

Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 14 (11.6.2018)

Graphenalgorithmen III



**UNI
FREIBURG**

Fabian Kuhn

Algorithmen und Komplexität

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

Baum:

- Zusammenhängender, ungerichteter Graph, ohne Zyklen

Äquivalente Definitionen:

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

Baum:

Zusammenhängender, ungerichteter Graph, **ohne Zyklen**

Anzahl Kanten in einem Baum: $n - 1$

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

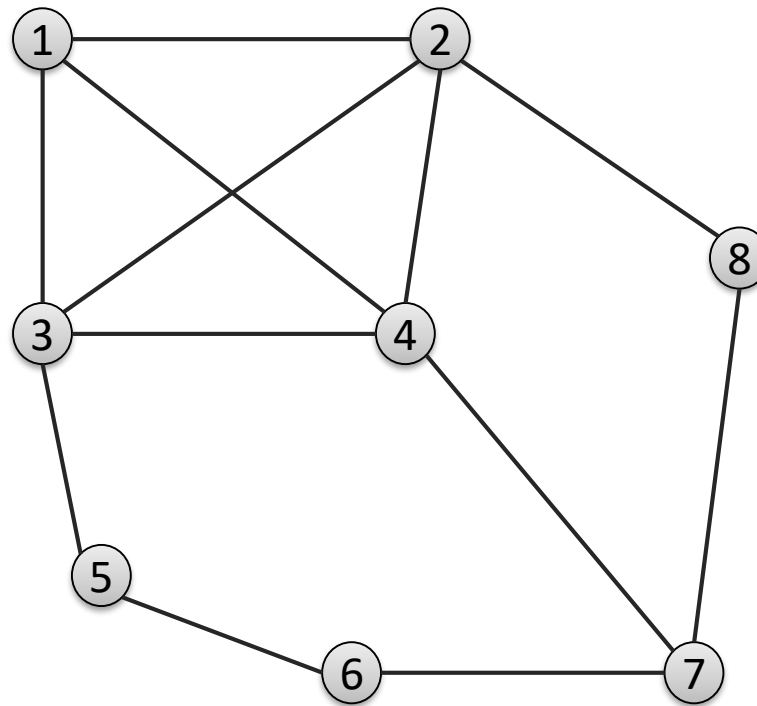
Anzahl Kanten eines Waldes: $n - k$

- $k =$ Anzahl Zusammenhangskomponenten

Spannbaum

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

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

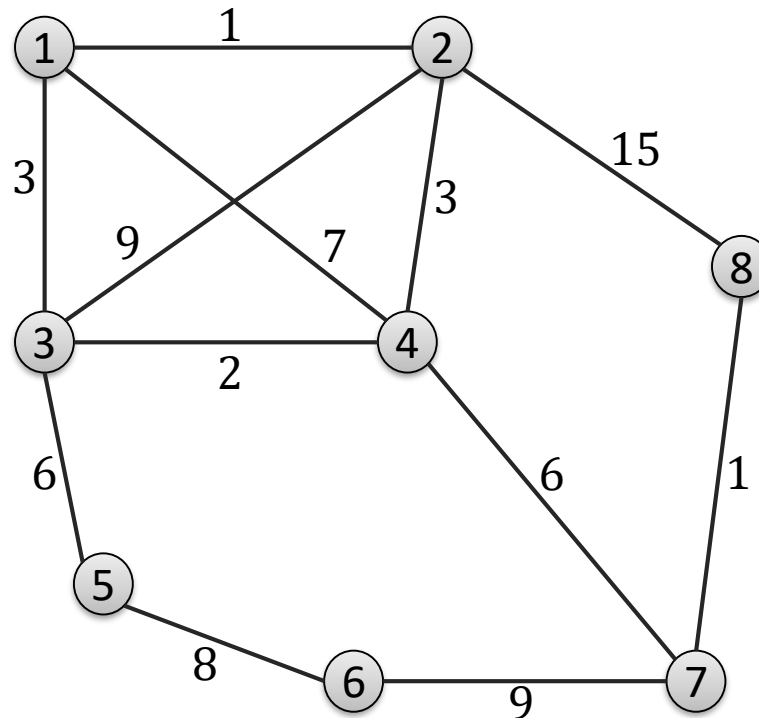


Minimaler Spannbaum

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

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

- Spannbaum mit kleinstem Gesamtgewicht



Minimale Spann­b­ume

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:

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...

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

- 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...

