Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Lehrstuhl für
INFORMATIK I
Algorithmen & Komplexität

Institut für Informatik

Algorithmen und Datenstrukturen

Wintersemester 2020/21

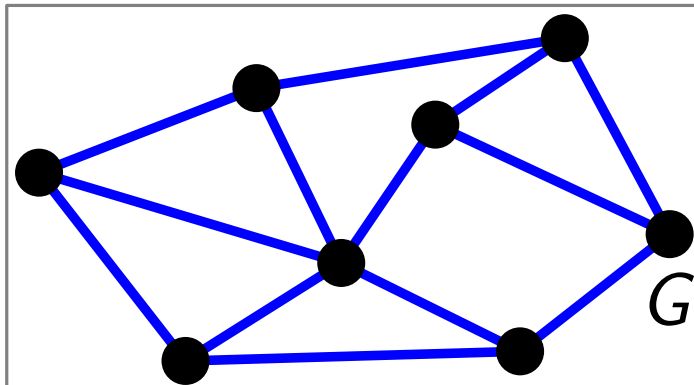
21. Vorlesung

Minimale Spannbäume

Prof. Dr. Alexander Wolff *Lehrstuhl für Informatik I*

Motivation

Gegeben: zusammenhängendes Straßennetz $G = (V, E)$
mit Kantengewichten $w\colon E \to \mathbb{R}_{>0}$,
das eine Menge V von n Städten verbindet.



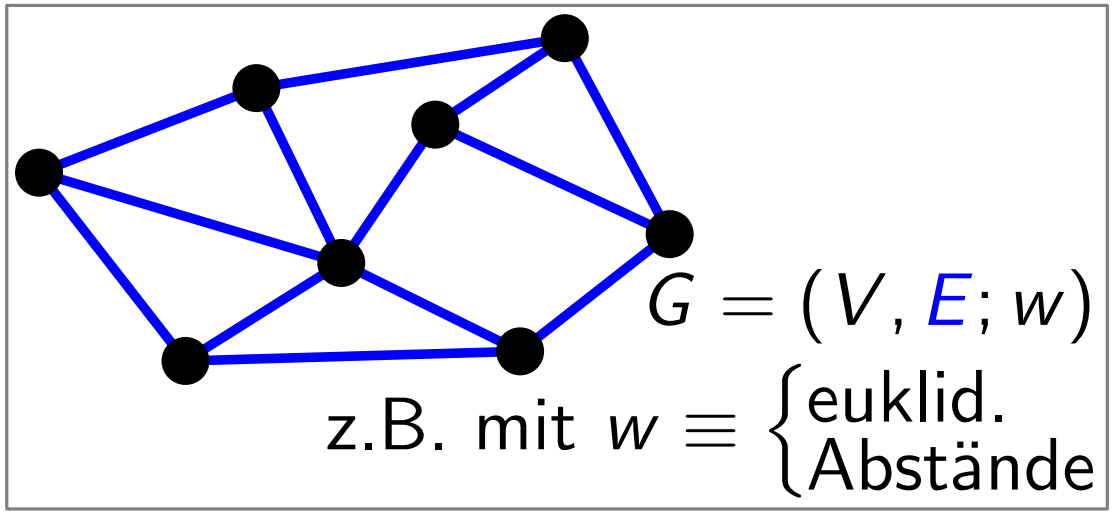
$G = (V, E; w)$

z.B. mit $w \equiv \begin{cases} \text{euklid.} \\ \text{Abstände} \end{cases}$

Motivation

Gegeben: zusammenhängendes Straßennetz $G = (V, E)$
mit Kantengewichten $w\colon E \to \mathbb{R}_{>0}$,
das eine Menge V von n Städten verbindet.

Gesucht: Teilnetz $G' = (V, E')$ mit $E' \subseteq E$, so dass

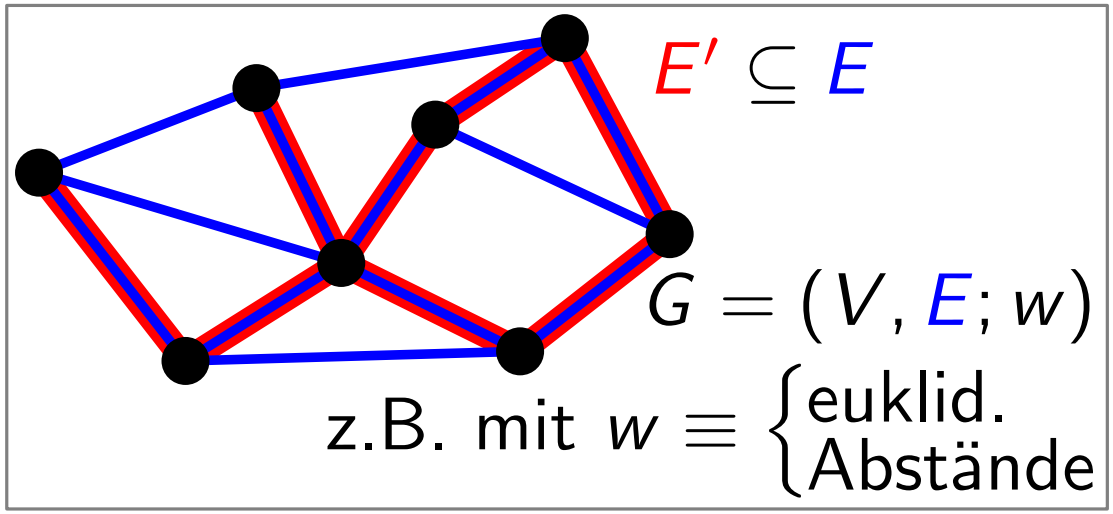


Motivation

Gegeben: zusammenhängendes Straßennetz $G = (V, E)$
mit Kantengewichten $w\colon E \to \mathbb{R}_{>0}$,
das eine Menge V von n Städten verbindet.

Gesucht: Teilnetz $G' = (V, E')$ mit $E' \subseteq E$, so dass

(1) man von jeder Stadt in G' zu jeder anderen kommen kann

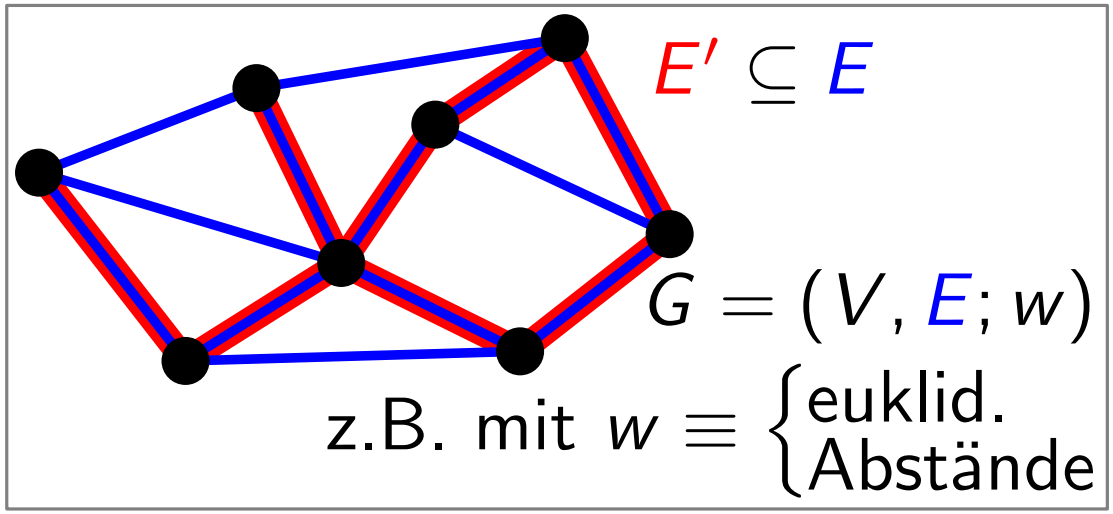


Motivation

Gegeben: zusammenhängendes Straßennetz $G = (V, E)$
mit Kantengewichten $w\colon E \to \mathbb{R}_{>0}$,
das eine Menge V von n Städten verbindet.

Gesucht: Teilnetz $G' = (V, E')$ mit $E' \subseteq E$, so dass

(1) man von jeder Stadt in G' zu jeder anderen kommen kann („G' spannt G auf“) und



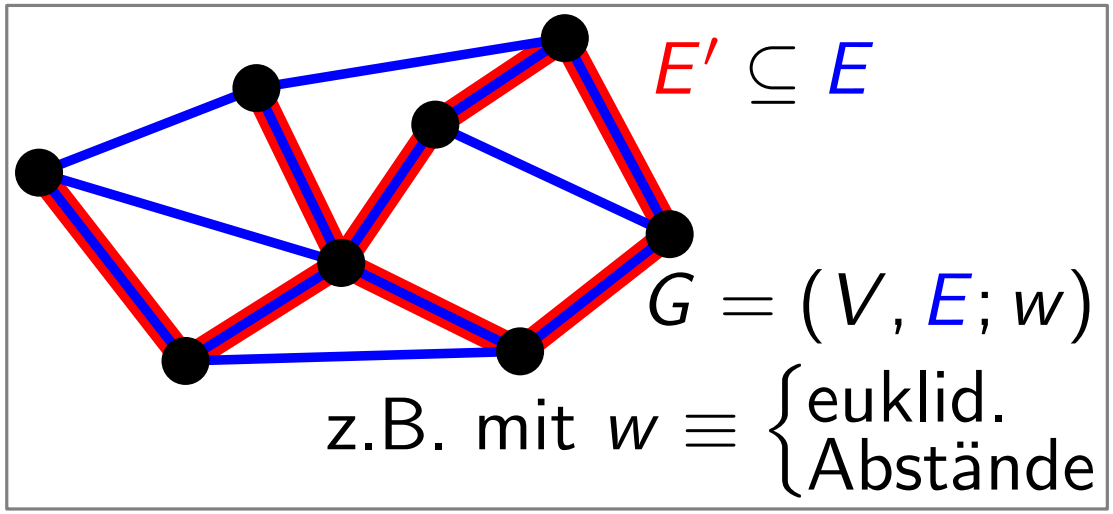
Motivation

Gegeben: zusammenhängendes Straßennetz $G = (V, E)$
mit Kantengewichten $w\colon E \to \mathbb{R}_{>0}$,
das eine Menge V von n Städten verbindet.

Gesucht: Teilnetz $G' = (V, E')$ mit $E' \subseteq E$, so dass

(1) man von jeder Stadt in G' zu jeder anderen kommen kann („G' spannt G auf“) und

(2) die „Schneeräumkosten“ $w(E') = \sum_{e \in E'} w(e)$ minimal sind unter allen Teilnetzen, die (1) erfüllen.



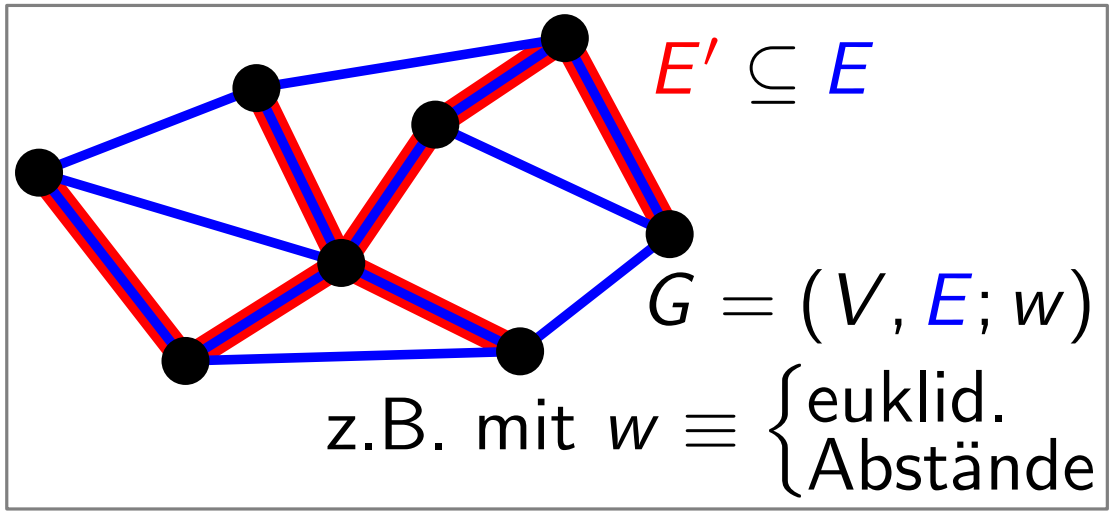
Motivation

Gegeben: zusammenhängendes Straßennetz $G = (V, E)$
mit Kantengewichten $w\colon E \to \mathbb{R}_{>0}$,
das eine Menge V von n Städten verbindet.

Gesucht: Teilnetz $G' = (V, E')$ mit $E' \subseteq E$, so dass

(1) man von jeder Stadt in G' zu jeder anderen kommen kann („G' spannt G auf“) und

(2) die „Schneeräumkosten“ $w(E') = \sum_{e \in E'} w(e)$ minimal sind unter allen Teilnetzen, die (1) erfüllen.



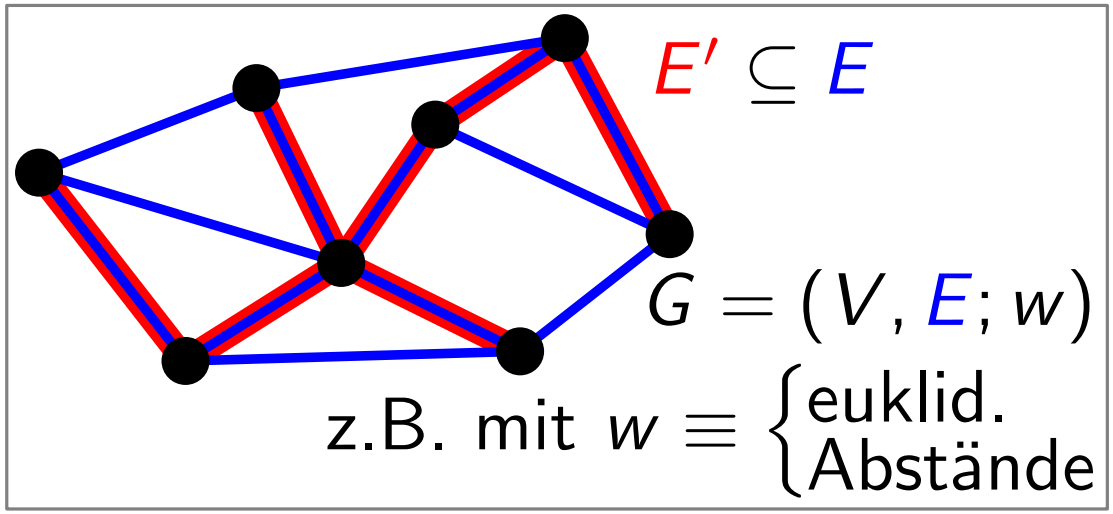
Motivation

Gegeben: zusammenhängendes Straßennetz $G = (V, E)$
mit Kantengewichten $w\colon E \to \mathbb{R}_{>0}$,
das eine Menge V von n Städten verbindet.

Gesucht: Teilnetz $G' = (V, E')$ mit $E' \subseteq E$, so dass

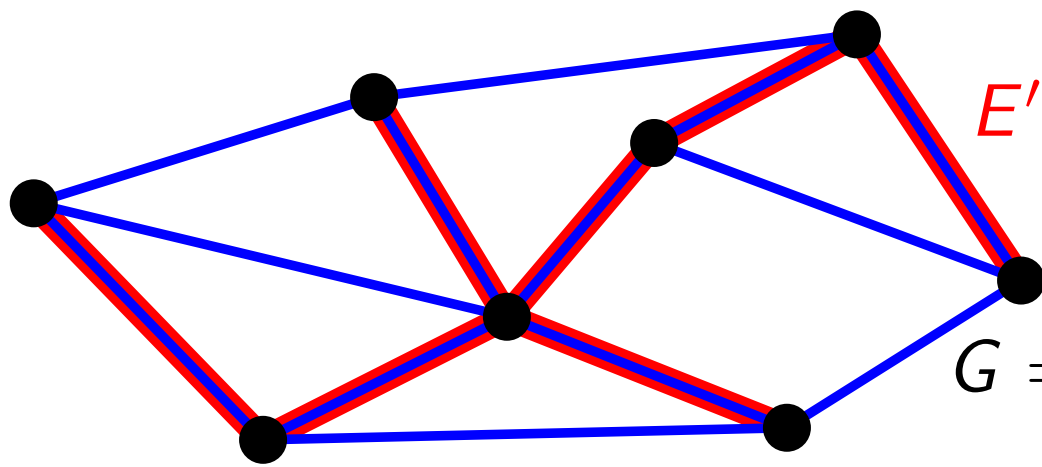
(1) man von jeder Stadt in G' zu jeder anderen kommen kann („G' spannt G auf“) und

(2) die „Schneeräumkosten“ $w(E') = \sum_{e \in E'} w(e)$ minimal sind unter allen Teilnetzen, die (1) erfüllen.



Beobachtung

Wegen der Minimalität von $w(E')$ gilt:



$E' \subseteq E$

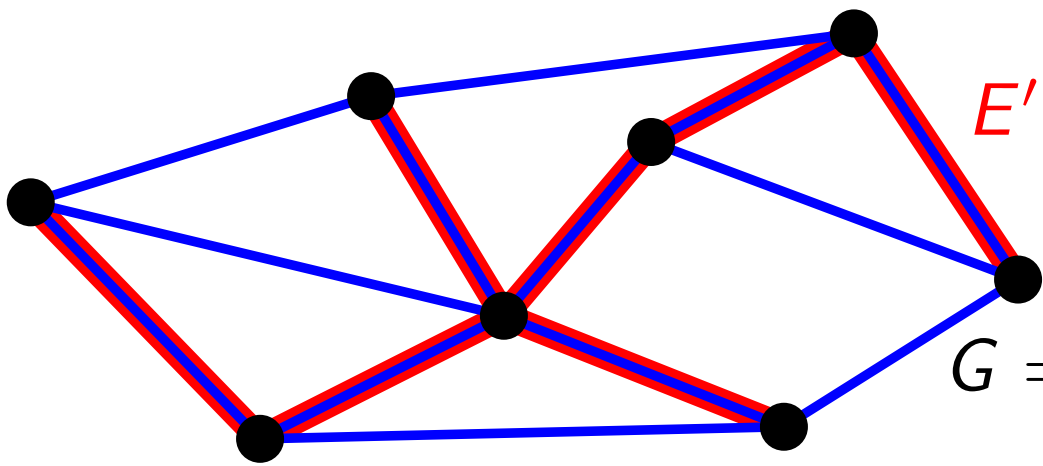
$G = (V, E)$

z.B. mit $w \equiv \begin{cases} \text{euklid.} \\ \text{Abstände} \end{cases}$

Beobachtung

Wegen der Minimalität von $w(E')$ gilt:

G' hat keine Kreise



$E' \subseteq E$

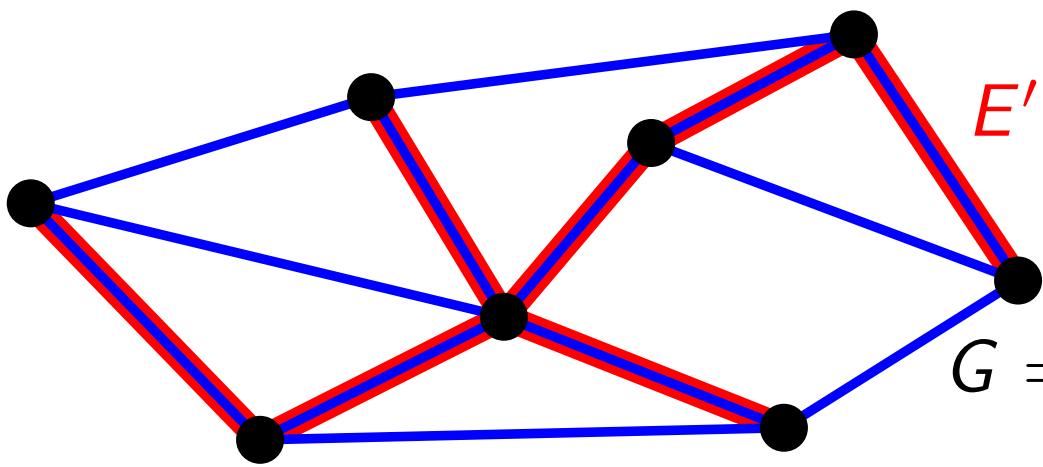
$G = (V, E)$

z.B. mit $w \equiv \begin{cases} \text{euklid.} \\ \text{Abstände} \end{cases}$

Beobachtung

Wegen der Minimalität von $w(E')$ gilt:

G' hat keine Kreise $\Rightarrow$ G' ist ein Wald.



$E' \subseteq E$

$G = (V, E)$

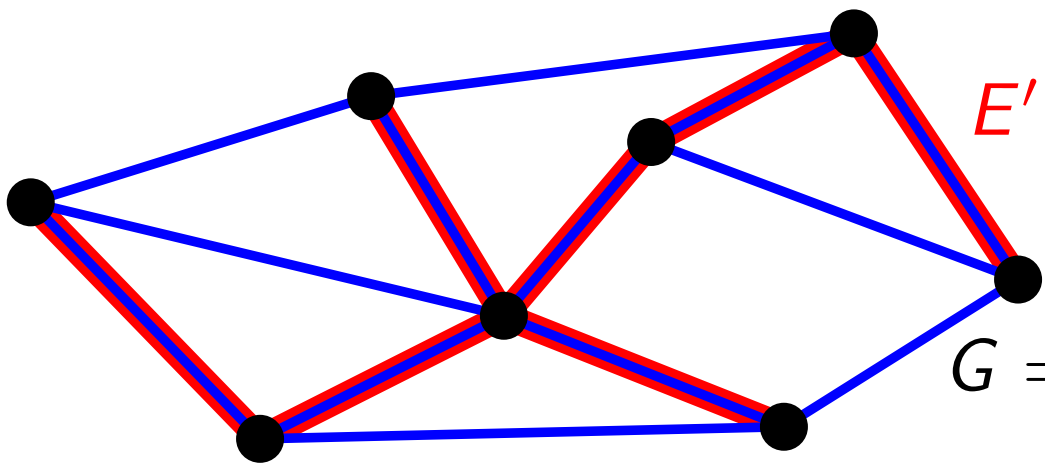
z.B. mit $w \equiv \begin{cases} \text{euklid.} \\ \text{Abstände} \end{cases}$

Beobachtung

Wegen der Minimalität von $w(E')$ gilt:

G' hat keine Kreise $\Rightarrow$ G' ist ein Wald.

G' „erbt“ Zusammenhang von G $\Rightarrow$ G' Baum.



$E' \subseteq E$

$G = (V, E)$

z.B. mit $w \equiv \begin{cases} \text{euklid.} \\ \text{Abstände} \end{cases}$

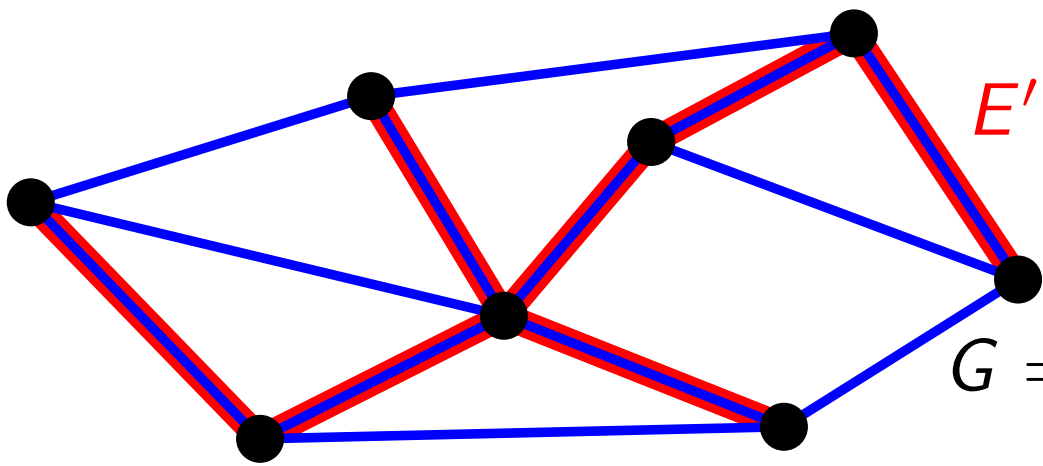
Beobachtung

Wegen der Minimalität von $w(E')$ gilt:

G' hat keine Kreise $\Rightarrow$ G' ist ein Wald.

G' „erbt“ Zusammenhang von G $\Rightarrow$ G' Baum.

G' spannt G auf $\Rightarrow$ G' ist Spannbaum von G.



$E' \subseteq E$

$G = (V, E)$

z.B. mit $w \equiv \begin{cases} \text{euklid.} \\ \text{Abstände} \end{cases}$

Beobachtung

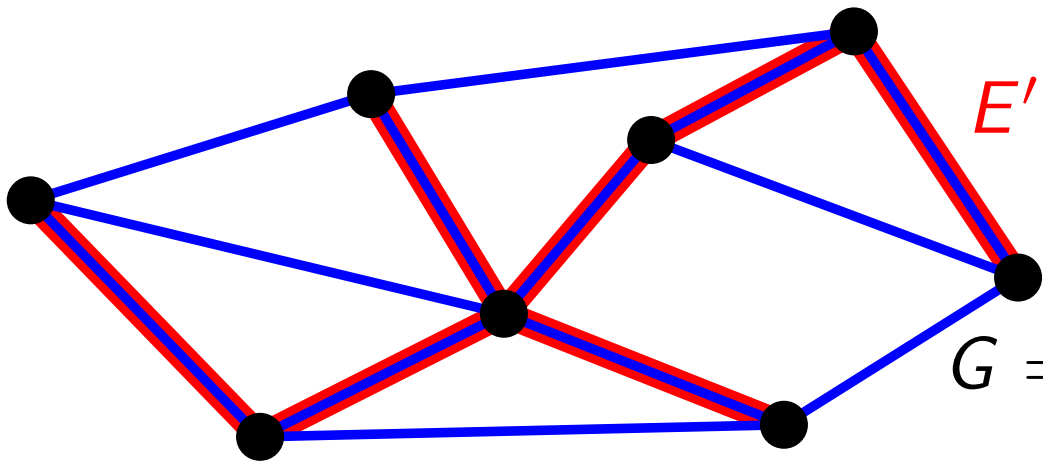
Wegen der Minimalität von $w(E')$ gilt:

G' hat keine Kreise $\Rightarrow$ G' ist ein Wald.

G' „erbt“ Zusammenhang von G $\Rightarrow$ G' Baum.

G' spannt G auf $\Rightarrow$ G' ist Spannbaum von G.

G' hat minimales Gewicht unter *allen* Spannbäumen von G.



$E' \subseteq E$

$G = (V, E)$

z.B. mit $w \equiv \begin{cases} \text{euklid.} \\ \text{Abstände} \end{cases}$

Beobachtung

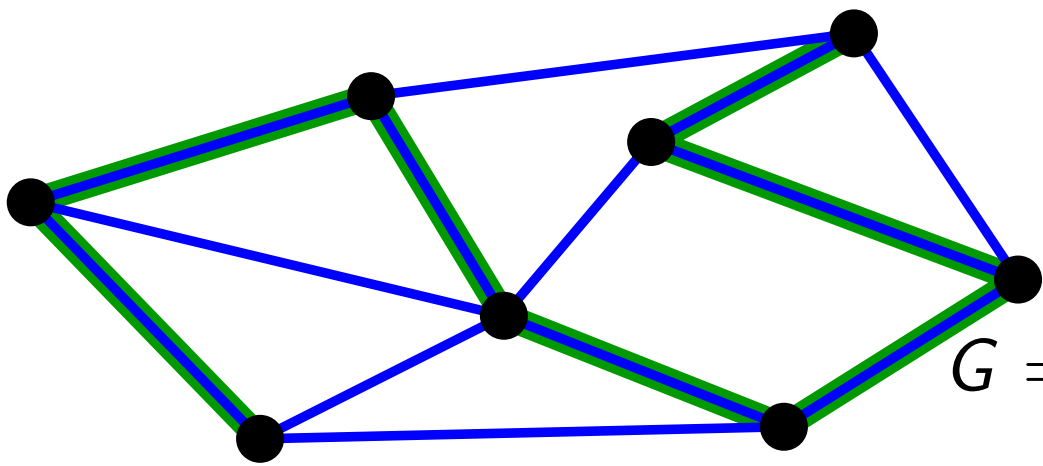
Wegen der Minimalität von $w(E')$ gilt:

G' hat keine Kreise $\Rightarrow$ G' ist ein Wald.

G' „erbt“ Zusammenhang von G $\Rightarrow$ G' Baum.

G' spannt G auf $\Rightarrow$ G' ist Spannbaum von G.

G' hat minimales Gewicht unter *allen* Spannbäumen von G.



$G = (V, E)$

z.B. mit $w \equiv \begin{cases} \text{euklid.} \\ \text{Abstände} \end{cases}$

Beobachtung

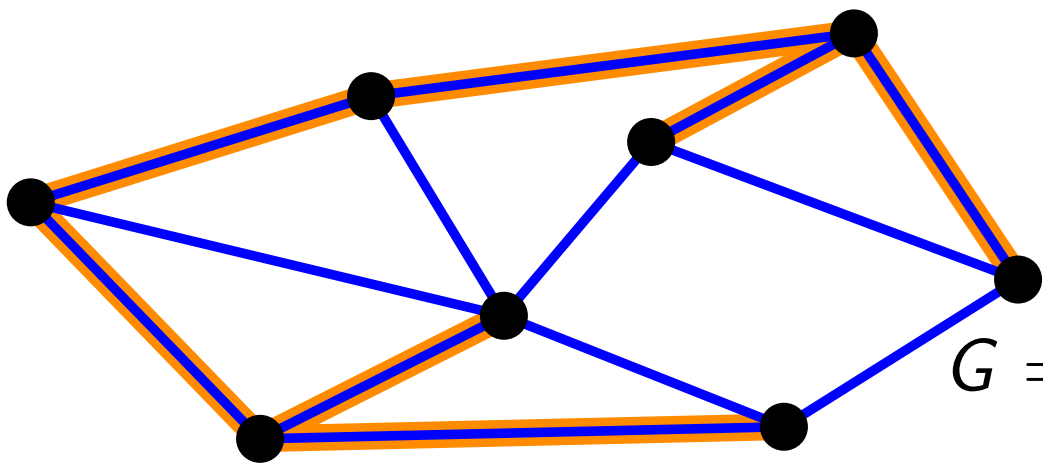
Wegen der Minimalität von $w(E')$ gilt:

G' hat keine Kreise $\Rightarrow$ G' ist ein Wald.

G' „erbt“ Zusammenhang von G $\Rightarrow$ G' Baum.

G' spannt G auf $\Rightarrow$ G' ist Spannbaum von G.

G' hat minimales Gewicht unter *allen* Spannbäumen von G.



$G = (V, E)$

z.B. mit $w \equiv \begin{cases} \text{euklid.} \\ \text{Abstände} \end{cases}$

Beobachtung

Wegen der Minimalität von $w(E')$ gilt:

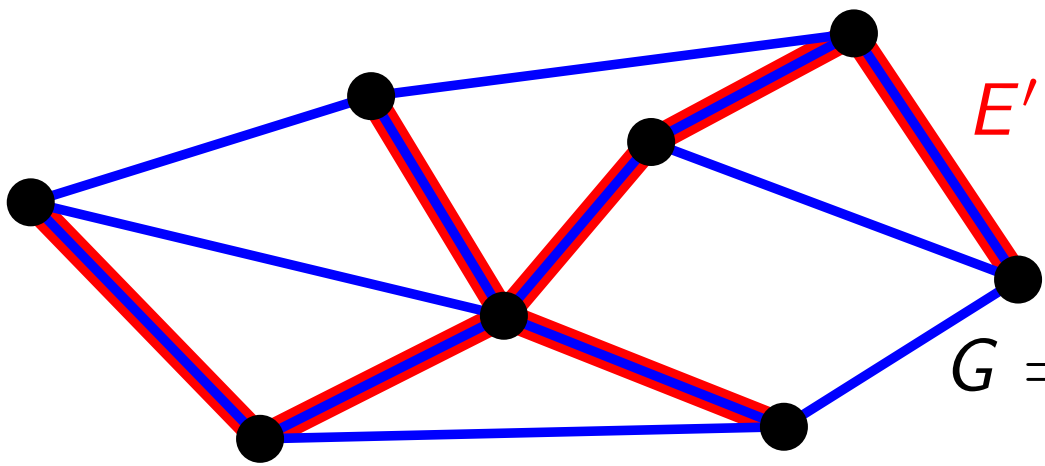
G' hat keine Kreise $\Rightarrow$ G' ist ein Wald.

G' „erbt“ Zusammenhang von G $\Rightarrow$ G' Baum.

G' spannt G auf $\Rightarrow$ G' ist Spannbaum von G.

G' hat minimales Gewicht unter *allen* Spannbäumen von G.

Wir nennen G' kurz *minimalen Spannbaum* von G.



$E' \subseteq E$

$G = (V, E)$

z.B. mit $w \equiv \begin{cases} \text{euklid.} \\ \text{Abstände} \end{cases}$

Beobachtung

Wegen der Minimalität von $w(E')$ gilt:

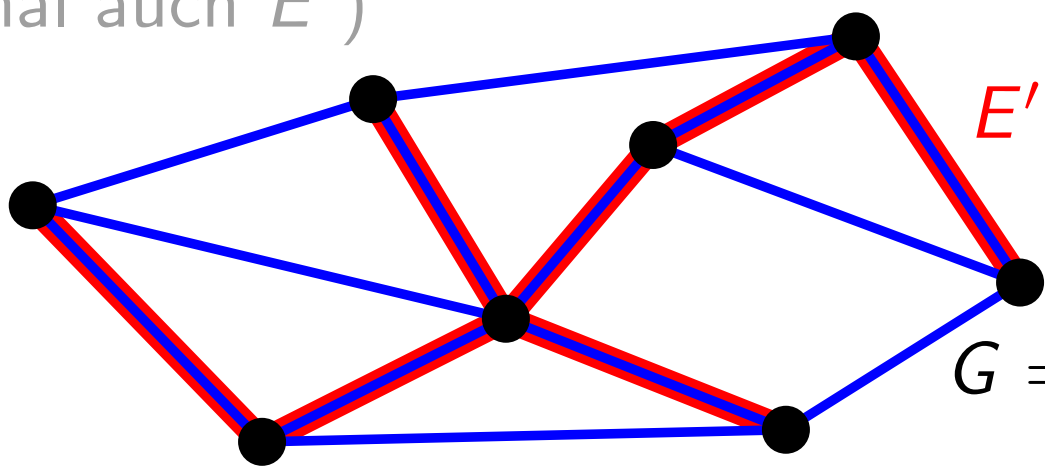
G' hat keine Kreise $\Rightarrow$ G' ist ein Wald.

G' „erbt“ Zusammenhang von G $\Rightarrow$ G' Baum.

G' spannt G auf $\Rightarrow$ G' ist Spannbaum von G.

G' hat minimales Gewicht unter *allen* Spannbäumen von G.

Wir nennen G' kurz *minimalen Spannbaum* von G.
(manchmal auch E')



$E' \subseteq E$

$G = (V, E)$

z.B. mit $w \equiv \begin{cases} \text{euklid.} \\ \text{Abstände} \end{cases}$

Beobachtung

Wegen der Minimalität von $w(E')$ gilt:

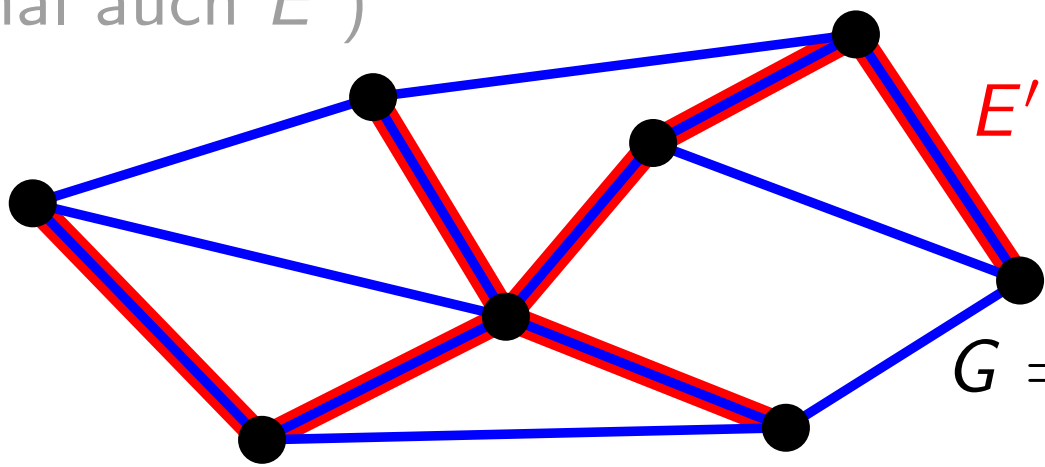
G' hat keine Kreise $\Rightarrow$ G' ist ein Wald.

G' „erbt“ Zusammenhang von G $\Rightarrow$ G' Baum.

G' spannt G auf $\Rightarrow$ G' ist Spannbaum von G.

G' hat minimales Gewicht unter *allen* Spannbäumen von G.

