SS 2021

Algorithmen und Datenstrukturen

11. Kapitel

Greedy-Algorithmen für Graphprobleme

Martin Dietzfelbinger

Juli 2021

11.1 Kürzeste Wege mit einem Startknoten: Der Algorithmus von Dijkstra

11.1 Kürzeste Wege mit einem Startknoten: Der Algorithmus von Dijkstra

Definition 11.1.1

1. Ein **gewichteter** Digraph $\boldsymbol{G = (V, E, c)}$ besteht aus einem Digraphen (V, E) und einer Funktion $c\colon E \to \mathbb{R}$, die jeder Kante (v, w) einen Wert $c(v, w)$ zuordnet.

11.1 Kürzeste Wege mit einem Startknoten: Der Algorithmus von Dijkstra

Definition 11.1.1

1. Ein **gewichteter** Digraph $\boldsymbol{G = (V, E, c)}$ besteht aus einem Digraphen (V, E) und einer Funktion $c\colon E \to \mathbb{R}$, die jeder Kante (v, w) einen Wert $c(v, w)$ zuordnet.

   c steht für „**cost**“; $c(v, w)$ kann als „Kosten“ oder „Länge“ oder „Gewicht“ der Kante (v, w) interpretiert werden.

11.1 Kürzeste Wege mit einem Startknoten: Der Algorithmus von Dijkstra

Definition 11.1.1

1. Ein **gewichteter** Digraph $\boldsymbol{G = (V, E, c)}$ besteht aus einem Digraphen (V, E) und einer Funktion $c\colon E \to \mathbb{R}$, die jeder Kante (v, w) einen Wert $c(v, w)$ zuordnet.

   c steht für „**cost**“; $c(v, w)$ kann als „Kosten“ oder „Länge“ oder „Gewicht“ der Kante (v, w) interpretiert werden.

2. Ein gerichteter Kantenzug $p = (v_0, v_1, \ldots, v_k)$ in G hat Kosten/Länge/Gewicht

$$\boldsymbol{c(p)} = \sum_{1 \leq i \leq k} c(v_{i-1}, v_i).$$

3. Der **(gerichtete) Abstand** von $v, w \in V$ ist

$$\boldsymbol{d(v,w)} := \min\{c(p) \mid p \text{ Kantenzug von } v \text{ nach } w\}$$

3. Der **(gerichtete) Abstand** von $v, w \in V$ ist

$$\boldsymbol{d(v,w)} := \min\{c(p) \mid p \text{ Kantenzug von } v \text{ nach } w\}$$

($= \boldsymbol{\infty}$, falls kein Kantenzug von v nach w existiert;

3. Der **(gerichtete) Abstand** von $v, w \in V$ ist

$$\boldsymbol{d(v,w)} := \min\{c(p) \mid p \text{ Kantenzug von } v \text{ nach } w\}$$

($= \boldsymbol{\infty}$, falls kein Kantenzug von v nach w existiert;
$= \boldsymbol{-\infty}$, falls es von v nach w Kantenzüge mit beliebig stark negativen Kosten gibt.)

3. Der **(gerichtete) Abstand** von $v, w \in V$ ist

$$\boldsymbol{d(v,w)} := \min\{c(p) \mid p \text{ Kantenzug von } v \text{ nach } w\}$$

($= \boldsymbol{\infty}$, falls kein Kantenzug von v nach w existiert;
$= \boldsymbol{-\infty}$, falls es von v nach w Kantenzüge mit beliebig stark negativen Kosten gibt.)

Klar: $d(v,v) \leq 0$ (wegen des Kantenzugs (v) mit Kosten 0).

3. Der **(gerichtete) Abstand** von $v, w \in V$ ist

$$\boldsymbol{d(v,w)} := \min\{c(p) \mid p \text{ Kantenzug von } v \text{ nach } w\}$$

($= \boldsymbol{\infty}$, falls kein Kantenzug von v nach w existiert;
$= \boldsymbol{-\infty}$, falls es von v nach w Kantenzüge mit beliebig stark negativen Kosten gibt.)

Klar: $d(v,v) \leq 0$ (wegen des Kantenzugs (v) mit Kosten 0).