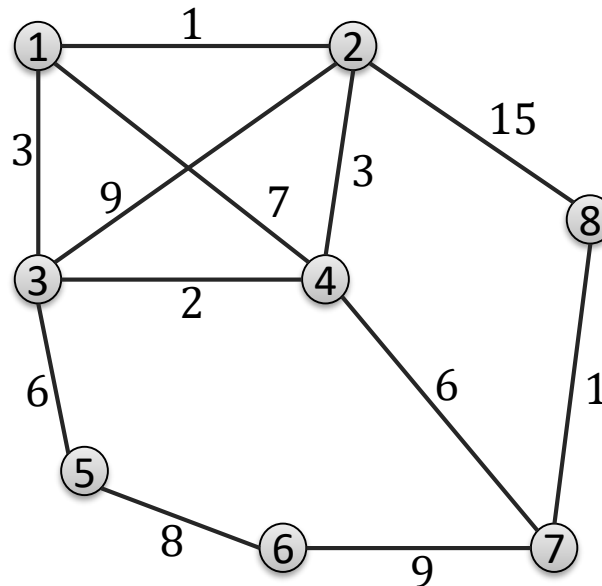
Schnitt $(S, V \setminus S)$ für $S \subseteq V$:

- 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

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 .



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 .

Kruskal's MST-Algorithmus

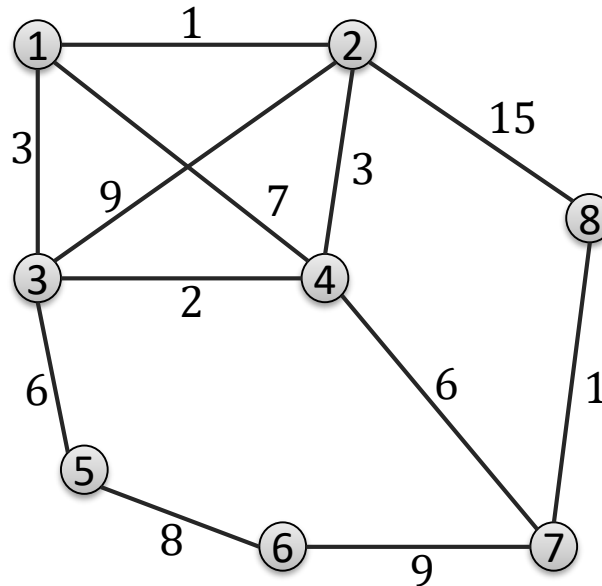
$A = \emptyset$

while A ist kein Spannbaum **do**

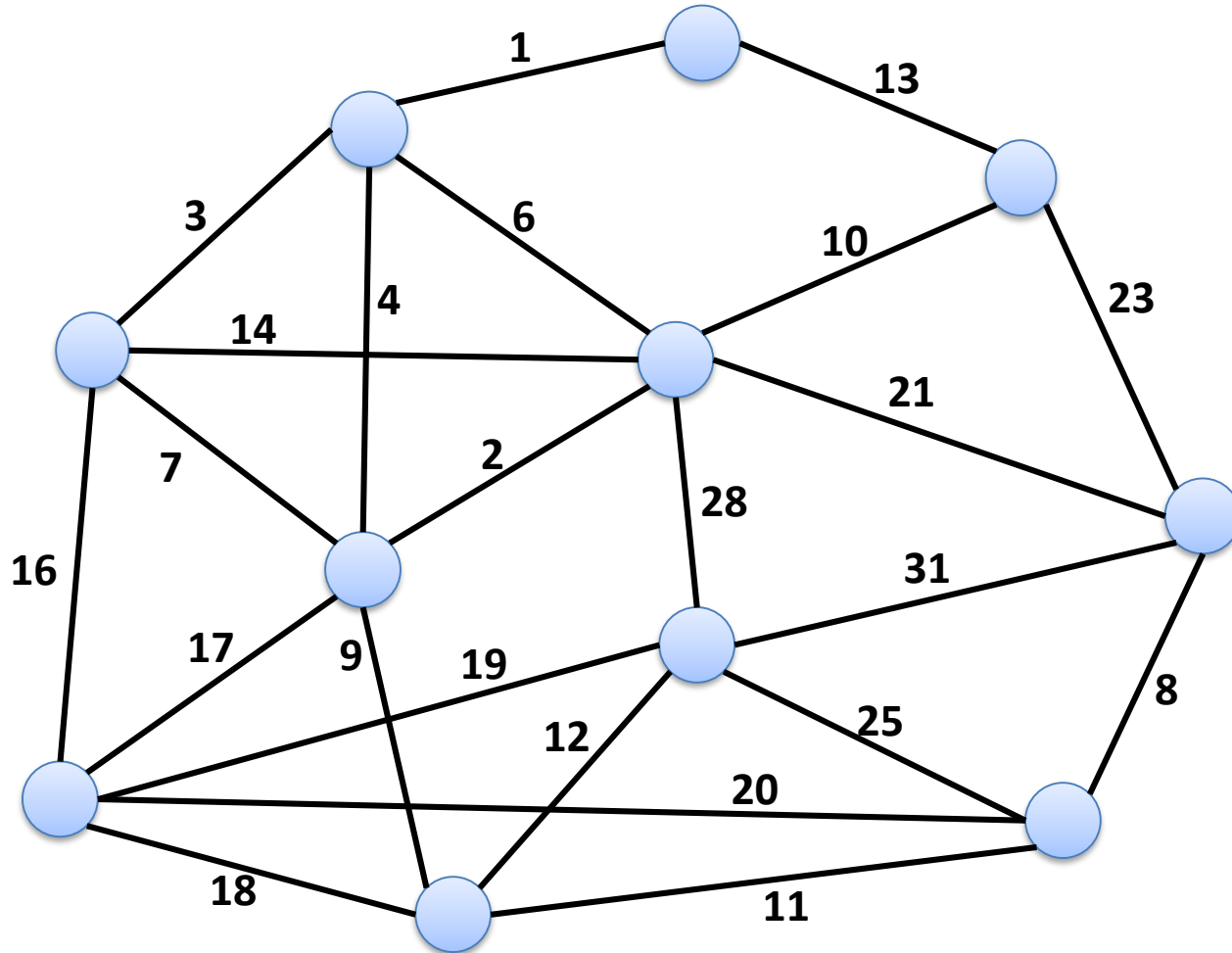
$e = \{u, v\}$ ist Kante mit kleinstem Gewicht,
so dass $A \cup \{\{u, v\}\}$ keinen Zyklus enthält