Wir nennen G' kurz *minimalen Spannbaum* von G.
(manchmal auch E')



$E' \subseteq E$

$G = (V, E)$

Beob. $|E'| =$ **?**

z.B. mit $w \equiv \begin{cases} \text{euklid.} \\ \text{Abstände} \end{cases}$

Beobachtung

Wegen der Minimalität von $w(E')$ gilt:

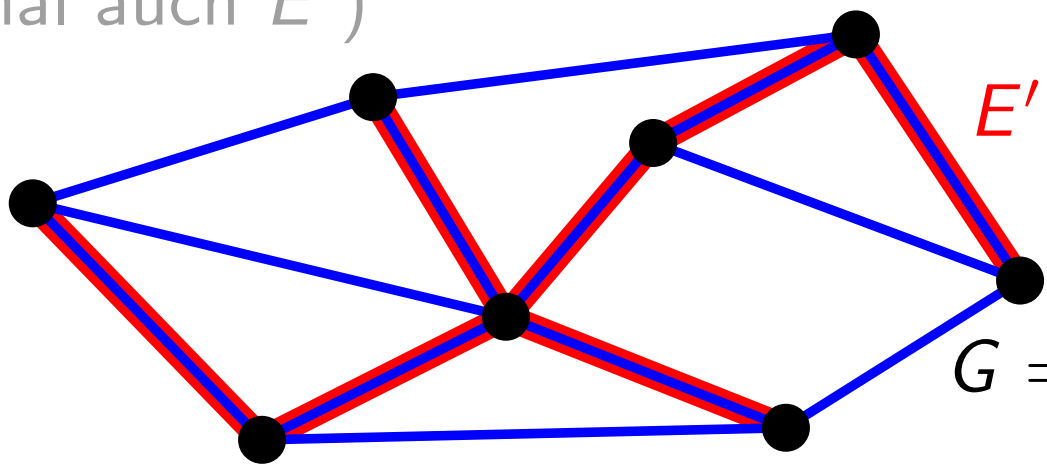
G' hat keine Kreise $\Rightarrow$ G' ist ein Wald.

G' „erbt“ Zusammenhang von G $\Rightarrow$ G' Baum.

G' spannt G auf $\Rightarrow$ G' ist Spannbaum von G.

G' hat minimales Gewicht unter *allen* Spannbäumen von G.

Wir nennen G' kurz *minimalen Spannbaum* von G.
(manchmal auch E')



$E' \subseteq E$

$G = (V, E)$

Beob. $|E'| = |V| - 1$

z.B. mit $w \equiv \begin{cases} \text{euklid.} \\ \text{Abstände} \end{cases}$

Generischer Min.-Spannbaum-Algorithmus

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

Generischer Min.-Spannbaum-Algorithmus

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$

while $|A| < |V| - 1$ **do**

    // *Invariante:* A ist Teilmenge eines min. Spannbaums von G

    finde Kante uv, die *sicher* für A ist

    $A = A \cup \{uv\}$

return A

Generischer Min.-Spannbaum-Algorithmus

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$

while $|A| < |V| - 1$ **do**

  // *Invariante:* A ist Teilmenge eines min. Spannbaums von G

  finde Kante uv, die *sicher* für A ist

  $A = A \cup \{uv\}$

return A

Generischer Min.-Spannbaum-Algorithmus

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$

while $|A| < |V| - 1$ **do**

  // *Invariante:* A ist Teilmenge eines min. Spannbaums von G

  finde Kante uv, die *sicher* für A ist

  $A = A \cup \{uv\}$

return A

Wir sagen uv ist *sicher* für A, falls Invariante für $A \cup \{uv\}$ gilt.

Generischer Min.-Spannbaum-Algorithmus

```
GenericMST(UndirectedConnectedGraph G, EdgeWeights w)
  A = ∅
  while |A| < |V| − 1 do
      // Invariante: A ist Teilmenge eines min. Spannbaums von G
      finde Kante uv, die sicher für A ist
      A = A ∪ {uv}
  return A
```

Wir sagen *uv* ist *sicher* für A, falls Invariante für $A \cup \{uv\}$ gilt.

Beob. Dies ist ein sogenannter *Greedy-Algorithmus*!

Generischer Min.-Spannbaum-Algorithmus

```
GenericMST(UndirectedConnectedGraph G, EdgeWeights w)
  A = ∅
  while |A| < |V| − 1 do
    // Invariante: A ist Teilmenge eines min. Spannbaums von G
    finde Kante uv, die sicher für A ist
    A = A ∪ {uv}
  return A
```

Wir sagen *uv* ist *sicher* für A, falls Invariante für $A \cup \{uv\}$ gilt.

Beob. Dies ist ein sogenannter *Greedy-Algorithmus*!

Frage: Gibt's überhaupt immer eine sichere Kante?

Generischer Min.-Spannbaum-Algorithmus

```
GenericMST(UndirectedConnectedGraph G, EdgeWeights w)
  A = ∅
  while |A| < |V| − 1 do
    // Invariante: A ist Teilmenge eines min. Spannbaums von G
    finde Kante uv, die sicher für A ist
    A = A ∪ {uv}
  return A
```

Wir sagen *uv* ist *sicher* für A, falls Invariante für $A \cup \{uv\}$ gilt.

Beob. Dies ist ein sogenannter *Greedy-Algorithmus*!

Frage: Gibt's überhaupt immer eine sichere Kante?

Antwort: Ja!

Generischer Min.-Spannbaum-Algorithmus

```
GenericMST(UndirectedConnectedGraph G, EdgeWeights w)
  A = ∅
  while |A| < |V| − 1 do
    // Invariante: A ist Teilmenge eines min. Spannbaums von G
    finde Kante uv, die sicher für A ist
    A = A ∪ {uv}
  return A
```

Wir sagen uv ist *sicher* für A, falls Invariante für $A \cup \{uv\}$ gilt.

Beob. Dies ist ein sogenannter *Greedy-Algorithmus*!

Frage: Gibt's überhaupt immer eine sichere Kante?

Antwort: Ja! – *Per Induktion!*

Generischer Min.-Spannbaum-Algorithmus

```
GenericMST(UndirectedConnectedGraph G, EdgeWeights w)
  A = ∅
  while |A| < |V| − 1 do
      // Invariante: A ist Teilmenge eines min. Spannbaums von G
      finde Kante uv, die sicher für A ist
      A = A ∪ {uv}
  return A
```

Wir sagen *uv* ist *sicher* für A, falls Invariante für $A \cup \{uv\}$ gilt.

Beob. Dies ist ein sogenannter *Greedy-Algorithmus*!

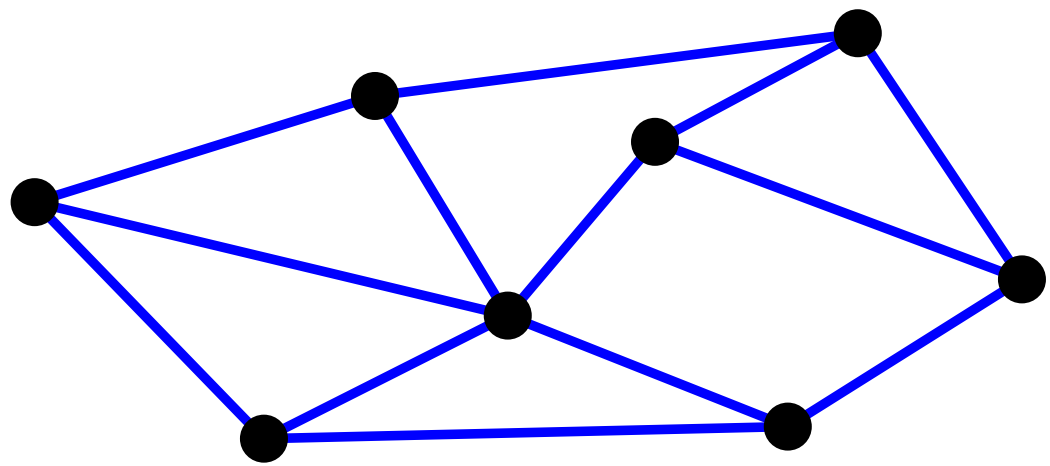
Frage: Gibt's überhaupt immer eine sichere Kante?

Antwort: Ja! – *Per Induktion!*

Frage: Aber wie findet man eine – ohne schon einen minimalen Spannbaum zu kennen?

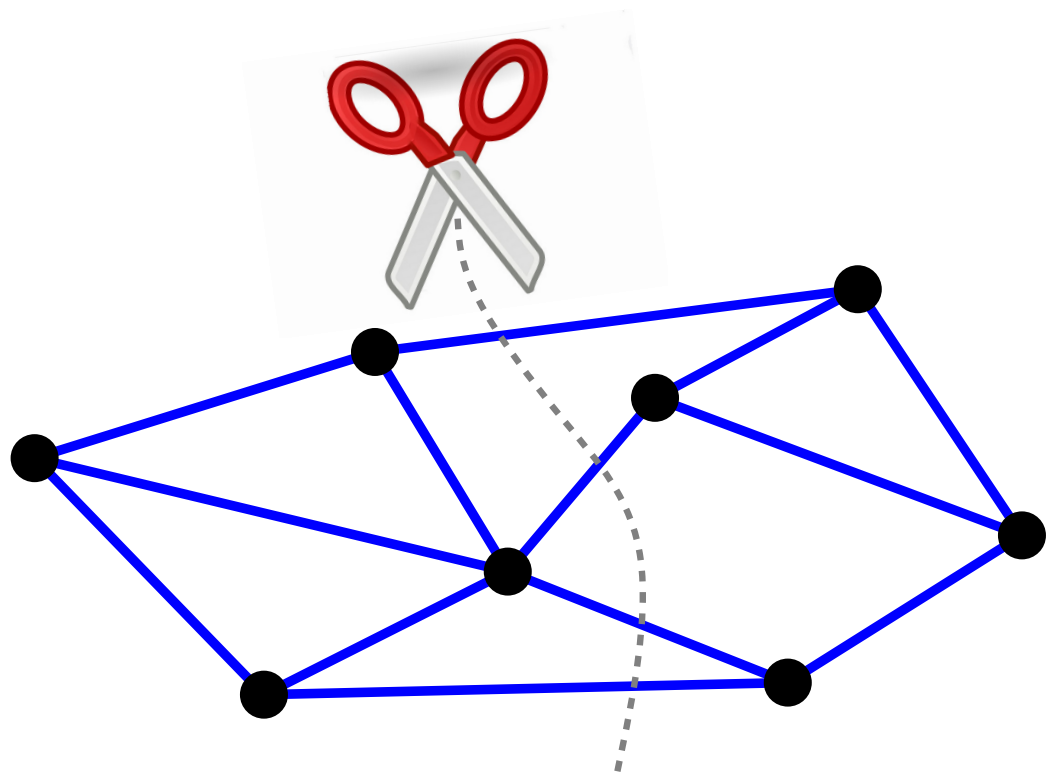
Schnitte und leichte Kanten

Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung*) von V.



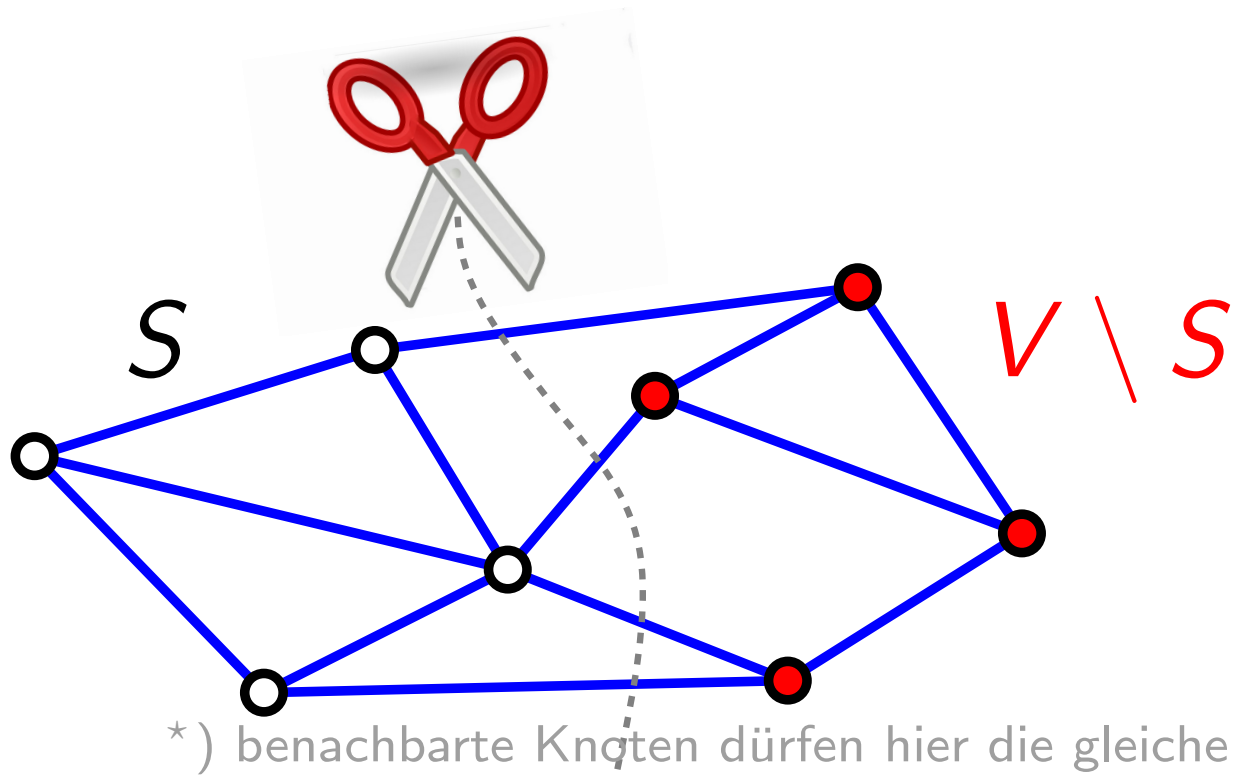
Schnitte und leichte Kanten

Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung*) von V.



Schnitte und leichte Kanten

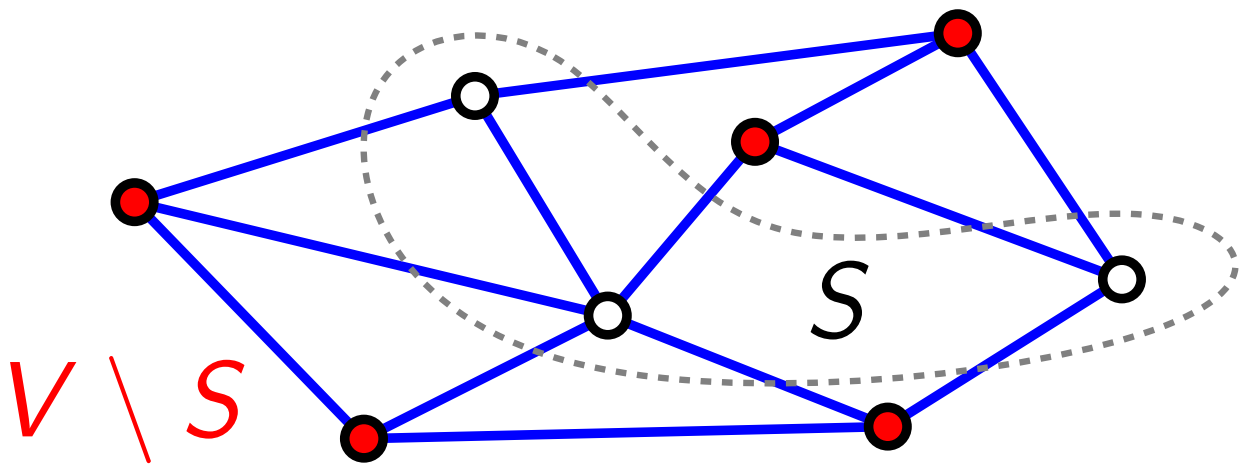
Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung*) von V.



*) benachbarte Knoten dürfen hier die gleiche Farbe haben.

Schnitte und leichte Kanten

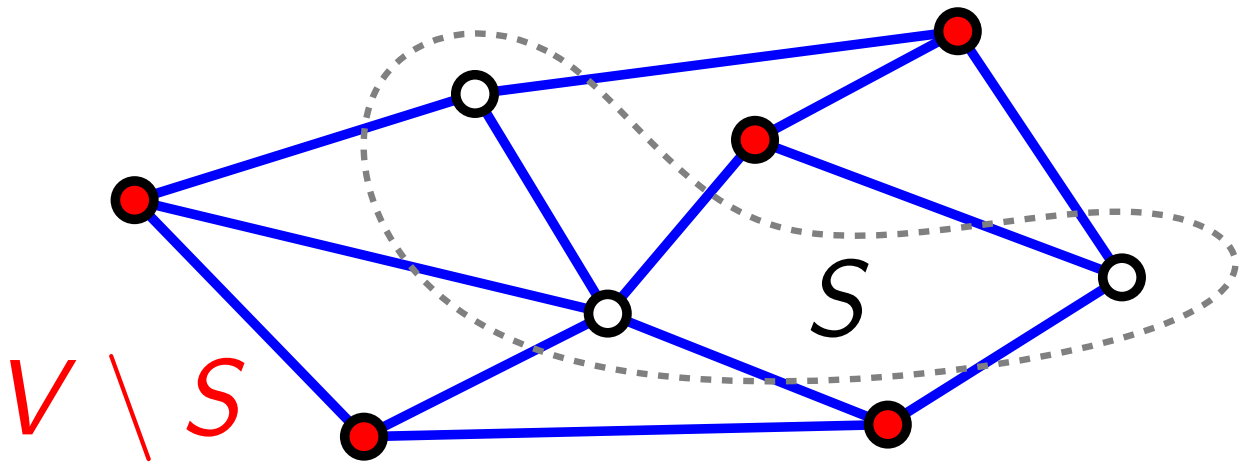
Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung*) von V.



Schnitte und leichte Kanten

Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung*) von V.

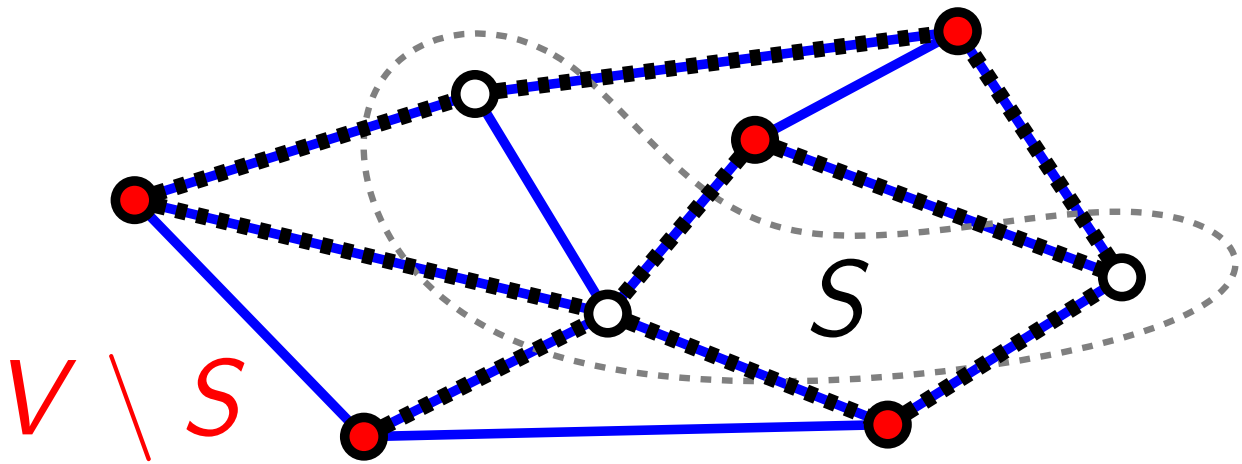
Eine Kante e *kreuzt* $(S, V \setminus S)$, wenn ein Endpunkt von e in S und der andere in $V \setminus S$ liegt.



Schnitte und leichte Kanten

Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung*) von V.

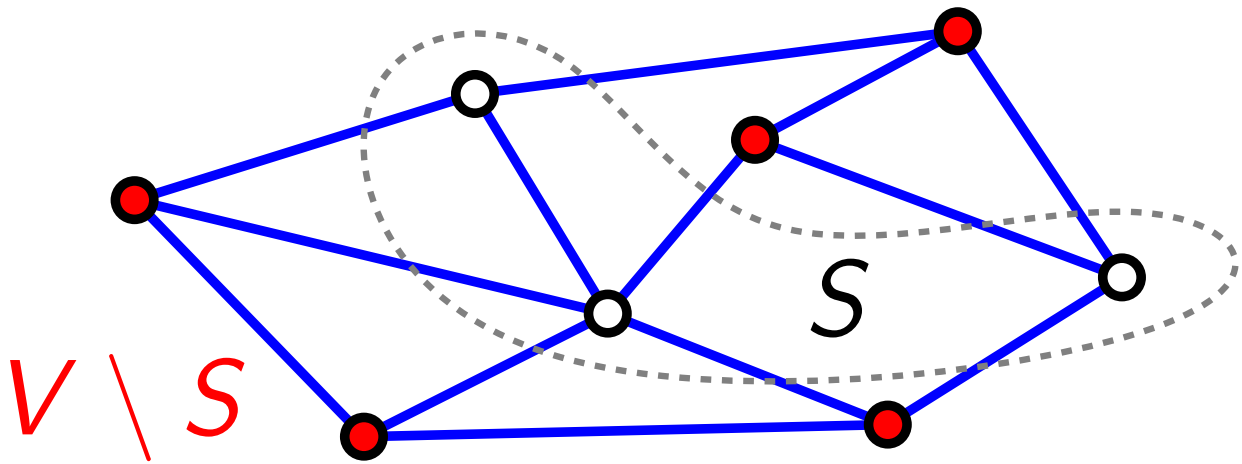
Eine Kante e *kreuzt* $(S, V \setminus S)$, wenn ein Endpunkt von e in S und der andere in $V \setminus S$ liegt.



Schnitte und leichte Kanten

Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung*) von V.

Eine Kante e *kreuzt* $(S, V \setminus S)$, wenn ein Endpunkt von e in S und der andere in $V \setminus S$ liegt.

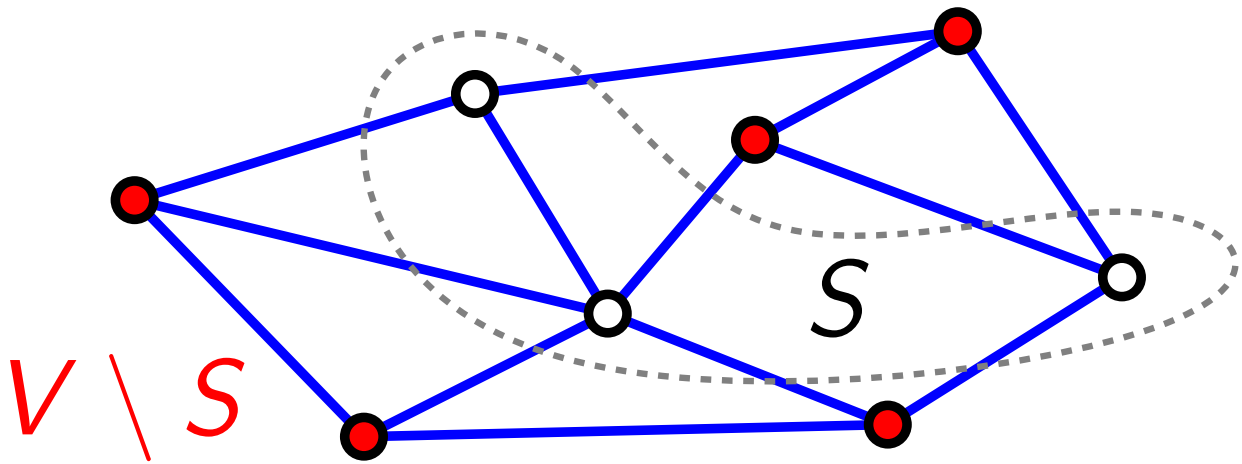


Schnitte und leichte Kanten

Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung*) von V.

Eine Kante e *kreuzt* $(S, V \setminus S)$, wenn ein Endpunkt von e in S und der andere in $V \setminus S$ liegt.

Ein Schnitt *respektiert* eine Kantenmenge A, wenn keine Kante in A den Schnitt kreuzt.

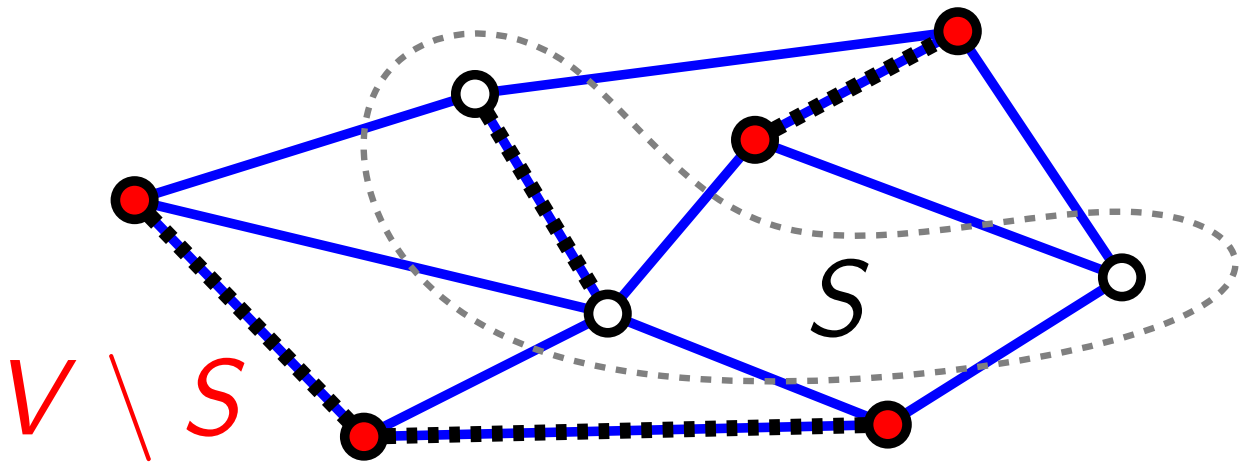


Schnitte und leichte Kanten

Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung*) von V.

Eine Kante e *kreuzt* $(S, V \setminus S)$, wenn ein Endpunkt von e in S und der andere in $V \setminus S$ liegt.

Ein Schnitt *respektiert* eine Kantenmenge A, wenn keine Kante in A den Schnitt kreuzt.

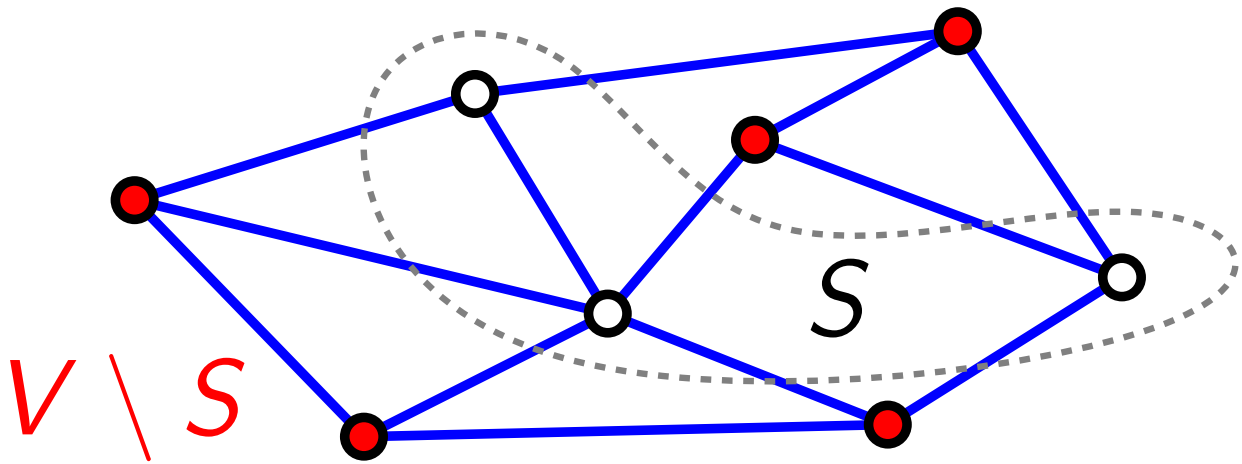


Schnitte und leichte Kanten

Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung*) von V.

Eine Kante e *kreuzt* $(S, V \setminus S)$, wenn ein Endpunkt von e in S und der andere in $V \setminus S$ liegt.

Ein Schnitt *respektiert* eine Kantenmenge A, wenn keine Kante in A den Schnitt kreuzt.



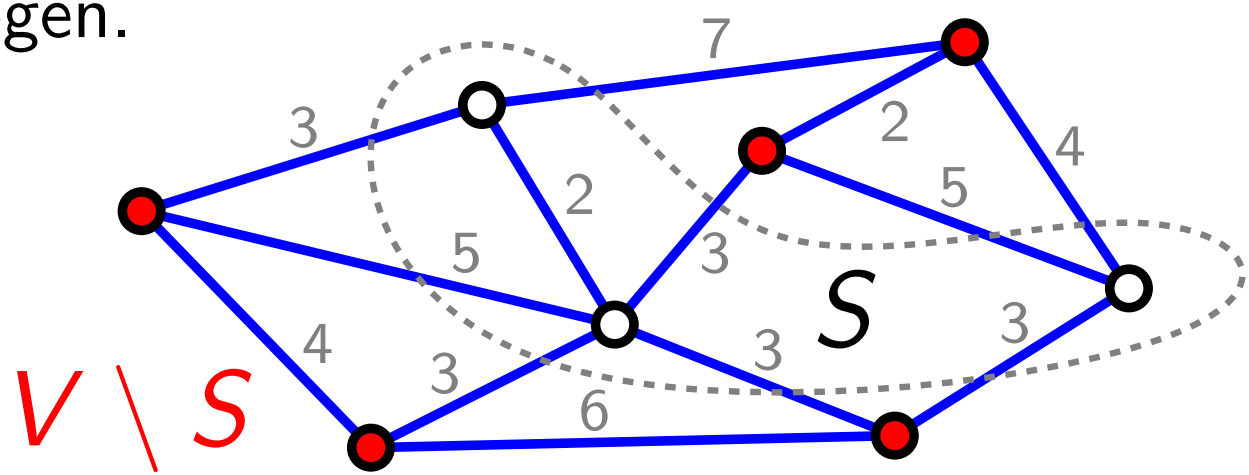
Schnitte und leichte Kanten

Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung*) von V.

Eine Kante e *kreuzt* $(S, V \setminus S)$, wenn ein Endpunkt von e in S und der andere in $V \setminus S$ liegt.

Ein Schnitt *respektiert* eine Kantenmenge A, wenn keine Kante in A den Schnitt kreuzt.

Eine Kante e, die einen Schnitt kreuzt, ist *leicht*, wenn alle Kanten, die den Schnitt kreuzen, mindestens $w(e)$ wiegen.



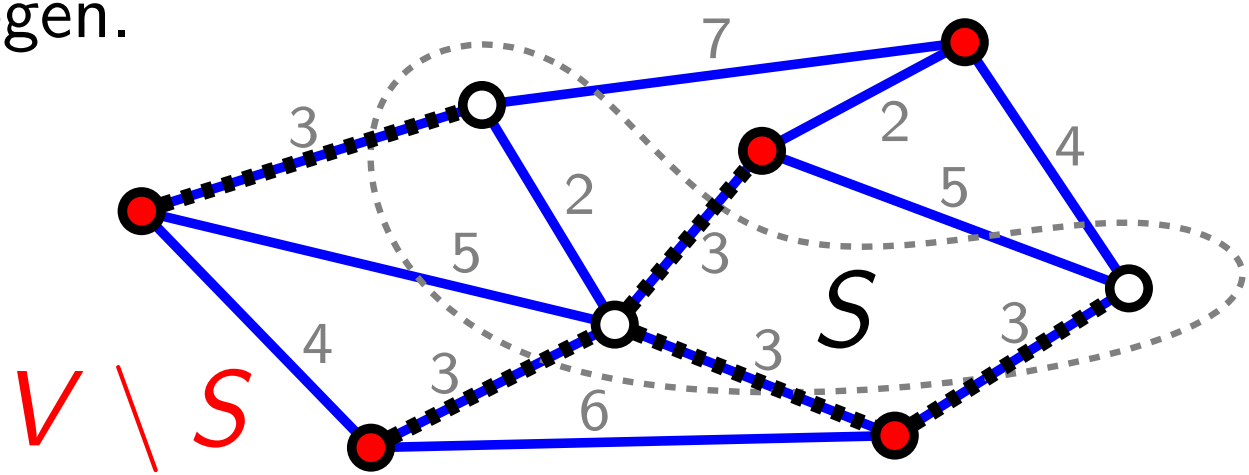
Schnitte und leichte Kanten

Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung*) von V.

Eine Kante e *kreuzt* $(S, V \setminus S)$, wenn ein Endpunkt von e in S und der andere in $V \setminus S$ liegt.

Ein Schnitt *respektiert* eine Kantenmenge A, wenn keine Kante in A den Schnitt kreuzt.

Eine Kante e, die einen Schnitt kreuzt, ist *leicht*, wenn alle Kanten, die den Schnitt kreuzen, mindestens $w(e)$ wiegen.



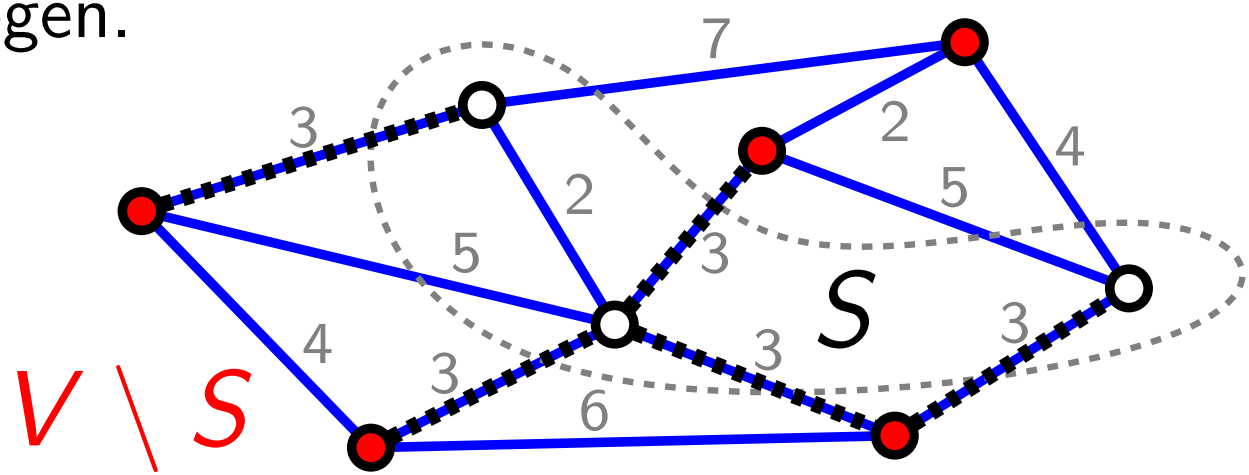
Schnitte und leichte Kanten

Def. Ein *Schnitt* $(S, V \setminus S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Zerlegung (od. Zweifärbung*) von V.

Eine Kante e *kreuzt* $(S, V \setminus S)$, wenn ein Endpunkt von e in S und der andere in $V \setminus S$ liegt.

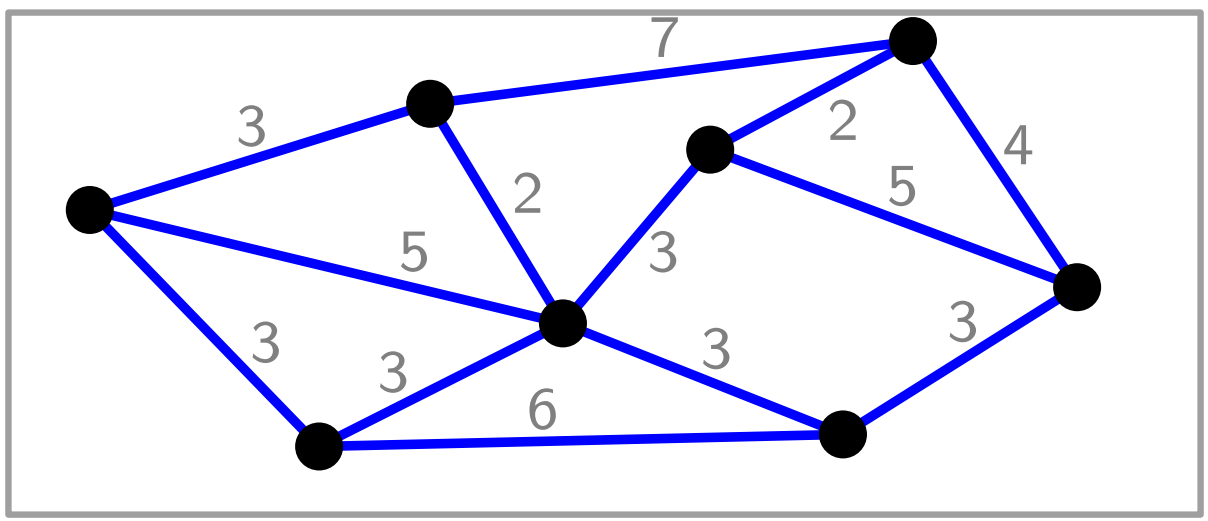
Ein Schnitt *respektiert* eine Kantenmenge A, wenn keine Kante in A den Schnitt kreuzt.

Eine Kante e, die einen Schnitt kreuzt, ist *leicht*, wenn alle Kanten, die den Schnitt kreuzen, mindestens $w(e)$ wiegen.



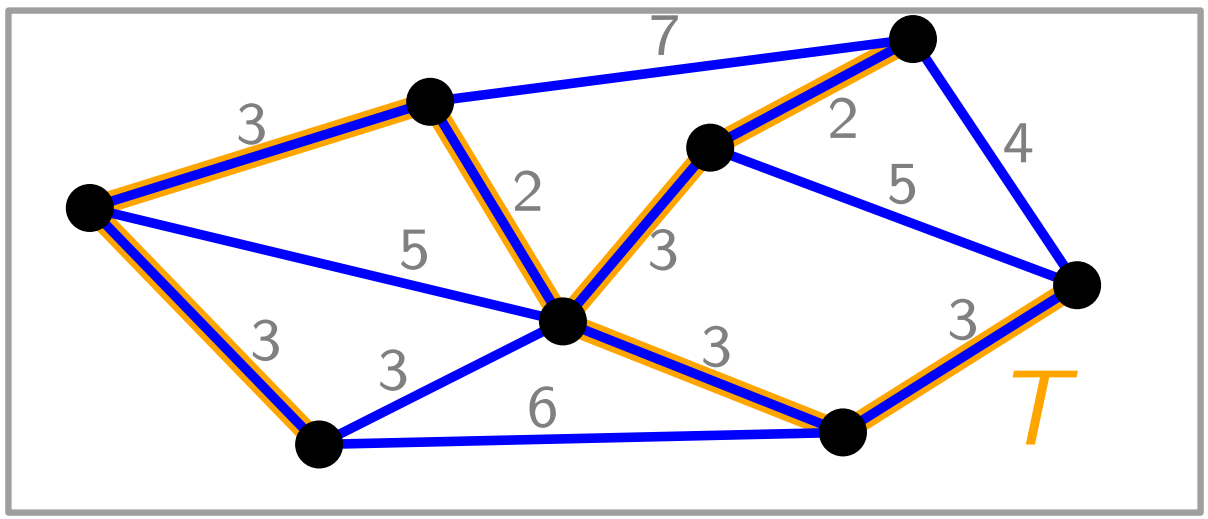
Erweiterungssatz

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.