Bemerkung

Wenn alle Kantengewichte ≥ 0 sind, gilt:

$d(v,w) =$ minimale Länge eines (einfachen) **Weges** von v nach w.

3. Der **(gerichtete) Abstand** von $v, w \in V$ ist

$$\boldsymbol{d(v,w)} := \min\{c(p) \mid p \text{ Kantenzug von } v \text{ nach } w\}$$

($= \boldsymbol{\infty}$, falls kein Kantenzug von v nach w existiert;
$= \boldsymbol{-\infty}$, falls es von v nach w Kantenzüge mit beliebig stark negativen Kosten gibt.)

Klar: $d(v,v) \leq 0$ (wegen des Kantenzugs (v) mit Kosten 0).

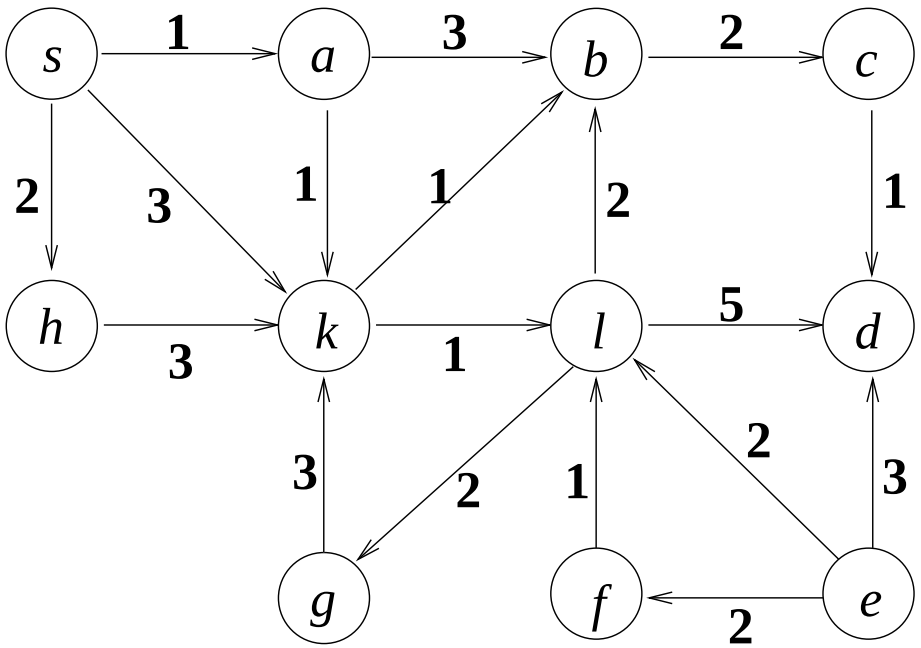
Bemerkung

Wenn alle Kantengewichte ≥ 0 sind, gilt:

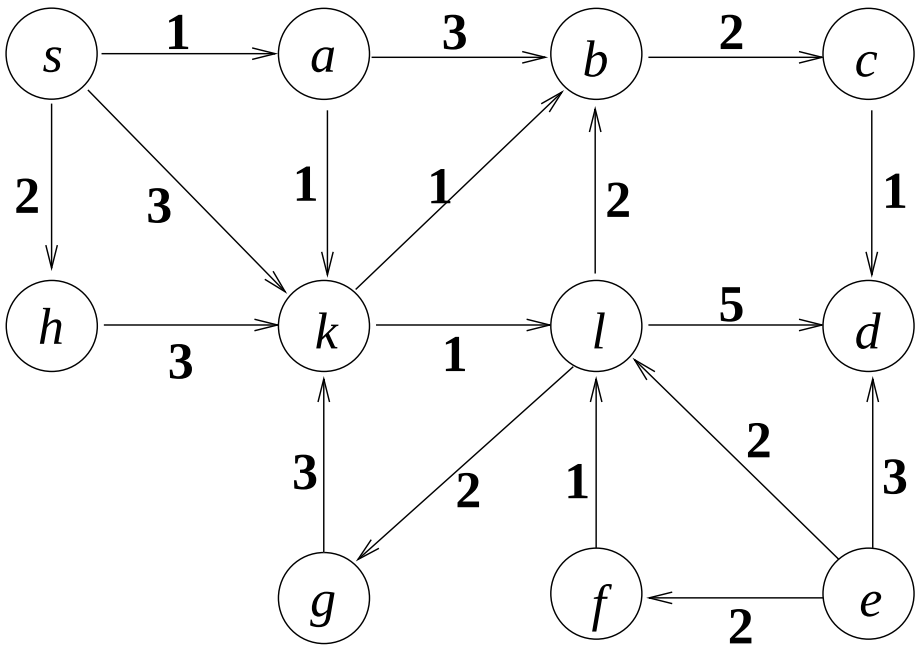
$$d(v,w) = \text{minimale Länge eines (einfachen) } \textbf{Weges} \text{ von } v \text{ nach } w.$$

