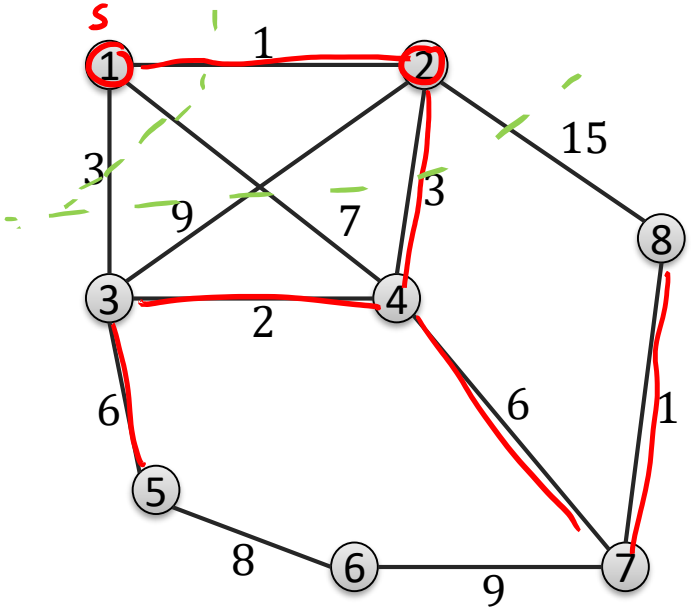
$A = \emptyset$
while A ist kein Spannbaum **do**
    Finde sichere Kante $\{u, v\}$ für A
    $A = A \cup \{\{u, v\}\}$
return A

- **Idee:** A ist immer ein zusammenhängender Teilbaum
  - Starte bei einem beliebigen Knoten $s \in V$
  - Baum wächst von s aus, indem immer eine leichte Schnittkante des durch A induzierten Schnitts hinzugefügt wird.