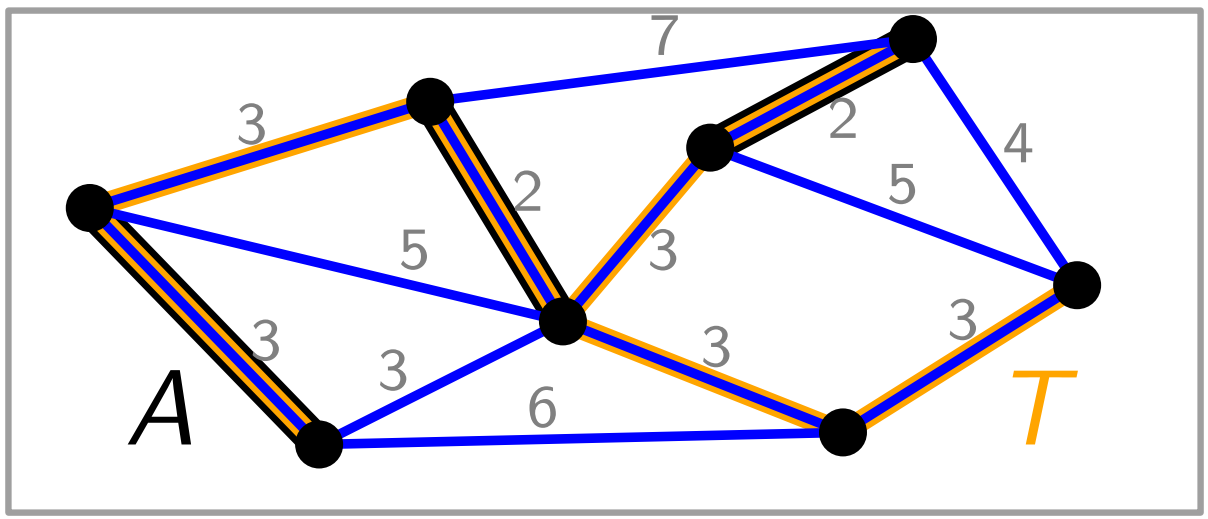
Erweiterungssatz

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.



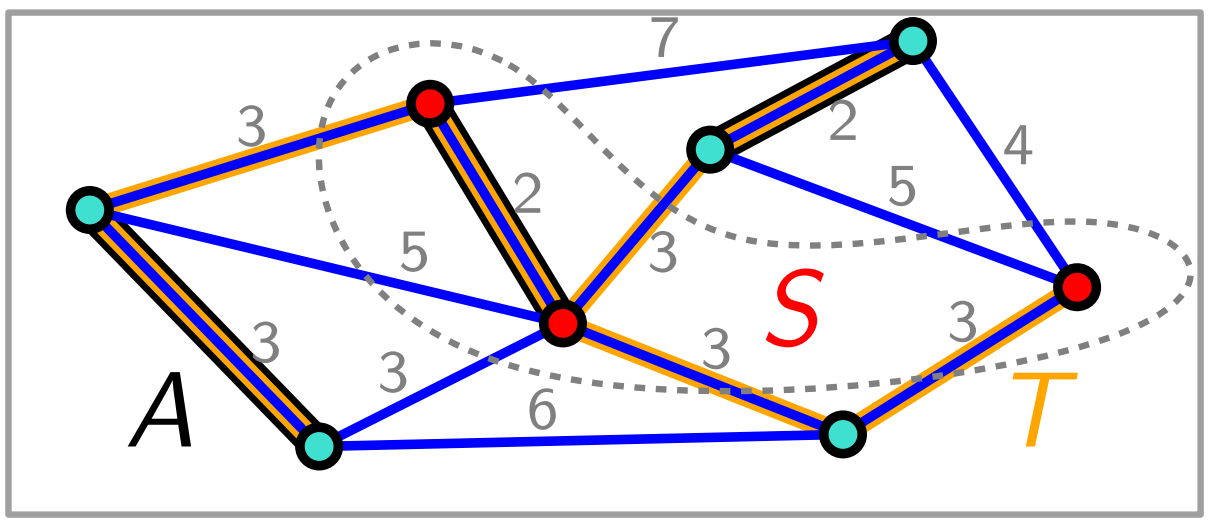
Erweiterungssatz

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei **A** Teilmenge von T.



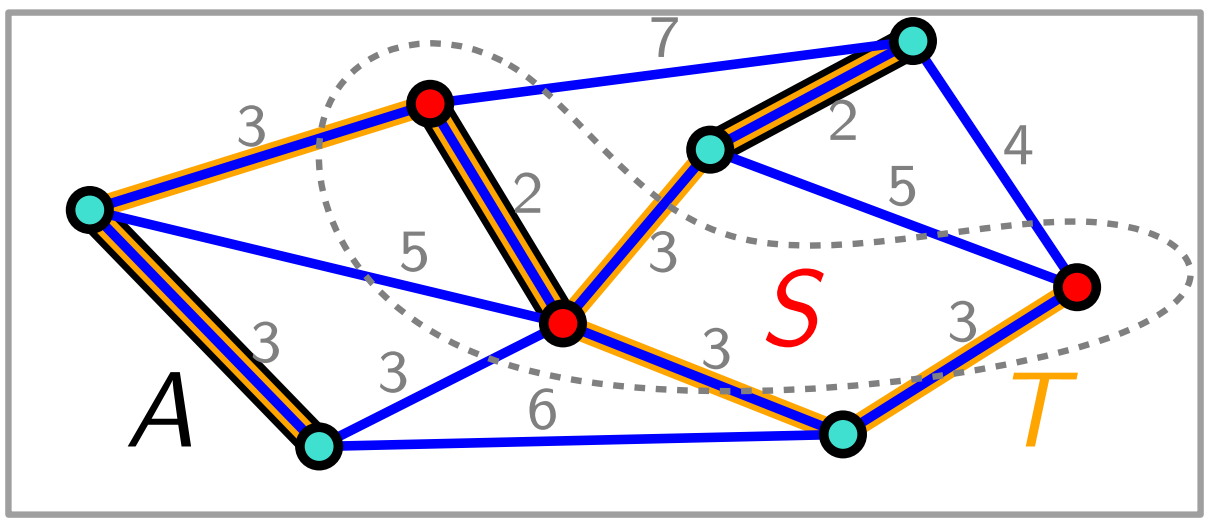
Erweiterungssatz

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei **A** Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der **A** respektiert.



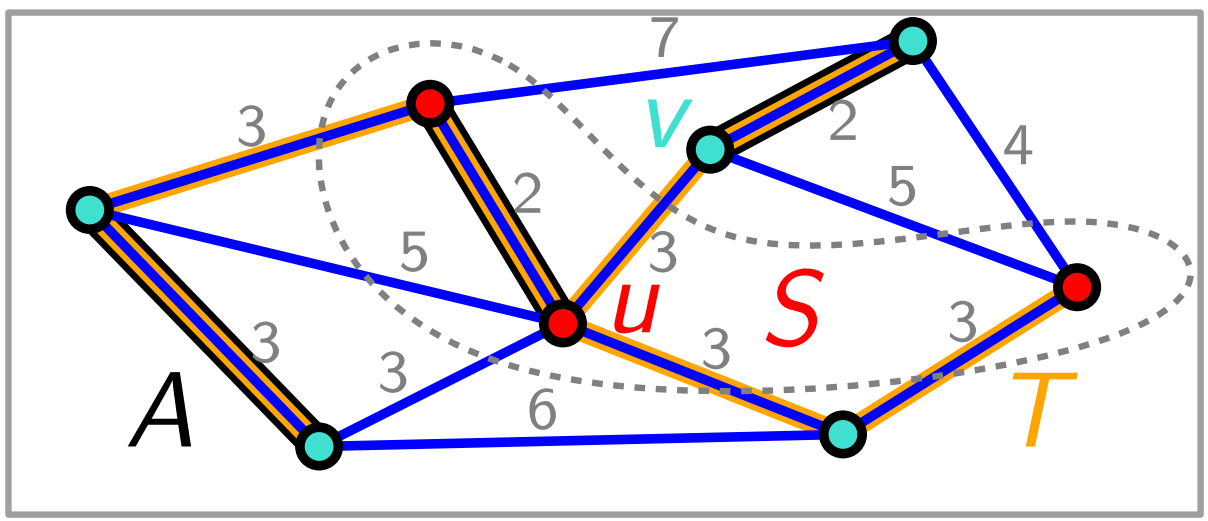
Erweiterungssatz

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei **A** Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der **A** respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.



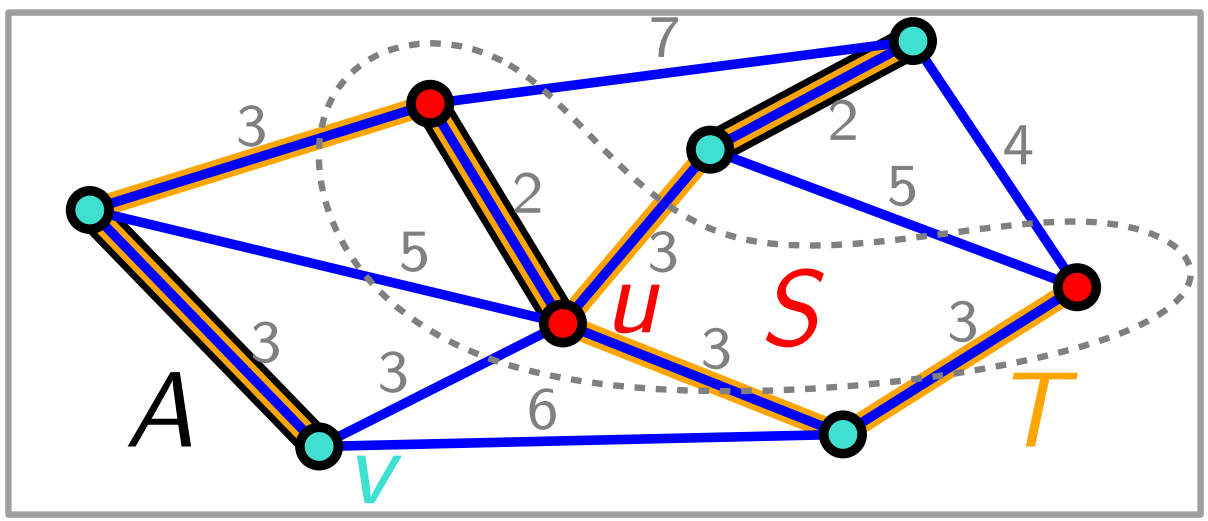
Erweiterungssatz

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei **A** Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der **A** respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.



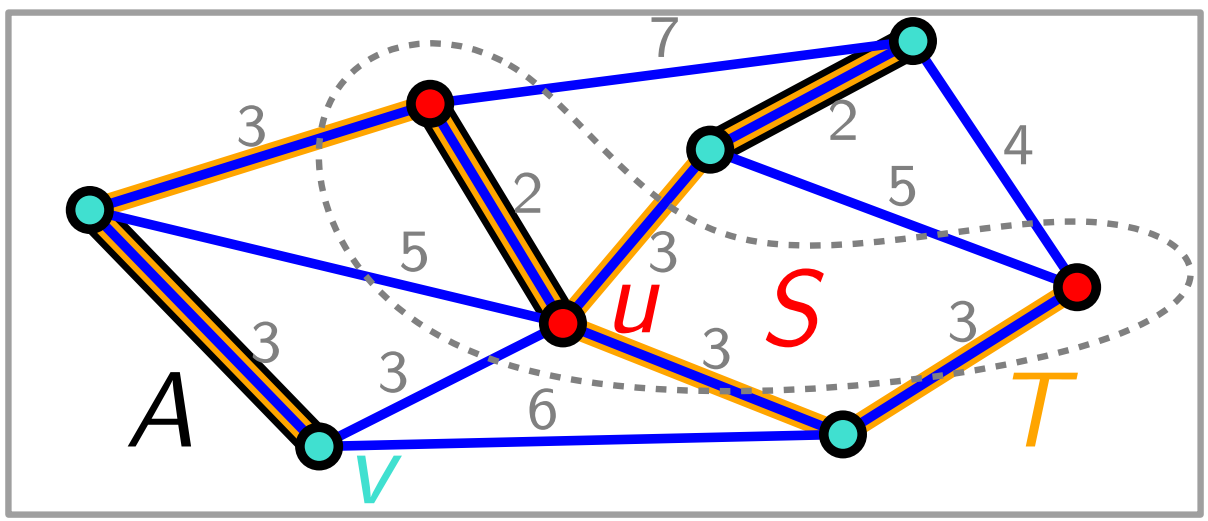
Erweiterungssatz

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei **A** Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der **A** respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.



Erweiterungssatz

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei **A** Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der **A** respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für **A**.



Erweiterungssatz

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.

Sei T Kantenmenge eines min. Spannbaums von G.

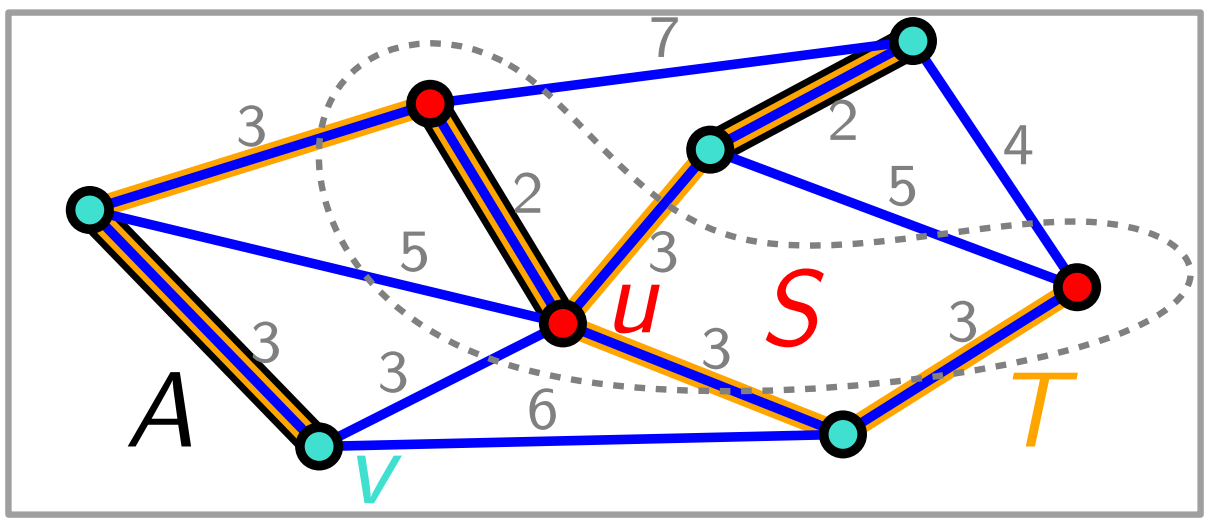
Sei **A** Teilmenge von T.

Sei $(S, V \setminus S)$ ein Schnitt, der **A** respektiert.

Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.

Dann ist uv sicher für **A**,

d.h. G hat einen min. Spannbaum, der $A \cup \{uv\}$ enthält.



Erweiterungssatz

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.

Sei T Kantenmenge eines min. Spannbaums von G.

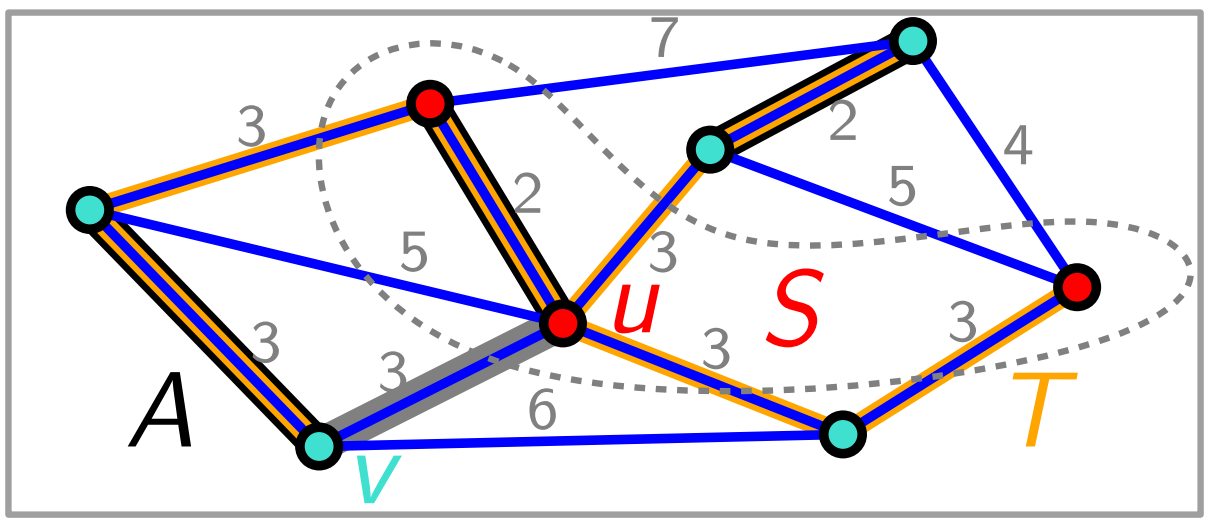
Sei **A** Teilmenge von T.

Sei $(S, V \setminus S)$ ein Schnitt, der **A** respektiert.

Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.

Dann ist uv sicher für **A**,

d.h. G hat einen min. Spannbaum, der $A \cup \{uv\}$ enthält.



Erweiterungssatz

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.

Sei T Kantenmenge eines min. Spannbaums von G.

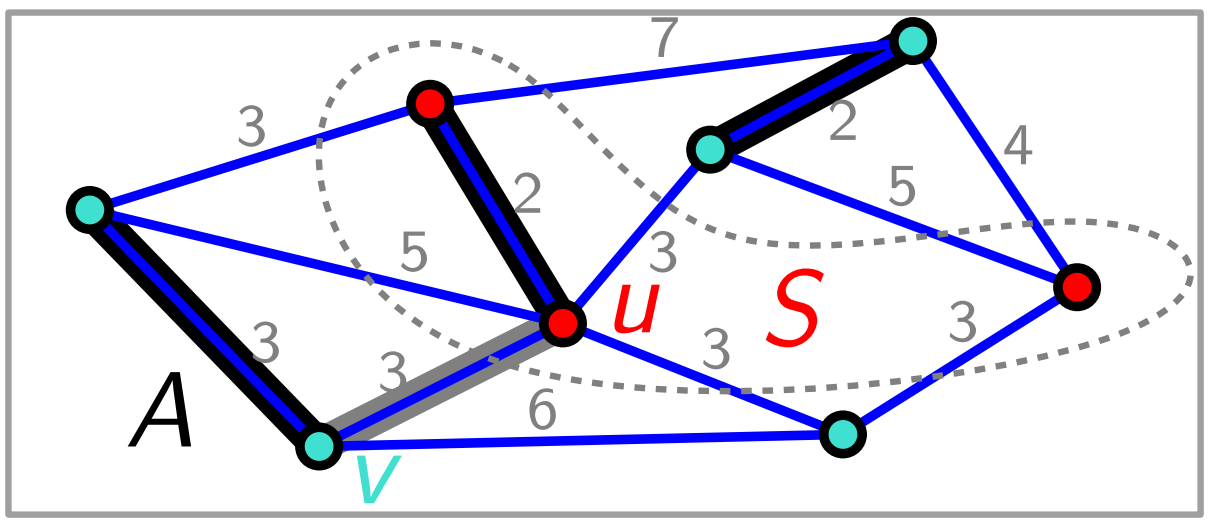
Sei **A** Teilmenge von T.

Sei $(S, V \setminus S)$ ein Schnitt, der **A** respektiert.

Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.

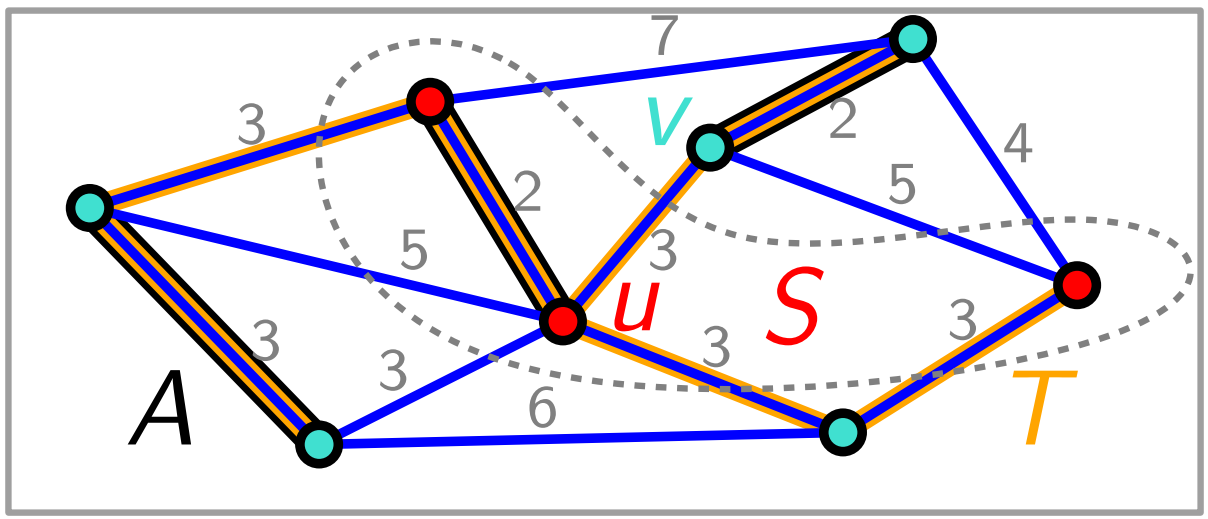
Dann ist uv sicher für **A**,

d.h. G hat einen min. Spannbaum, der $A \cup \{uv\}$ enthält.



Beweis

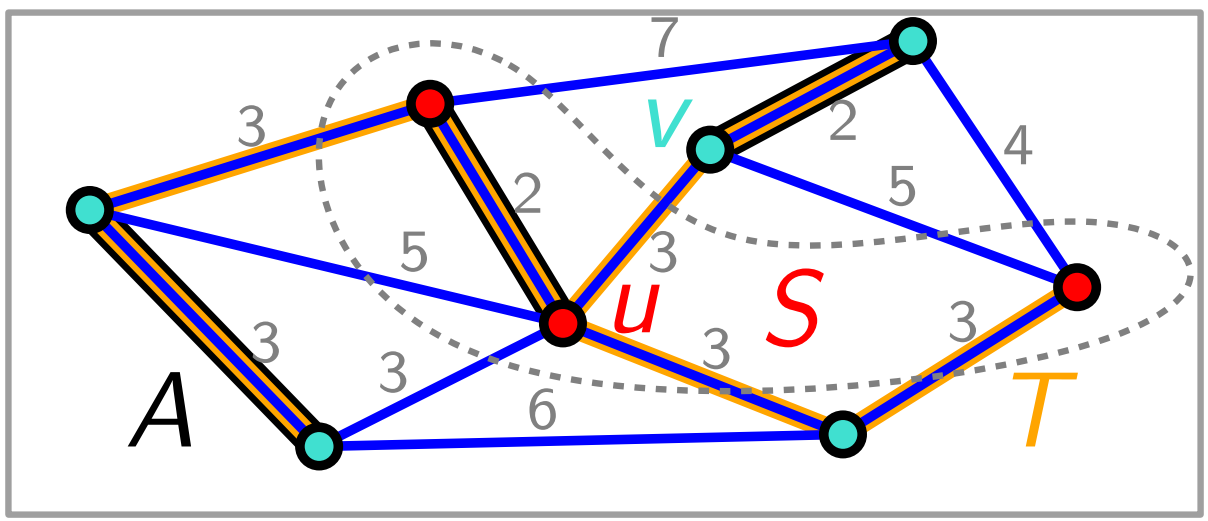
Satz. ... Dann ist uv sicher für A.



Beweis

Satz. ... Dann ist uv sicher für A.

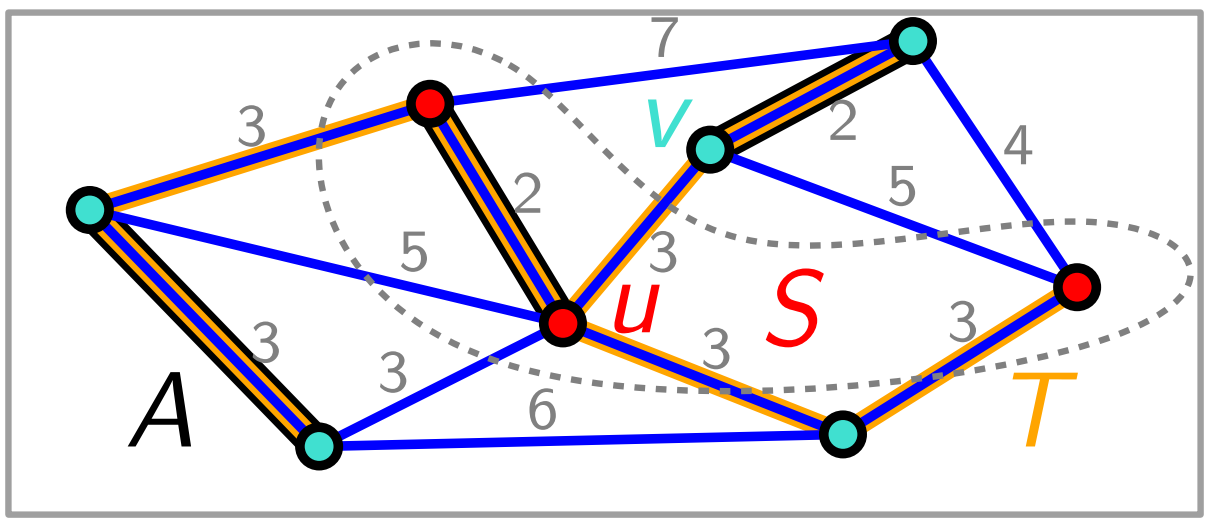
Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.



Beweis

Satz. ... Dann ist uv sicher für A.

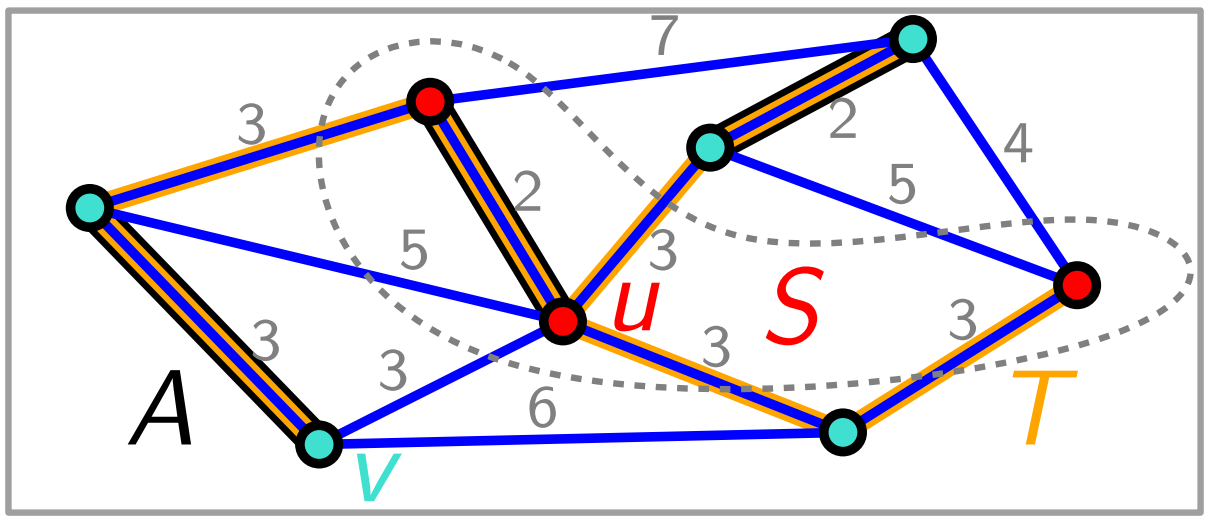
Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.
Falls $uv \in T$, fertig.



Beweis

Satz. ... Dann ist uv sicher für A.

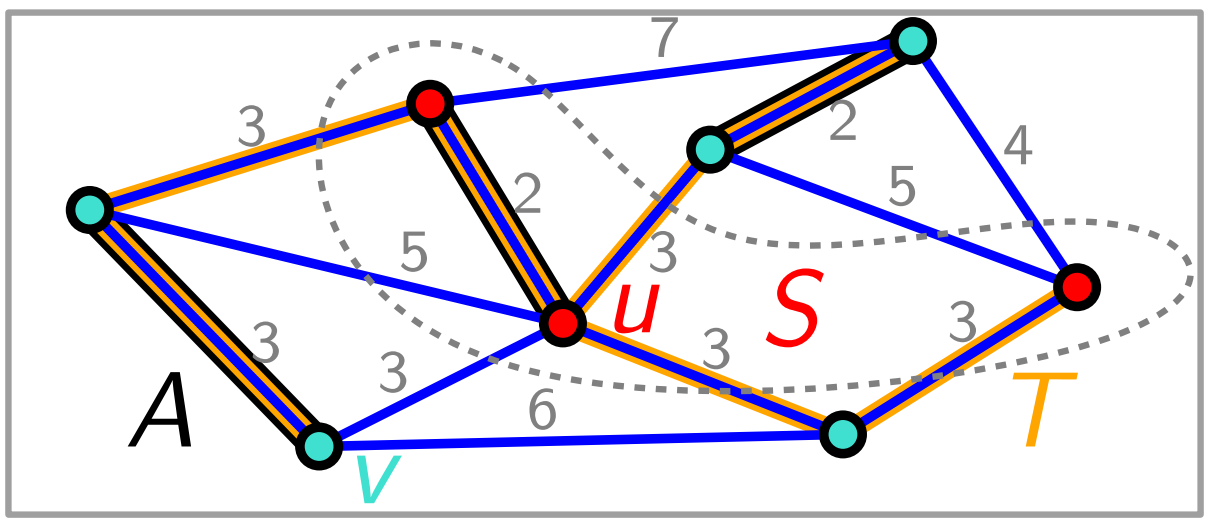
Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.
Falls $uv \in T$, fertig. Also $uv \notin T$.



Beweis

Satz. ... Dann ist uv sicher für A.

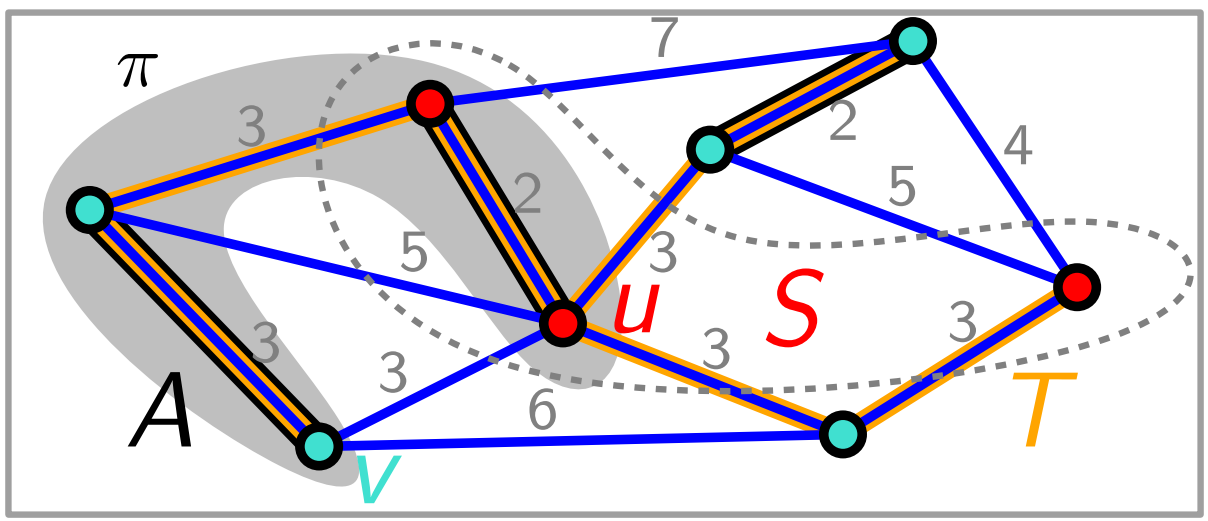
Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.
Falls $uv \in T$, fertig. Also $uv \notin T$. Sei π u-v-Pfad in T.



Beweis

Satz. ... Dann ist uv sicher für A.

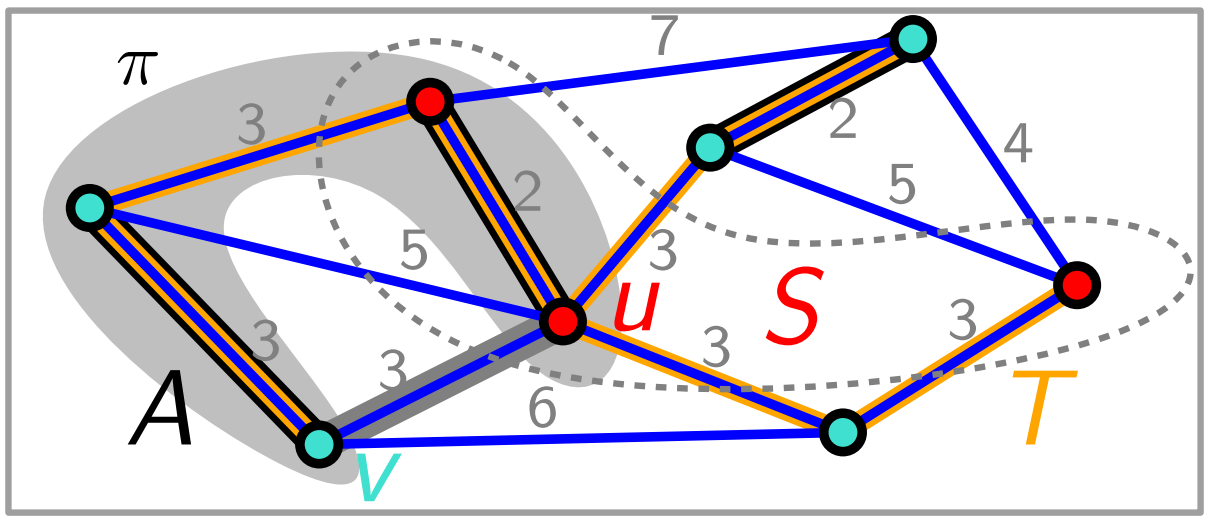
Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.
Falls $uv \in T$, fertig. Also $uv \notin T$. Sei π u-v-Pfad in T.



Beweis

Satz. ... Dann ist uv sicher für A.

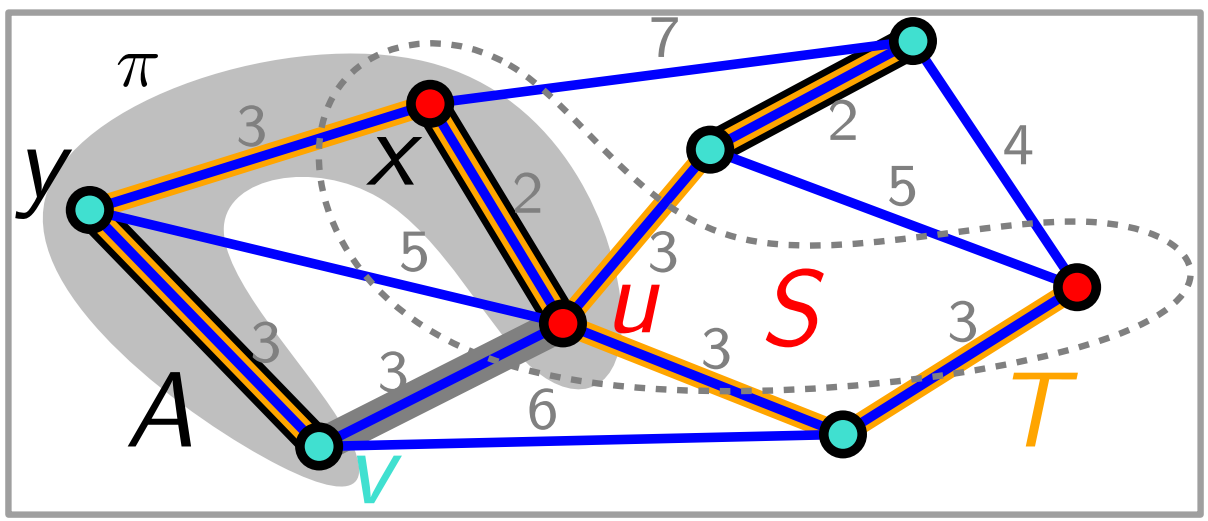
Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.
Falls $uv \in T$, fertig. Also $uv \notin T$. Sei π u-v-Pfad in T.
$\Rightarrow \pi + uv$ ist Kreis (wobei uv $(S, V \setminus S)$ kreuzt)



Beweis

Satz. ... Dann ist uv sicher für A.