(Man kann aus einem Kantenzug von v nach w Kreise herausschneiden, ohne die Länge zu vergrößern.)

Beispiel: Digraph mit nichtnegativen Kantenkosten.

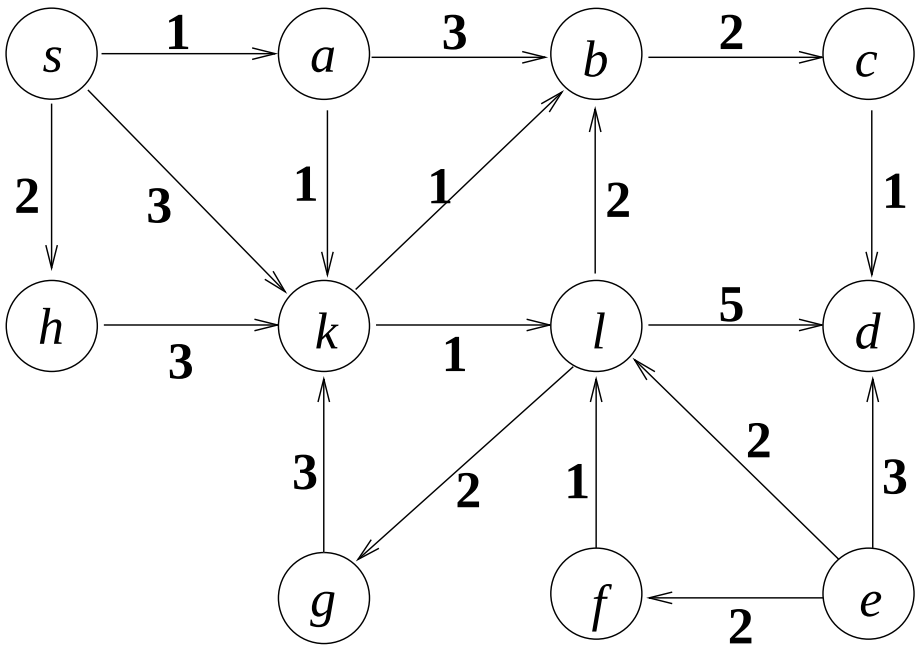


Beispiel: Digraph mit nichtnegativen Kantenkosten.