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

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



Kruskal's MST-Algorithmus: Beispiel



Kruskal's Algorithmus ist ein typisches Beispiel eines sogenannten

Greedy Algorithmus

- Wir beginnen mit einer leeren Lösungsmenge (Kantenmenge)
- In jedem Schritt wird die Teillösung erweitert sodass 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...

Kruskals Algorithmus (Pseudo-Code)

1. $A = \emptyset$
2. Sort edges by weight (ascending)
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:** $O(m \log m) = O(m \log n)$ für's Sortieren,
- zuzüglich zur Laufzeit für die Verwaltung der Komponenten
- Letzteres geht im Allgemeinen schneller als das Sortieren...

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** 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. $U.union(S_u, S_v)$

Beste Union-Find Datenstruktur

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

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

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

Laufzeit Kruskal

- 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

- Im Allgemeinen ist der **MST nicht eindeutig**

Satz: Bei paarweise verschiedenen Kantengewichten ist der MST eindeutig.

- 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

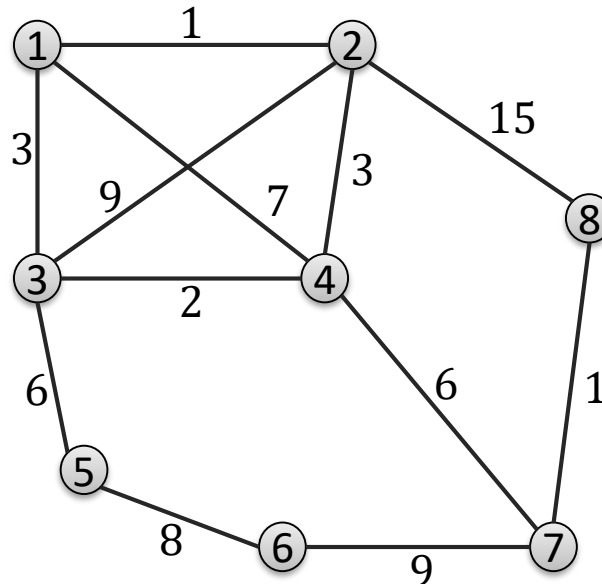
$A = \emptyset$

while A ist kein Spannbaum **do**

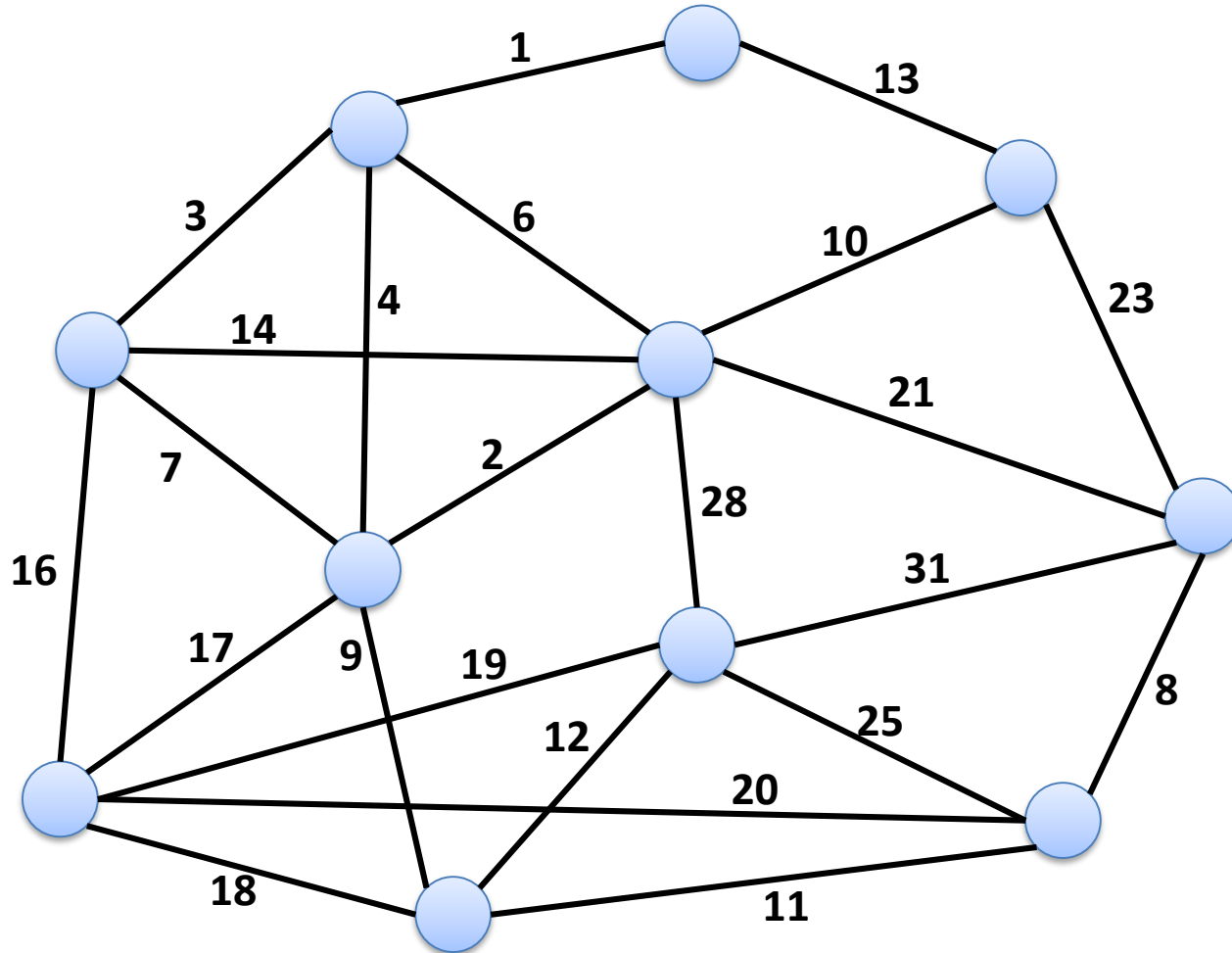
$e = \{u, v\}$ ist Kante mit kleinstem Gewicht,
so dass $u \in A$ und $v \notin A$

$A = A \cup \{e\}$

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



Prims MST-Algorithmus: Beispiel



Konkretisierung von Prim's Algorithmus

- Knoten, welche im Teilbaum A sind, heissen markiert
- Für Knoten $u \notin A$:
 - $\alpha(u)$ ist der nächste 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**

$u.\text{marked} = \text{false}; d(u) = \infty; \alpha(u) = \text{NULL}$

$s.\text{marked} = \text{false}; d(s) = 0; A = \emptyset$ // 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**

$\alpha(v) = u; d(v) = w(\{u, v\})$

$u.\text{marked} = \text{true}$

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

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 Prim's Algorithmus

```
H = create(); A = ∅  
for all u ∈ V \ {s} do  
    H.insert(u, ∞);  $\alpha(u) = \text{NULL}$   
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 ≠ s then A = A ∪ {u,  $\alpha(u)$ }
```

Anzahl Priority Queue Operationen

- **create**
- **insert**
- **getMin / deleteMin**
- **decreaseKey**

Gesamt-Laufzeit