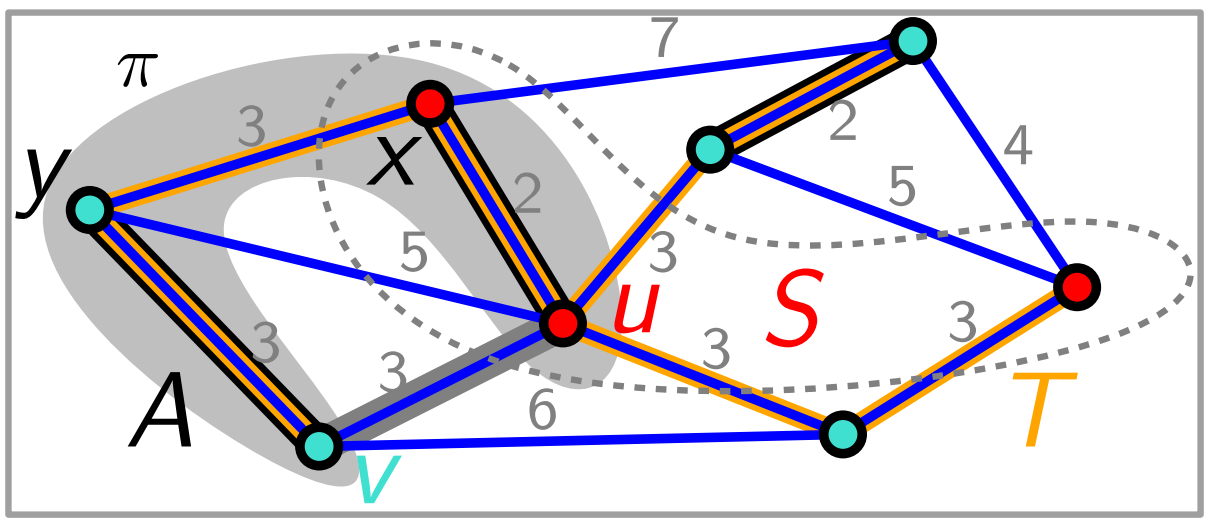
Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.
Falls $uv \in T$, fertig. Also $uv \notin T$. Sei π u-v-Pfad in T.
$\Rightarrow \pi + uv$ ist Kreis (wobei uv $(S, V \setminus S)$ kreuzt)
$\Rightarrow$ Kreis enthält zweite Kante xy, die $(S, V \setminus S)$ kreuzt.



Beweis

Satz. ... Dann ist uv sicher für A.

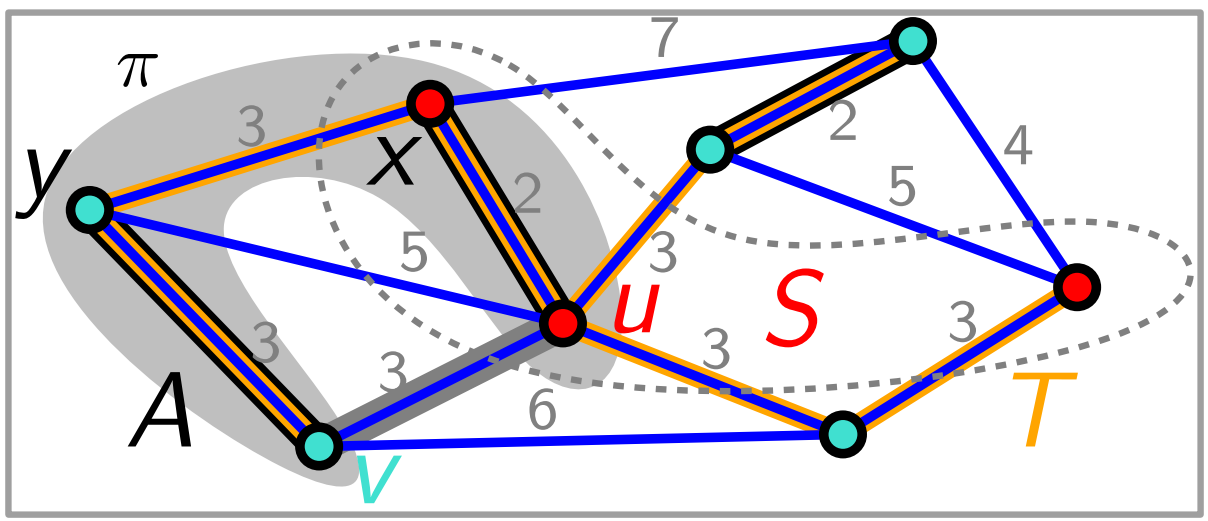
Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.
Falls $uv \in T$, fertig. Also $uv \notin T$. Sei π u-v-Pfad in T.
$\Rightarrow \pi + uv$ ist Kreis (wobei uv $(S, V \setminus S)$ kreuzt)
$\Rightarrow$ Kreis enthält zweite Kante xy, die $(S, V \setminus S)$ kreuzt.
$\Rightarrow T'$ ist auch Spannbaum von G.



Beweis

Satz. ... Dann ist uv sicher für A.

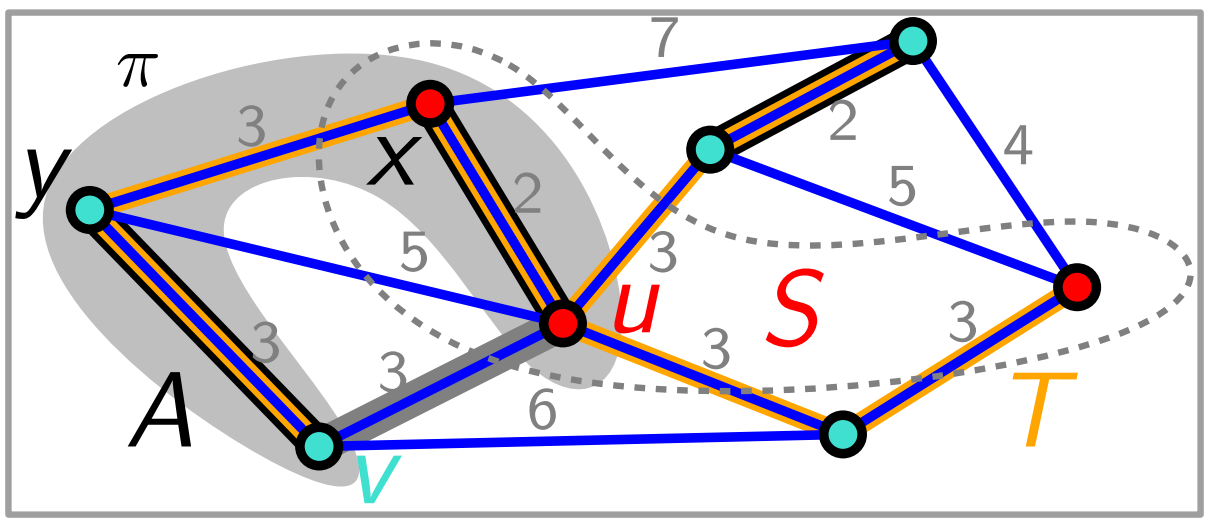
Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.
Falls $uv \in T$, fertig. Also $uv \notin T$. Sei π u-v-Pfad in T.
$\Rightarrow \pi + uv$ ist Kreis (wobei uv $(S, V \setminus S)$ kreuzt)
$\Rightarrow$ Kreis enthält zweite Kante xy, die $(S, V \setminus S)$ kreuzt.
$\Rightarrow T' = (T \cup \{uv\}) \setminus \{xy\}$ ist auch Spannbaum von G.



Beweis

Satz. ... Dann ist uv sicher für A.

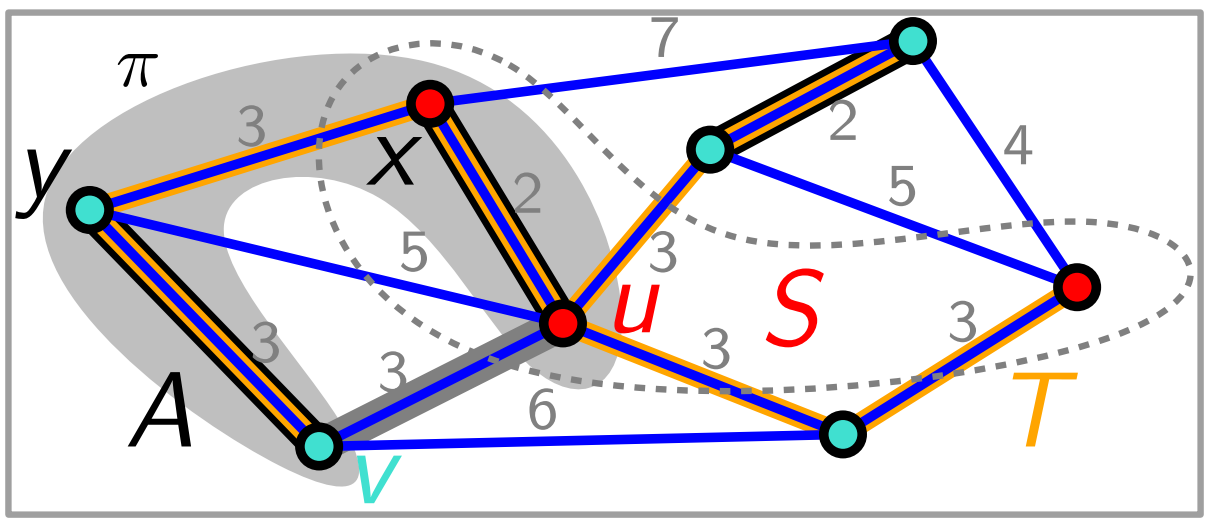
Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.
Falls $uv \in T$, fertig. Also $uv \notin T$. Sei π u-v-Pfad in T.
$\Rightarrow \pi + uv$ ist Kreis (wobei uv $(S, V \setminus S)$ kreuzt)
$\Rightarrow$ Kreis enthält zweite Kante xy, die $(S, V \setminus S)$ kreuzt.
$\Rightarrow T' = (T \cup \{uv\}) \setminus \{xy\}$ ist auch Spannbaum von G.
$w(T') = w(T) + w(uv) - w(xy)$



Beweis

Satz. ... Dann ist uv sicher für A.

Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.
Falls $uv \in T$, fertig. Also $uv \notin T$. Sei π u-v-Pfad in T.
$\Rightarrow \pi + uv$ ist Kreis (wobei uv $(S, V \setminus S)$ kreuzt)
$\Rightarrow$ Kreis enthält zweite Kante xy, die $(S, V \setminus S)$ kreuzt.
$\Rightarrow T' = (T \cup \{uv\}) \setminus \{xy\}$ ist auch Spannbaum von G.
$w(T') = w(T) + \underbrace{w(uv) - w(xy)}$

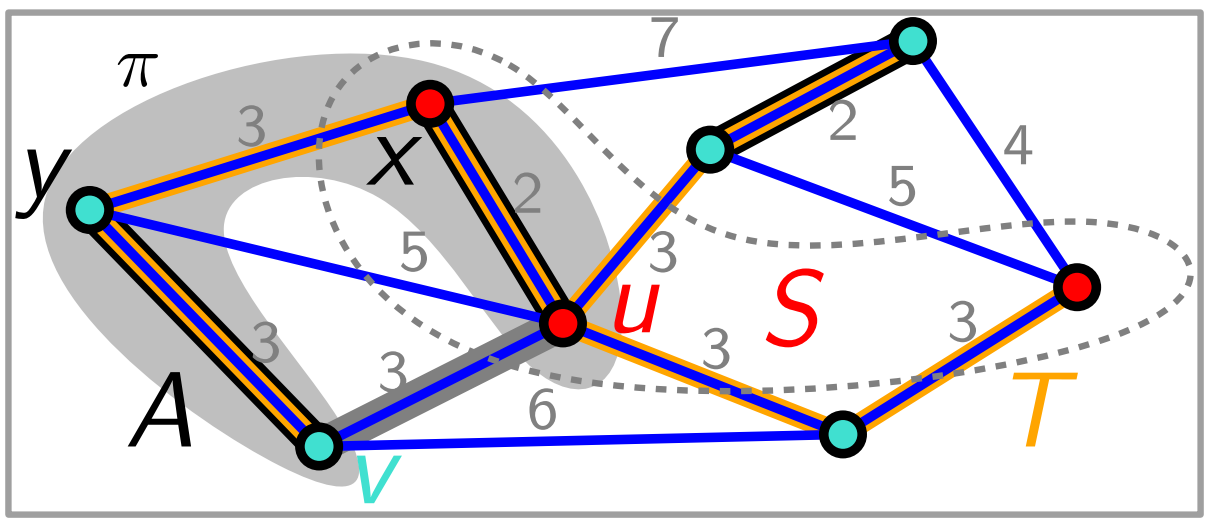


Beweis

Satz. ... Dann ist uv sicher für A.

Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.
Falls $uv \in T$, fertig. Also $uv \notin T$. Sei π u-v-Pfad in T.
$\Rightarrow \pi + uv$ ist Kreis (wobei uv $(S, V \setminus S)$ kreuzt)
$\Rightarrow$ Kreis enthält zweite Kante xy, die $(S, V \setminus S)$ kreuzt.
$\Rightarrow T' = (T \cup \{uv\}) \setminus \{xy\}$ ist auch Spannbaum von G.

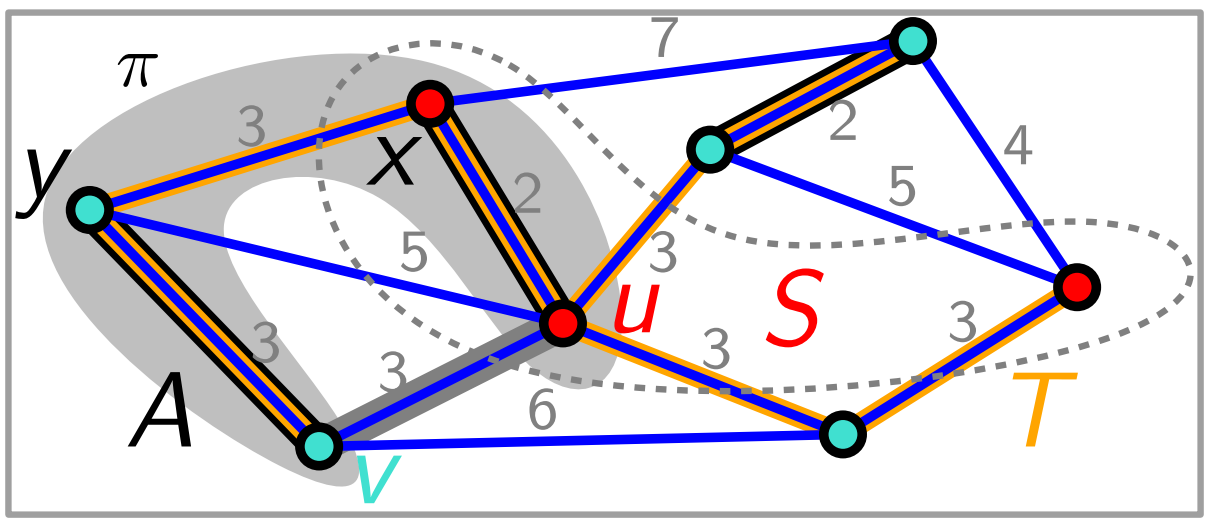
$$w(T') = w(T) + \underbrace{w(uv) - w(xy)}_{\leq 0}$$



Beweis

Satz. ... Dann ist uv sicher für A.

Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.
Falls $uv \in T$, fertig. Also $uv \notin T$. Sei π u-v-Pfad in T.
$\Rightarrow \pi + uv$ ist Kreis (wobei uv $(S, V \setminus S)$ kreuzt)
$\Rightarrow$ Kreis enthält zweite Kante xy, die $(S, V \setminus S)$ kreuzt.
$\Rightarrow T' = (T \cup \{uv\}) \setminus \{xy\}$ ist auch Spannbaum von G.
$w(T') = w(T) + \underbrace{w(uv) - w(xy)}_{\leq 0, \text{ da } uv \text{ leicht bzgl. } (S, V \setminus S)}$

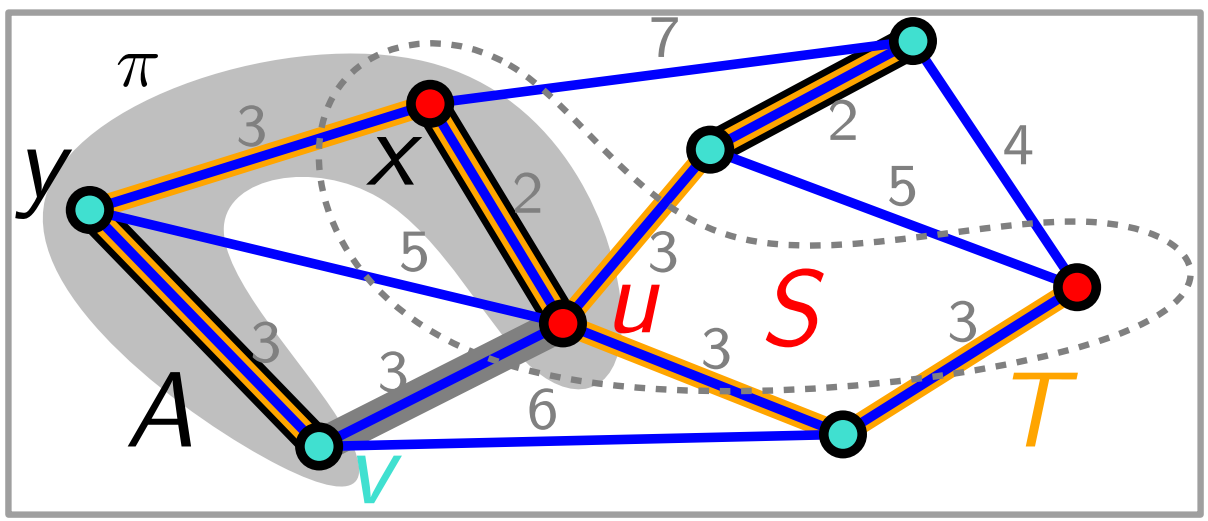


Beweis

Satz. ... Dann ist uv sicher für A.

Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.
Falls $uv \in T$, fertig. Also $uv \notin T$. Sei π u-v-Pfad in T.
$\Rightarrow \pi + uv$ ist Kreis (wobei uv $(S, V \setminus S)$ kreuzt)
$\Rightarrow$ Kreis enthält zweite Kante xy, die $(S, V \setminus S)$ kreuzt.
$\Rightarrow T' = (T \cup \{uv\}) \setminus \{xy\}$ ist auch Spannbaum von G.

$$w(T') = w(T) + \underbrace{w(uv) - w(xy)}_{\leq 0, \text{ da } uv \text{ leicht bzgl. } (S, V \setminus S)} \leq w(T)$$

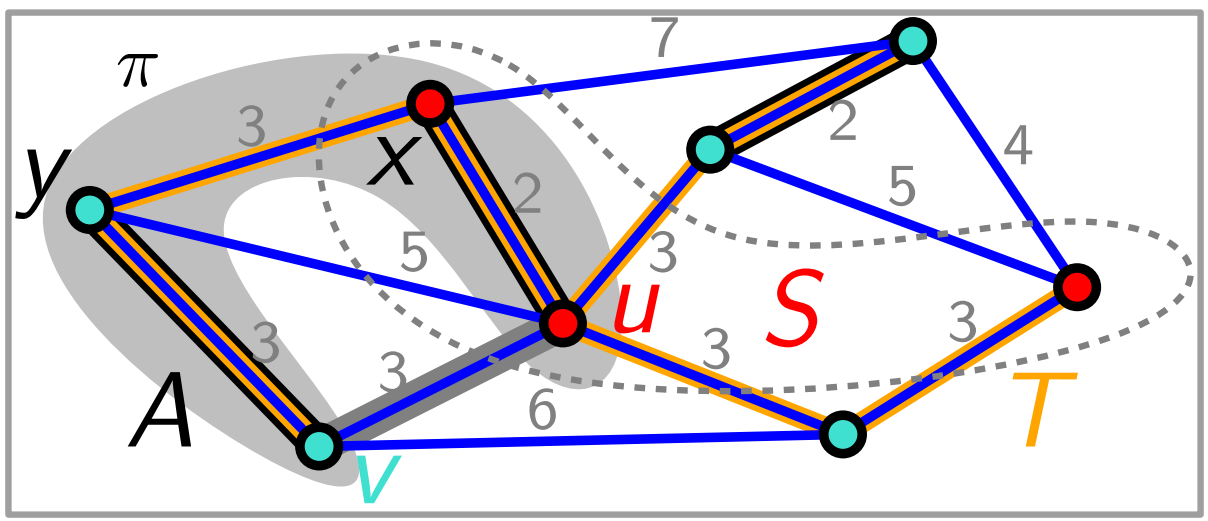


Beweis

Satz. ... Dann ist uv sicher für A.

Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.
Falls $uv \in T$, fertig. Also $uv \notin T$. Sei π u-v-Pfad in T.
$\Rightarrow \pi + uv$ ist Kreis (wobei uv $(S, V \setminus S)$ kreuzt)
$\Rightarrow$ Kreis enthält zweite Kante xy, die $(S, V \setminus S)$ kreuzt.
$\Rightarrow T' = (T \cup \{uv\}) \setminus \{xy\}$ ist auch Spannbaum von G.

$$w(T') = w(T) + \underbrace{w(uv) - w(xy)}_{\leq 0, \text{ da } uv \text{ leicht bzgl. } (S, V \setminus S)} \leq w(T)$$



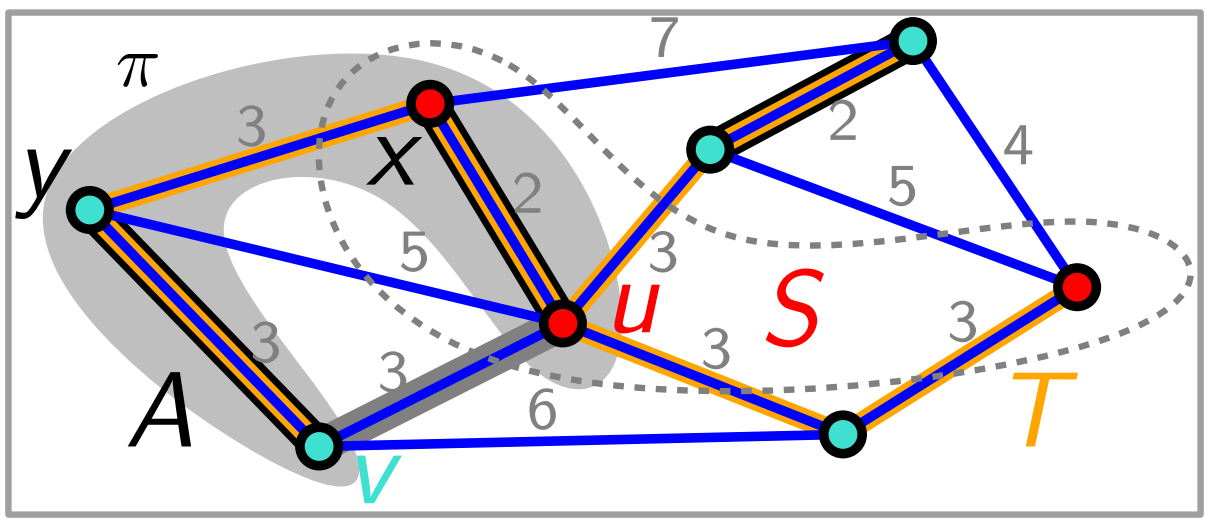
$\Rightarrow$ T' ist *minimaler* Spannbaum von G.

Beweis

Satz. ... Dann ist uv sicher für A.

Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.
Falls $uv \in T$, fertig. Also $uv \notin T$. Sei π u-v-Pfad in T.
$\Rightarrow \pi + uv$ ist Kreis (wobei uv $(S, V \setminus S)$ kreuzt)
$\Rightarrow$ Kreis enthält zweite Kante xy, die $(S, V \setminus S)$ kreuzt.
$\Rightarrow T' = (T \cup \{uv\}) \setminus \{xy\}$ ist auch Spannbaum von G.

$$w(T') = w(T) + \underbrace{w(uv) - w(xy)}_{\leq 0, \text{ da } uv \text{ leicht bzgl. } (S, V \setminus S)} \leq w(T)$$



$\Rightarrow T'$ ist *minimaler* Spannbaum von G.

Und: $A \cup \{uv\} \subseteq T'$.

Beweis

Satz. ... Dann ist uv sicher für A.

Beweis. *Zeige:* G hat min. Spannbaum, der $A \cup \{uv\}$ enthält.

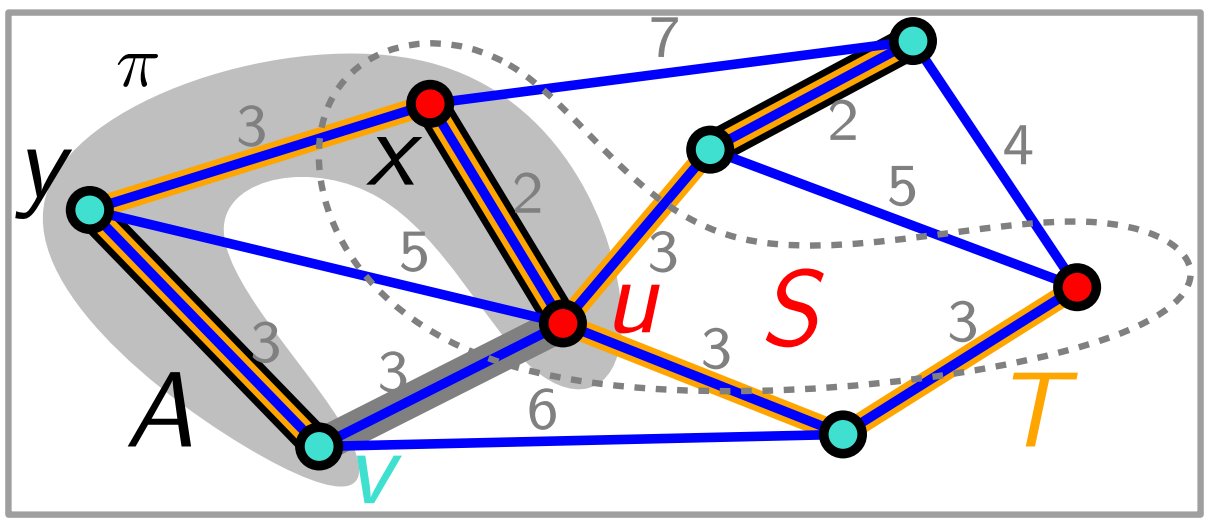
Falls $uv \in T$, fertig. Also $uv \notin T$. Sei π u-v-Pfad in T.

$\Rightarrow \pi + uv$ ist Kreis (wobei uv $(S, V \setminus S)$ kreuzt)

$\Rightarrow$ Kreis enthält zweite Kante xy, die $(S, V \setminus S)$ kreuzt.

$\Rightarrow T' = (T \cup \{uv\}) \setminus \{xy\}$ ist auch Spannbaum von G.

$$w(T') = w(T) + \underbrace{w(uv) - w(xy)}_{\leq 0, \text{ da } uv \text{ leicht bzgl. } (S, V \setminus S)} \leq w(T)$$



$\Rightarrow T'$ ist *minimaler* Spannbaum von G.

Und: $A \cup \{uv\} \subseteq T'$.

$\Rightarrow uv$ ist sicher für A. $\square$

Zurück zum Algorithmus

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$

while $|A| < |V| - 1$ **do**

    // *INV:* $A \subseteq$ min. Spannbaum von G

    finde Kante uv, die *sicher* für A ist

    $A = A \cup \{uv\}$

return A

Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

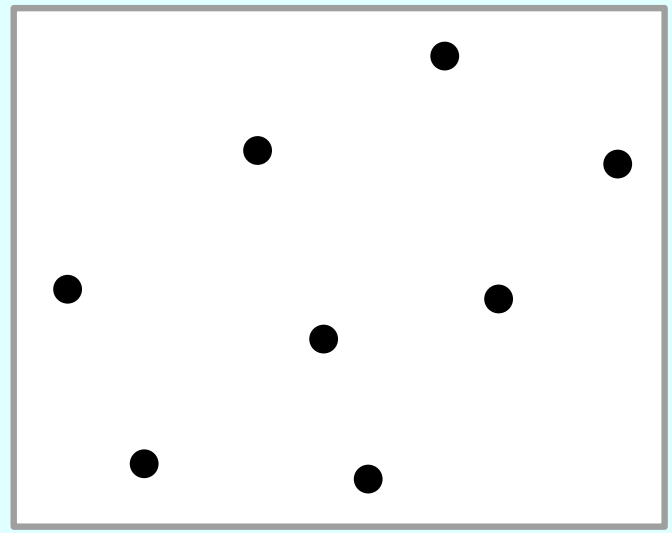
```
GenericMST(UndirectedConnectedGraph G, EdgeWeights w)
  A = ∅
  while |A| < |V| − 1 do
      // INV: A ⊆ min. Spannbaum von G
      finde Kante uv, die sicher für A ist
      A = A ∪ {uv}
  return A
```

Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

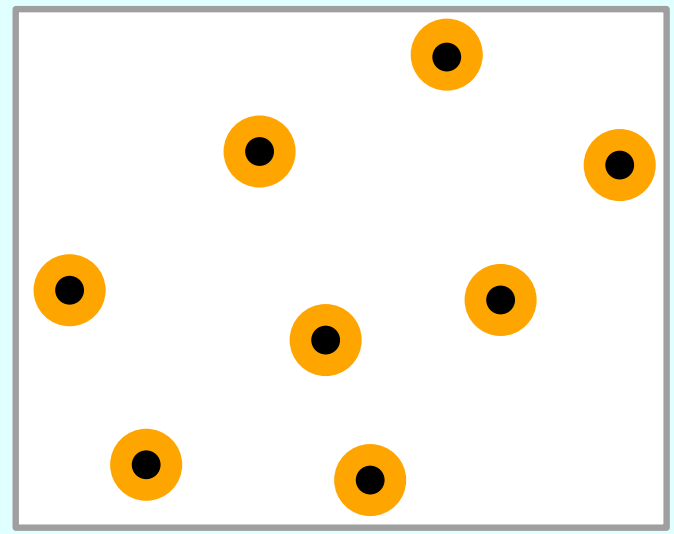


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

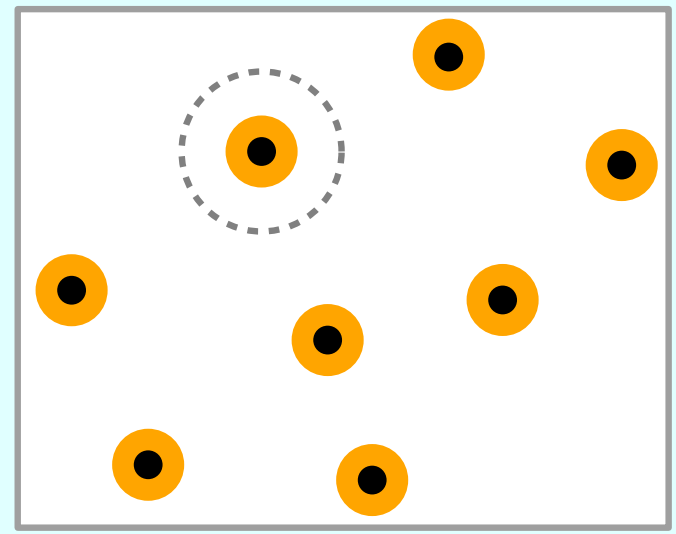


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

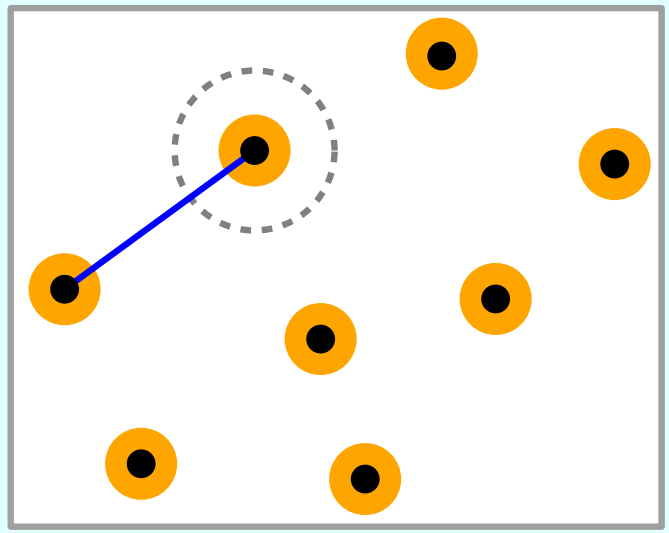


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

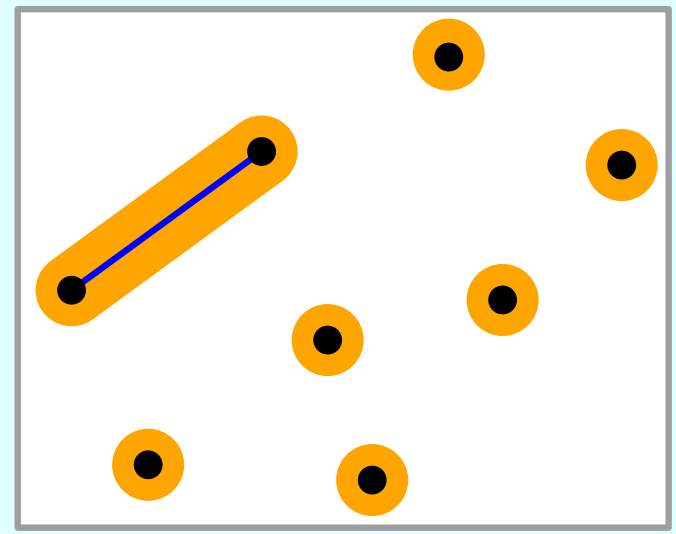


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

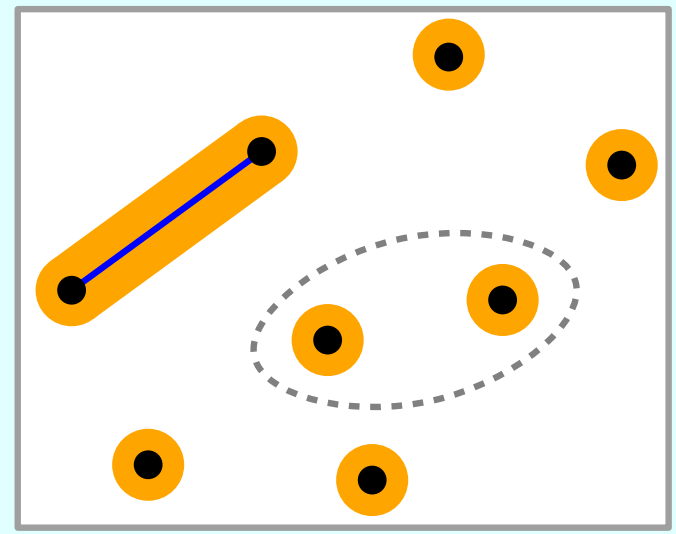


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

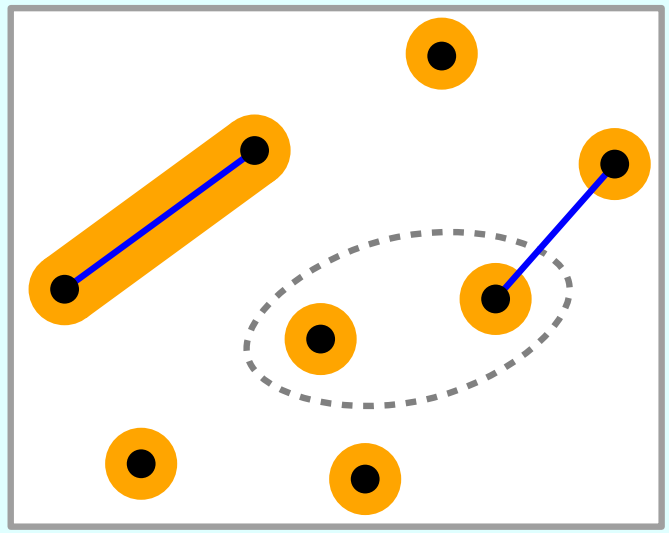


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

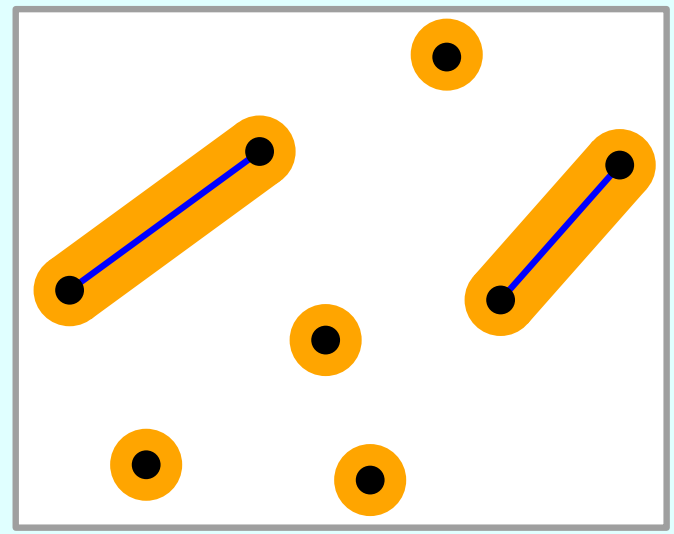


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

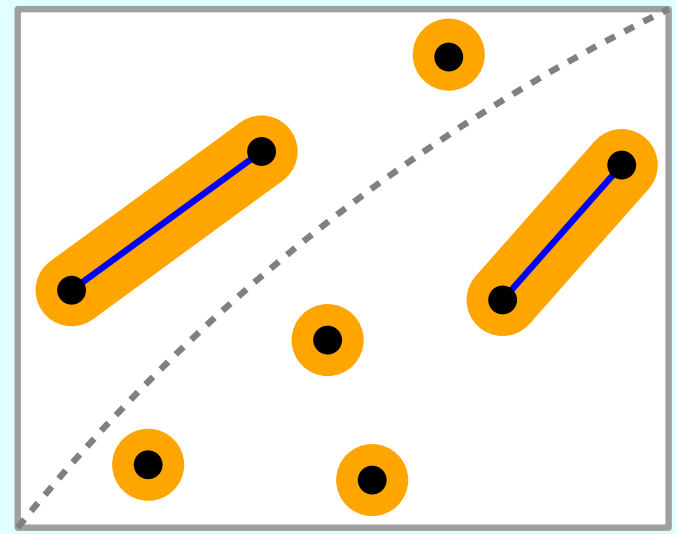


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

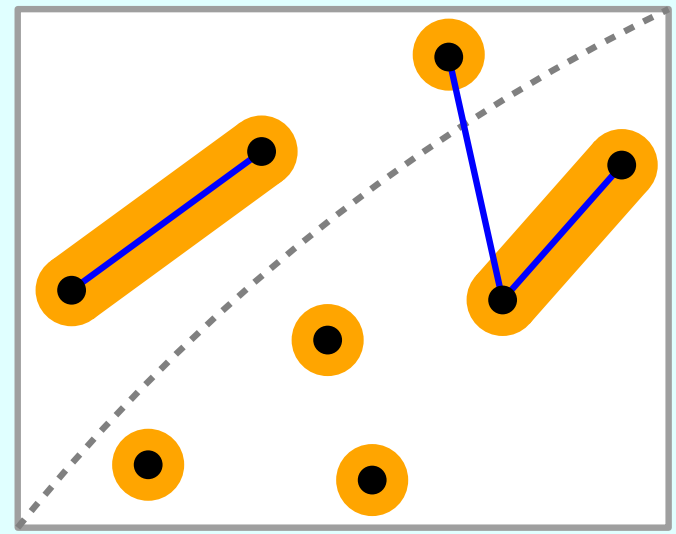


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

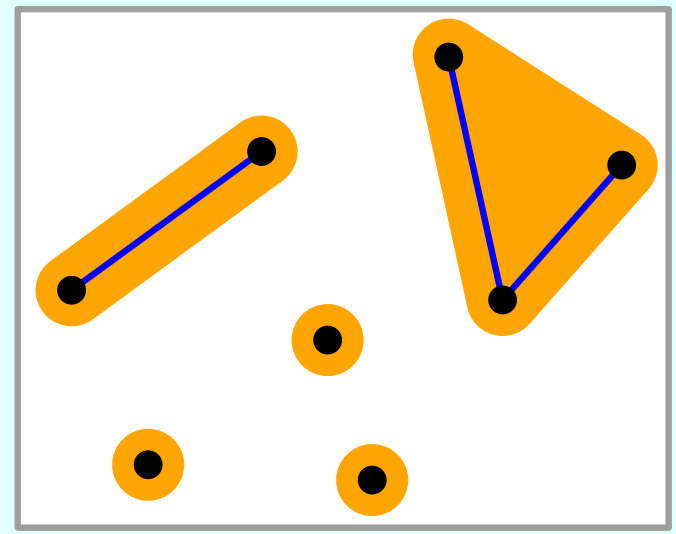


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

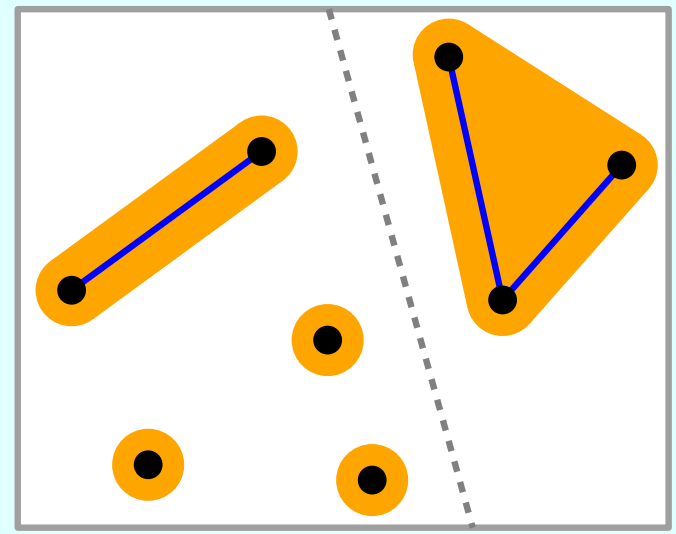


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

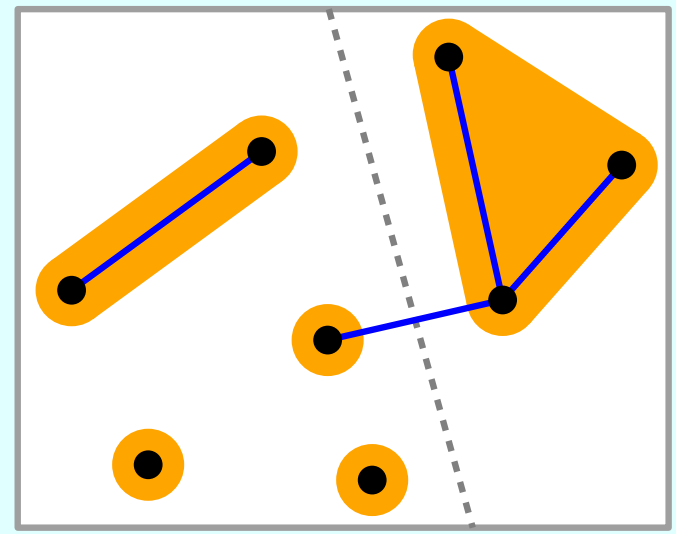


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

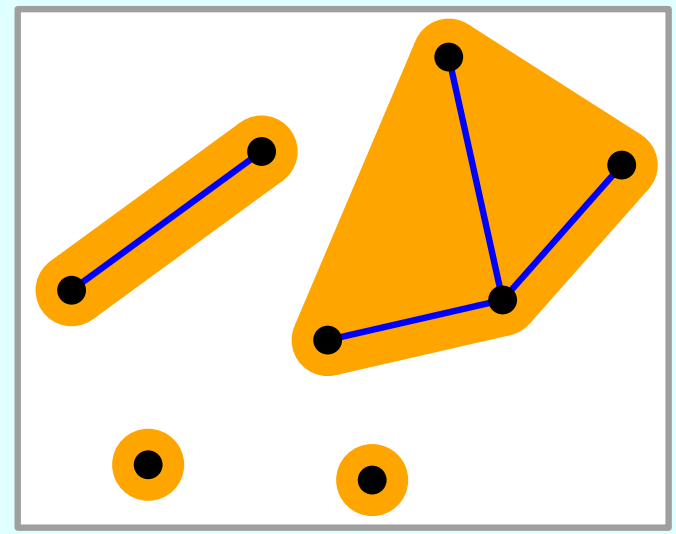


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

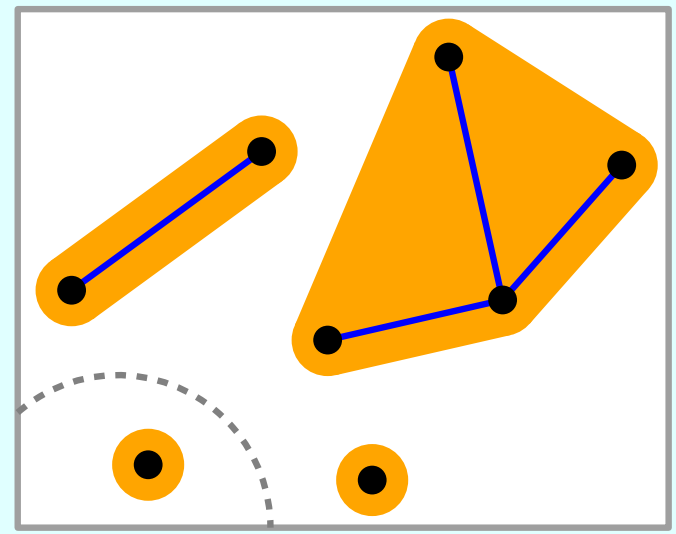


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

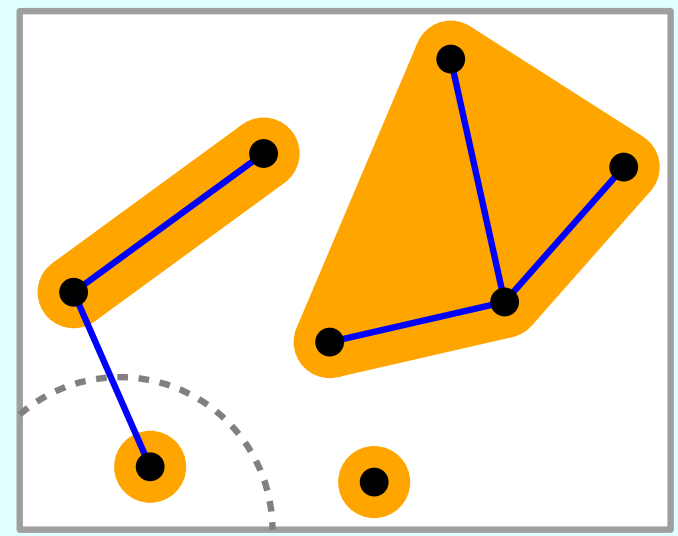


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

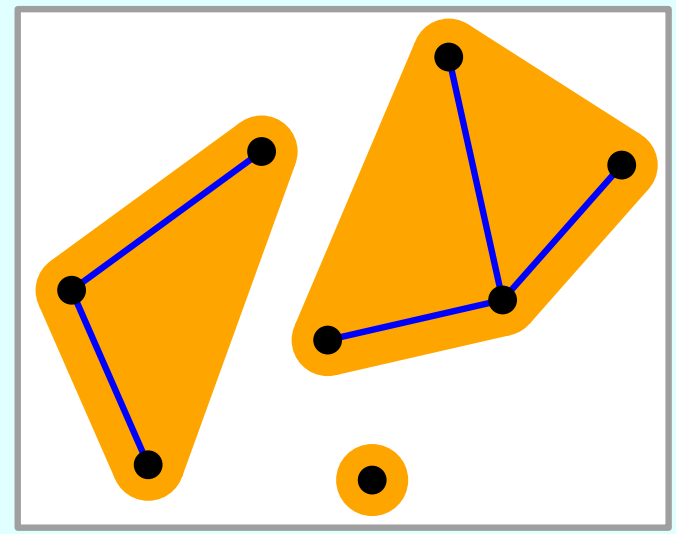


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

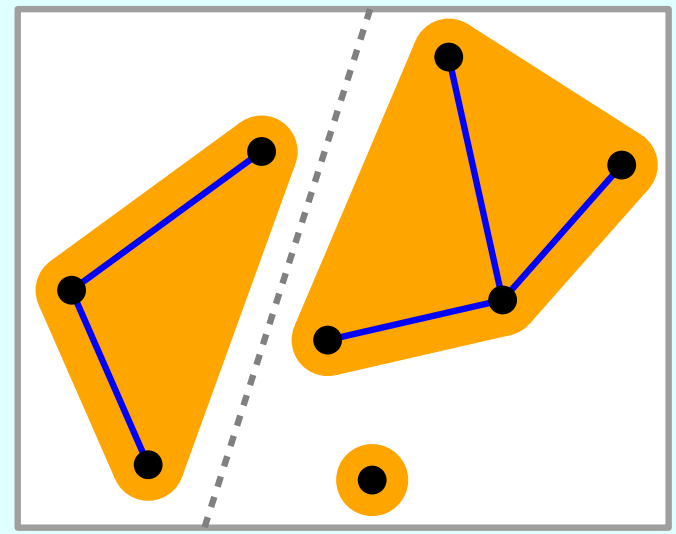


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

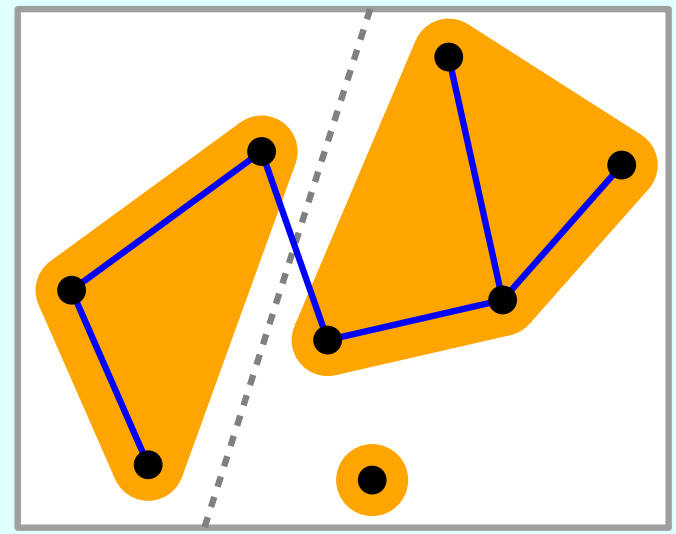


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

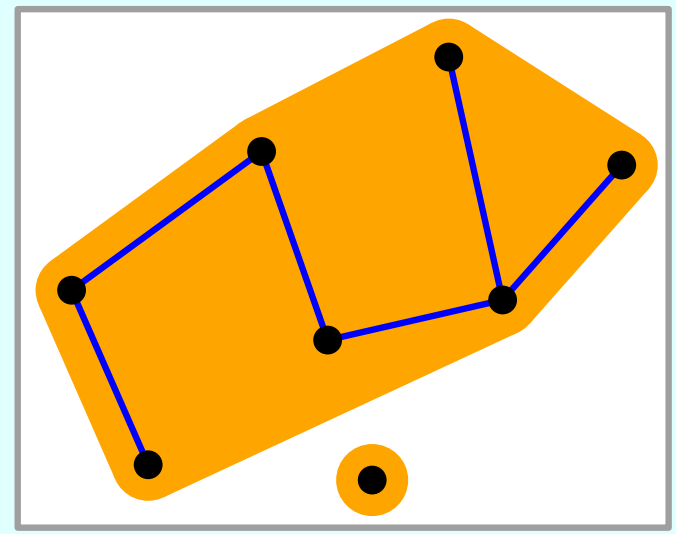


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

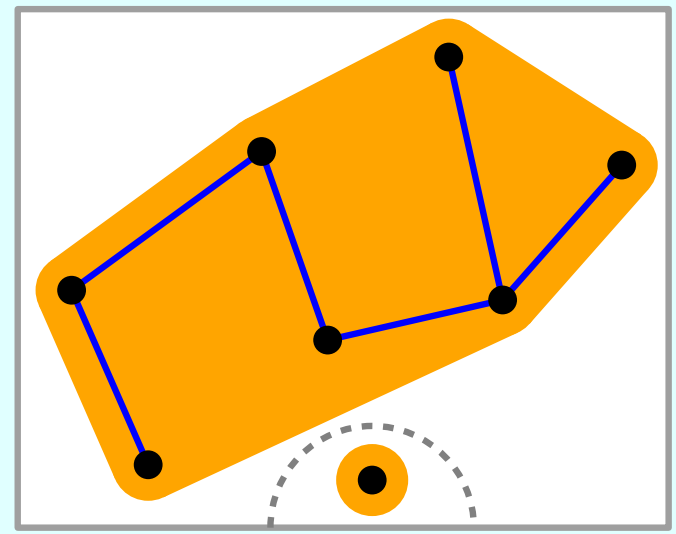


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

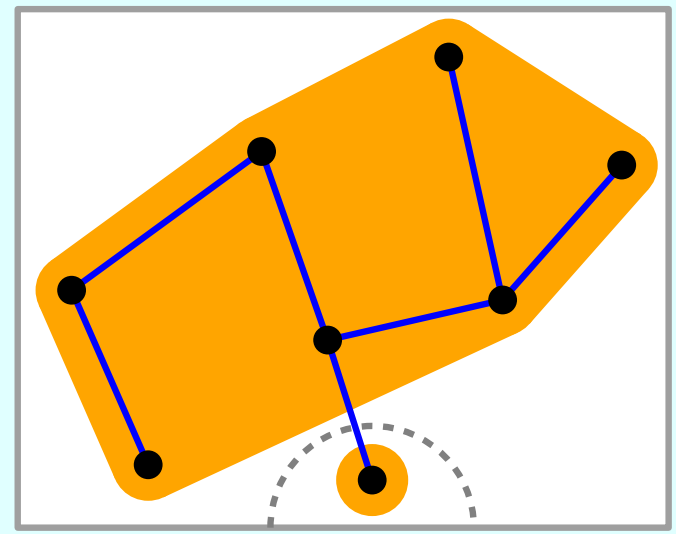


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A

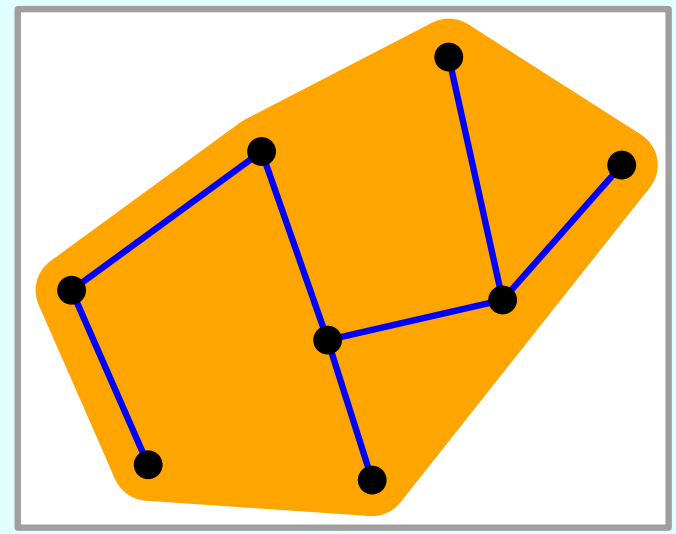


Zurück zum Algorithmus

Satz. Sei $G = (V, E; w)$ ein zshg., gewichteter, unger. Graph.
Sei T Kantenmenge eines min. Spannbaums von G.
Sei A Teilmenge von T.
Sei $(S, V \setminus S)$ ein Schnitt, der A respektiert.
Sei $uv \in E$ leicht bzgl. $(S, V \setminus S)$.
Dann ist uv sicher für A.

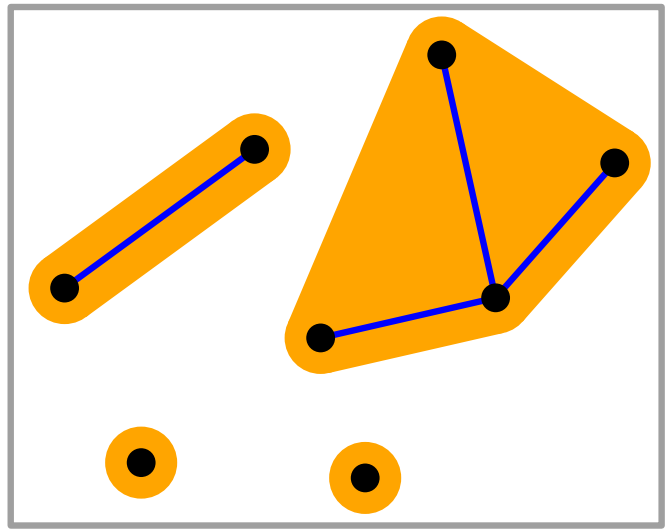
GenericMST(UndirectedConnectedGraph G, EdgeWeights w)

$A = \emptyset$
while $|A| < |V| - 1$ **do**
  // *INV:* $A \subseteq$ min. Spannbaum von G
  finde Kante uv, die *sicher* für A ist
  $A = A \cup \{uv\}$
return A



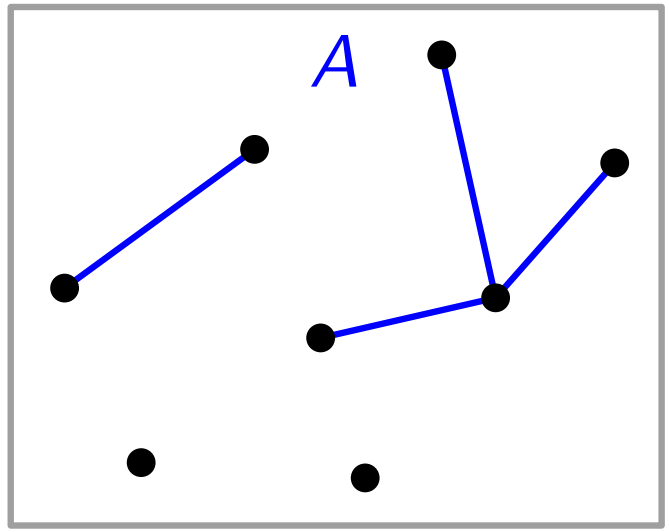
Zusammenhangskomponenten

Def. Eine *Zusammenhangskomponente* eines Graphen ist ein Teilgraph, der von einer nicht vergrößerbaren („*inklusionsmaximalen*“) zusammenhängenden Menge von Knoten *induziert* wird.



Zusammenhangskomponenten

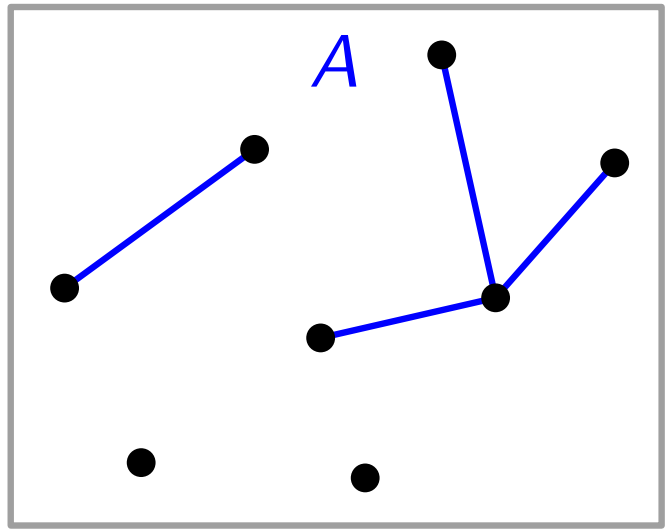
Def. Eine *Zusammenhangskomponente* eines Graphen ist ein Teilgraph, der von einer nicht vergrößerbaren („*inklusionsmaximalen*“) zusammenhängenden Menge von Knoten *induziert* wird.



Korollar. $G = (V, E)$ wie gehabt.
$A \subseteq E$ in einem min. Spannbaum von G enthalten.

Zusammenhangskomponenten

Def. Eine *Zusammenhangskomponente* eines Graphen ist ein Teilgraph, der von einer nicht vergrößerbaren („*inklusionsmaximalen*“) zusammenhängenden Menge von Knoten *induziert* wird.



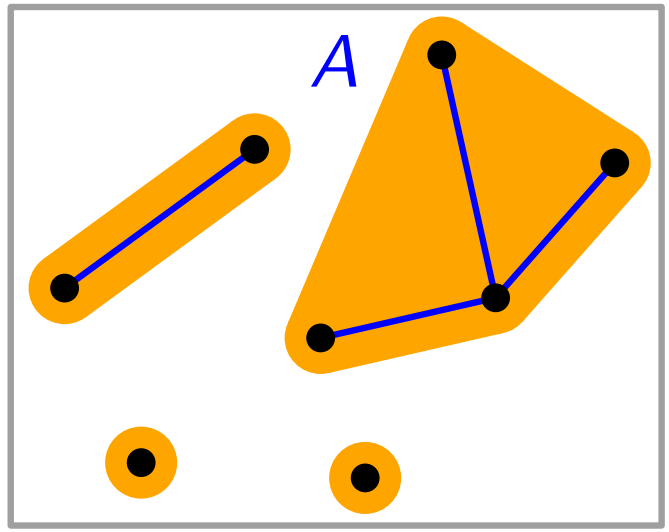
G_A

Korollar. $G = (V, E)$ wie gehabt.
$A \subseteq E$ in einem min. Spannbaum von G enthalten.

Wald $G_A = (V, A)$

Zusammenhangskomponenten

Def. Eine *Zusammenhangskomponente* eines Graphen ist ein Teilgraph, der von einer nicht vergrößerbaren („*inklusionsmaximalen*“) zusammenhängenden Menge von Knoten *induziert* wird.

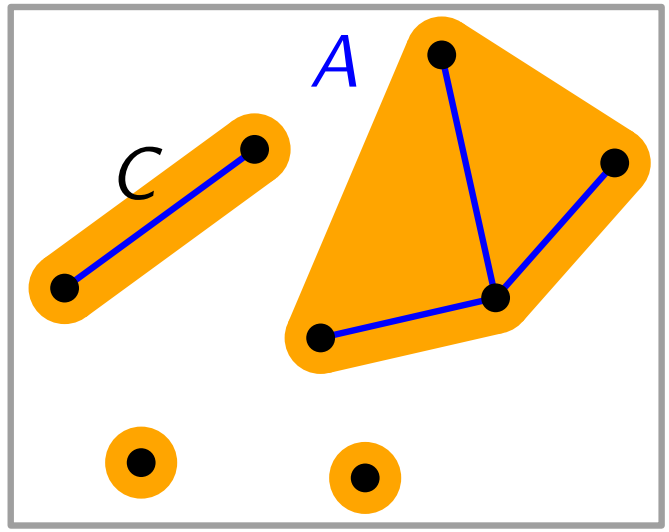


G_A

Korollar. $G = (V, E)$ wie gehabt.
$A \subseteq E$ in einem min. Spannbaum von G enthalten.
Wald $G_A = (V, A)$

Zusammenhangskomponenten

Def. Eine *Zusammenhangskomponente* eines Graphen ist ein Teilgraph, der von einer nicht vergrößerbaren („*inklusionsmaximalen*“) zusammenhängenden Menge von Knoten *induziert* wird.

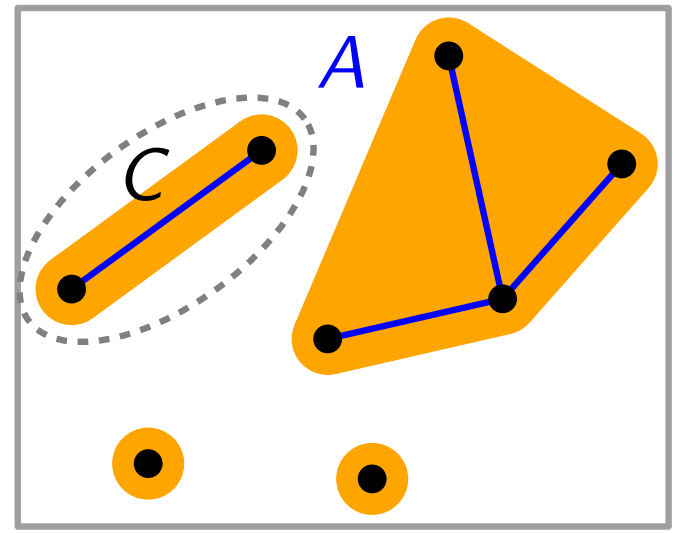


G_A

Korollar. $G = (V, E)$ wie gehabt.
$A \subseteq E$ in einem min. Spannbaum von G enthalten.
$C = (V_C, E_C)$ Zshgskomp. des Waldes $G_A = (V, A)$.

Zusammenhangskomponenten

Def. Eine *Zusammenhangskomponente* eines Graphen ist ein Teilgraph, der von einer nicht vergrößerbaren („*inklusionsmaximalen*“) zusammenhängenden Menge von Knoten *induziert* wird.



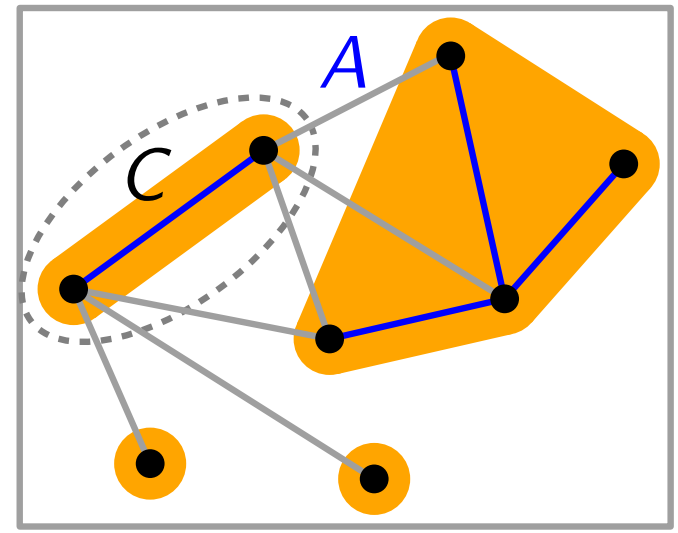
G_A

Korollar. $G = (V, E)$ wie gehabt.
$A \subseteq E$ in einem min. Spannbaum von G enthalten.
$C = (V_C, E_C)$ Zshgskomp. des Waldes $G_A = (V, A)$.

$(V_C, V \setminus V_C)$

Zusammenhangskomponenten

Def. Eine *Zusammenhangskomponente* eines Graphen ist ein Teilgraph, der von einer nicht vergrößerbaren („*inklusionsmaximalen*“) zusammenhängenden Menge von Knoten *induziert* wird.



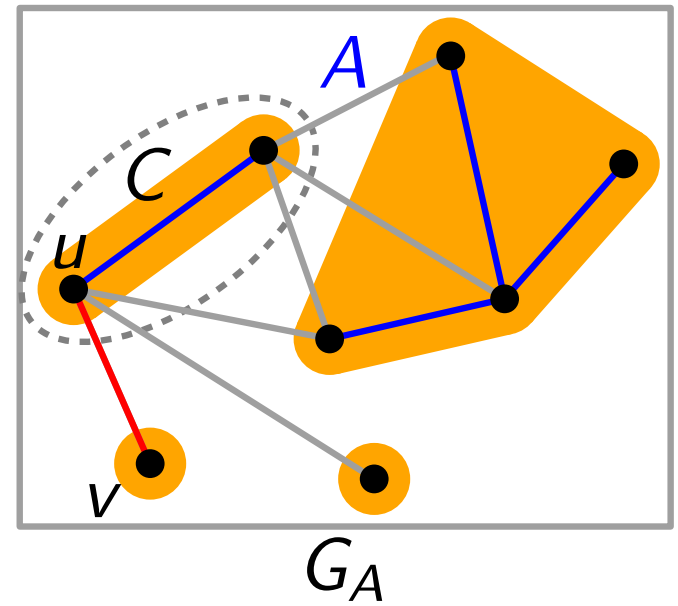
G_A

Korollar. $G = (V, E)$ wie gehabt.
$A \subseteq E$ in einem min. Spannbaum von G enthalten.
$C = (V_C, E_C)$ Zshgskomp. des Waldes $G_A = (V, A)$.

$(V_C, V \setminus V_C)$

Zusammenhangskomponenten

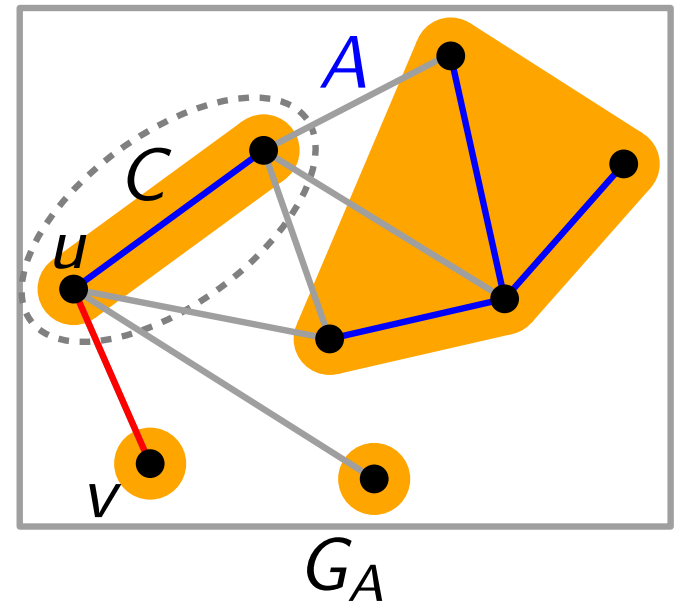
Def. Eine *Zusammenhangskomponente* eines Graphen ist ein Teilgraph, der von einer nicht vergrößerbaren („*inklusionsmaximalen*“) zusammenhängenden Menge von Knoten *induziert* wird.



Korollar. $G = (V, E)$ wie gehabt.
$A \subseteq E$ in einem min. Spannbaum von G enthalten.
$C = (V_C, E_C)$ Zshgskomp. des Waldes $G_A = (V, A)$.
uv leicht bzgl. $(V_C, V \setminus V_C)$

Zusammenhangskomponenten

Def. Eine *Zusammenhangskomponente* eines Graphen ist ein Teilgraph, der von einer nicht vergrößerbaren („*inklusionsmaximalen*“) zusammenhängenden Menge von Knoten *induziert* wird.

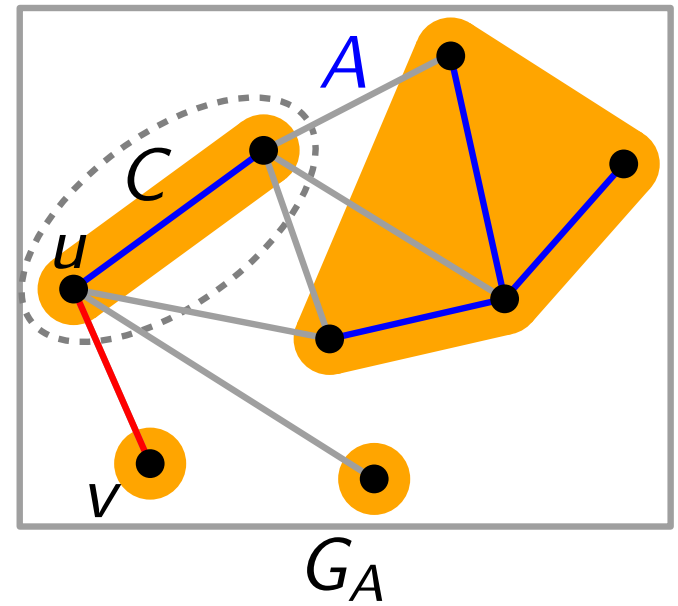


Korollar. $G = (V, E)$ wie gehabt.
$A \subseteq E$ in einem min. Spannbaum von G enthalten.
$C = (V_C, E_C)$ Zshgskomp. des Waldes $G_A = (V, A)$.
uv leicht bzgl. $(V_C, V \setminus V_C)$

Dann gilt:

Zusammenhangskomponenten

Def. Eine *Zusammenhangskomponente* eines Graphen ist ein Teilgraph, der von einer nicht vergrößerbaren („*inklusionsmaximalen*“) zusammenhängenden Menge von Knoten *induziert* wird.



Korollar. $G = (V, E)$ wie gehabt.
$A \subseteq E$ in einem min. Spannbaum von G enthalten.
$C = (V_C, E_C)$ Zshgskomp. des Waldes $G_A = (V, A)$.
uv leicht bzgl. $(V_C, V \setminus V_C)$

Dann gilt: uv ist sicher für A.