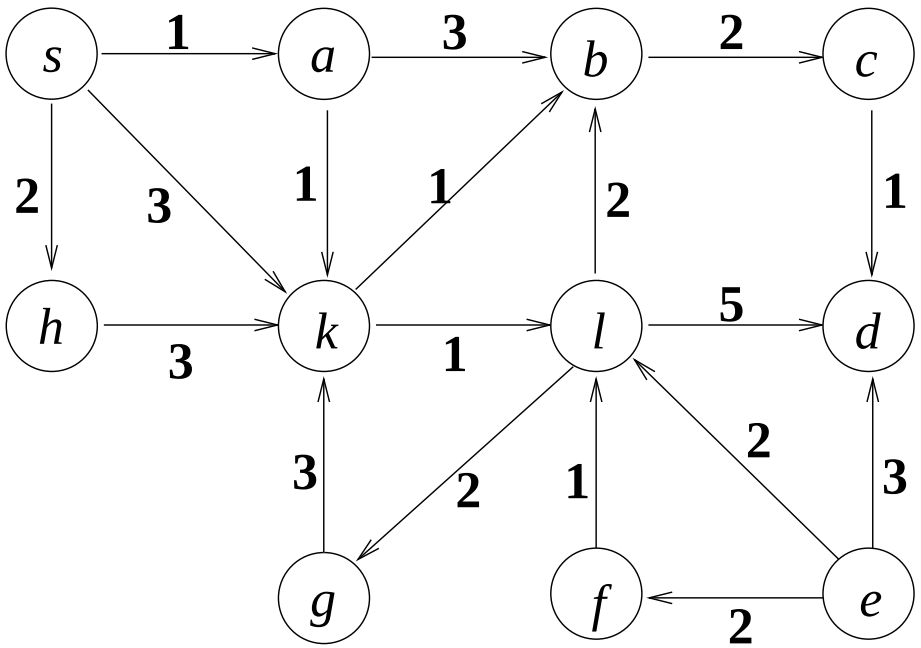
$c((s, a, b, c)) =$

Beispiel: Digraph mit nichtnegativen Kantenkosten.



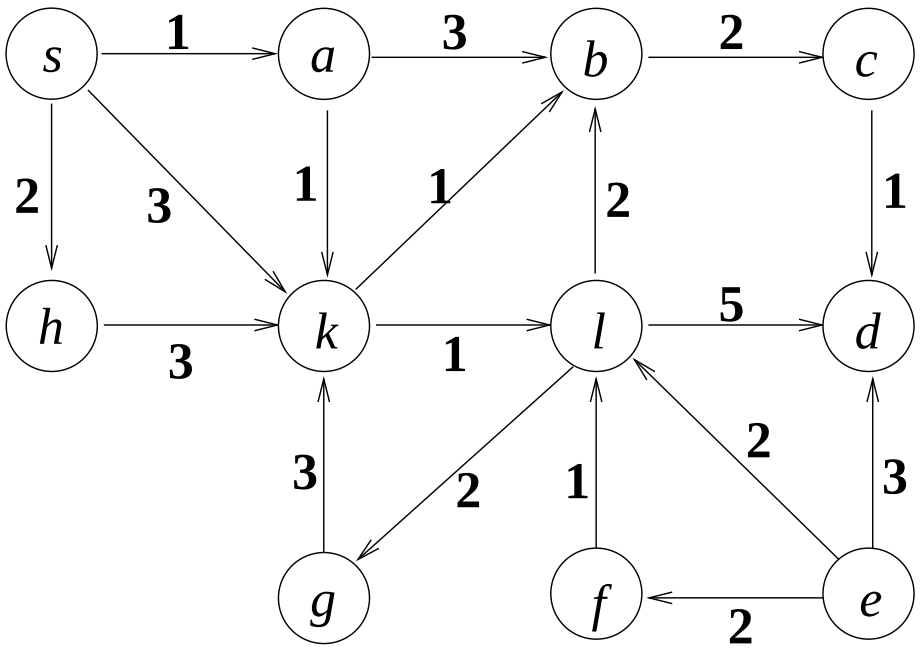
$c((s, a, b, c)) = 6$

Beispiel: Digraph mit nichtnegativen Kantenkosten.



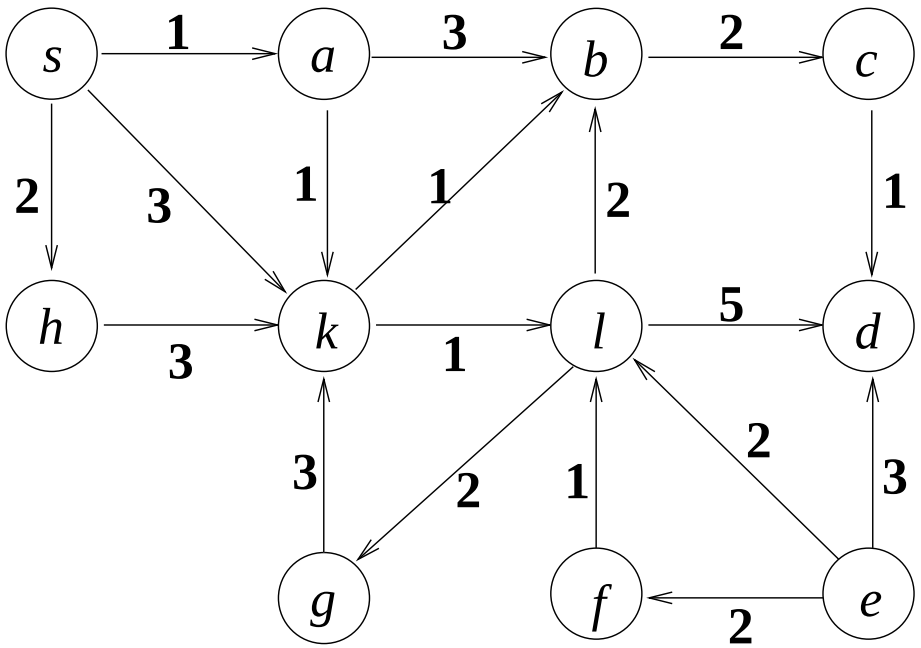
$c((s,a,b,c)) = 6, \quad c((s,a,k,b,c)) =$