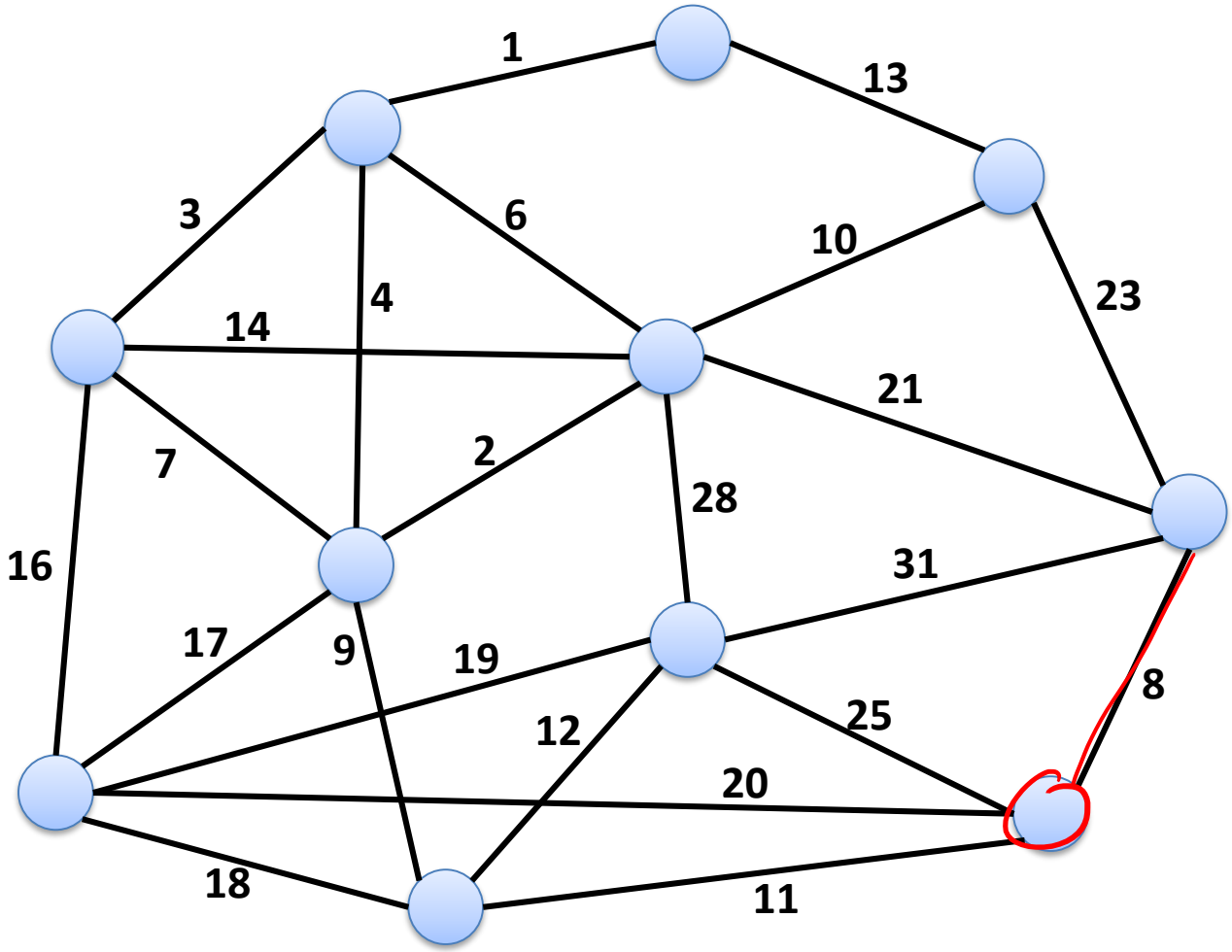
Prims MST-Algorithmus

$A = \emptyset$
while A ist kein Spannbaum **do**
    $e = \{u, v\}$ ist Kante mit kleinstem Gewicht, so dass $u \in A$ und $v \notin A$
    $A = A \cup \{\{u, v\}\}$

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



Prims MST-Algorithmus: Beispiel



Konkretisierung von Prims Algorithmus

- Knoten, welche im Teilbaum A sind, heissen markiert
- Für Knoten $u \notin A$:
  - $\alpha(u)$ ist der näheste Nachbar von u im durch A bestimmten Teilbaum
  - $d(u) = \text{dist}(u, \alpha(u))$ (oder ∞ falls $\alpha(u) = \text{NULL}$)

for all $u \in V \setminus \{s\}$ **do**
    u.marked = false; $d(u) = \infty$; $\alpha(u) = \text{NULL}$
s.marked = false; $d(s) = 0$; $A = \emptyset$ // Wir starten bei Knoten s
while there are unmarked nodes **do**
    u = unmarked node with minimal $d(u)$
    for all unmarked neighbors v of u **do**
        if $w(\{u,v\}) < d(v)$ then
            $\alpha(v) = u$; $d(v) = w(\{u,v\})$
    u.marked = true
    if $u \neq s$ **then** $A = A \cup \{\{u, \alpha(u)\}\}$

Initialisierung

Prioritätswarteschlange

UNI FREIBURG

Priority Queue:

- Verwaltet eine Menge von *(key,value)*-Paaren

Operationen:

- *create*: erzeugt einen leeren Heap
- *H.insert(x, key)*: fügt Element *x* mit Schlüssel *key* ein
- *H.getMin()*: gibt Element mit kleinstem Schlüssel zurück
- *H.deleteMin()*: löscht Element mit kleinstem Schlüssel und gibt es zurück
- *H.decreaseKey(x, newkey)* :

  Falls *newkey* kleiner als der aktuelle Schlüssel des Elementes *x (gegeben durch einen pointer)* ist, wird der Schlüssel von *x* auf *newkey* gesetzt

Implementierung von Prims Algorithmus

UNI FREIBURG

H = create(); $A = \emptyset$

for all $u \in V \setminus \{s\}$ **do**

    H.insert(u, ∞); $\alpha(u)$ = NULL

H.insert(s, 0)

while H is not empty **do**

    u = H.deleteMin()

    for all unmarked neighbors v of u **do**

        if $w(\{u,v\}) < d(v)$ **then**

            H.decreaseKey(v, $w(\{u,v\})$)

            $\alpha(v) = u$

    if $u \neq s$ **then** $A = A \cup \{\{u, \alpha(u)\}\}$

Analyse von Prims MST-Algorithmus

UNI FREIBURG

Anzahl Priority Queue Operationen

- **create** 1
- **insert** n
- **getMin / deleteMin** n
- **decreaseKey** m

Gesamt-Laufzeit

$$O\left(n \cdot \left(T(\text{insert}) + T(\text{deleteMin})\right) + m\, T(\text{decreaseKey})\right)$$