Der Algorithmus von Jarník-Prim (1930/1957)

```
Dijkstra(WeightedGraph G = (V, E; w), Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V, d)   // Gewichtung
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            Relax (u, v; w)
```

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)

```
Initialize(G, s)
Q = new PriorityQueue(V, d)   // Gewichtung
while not Q.Empty() do
    u = Q.ExtractMin()
    foreach v ∈ Adj[u] do
        Relax(u, v; w)
```

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)

```
Initialize(G, s)
Q = new PriorityQueue(V, d)   // Gewichtung
while not Q.Empty() do
    u = Q.ExtractMin()
    foreach v ∈ Adj[u] do
        Relax(u, v; w)
```

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)

```
Initialize(G, s)
Q = new PriorityQueue(V, d)   // Gewichtung
while not Q.Empty() do
    u = Q.ExtractMin()
    foreach v ∈ Adj[u] do
        Relax'(u, v; w)
```

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)

```
Initialize(G, s)
Q = new PriorityQueue(V, d)   // Gewichtung
while not Q.Empty() do
    u = Q.ExtractMin()
    foreach v ∈ Adj[u] do
        Relax′(u, v; w)
```

Relax $(u, v; w)$

```
if v.d > u.d + w(u, v) then
    v.d = u.d + w(u, v)
    v.π = u
    Q.DecreaseKey(v, v.d)
```

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  Q = **new** PriorityQueue(V, d) // Gewichtung
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in$ Adj[u] **do**
      Relax′($u, v; w$)

Relax′($u, v; w$)
  if $v.d > ~~u.d +~~ w(u, v)$ **then**
    $v.d = ~~u.d +~~ w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey($v, v.d$)

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  Q = **new** PriorityQueue(V, d) // Gewichtung
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in$ Adj$[u]$ **do**
      Relax′$(u, v; w)$

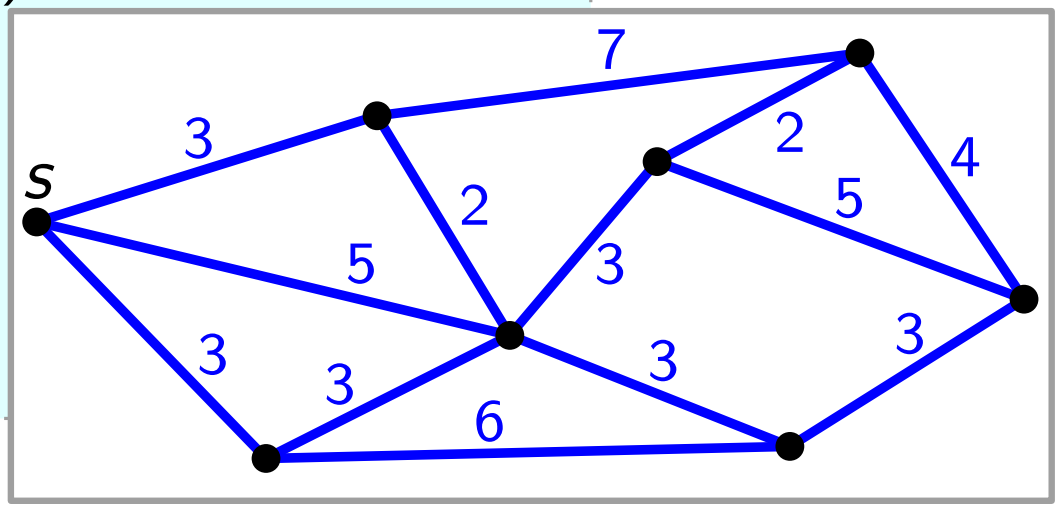
$v \in Q$ **and ...**

Relax′$(u, v; w)$
  if $v.d > ~~u.d +~~ w(u, v)$ **then**
    $v.d = ~~u.d +~~ w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey$(v, v.d)$

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  Q = **new** PriorityQueue(V, d) // Gewichtung
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in$ Adj[u] **do**
      Relax′($u, v; w$)



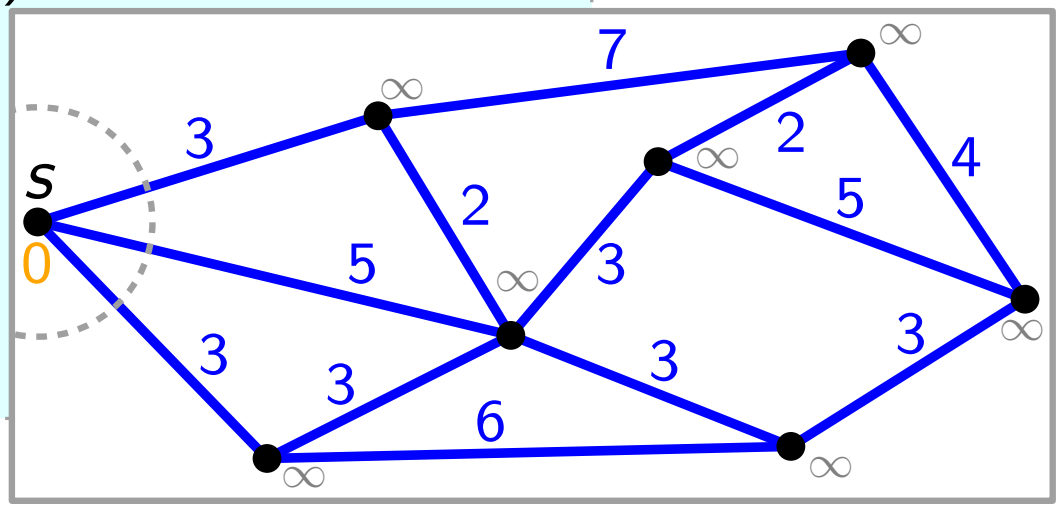
$v \in Q$ **and ...**

Relax′($u, v; w$)
  if $v.d > ~~u.d +~~ w(u, v)$ **then**
    $v.d = ~~u.d +~~ w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey($v, v.d$)

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  Q = **new** PriorityQueue(V, d) // Gewichtung
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in$ Adj[u] **do**
      Relax′($u, v; w$)



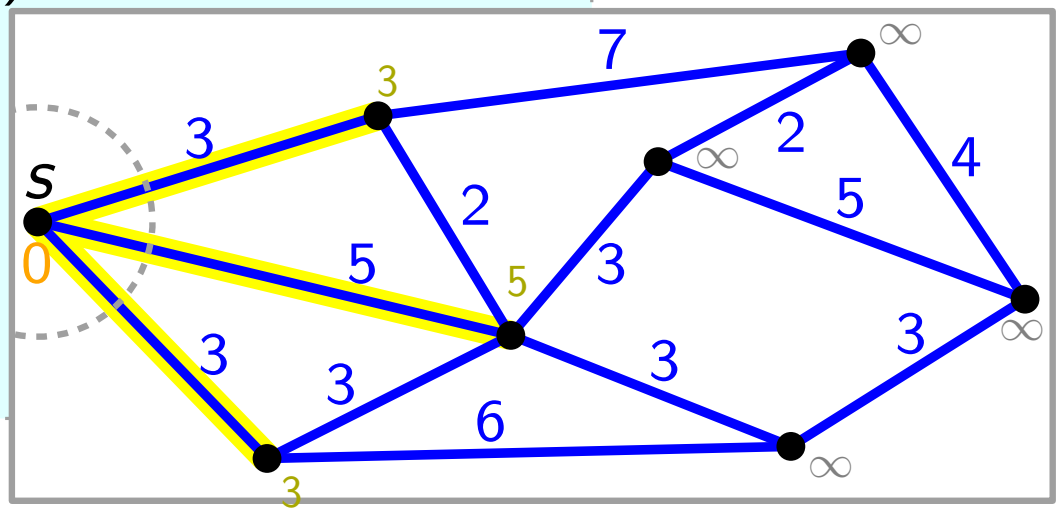
$v \in Q$ **and ...**

Relax′($u, v; w$)
  if $v.d > ~~u.d +~~ w(u, v)$ **then**
    $v.d = ~~u.d +~~ w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey($v, v.d$)

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  Q = **new** PriorityQueue(V, d) // Gewichtung
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in$ Adj$[u]$ **do**
      Relax′$(u, v; w)$



$v \in Q$ **and ...**

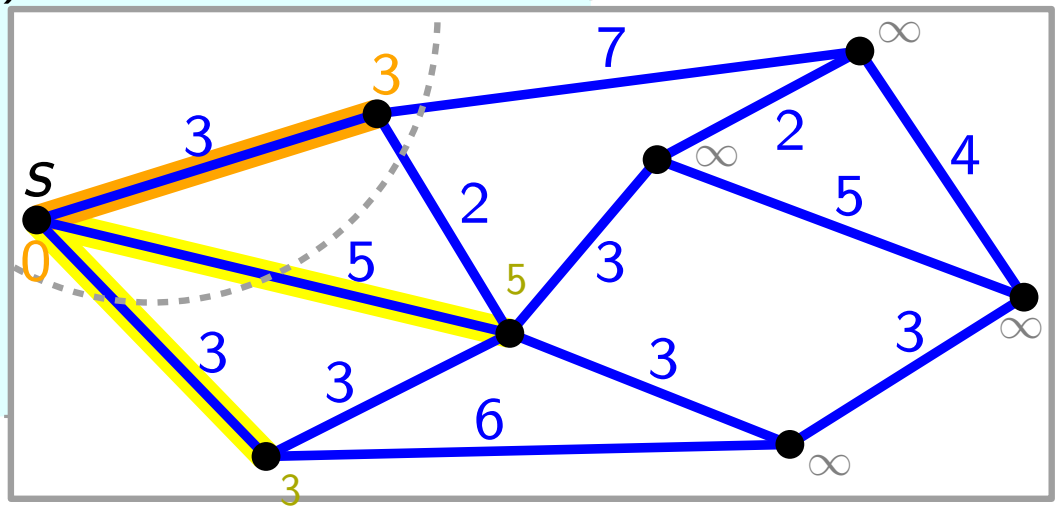
Relax′$(u, v; w)$
  if $v.d > ~~u.d +~~ w(u, v)$ **then**
    $v.d = ~~u.d +~~ w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey$(v, v.d)$

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)

```
Initialize(G, s)
Q = new PriorityQueue(V, d)   // Gewichtung
while not Q.Empty() do
    u = Q.ExtractMin()
    foreach v ∈ Adj[u] do
        Relax′(u, v; w)
```



$v \in Q$ **and ...**

Relax′$(u, v; w)$

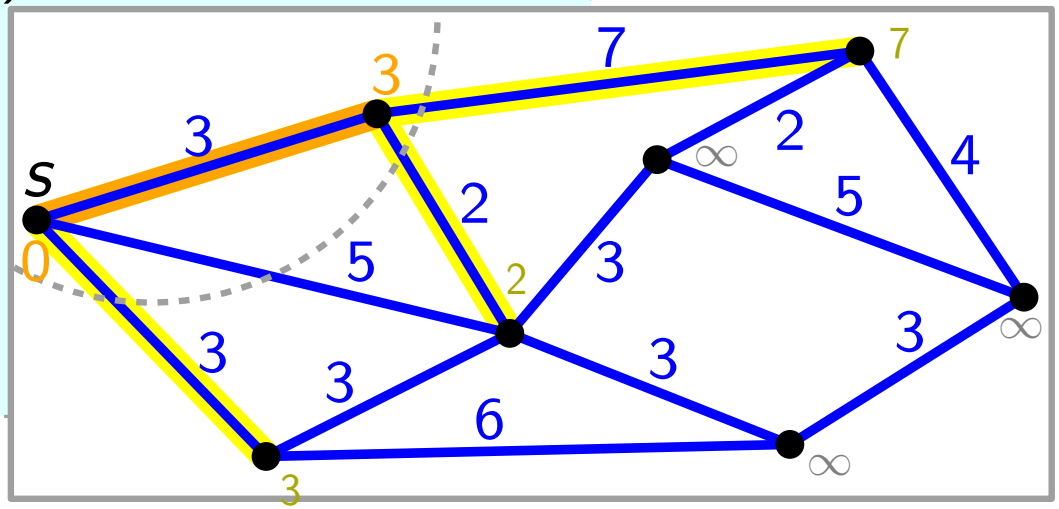
```
if v.d > ~~u.d +~~ w(u, v) then
    v.d = ~~u.d +~~ w(u, v)
    v.π = u
    Q.DecreaseKey(v, v.d)
```

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)

```
Initialize(G, s)
Q = new PriorityQueue(V, d)   // Gewichtung
while not Q.Empty() do
    u = Q.ExtractMin()
    foreach v ∈ Adj[u] do
        Relax′(u, v; w)
```



$v \in Q$ **and ...**

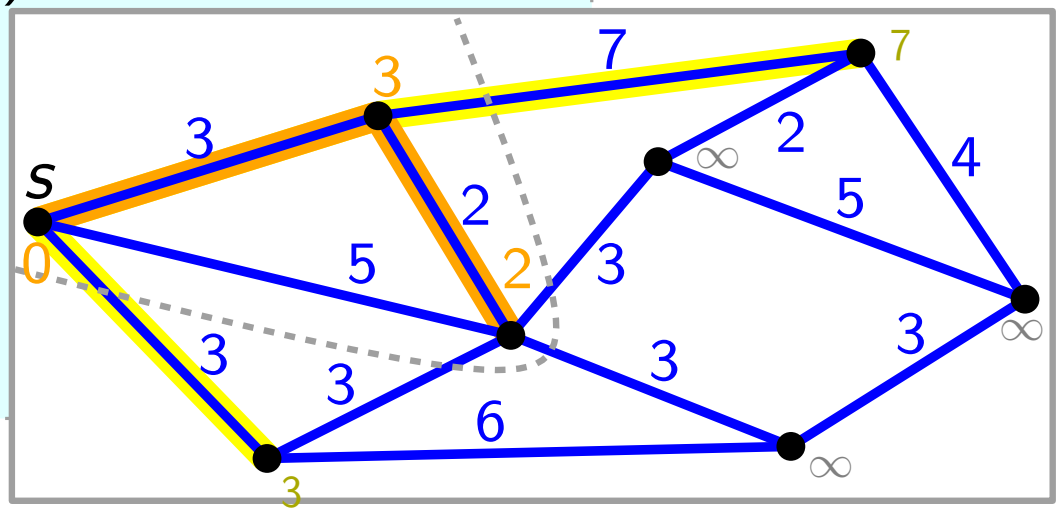
Relax′$(u, v; w)$

```
if v.d > ~~u.d +~~ w(u, v) then
    v.d = ~~u.d +~~ w(u, v)
    v.π = u
    Q.DecreaseKey(v, v.d)
```

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  Q = **new** PriorityQueue(V, d) // Gewichtung
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in$ Adj$[u]$ **do**
      Relax′$(u, v; w)$



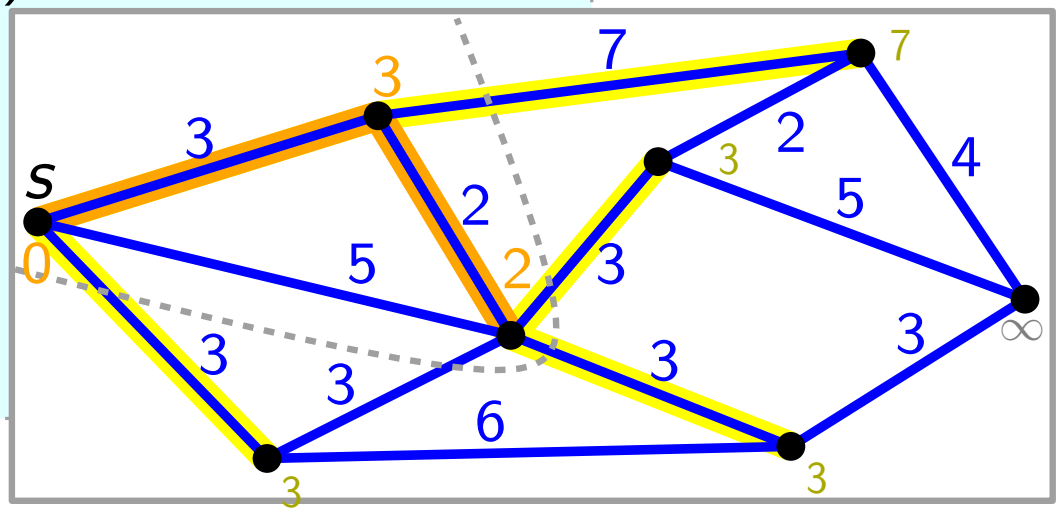
$v \in Q$ **and ...**

Relax′$(u, v; w)$
  if $v.d > ~~u.d +~~ w(u, v)$ **then**
    $v.d = ~~u.d +~~ w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey($v, v.d$)

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  Q = **new** PriorityQueue(V, d) // Gewichtung
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in$ Adj[u] **do**
      Relax′($u, v; w$)



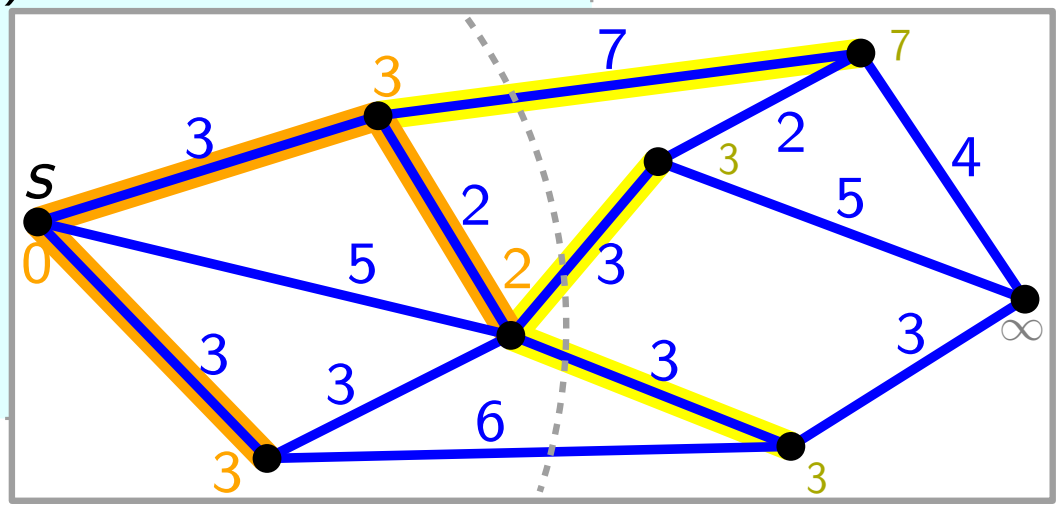
$v \in Q$ **and ...**

Relax′($u, v; w$)
  if $v.d > ~~u.d +~~ w(u, v)$ **then**
    $v.d = ~~u.d +~~ w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey($v, v.d$)

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  Q = **new** PriorityQueue(V, d) // Gewichtung
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in$ Adj[u] **do**
      Relax′($u, v; w$)



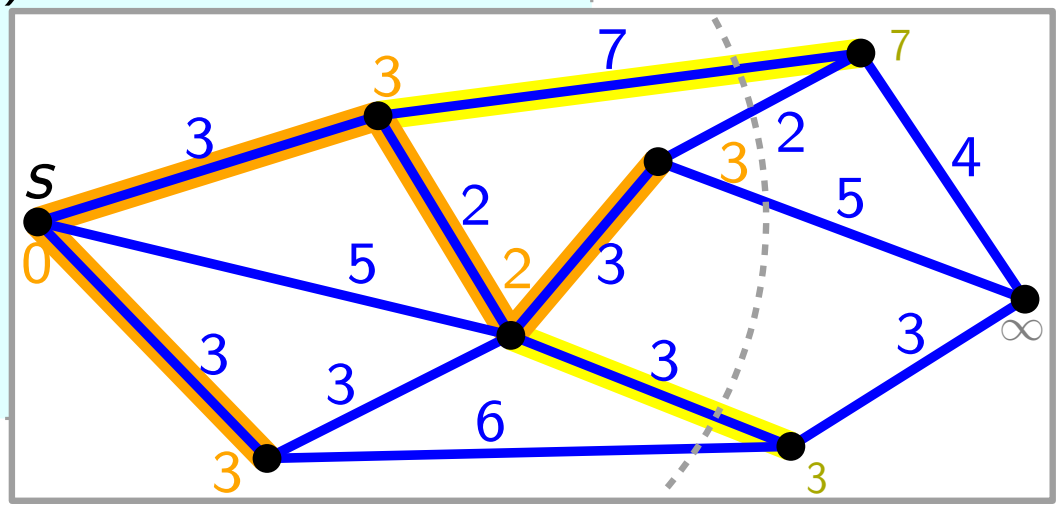
$v \in Q$ **and ...**

Relax′($u, v; w$)
  if $v.d > ~~u.d +~~ w(u, v)$ **then**
    $v.d = ~~u.d +~~ w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey($v, v.d$)

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  Q = **new** PriorityQueue(V, d) // Gewichtung
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in$ Adj$[u]$ **do**
      Relax′$(u, v; w)$



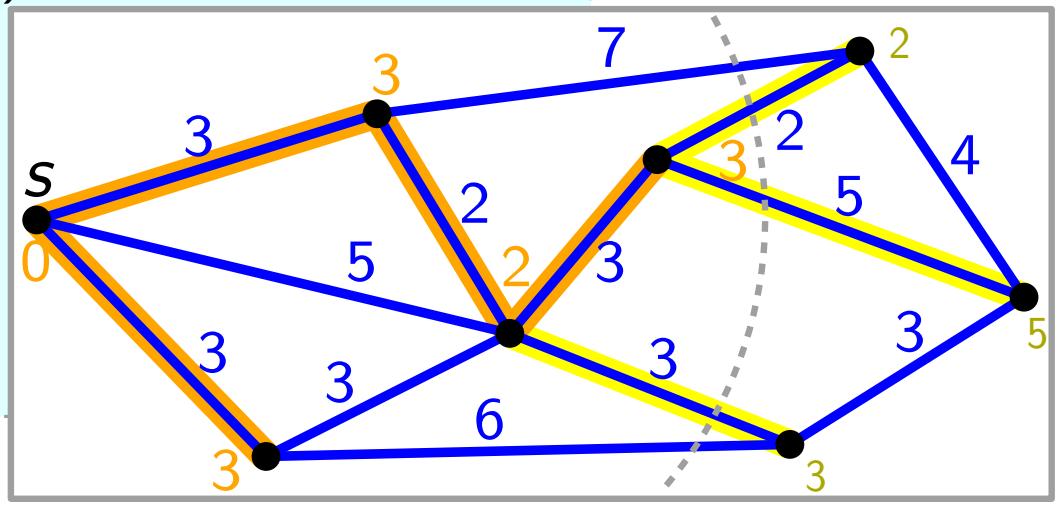
$v \in Q$ **and ...**

Relax′$(u, v; w)$
  if $v.d > ~~u.d +~~ w(u, v)$ **then**
    $v.d = ~~u.d +~~ w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey$(v, v.d)$

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  Q = **new** PriorityQueue(V, d) // Gewichtung
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in$ Adj[u] **do**
      Relax′($u, v; w$)



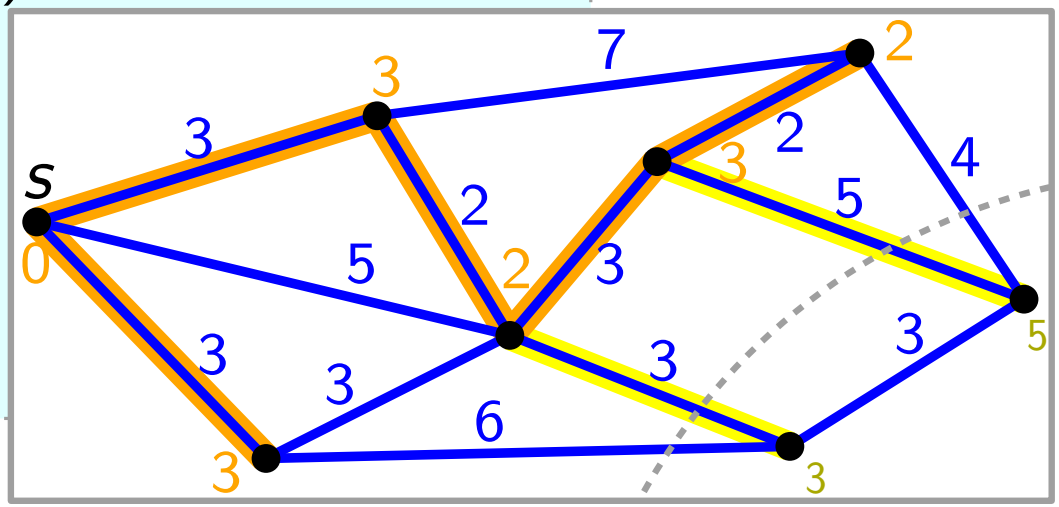
$v \in Q$ **and ...**

Relax′($u, v; w$)
  if $v.d > ~~u.d +~~ w(u, v)$ **then**
    $v.d = ~~u.d +~~ w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey($v, v.d$)

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  Q = **new** PriorityQueue(V, d) // Gewichtung
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in$ Adj[u] **do**
      Relax′($u, v; w$)



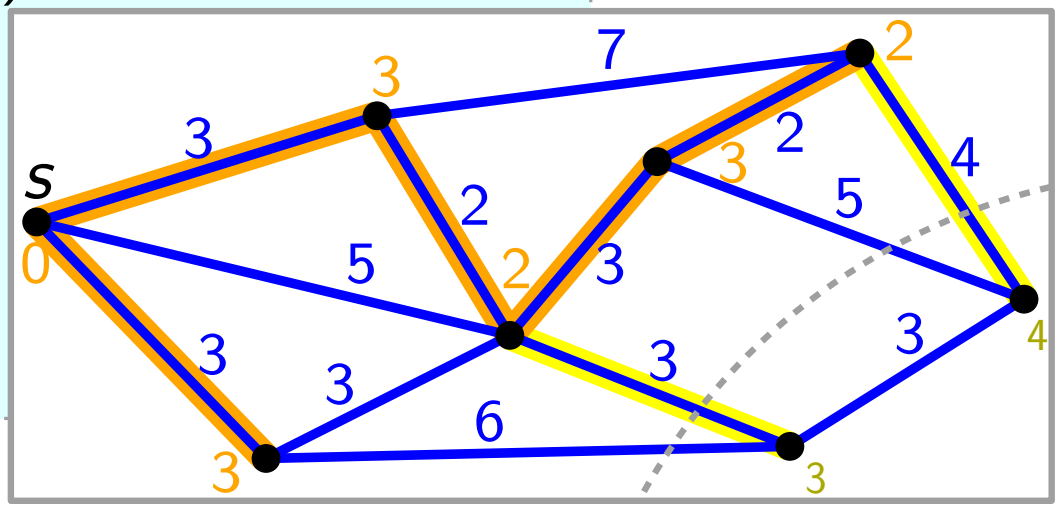
$v \in Q$ **and ...**

Relax′($u, v; w$)
  if $v.d > ~~u.d +~~ w(u, v)$ **then**
    $v.d = ~~u.d +~~ w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey($v, v.d$)

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  Q = **new** PriorityQueue(V, d) // Gewichtung
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in$ Adj[u] **do**
      Relax′($u, v; w$)



$v \in Q$ **and ...**

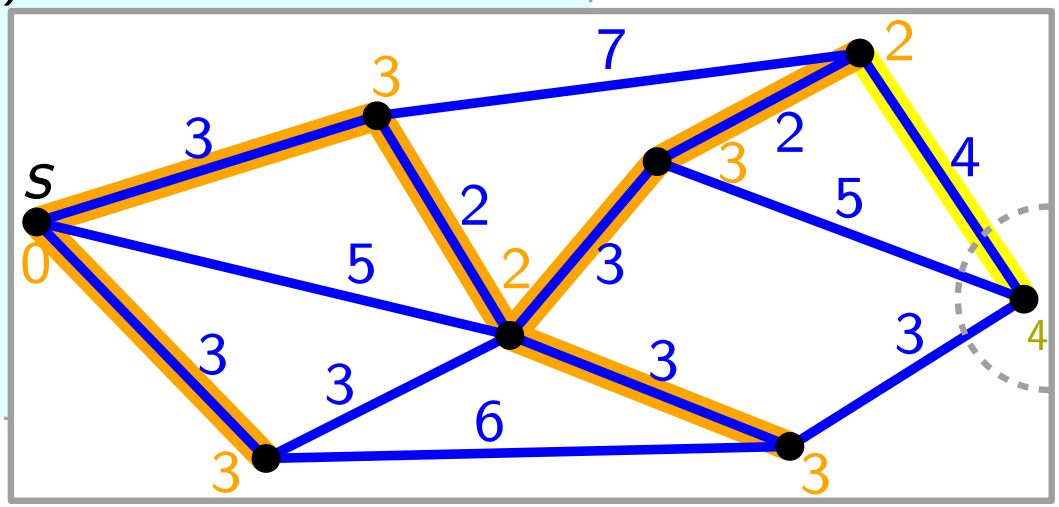
Relax′($u, v; w$)
  if $v.d > ~~u.d +~~ w(u, v)$ **then**
    $v.d = ~~u.d +~~ w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey($v, v.d$)

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)

```
Initialize(G, s)
Q = new PriorityQueue(V, d)   // Gewichtung
while not Q.Empty() do
    u = Q.ExtractMin()
    foreach v ∈ Adj[u] do
        Relax′(u, v; w)
```



$v \in Q$ **and ...**

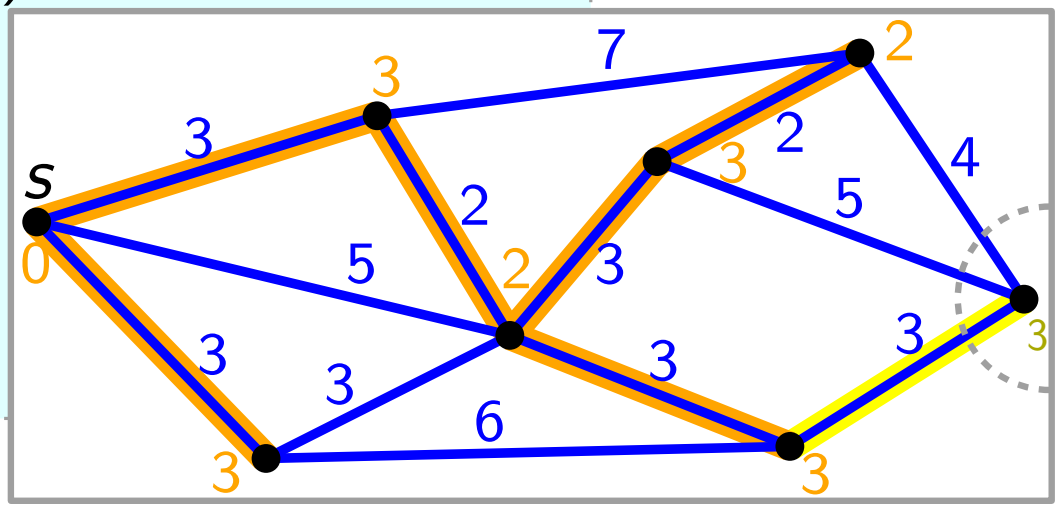
Relax′$(u, v; w)$

```
if v.d > ~~u.d +~~ w(u, v) then
    v.d = ~~u.d +~~ w(u, v)
    v.π = u
    Q.DecreaseKey(v, v.d)
```

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  Q = **new** PriorityQueue(V, d) // Gewichtung
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in$ Adj[u] **do**
      Relax′($u, v; w$)



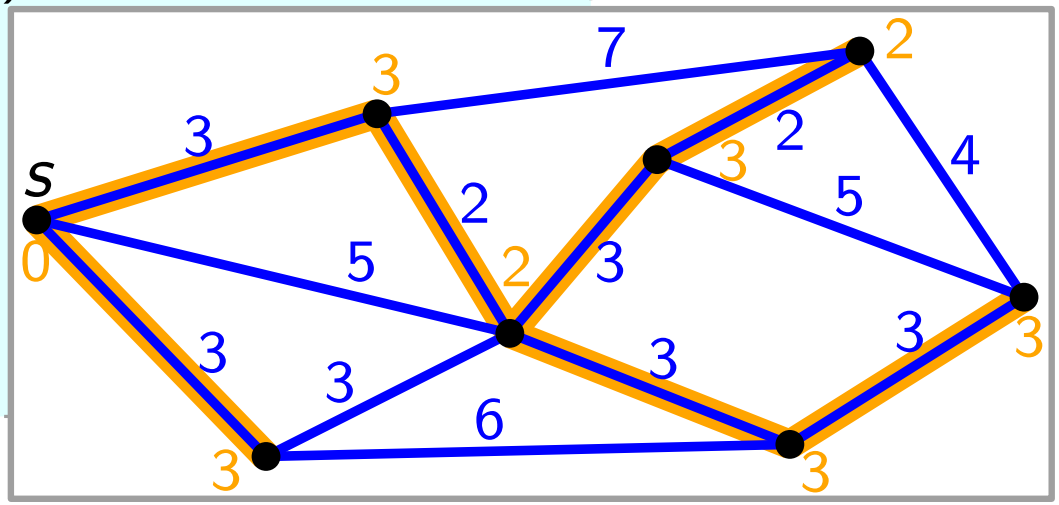
$v \in Q$ **and ...**

Relax′($u, v; w$)
  if $v.d > ~~u.d +~~ w(u, v)$ **then**
    $v.d = ~~u.d +~~ w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey($v, v.d$)

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  Q = **new** PriorityQueue(V, d) // Gewichtung
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in$ Adj[u] **do**
      Relax′($u, v; w$)



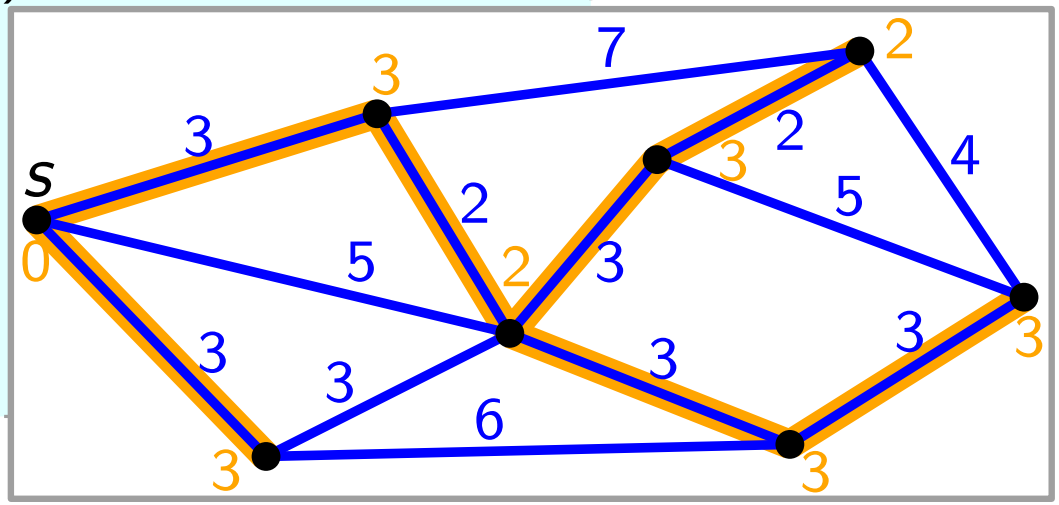
$v \in Q$ **and ...**

Relax′($u, v; w$)
  if $v.d > ~~u.d +~~ w(u, v)$ **then**
    $v.d = ~~u.d +~~ w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey($v, v.d$)

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  Q = **new** PriorityQueue(V, d) // Gewichtung
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in$ Adj[u] **do**
      Relax′($u, v; w$)



$v \in Q$ **and ...**

Relax′($u, v; w$)
  if $v.d > ~~u.d +~~ w(u, v)$ **then**
    $v.d = ~~u.d +~~ w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey($v, v.d$)

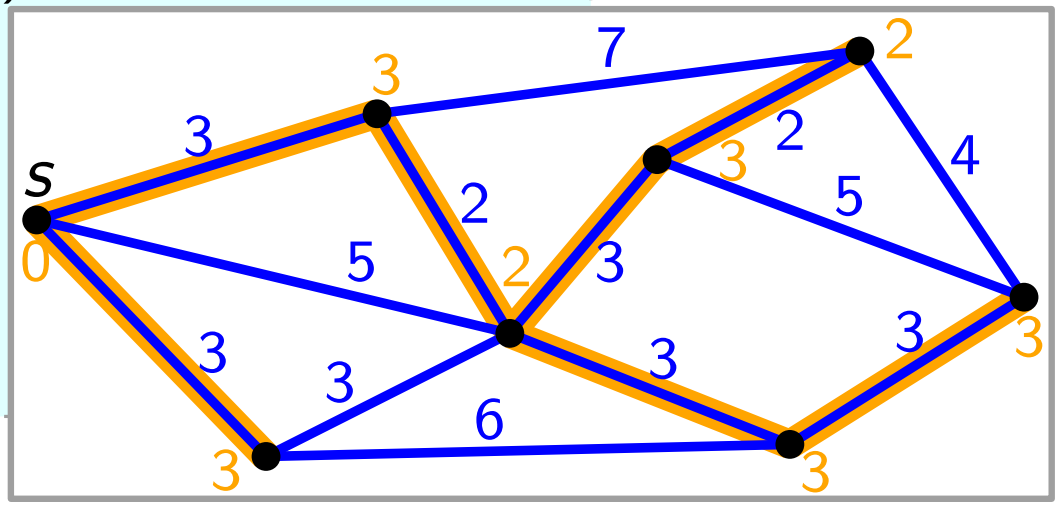
Korrektheit?

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)

```
Initialize(G, s)
Q = new PriorityQueue(V, d)   // Gewichtung
while not Q.Empty() do
    u = Q.ExtractMin()
    foreach v ∈ Adj[u] do
        Relax′(u, v; w)
```



$v \in Q$ **and ...**

Relax′$(u, v; w)$

```
if v.d > ~~u.d +~~ w(u, v) then
    v.d = ~~u.d +~~ w(u, v)
    v.π = u
    Q.DecreaseKey(v, v.d)
```

Korrektheit? ✓

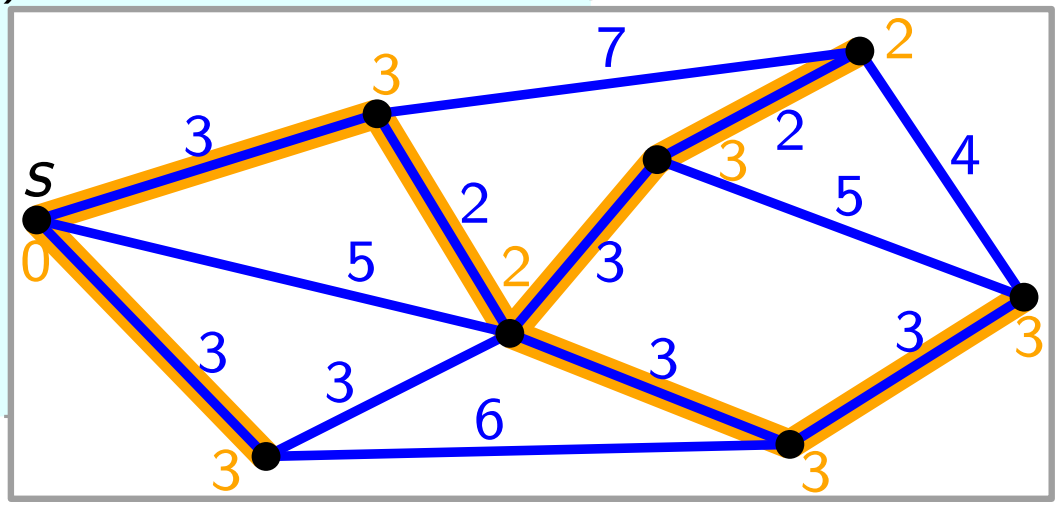
Folgt aus Korollar:
$A = \{\{u, u.\pi\} : u \notin Q\}$,
Kante $\{u, u.\pi\}$ immer
sicher bzgl. $(Q^\star, V \setminus Q^\star)$,
wobei $Q^\star = Q \cup \{u\}$.

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)

```
Initialize(G, s)
Q = new PriorityQueue(V, d)   // Gewichtung
while not Q.Empty() do
    u = Q.ExtractMin()
    foreach v ∈ Adj[u] do
        Relax′(u, v; w)
```



$v \in Q$ **and ...**

Relax′$(u, v; w)$

```
if v.d > ~~u.d +~~ w(u, v) then
    v.d = ~~u.d +~~ w(u, v)
    v.π = u
    Q.DecreaseKey(v, v.d)
```

Korrektheit? ✓

Laufzeit?

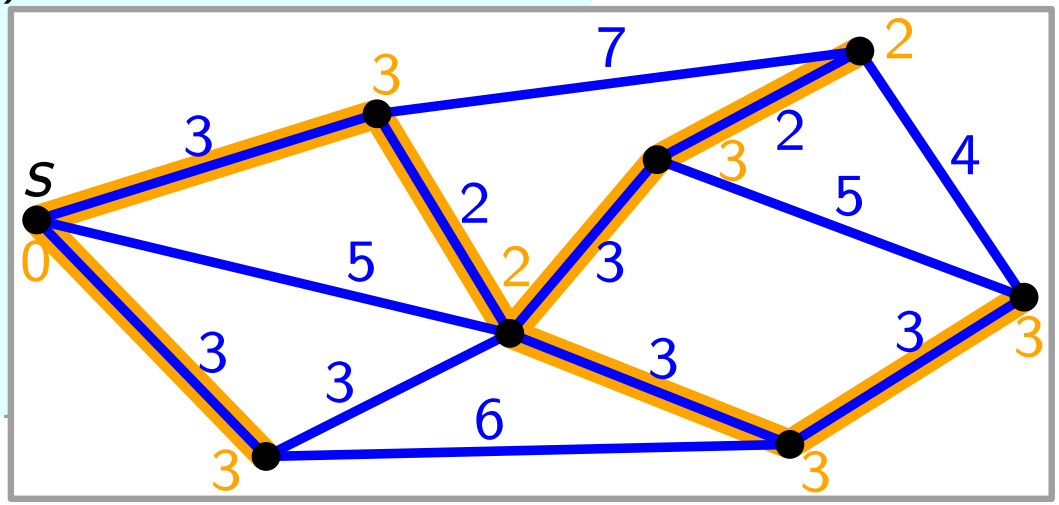
Folgt aus Korollar:
$A = \{\{u, u.\pi\} : u \notin Q\}$,
Kante $\{u, u.\pi\}$ immer
sicher bzgl. $(Q^\star, V \setminus Q^\star)$,
wobei $Q^\star = Q \cup \{u\}$.