Beispiel: Digraph mit nichtnegativen Kantenkosten.



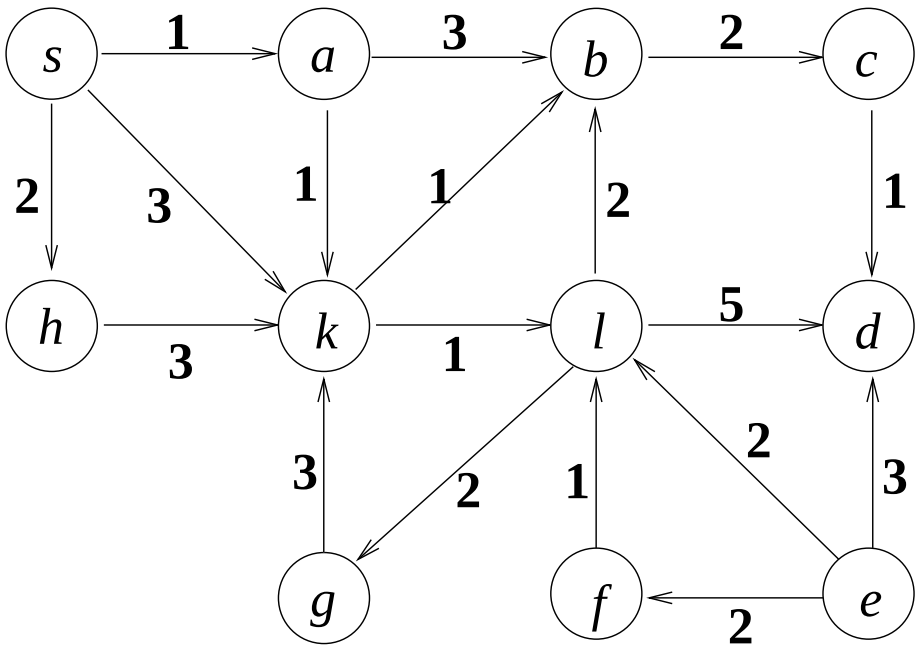
$c((s,a,b,c)) = 6, \quad c((s,a,k,b,c)) = 5$

Beispiel: Digraph mit nichtnegativen Kantenkosten.



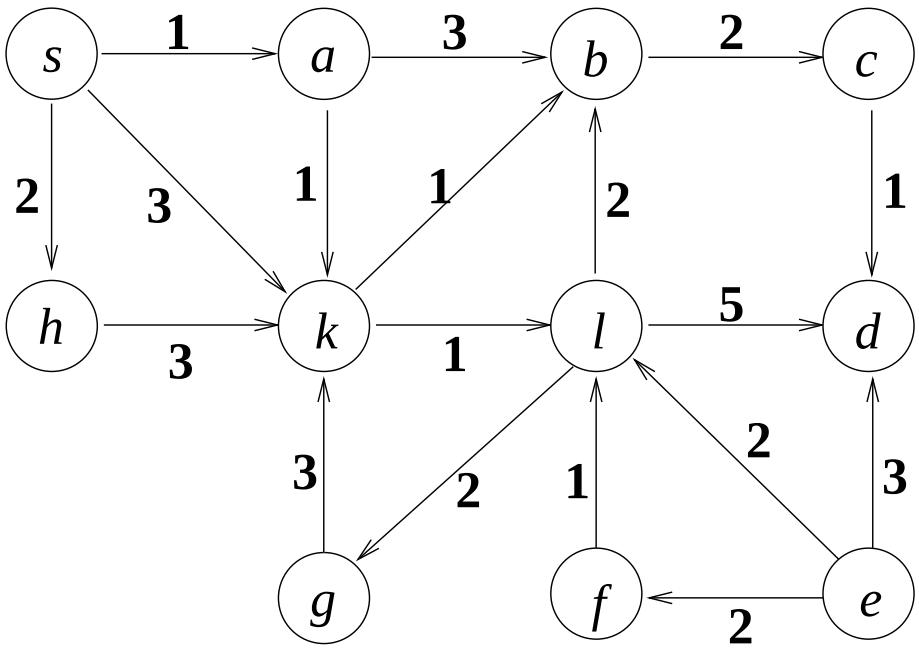
$c((s,a,b,c)) = 6, \quad c((s,a,k,b,c)) = 5, \quad d(s,c) =$