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)

```
Initialize(G, s)
Q = new PriorityQueue(V, d)   // Gewichtung
while not Q.Empty() do
    u = Q.ExtractMin()
    foreach v ∈ Adj[u] do
        Relax'(u, v; w)
```



$v \in Q$ **and ...**

Relax′$(u, v; w)$

```
if v.d > ~~u.d +~~ w(u, v) then
    v.d = ~~u.d +~~ w(u, v)
    v.π = u
    Q.DecreaseKey(v, v.d)
```

Korrektheit? ✓

Folgt aus Korollar:
$A = \{\{u, u.\pi\} : u \notin Q\}$,
Kante $\{u, u.\pi\}$ immer
sicher bzgl. $(Q^\star, V \setminus Q^\star)$,
wobei $Q^\star = Q \cup \{u\}$.

Laufzeit?

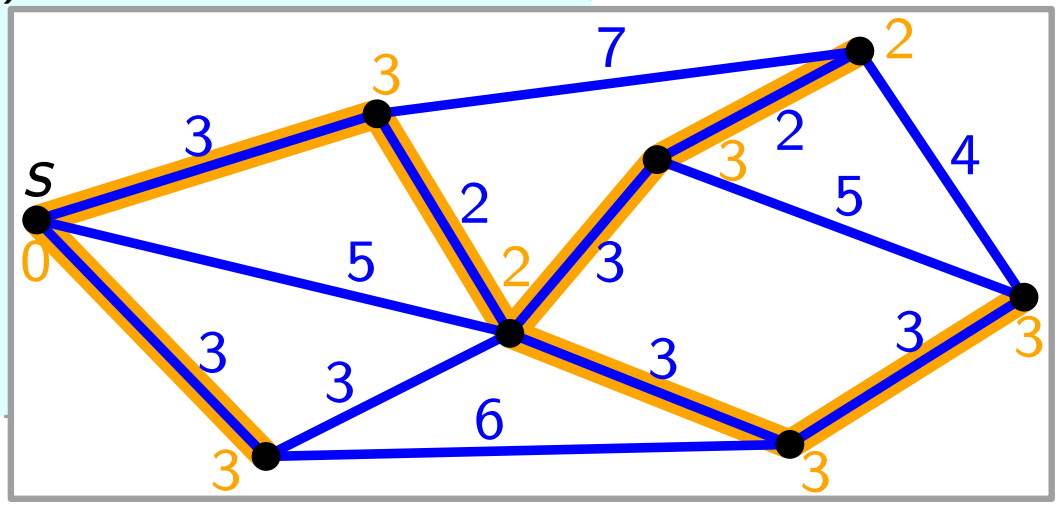
$O(\square \cdot \text{DecreaseKey} + \square \cdot \text{ExtractMin})$

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)

```
Initialize(G, s)
Q = new PriorityQueue(V, d)   // Gewichtung
while not Q.Empty() do
    u = Q.ExtractMin()
    foreach v ∈ Adj[u] do
        Relax'(u, v; w)
```



$v \in Q$ **and ...**

Relax′$(u, v; w)$

```
if v.d > ~~u.d +~~ w(u, v) then
    v.d = ~~u.d +~~ w(u, v)
    v.π = u
    Q.DecreaseKey(v, v.d)
```

Korrektheit? ✓

Folgt aus Korollar:
$A = \{\{u, u.\pi\} : u \notin Q\}$,
Kante $\{u, u.\pi\}$ immer
sicher bzgl. $(Q^\star, V \setminus Q^\star)$,
wobei $Q^\star = Q \cup \{u\}$.

Laufzeit?

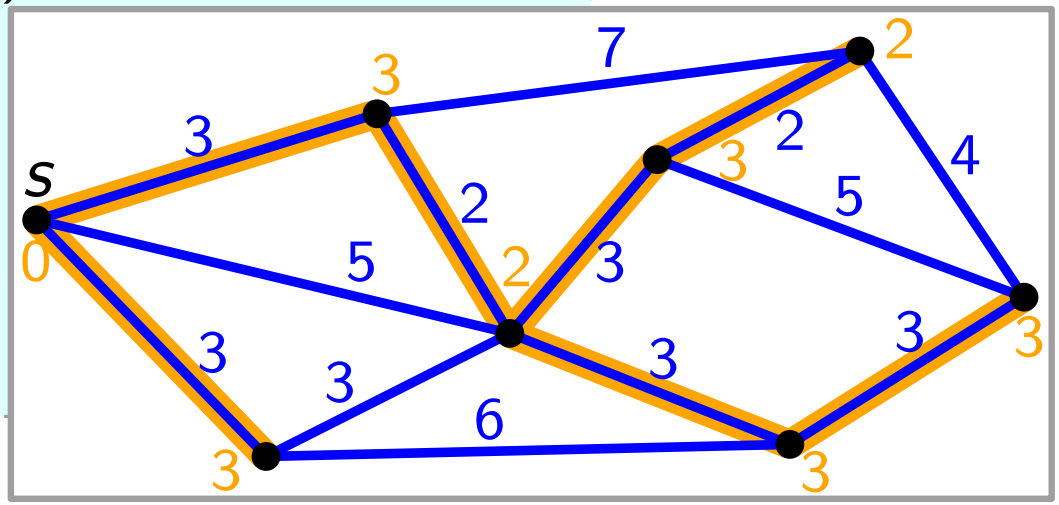
$O(|E| \cdot \text{DecreaseKey} + |V| \cdot \text{ExtractMin})$

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)

```
Initialize(G, s)
Q = new PriorityQueue(V, d)   // Gewichtung
while not Q.Empty() do
    u = Q.ExtractMin()
    foreach v ∈ Adj[u] do
        Relax'(u, v; w)
```



$v \in Q$ **and ...**

Relax′$(u, v; w)$

```
if v.d > ~~u.d +~~ w(u, v) then
    v.d = ~~u.d +~~ w(u, v)
    v.π = u
    Q.DecreaseKey(v, v.d)
```

Korrektheit? ✓

Folgt aus Korollar:
$A = \{\{u, u.\pi\} : u \notin Q\}$,
Kante $\{u, u.\pi\}$ immer
sicher bzgl. $(Q^\star, V \setminus Q^\star)$,
wobei $Q^\star = Q \cup \{u\}$.

Laufzeit?

$O(|E| \cdot \text{DecreaseKey} + |V| \cdot \text{ExtractMin})$

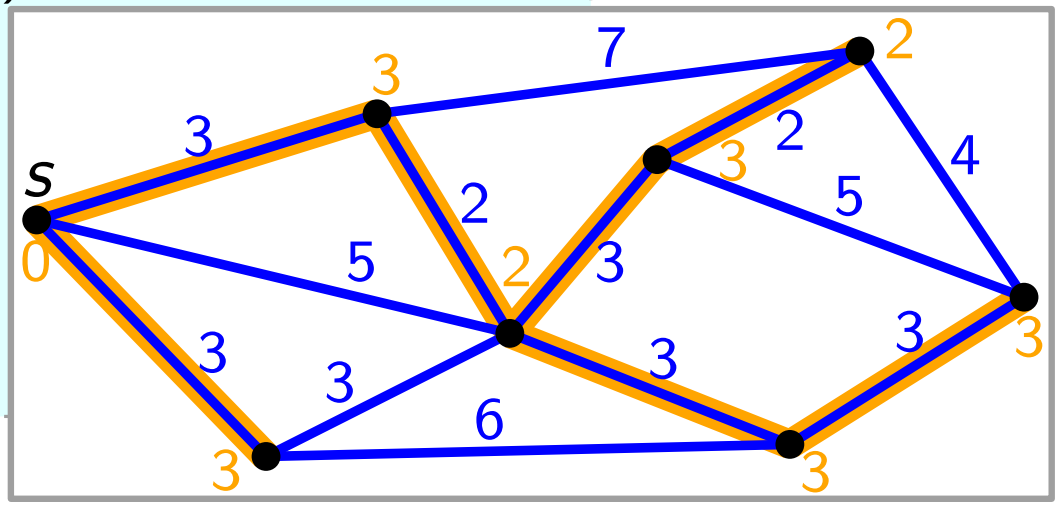
$\Rightarrow O((E + V) \log V)$ [Heap/RS-Baum]

Der Algorithmus von Jarník-Prim (1930/1957)

JarníkPrimMST ← Undirected

~~Dijkstra~~(WeightedGraph $G = (V, E; w)$, Vertex s)

```
Initialize(G, s)
Q = new PriorityQueue(V, d)   // Gewichtung
while not Q.Empty() do
    u = Q.ExtractMin()
    foreach v ∈ Adj[u] do
        Relax'(u, v; w)
```



$v \in Q$ **and ...**

Relax′$(u, v; w)$

```
if v.d > ~~u.d +~~ w(u, v) then
    v.d = ~~u.d +~~ w(u, v)
    v.π = u
    Q.DecreaseKey(v, v.d)
```

Korrektheit? ✓

Folgt aus Korollar:
$A = \{\{u, u.\pi\} : u \notin Q\}$,
Kante $\{u, u.\pi\}$ immer
sicher bzgl. $(Q^\star, V \setminus Q^\star)$,
wobei $Q^\star = Q \cup \{u\}$.

Laufzeit?

$O(|E| \cdot \text{DecreaseKey} + |V| \cdot \text{ExtractMin})$

$\Rightarrow O((E + V) \log V)$ [Heap/RS-Baum]

$\Rightarrow O(E + V \log V)$ [Fibonacci-Heap]

Einschub: halbdynamische Mengen

Einschub: halbdynamische Mengen (wachsen nur, schrumpfen nicht)

Einschub: halbdynamische Mengen (wachsen nur, schrumpfen nicht)

Die halbdyn. Mengen zerlegen immer eine Grundmenge X.

Einschub: halbdynamische Mengen (wachsen nur, schrumpfen nicht)

Die halbdyn. Mengen zerlegen immer eine Grundmenge X.

Drei Operationen:

Einschub: halbdynamische Mengen (wachsen nur, schrumpfen nicht)

Die halbdyn. Mengen zerlegen immer eine Grundmenge X.

Drei Operationen:

MakeSet(Element x)

FindSet(Element x)

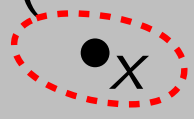
Union(Elem. x, Elem. y)

Einschub: halbdynamische Mengen (wachsen nur, schrumpfen nicht)

Die halbdyn. Mengen zerlegen immer eine Grundmenge X.

Drei Operationen:

MakeSet(Element x) legt die Menge $\{x\}$ an.



FindSet(Element x)

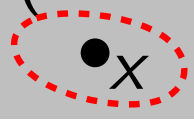
Union(Elem. x, Elem. y)

Einschub: halbdynamische Mengen (wachsen nur, schrumpfen nicht)

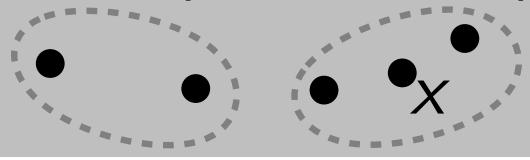
Die halbdyn. Mengen zerlegen immer eine Grundmenge X.

Drei Operationen:

MakeSet(Element x) — legt die Menge $\{x\}$ an.



FindSet(Element x) — liefert (Zeiger auf) die Menge zurück, die momentan x enthält.



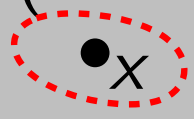
Union(Elem. x, Elem. y)

Einschub: halbdynamische Mengen (wachsen nur, schrumpfen nicht)

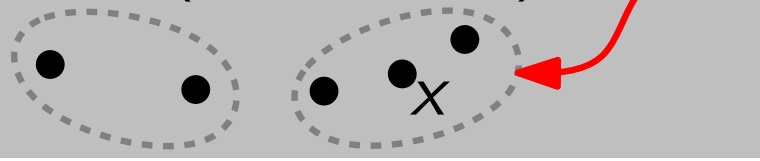
Die halbdyn. Mengen zerlegen immer eine Grundmenge X.

Drei Operationen:

MakeSet(Element x) legt die Menge $\{x\}$ an.



FindSet(Element x) liefert (Zeiger auf) die Menge zurück, die momentan x enthält.



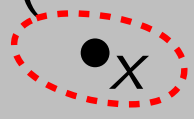
Union(Elem. x, Elem. y)

Einschub: halbdynamische Mengen (wachsen nur, schrumpfen nicht)

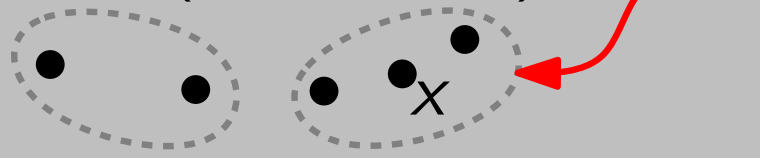
Die halbdyn. Mengen zerlegen immer eine Grundmenge X.

Drei Operationen:

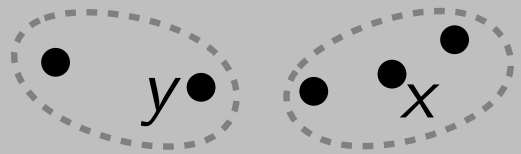
MakeSet(Element x) — legt die Menge $\{x\}$ an.



FindSet(Element x) — liefert (Zeiger auf) die Menge zurück, die momentan x enthält.



Union(Elem. x, Elem. y) — vereinigt die Mengen, die momentan x und y enthalten.

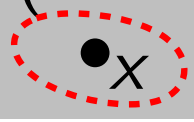


Einschub: halbdynamische Mengen (wachsen nur, schrumpfen nicht)

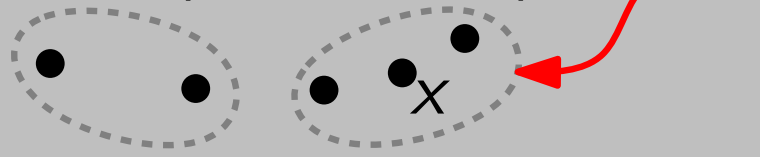
Die halbdyn. Mengen zerlegen immer eine Grundmenge X.

Drei Operationen:

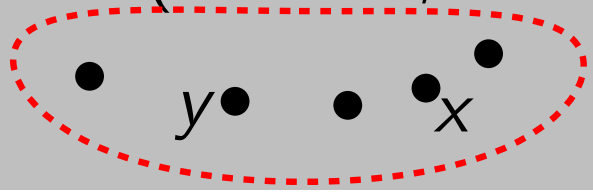
MakeSet(Element x) — legt die Menge $\{x\}$ an.



FindSet(Element x) — liefert (Zeiger auf) die Menge zurück, die momentan x enthält.



Union(Elem. x, Elem. y) — vereinigt die Mengen, die momentan x und y enthalten.

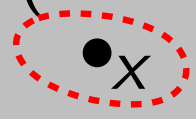


Einschub: halbdynamische Mengen (wachsen nur, schrumpfen nicht)

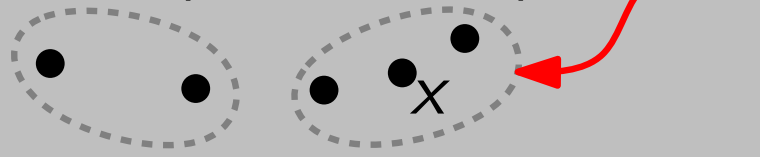
Die halbdyn. Mengen zerlegen immer eine Grundmenge X.

Drei Operationen:

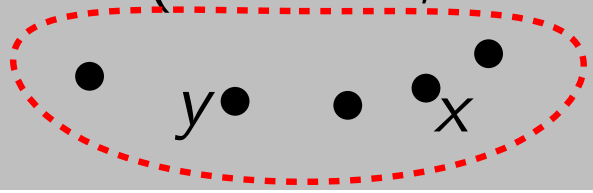
MakeSet(Element x) — legt die Menge $\{x\}$ an.



FindSet(Element x) — liefert (Zeiger auf) die Menge zurück, die momentan x enthält.



Union(Elem. x, Elem. y) — vereinigt die Mengen, die momentan x und y enthalten.



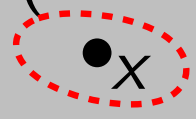
Satz. Eine Folge von m MakeSet-, Union- und FindSet-Oper., von denen n MakeSet-Oper. sind, benötigt $O($ $)$ Zeit

Einschub: halbdynamische Mengen (wachsen nur, schrumpfen nicht)

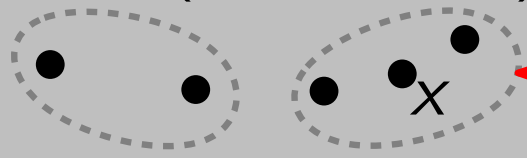
Die halbdyn. Mengen zerlegen immer eine Grundmenge X.

Drei Operationen:

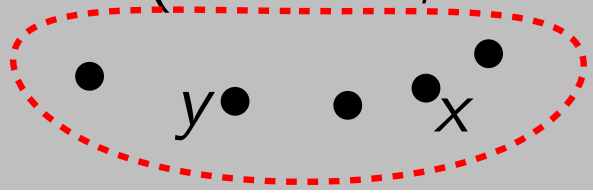
MakeSet(Element x) — legt die Menge $\{x\}$ an.



FindSet(Element x) — liefert (Zeiger auf) die Menge zurück, die momentan x enthält.



Union(Elem. x, Elem. y) — vereinigt die Mengen, die momentan x und y enthalten.



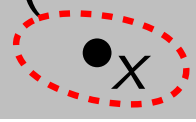
Satz. Eine Folge von m MakeSet-, Union- und FindSet-Oper., von denen n MakeSet-Oper. sind, benötigt $O(m \cdot \alpha(n))$ Zeit

Einschub: halbdynamische Mengen (wachsen nur, schrumpfen nicht)

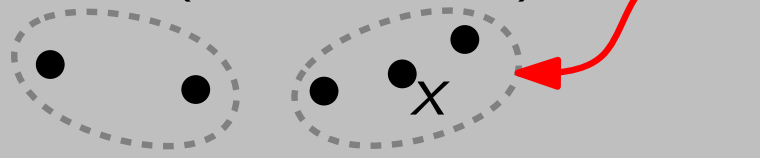
Die halbdyn. Mengen zerlegen immer eine Grundmenge X.

Drei Operationen:

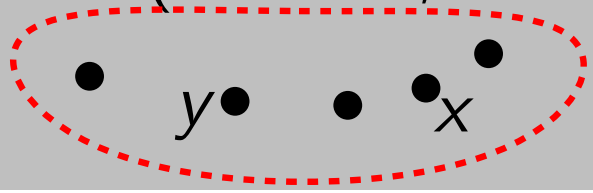
MakeSet(Element x) — legt die Menge $\{x\}$ an.



FindSet(Element x) — liefert (Zeiger auf) die Menge zurück, die momentan x enthält.



Union(Elem. x, Elem. y) — vereinigt die Mengen, die momentan x und y enthalten.



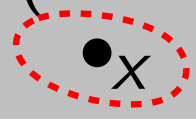
Satz. Eine Folge von m MakeSet-, Union- und FindSet-Oper., von denen n MakeSet-Oper. sind, benötigt $O(m \cdot \alpha(n))$ Zeit, wobei $\alpha(n) \leq 4$ für alle $n \leq 10^{80}$. Insbesondere $\alpha(n) \ll \log_{10} n$ für $n > 1$.

Einschub: halbdynamische Mengen (wachsen nur, schrumpfen nicht)

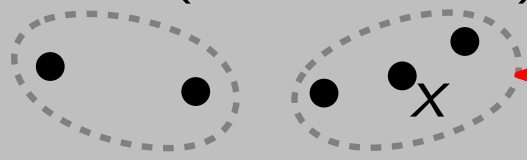
Die halbdyn. Mengen zerlegen immer eine Grundmenge X.

Drei Operationen:

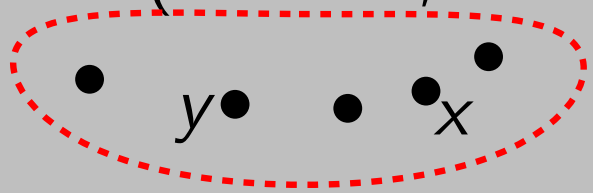
MakeSet(Element x) — legt die Menge $\{x\}$ an.



FindSet(Element x) — liefert (Zeiger auf) die Menge zurück, die momentan x enthält.



Union(Elem. x, Elem. y) — vereinigt die Mengen, die momentan x und y enthalten.

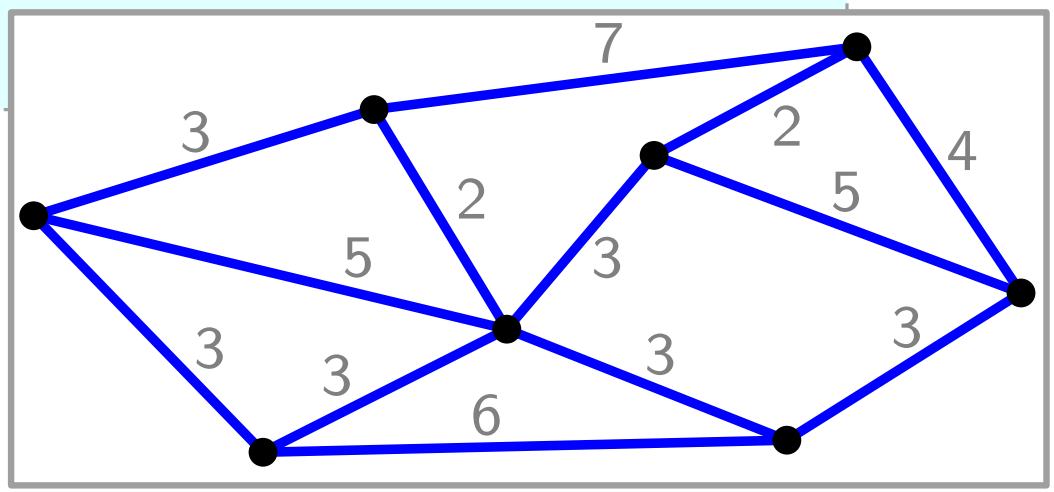


Satz. Eine Folge von m MakeSet-, Union- und FindSet-Oper., von denen n MakeSet-Oper. sind, benötigt $O(m \cdot \alpha(n))$ Zeit, wobei $\alpha(n) \leq 4$ für alle $n \leq 10^{80}$. Insbesondere $\alpha(n) \ll \log_{10} n$ für $n > 1$.

funktionales Inverses der extrem schnellwachsenden Ackermann-Funktion $A(n, n)$

Der Algorithmus von Kruskal

KruskalMST(WeightedUndirectedGraph $G = (V, E)$, w)
  $A = \emptyset$
  foreach $v \in V$ **do**
    MakeSet(v)



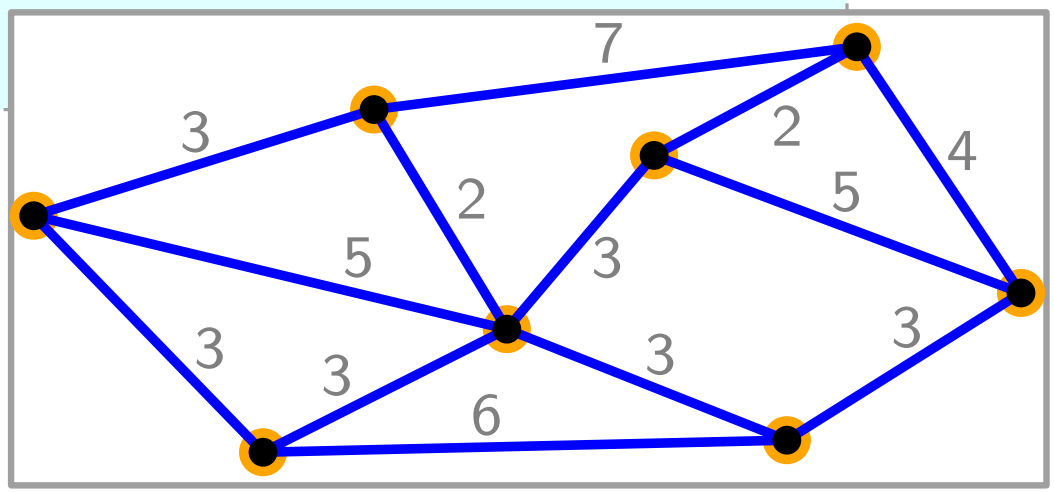
Der Algorithmus von Kruskal

KruskalMST(WeightedUndirectedGraph $G = (V, E)$, w)

$A = \emptyset$

foreach $v \in V$ **do**

  MakeSet(v)



Der Algorithmus von Kruskal

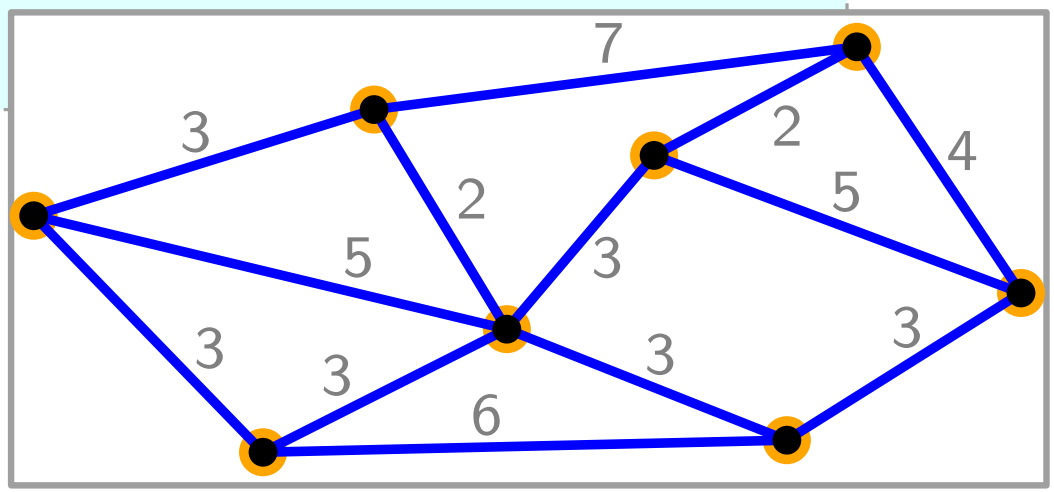
KruskalMST(WeightedUndirectedGraph $G = (V, E)$, w)

$A = \emptyset$

foreach $v \in V$ **do**

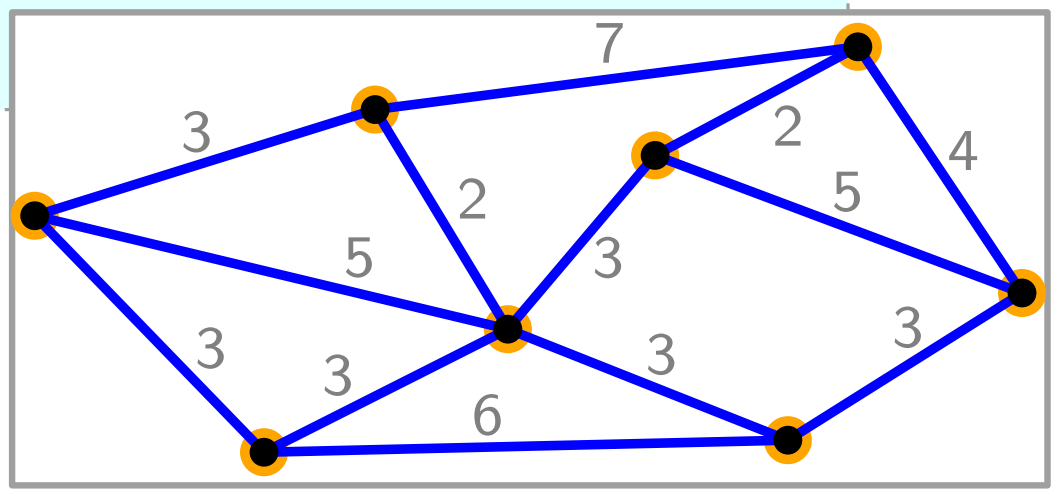
  MakeSet(v)

Sortiere E nicht-absteigend nach Gewicht w



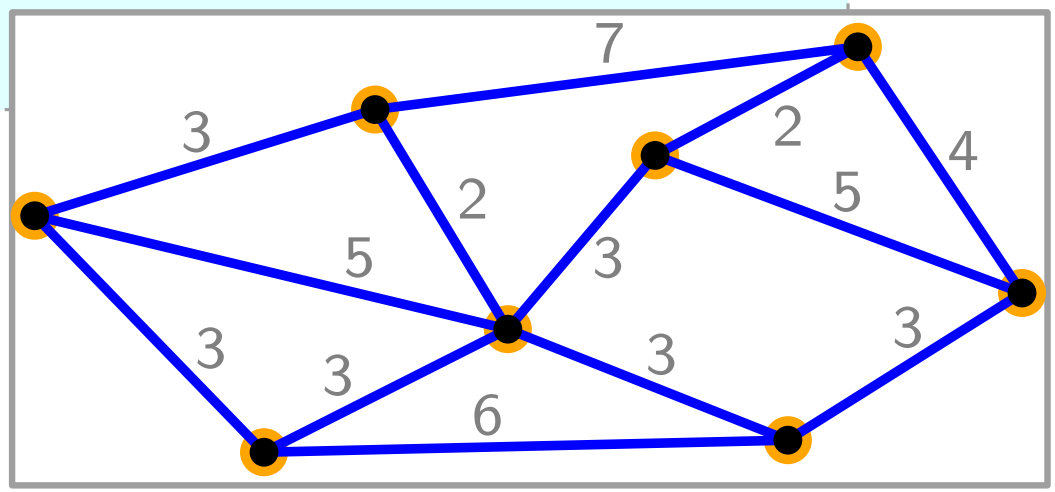
Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
  A = ∅
  foreach v ∈ V do
    MakeSet(v)
  Sortiere E nicht-absteigend nach Gewicht w
  foreach uv ∈ E do
    if FindSet(u) ≠ FindSet(v) then
      Was passiert hier?
  return A
```



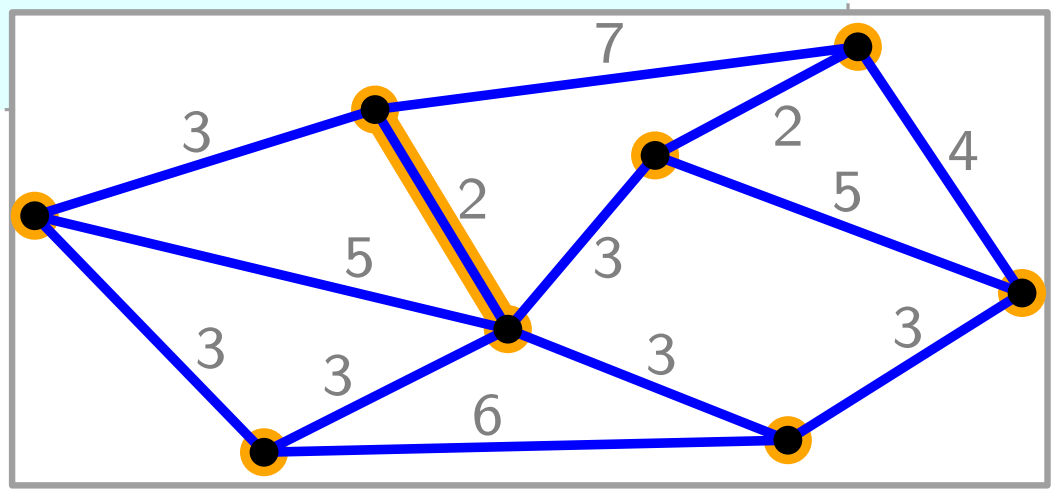
Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
    A = ∅
    foreach v ∈ V do
        MakeSet(v)
    Sortiere E nicht-absteigend nach Gewicht w
    foreach uv ∈ E do
        if FindSet(u) ≠ FindSet(v) then
            A = A ∪ {uv}
            Union(u, v)
    return A
```



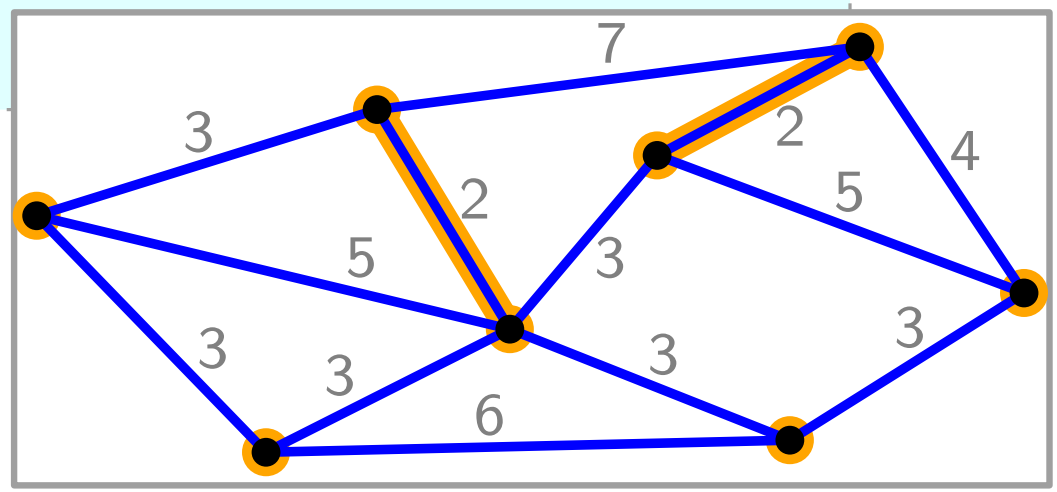
Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
    A = ∅
    foreach v ∈ V do
        MakeSet(v)
    Sortiere E nicht-absteigend nach Gewicht w
    foreach uv ∈ E do
        if FindSet(u) ≠ FindSet(v) then
            A = A ∪ {uv}
            Union(u, v)
    return A
```



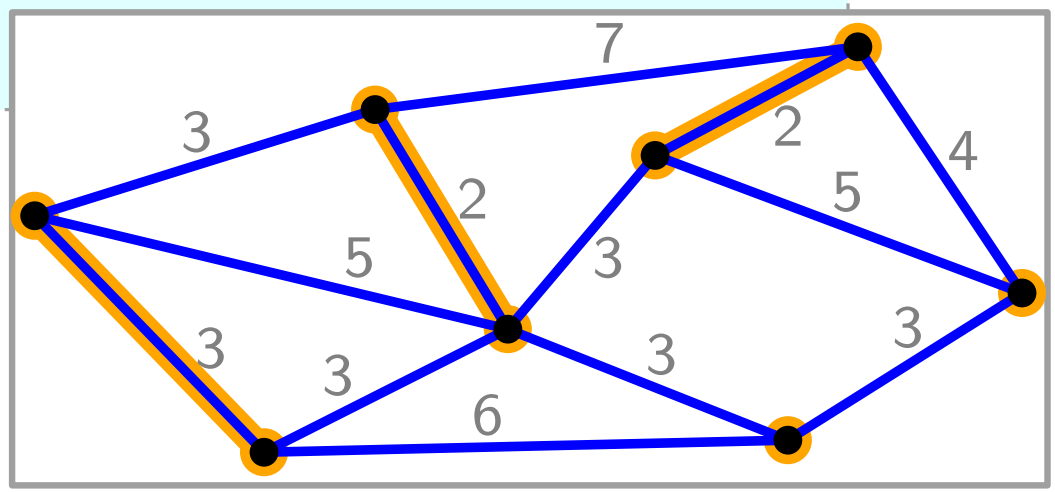
Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
    A = ∅
    foreach v ∈ V do
        MakeSet(v)
    Sortiere E nicht-absteigend nach Gewicht w
    foreach uv ∈ E do
        if FindSet(u) ≠ FindSet(v) then
            A = A ∪ {uv}
            Union(u, v)
    return A
```