Beispiel: Digraph mit nichtnegativen Kantenkosten.



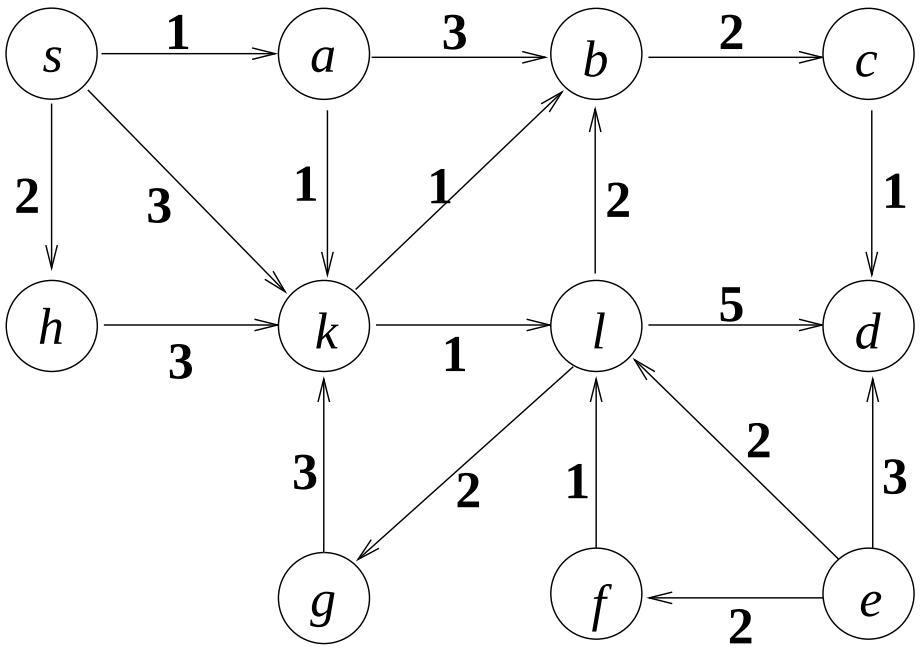
$c((s,a,b,c)) = 6, \quad c((s,a,k,b,c)) = 5, \quad d(s,c) = 5$

Beispiel: Digraph mit nichtnegativen Kantenkosten.