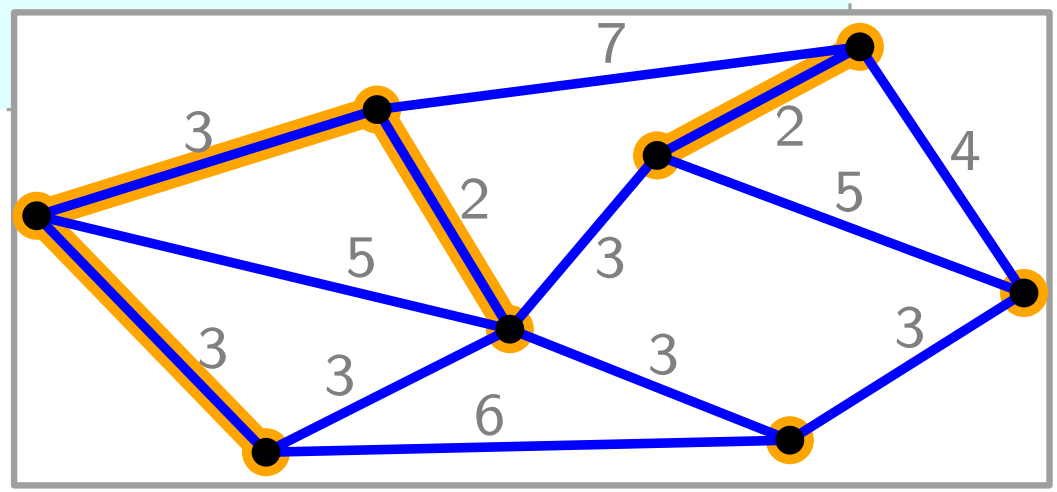
Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
  A = ∅
  foreach v ∈ V do
    MakeSet(v)
  Sortiere E nicht-absteigend nach Gewicht w
  foreach uv ∈ E do
    if FindSet(u) ≠ FindSet(v) then
      A = A ∪ {uv}
      Union(u, v)
  return A
```



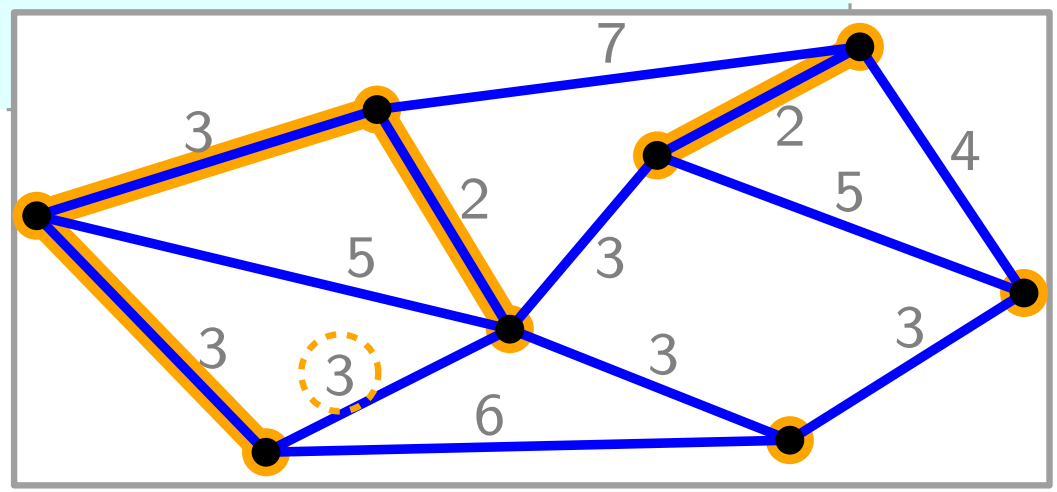
Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
  A = ∅
  foreach v ∈ V do
    MakeSet(v)
  Sortiere E nicht-absteigend nach Gewicht w
  foreach uv ∈ E do
    if FindSet(u) ≠ FindSet(v) then
      A = A ∪ {uv}
      Union(u, v)
  return A
```



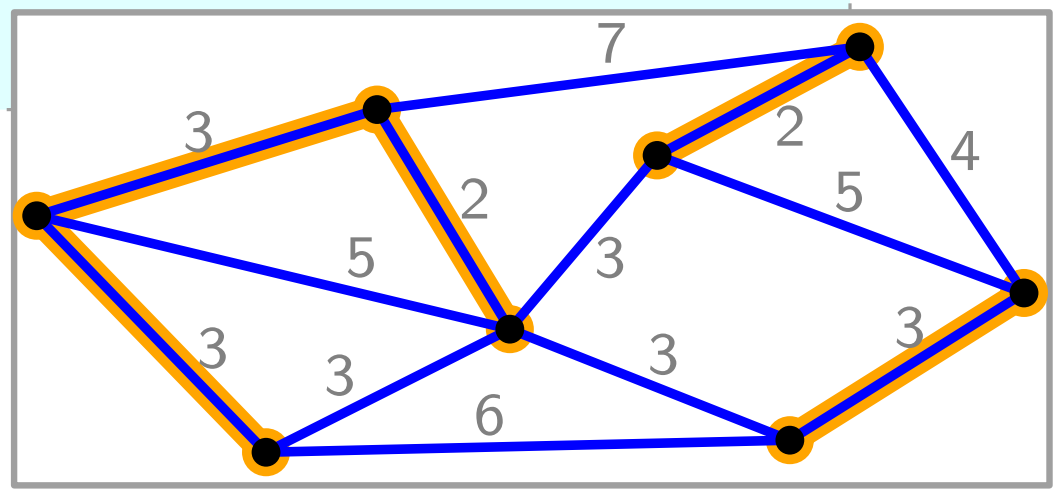
Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
    A = ∅
    foreach v ∈ V do
        MakeSet(v)
    Sortiere E nicht-absteigend nach Gewicht w
    foreach uv ∈ E do
        if FindSet(u) ≠ FindSet(v) then
            A = A ∪ {uv}
            Union(u, v)
    return A
```



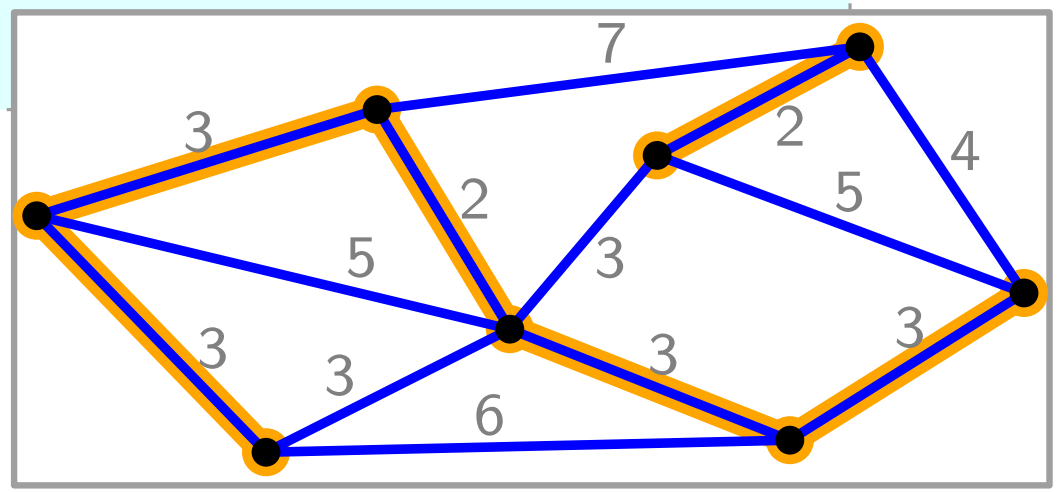
Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
  A = ∅
  foreach v ∈ V do
    MakeSet(v)
  Sortiere E nicht-absteigend nach Gewicht w
  foreach uv ∈ E do
    if FindSet(u) ≠ FindSet(v) then
      A = A ∪ {uv}
      Union(u, v)
  return A
```



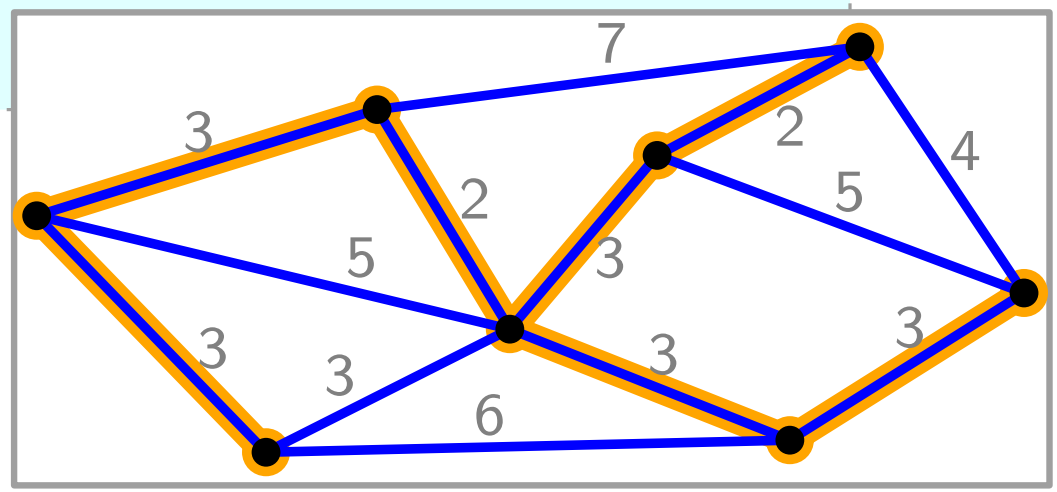
Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
    A = ∅
    foreach v ∈ V do
        MakeSet(v)
    Sortiere E nicht-absteigend nach Gewicht w
    foreach uv ∈ E do
        if FindSet(u) ≠ FindSet(v) then
            A = A ∪ {uv}
            Union(u, v)
    return A
```



Der Algorithmus von Kruskal

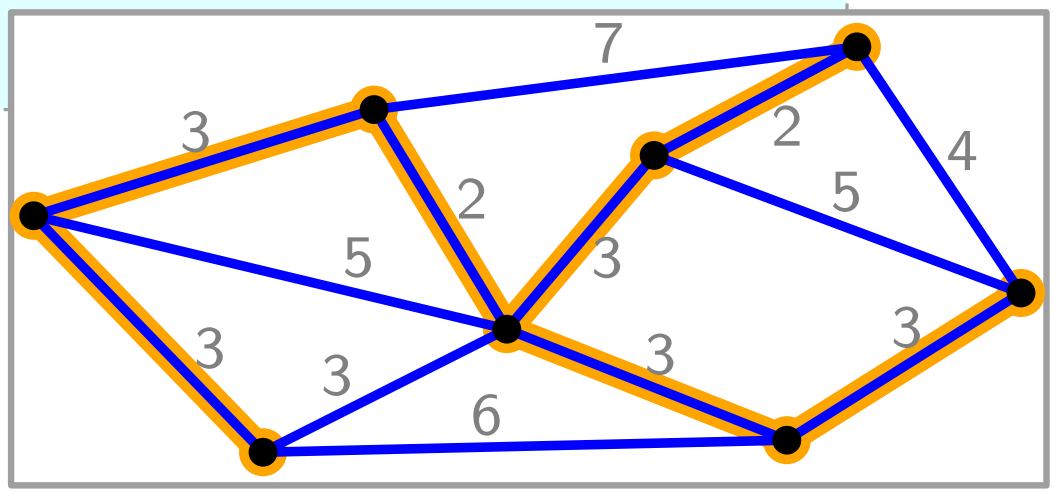
```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
    A = ∅
    foreach v ∈ V do
        MakeSet(v)
    Sortiere E nicht-absteigend nach Gewicht w
    foreach uv ∈ E do
        if FindSet(u) ≠ FindSet(v) then
            A = A ∪ {uv}
            Union(u, v)
    return A
```



Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
  A = ∅
  foreach v ∈ V do
    MakeSet(v)
  Sortiere E nicht-absteigend nach Gewicht w
  foreach uv ∈ E do
    if FindSet(u) ≠ FindSet(v) then
      A = A ∪ {uv}
      Union(u, v)
  return A
```

Laufzeit?

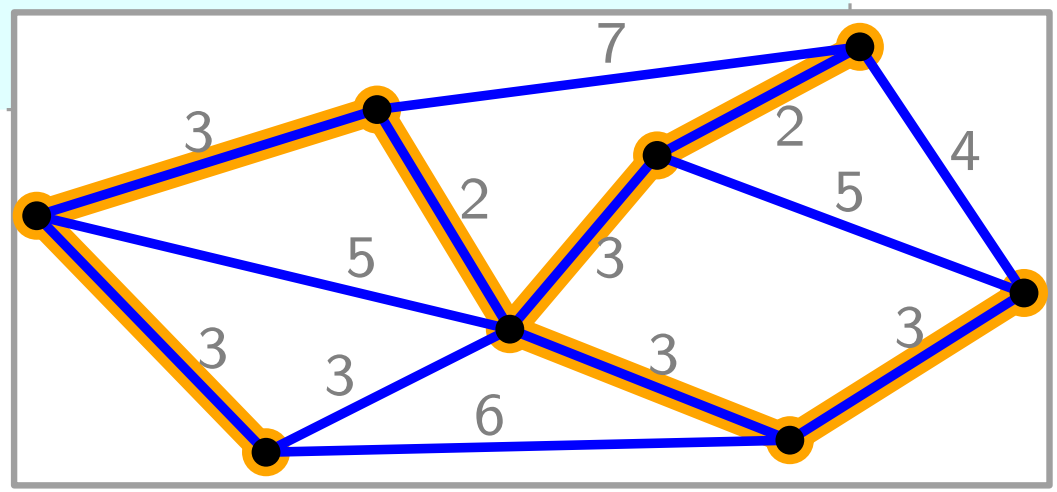


Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
    A = ∅
    foreach v ∈ V do
        MakeSet(v)
    Sortiere E nicht-absteigend nach Gewicht w
    foreach uv ∈ E do
        if FindSet(u) ≠ FindSet(v) then
            A = A ∪ {uv}
            Union(u, v)
    return A
```

Laufzeit?

·MakeSet + ·Union
+ ·FindSet + Sort(E)

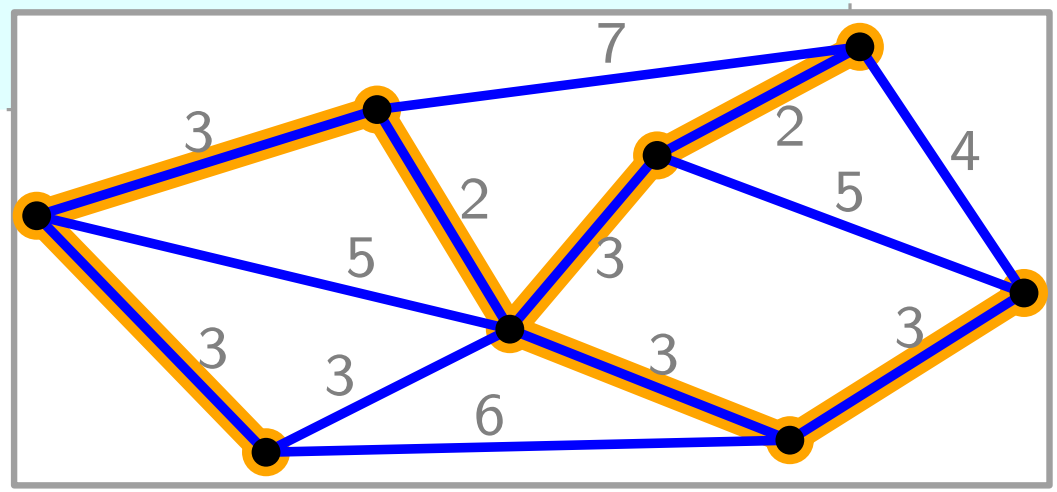


Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
  A = ∅
  foreach v ∈ V do
    MakeSet(v)
  Sortiere E nicht-absteigend nach Gewicht w
  foreach uv ∈ E do
    if FindSet(u) ≠ FindSet(v) then
      A = A ∪ {uv}
      Union(u, v)
  return A
```

Laufzeit?

$|V| \cdot \text{MakeSet} + \quad \cdot \text{Union} + \quad \cdot \text{FindSet} + \text{Sort}(E)$

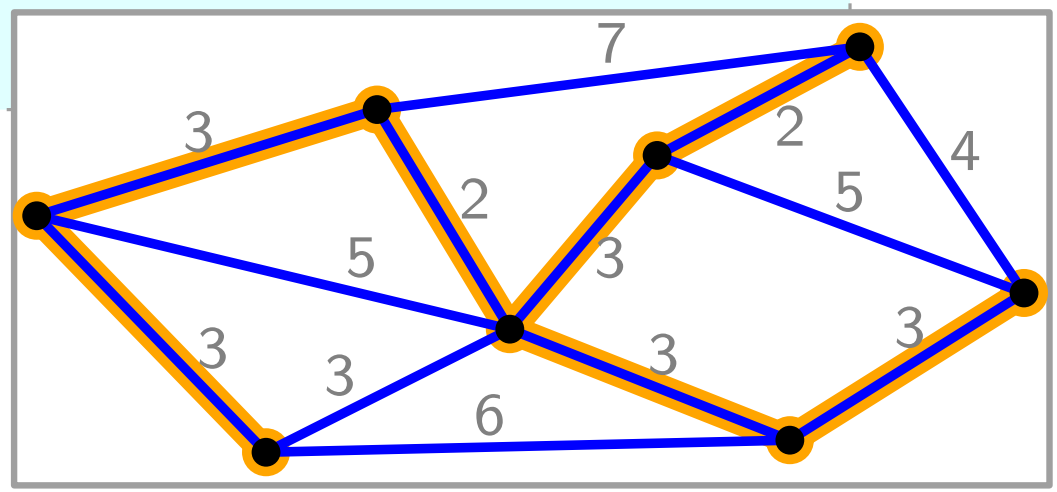


Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
    A = ∅
    foreach v ∈ V do
        MakeSet(v)
    Sortiere E nicht-absteigend nach Gewicht w
    foreach uv ∈ E do
        if FindSet(u) ≠ FindSet(v) then
            A = A ∪ {uv}
            Union(u, v)
    return A
```

Laufzeit?

$|V| \cdot \text{MakeSet} + (|V| - 1) \cdot \text{Union} + \quad \cdot \text{FindSet} + \text{Sort}(E)$

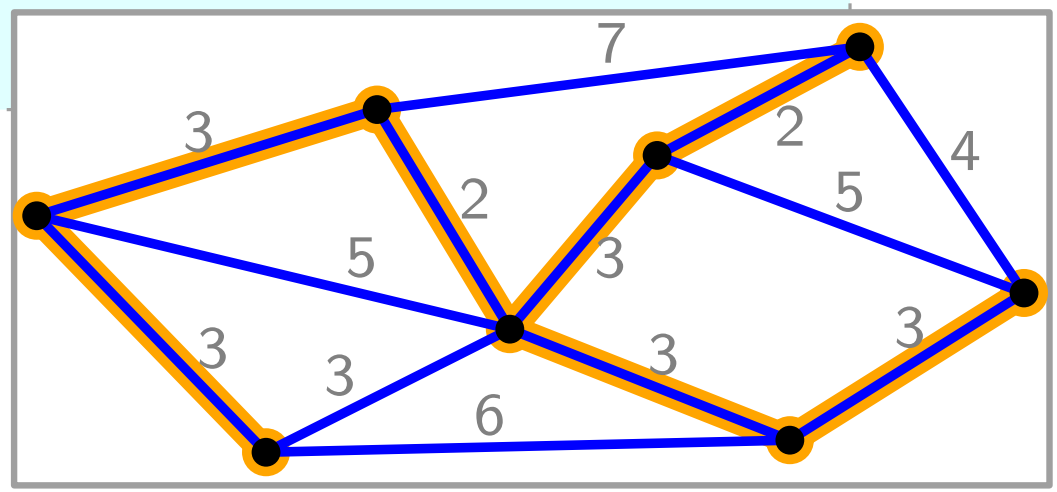


Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
    A = ∅
    foreach v ∈ V do
        MakeSet(v)
    Sortiere E nicht-absteigend nach Gewicht w
    foreach uv ∈ E do
        if FindSet(u) ≠ FindSet(v) then
            A = A ∪ {uv}
            Union(u, v)
    return A
```

Laufzeit?

$|V|\cdot\text{MakeSet} + (|V|-1)\cdot\text{Union} + 2|E|\cdot\text{FindSet} + \text{Sort}(E)$



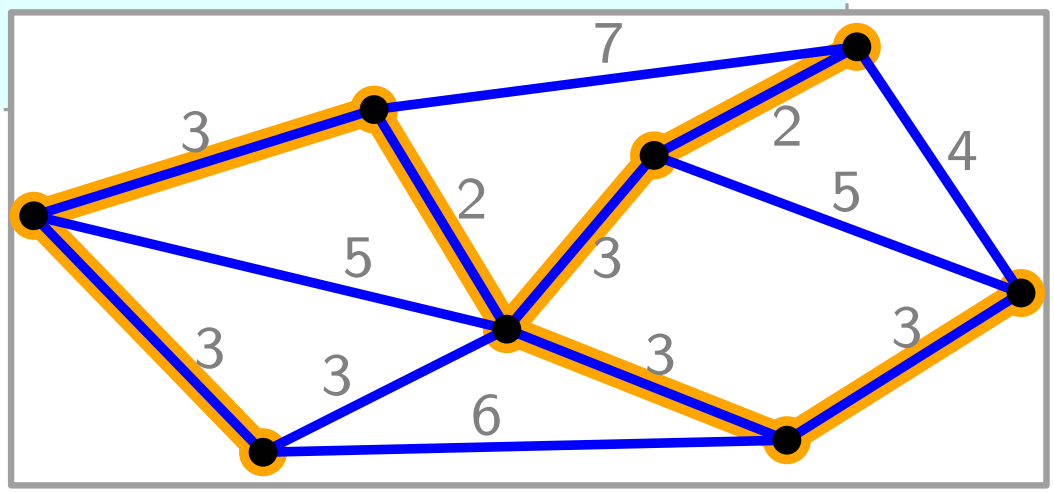
Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
    A = ∅
    foreach v ∈ V do
        MakeSet(v)
    Sortiere E nicht-absteigend nach Gewicht w
    foreach uv ∈ E do
        if FindSet(u) ≠ FindSet(v) then
            A = A ∪ {uv}
            Union(u, v)
    return A
```

Laufzeit?

$|V|\cdot\text{MakeSet} + (|V|-1)\cdot\text{Union} + 2|E|\cdot\text{FindSet} + \text{Sort}(E)$

$\in O(E \log V + E \log E)$



Der Algorithmus von Kruskal

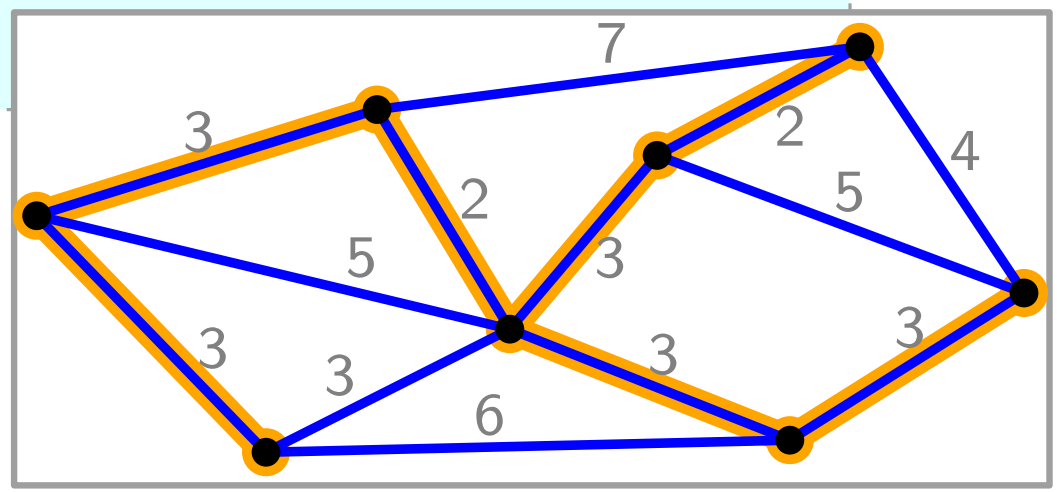
```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
    A = ∅
    foreach v ∈ V do
        MakeSet(v)
    Sortiere E nicht-absteigend nach Gewicht w
    foreach uv ∈ E do
        if FindSet(u) ≠ FindSet(v) then
            A = A ∪ {uv}
            Union(u, v)
    return A
```

Laufzeit?

$|V|\cdot\text{MakeSet} + (|V|-1)\cdot\text{Union} + 2|E|\cdot\text{FindSet} + \text{Sort}(E)$

$\in O(E \log V + E \log E)$

$=$

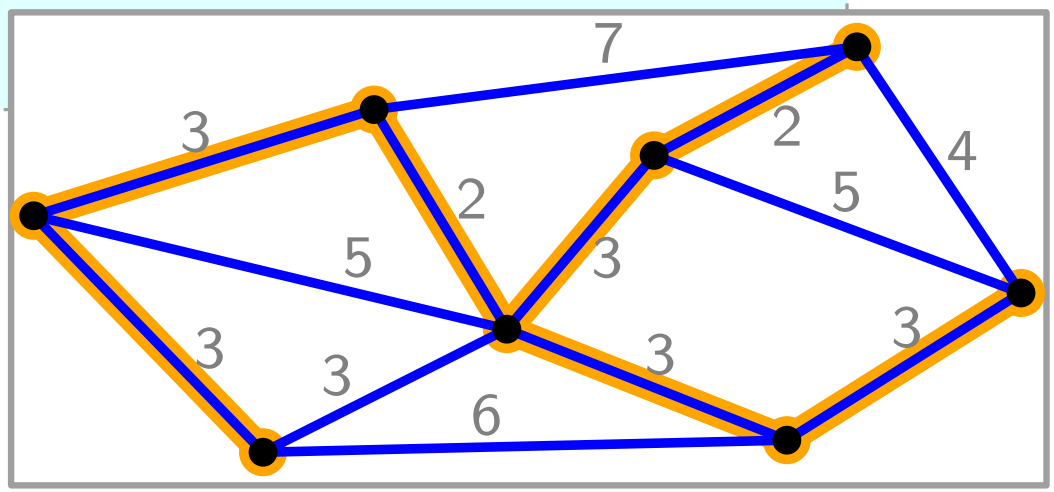


Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
    A = ∅
    foreach v ∈ V do
        MakeSet(v)
    Sortiere E nicht-absteigend nach Gewicht w
    foreach uv ∈ E do
        if FindSet(u) ≠ FindSet(v) then
            A = A ∪ {uv}
            Union(u, v)
    return A
```

Laufzeit?

$|V|\cdot\text{MakeSet} + (|V|-1)\cdot\text{Union}$
$+ 2|E|\cdot\text{FindSet} + \text{Sort}(E)$
$\in O(E \log V + E \log E)$
$= O(E \log V)$!

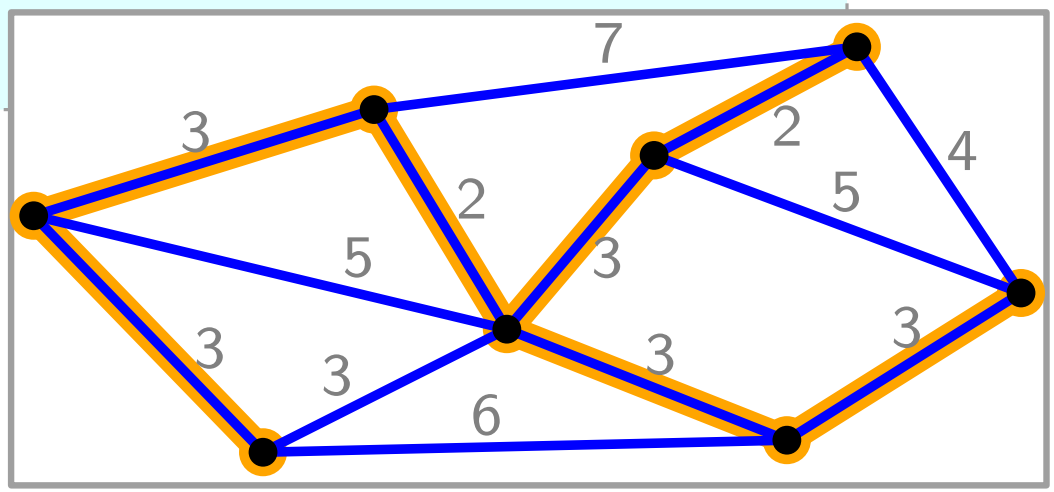


Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
    A = ∅
    foreach v ∈ V do
        MakeSet(v)
    Sortiere E nicht-absteigend nach Gewicht w
    foreach uv ∈ E do
        if FindSet(u) ≠ FindSet(v) then
            A = A ∪ {uv}
            Union(u, v)
    return A
```

Laufzeit?

$|V|\cdot\text{MakeSet} + (|V|-1)\cdot\text{Union}$
$+ 2|E|\cdot\text{FindSet} + \text{Sort}(E)$
$\in O(E \log V + E \log E)$
$= O(E \log V)$! Warum??

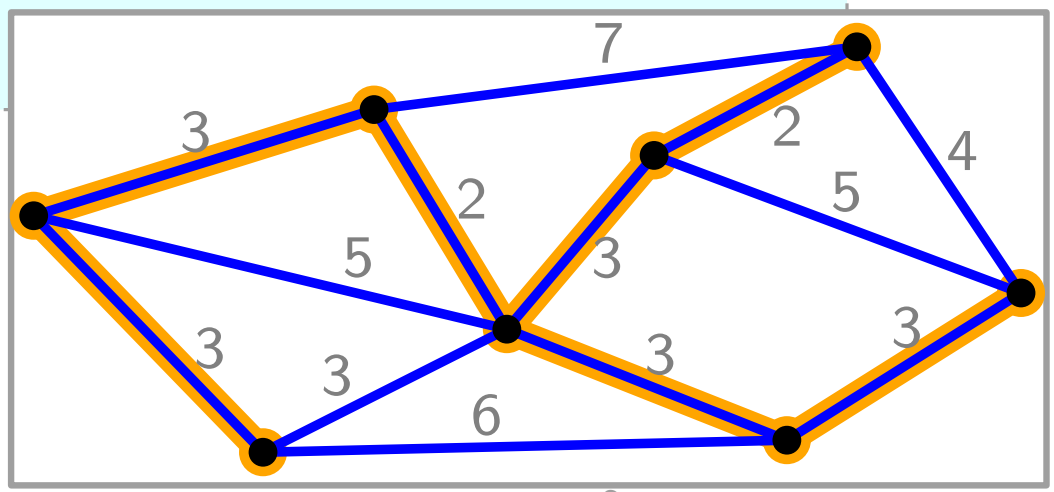


Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
    A = ∅
    foreach v ∈ V do
        MakeSet(v)
    Sortiere E nicht-absteigend nach Gewicht w
    foreach uv ∈ E do
        if FindSet(u) ≠ FindSet(v) then
            A = A ∪ {uv}
            Union(u, v)
    return A
```

Laufzeit?

$|V|\cdot\text{MakeSet} + (|V|-1)\cdot\text{Union}$
$+ 2|E|\cdot\text{FindSet} + \text{Sort}(E)$
$\in O(E \log V + E \log E)$
$= O(E \log V)$! Warum??



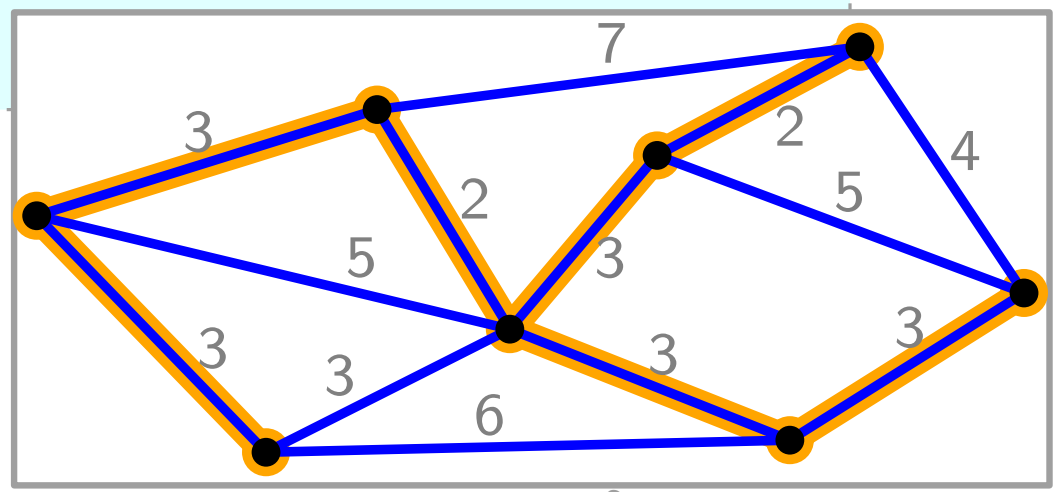
Weil $O(\log E) \subseteq O(\log V^2) = O(\log V)$.

Der Algorithmus von Kruskal

```
KruskalMST(WeightedUndirectedGraph G = (V, E), w)
    A = ∅
    foreach v ∈ V do
        MakeSet(v)
    Sortiere E nicht-absteigend nach Gewicht w
    foreach uv ∈ E do
        if FindSet(u) ≠ FindSet(v) then
            A = A ∪ {uv}
            Union(u, v)
    return A
```

Laufzeit?

$|V|\cdot\text{MakeSet} + (|V|-1)\cdot\text{Union}$
$+ 2|E|\cdot\text{FindSet} + \text{Sort}(E)$
$\in O(E \log V + E \log E)$
$= O(E \log V)$! Warum??



Weil $O(\log E) \subseteq O(\log V^2) = O(\log V)$. Ah...

Übersicht: Algo. für min. Spannbäume

Jarník-Prim

- geht (wie Dijkstra / BFS) wellenförmig von einem Startknoten aus

Kruskal

- bearbeitet Kanten nach aufsteigendem (genauer: nicht-absteig.) Gewicht

Übersicht: Algo. für min. Spannbäume

Jarník-Prim

- geht (wie Dijkstra / BFS) wellenförmig von einem Startknoten aus

Kruskal

- bearbeitet Kanten nach aufsteigendem (genauer: nicht-absteig.) Gewicht

Übersicht: Algo. für min. Spannbäume

Jarník-Prim

- geht (wie Dijkstra / BFS) wellenförmig von einem Startknoten aus
- aktuelle Kantenmenge zusammenhängend

Kruskal

- bearbeitet Kanten nach aufsteigendem (genauer: nicht-absteig.) Gewicht
- nach Einfügen der i. Kante gibt es $n - i$ Zshgskomp.

Übersicht: Algo. für min. Spannbäume

Jarník-Prim

- geht (wie Dijkstra / BFS) wellenförmig von einem Startknoten aus
- aktuelle Kantenmenge zusammenhängend
- Laufzeit $O(E + V \log V)$

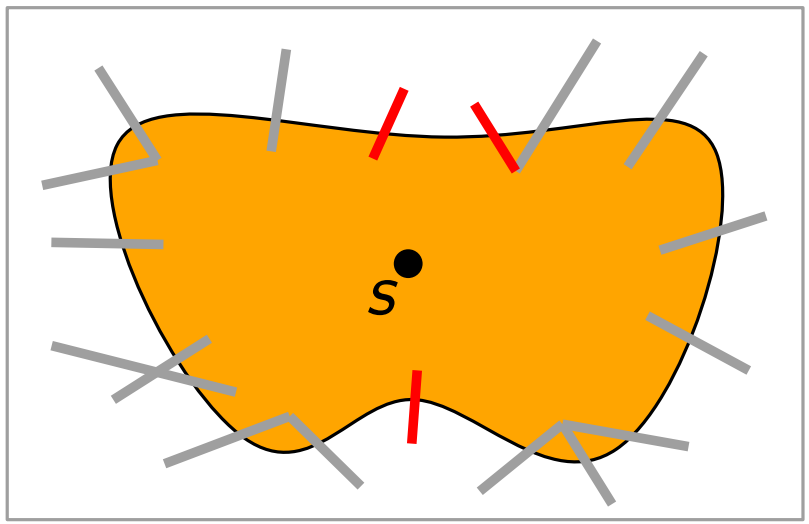
Kruskal

- bearbeitet Kanten nach aufsteigendem (genauer: nicht-absteig.) Gewicht
- nach Einfügen der i. Kante gibt es $n - i$ Zshgskomp.
- Laufzeit $O(E \log V)$

Übersicht: Algo. für min. Spannbäume

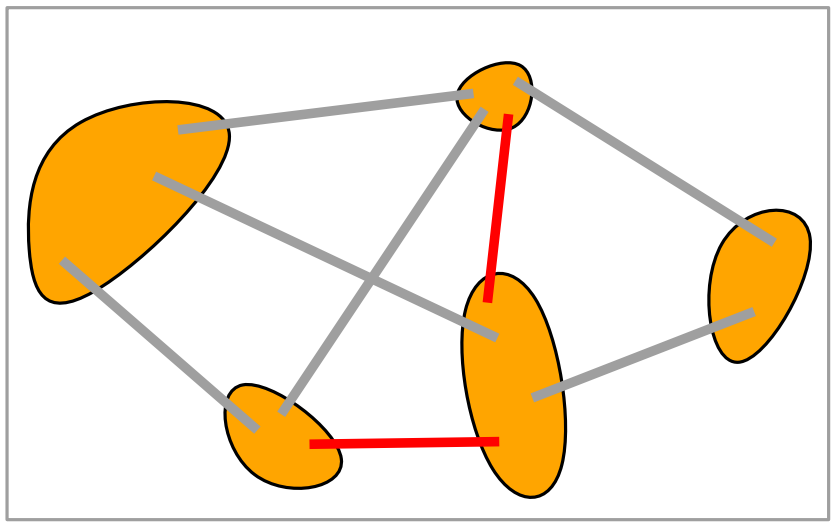
Jarník-Prim

- geht (wie Dijkstra / BFS) wellenförmig von einem Startknoten aus
- aktuelle Kantenmenge zusammenhängend
- Laufzeit $O(E + V \log V)$



Kruskal

- bearbeitet Kanten nach aufsteigendem (genauer: nicht-absteig.) Gewicht
- nach Einfügen der i. Kante gibt es $n - i$ Zshgskomp.
- Laufzeit $O(E \log V)$



Übersicht: Algo. für min. Spannbäume

Jarník-Prim

- geht (wie Dijkstra / BFS) wellenförmig von einem Startknoten aus
- aktuelle Kantenmenge zusammenhängend
- Laufzeit $O(E + V \log V)$

Kruskal

- bearbeitet Kanten nach aufsteigendem (genauer: nicht-absteig.) Gewicht
- nach Einfügen der i. Kante gibt es $n - i$ Zshgskomp.
- Laufzeit $O(E \log V)$

```
GenericMST(UndirectedConnectedGraph G, EdgeWeights w)
  A = ∅
  while |A| < |V| − 1 do
      // Invariante: A ist Teilmenge eines min. Spannbaums von G
      finde Kante uv, die sicher für A ist
      A = A ∪ {uv}
  return A
```

Übersicht: Algo. für min. Spannbäume

Jarník-Prim

- geht (wie Dijkstra / BFS) wellenförmig von einem Startknoten aus
- aktuelle Kantenmenge zusammenhängend
- Laufzeit $O(E + V \log V)$

Kruskal

- bearbeitet Kanten nach aufsteigendem (genauer: nicht-absteig.) Gewicht
- nach Einfügen der i. Kante gibt es $n - i$ Zshgskomp.
- Laufzeit $O(E \log V)$

```
GenericMST(UndirectedConnectedGraph G, EdgeWeights w)
  A = ∅
  while |A| < |V| − 1 do
      // Invariante: A ist Teilmenge eines min. Spannbaums von G
      finde Kante uv, die sicher für A ist
      A = A ∪ {uv}
  return A
```