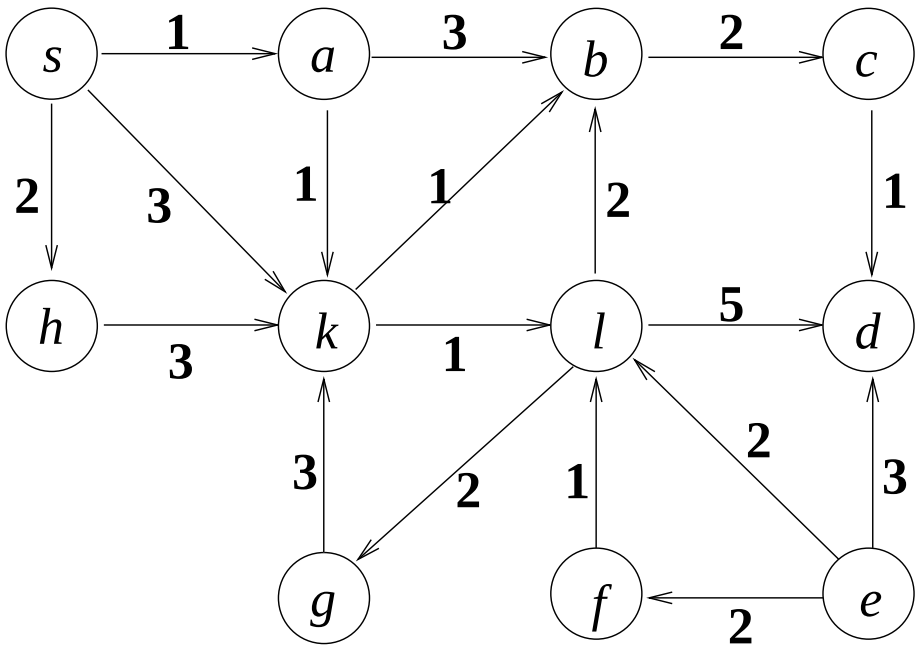
$c((s,a,b,c)) = 6, \quad c((s,a,k,b,c)) = 5, \quad d(s,c) = 5;$
$d(s,s) =$

Beispiel: Digraph mit nichtnegativen Kantenkosten.



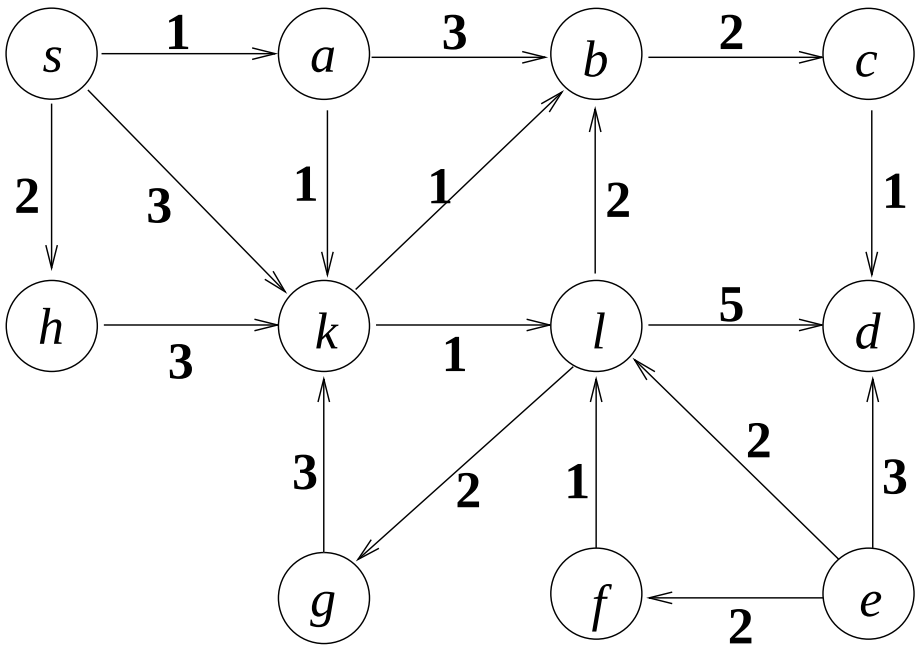
$c((s,a,b,c)) = 6, \quad c((s,a,k,b,c)) = 5, \quad d(s,c) = 5;$
$d(s,s) = 0$

Beispiel: Digraph mit nichtnegativen Kantenkosten.



$c((s,a,b,c)) = 6, \quad c((s,a,k,b,c)) = 5, \quad d(s,c) = 5;$
$d(s,s) = 0;$
$d(s,e) = d(s,f) =$

Beispiel: Digraph mit nichtnegativen Kantenkosten.



$c((s,a,b,c)) = 6, \quad c((s,a,k,b,c)) = 5, \quad d(s,c) = 5;$
$d(s,s) = 0;$
$d(s,e) = d(s,f) = \infty.$

Hier betrachten wir einen Algorithmus für das Problem

„Single-Source-Shortest-Paths (SSSP)“

(Kürzeste Wege von einem Startknoten aus).

Hier betrachten wir einen Algorithmus für das Problem

„Single-Source-Shortest-Paths (SSSP)“

(Kürzeste Wege von einem Startknoten aus).

Gegeben:

Gewichteter Digraph $G = (V, E, c)$ mit Kantenkosten $c(v, w) \geq 0$ und $s \in V$.

Gesucht:

Für jedes $v \in V$ der Abstand $d(s, v)$ und im Fall $d(s, v) < \infty$ ein Weg von s nach v der Länge $d(s, v)$.

Hier betrachten wir einen Algorithmus für das Problem

„Single-Source-Shortest-Paths (SSSP)“

(Kürzeste Wege von einem Startknoten aus).

Gegeben:

Gewichteter Digraph $G = (V, E, c)$ mit Kantenkosten $c(v, w) \geq 0$ und $s \in V$.

Gesucht:

Für jedes $v \in V$ der Abstand $d(s, v)$ und im Fall $d(s, v) < \infty$ ein Weg von s nach v der Länge $d(s, v)$.

Der **Algorithmus von Dijkstra** löst dieses Problem.

Hier betrachten wir einen Algorithmus für das Problem

„Single-Source-Shortest-Paths (SSSP)“

(Kürzeste Wege von einem Startknoten aus).

Gegeben:

Gewichteter Digraph $G = (V, E, c)$ mit Kantenkosten $c(v, w) \geq 0$ und $s \in V$.

Gesucht:

Für jedes $v \in V$ der Abstand $d(s, v)$ und im Fall $d(s, v) < \infty$ ein Weg von s nach v der Länge $d(s, v)$.

Der **Algorithmus von Dijkstra** löst dieses Problem.

(Aussprache: „Daik-stra“.)
Edsger W. Dijkstra, 1930–2002, niederländischer Informatiker, Pionier der „Strukturierten Programmierung“, Erfinder des Semaphorkonzepts für die Synchronisation von Prozessen, Turingpreis 1972.

(Zündende) Idee:

Eine Kante (v, w) wird als „Einbahnstraßen-Zündschnur“ mit *Länge* $c(v, w)$ gedacht.

(Zündende) Idee:

Eine Kante (v, w) wird als „Einbahnstraßen-Zündschnur“ mit *Länge* $c(v, w)$ gedacht.

Die Zündschnüre brennen mit konstanter Geschwindigkeit 1 pro Sekunde.

(Zündende) Idee:

Eine Kante (v, w) wird als „Einbahnstraßen-Zündschnur“ mit *Länge* $c(v, w)$ gedacht.

Die Zündschnüre brennen mit konstanter Geschwindigkeit 1 pro Sekunde.

Wenn das Feuer einen Knoten v erstmals erreicht, zum Zeitpunkt $t = t_v$, fangen alle Zündschnüre, die von v ausgehen, an zu brennen.

(Zündende) Idee:

Eine Kante (v, w) wird als „Einbahnstraßen-Zündschnur“ mit *Länge* $c(v, w)$ gedacht.

Die Zündschnüre brennen mit konstanter Geschwindigkeit 1 pro Sekunde.

Wenn das Feuer einen Knoten v erstmals erreicht, zum Zeitpunkt $t = t_v$, fangen alle Zündschnüre, die von v ausgehen, an zu brennen.

Wenn später oder gleichzeitig das Feuer über andere Schnüre nochmals bei v ankommt, passiert nichts weiter.

(Zündende) Idee:

Eine Kante (v, w) wird als „Einbahnstraßen-Zündschnur“ mit *Länge* $c(v, w)$ gedacht.

Die Zündschnüre brennen mit konstanter Geschwindigkeit 1 pro Sekunde.

Wenn das Feuer einen Knoten v erstmals erreicht, zum Zeitpunkt $t = t_v$, fangen alle Zündschnüre, die von v ausgehen, an zu brennen.

Wenn später oder gleichzeitig das Feuer über andere Schnüre nochmals bei v ankommt, passiert nichts weiter.

Zum Zeitpunkt $t_s = 0$ halten wir ein Zündholz an den Knoten s.

(Zündende) Idee:

Eine Kante (v, w) wird als „Einbahnstraßen-Zündschnur“ mit *Länge* $c(v, w)$ gedacht.

Die Zündschnüre brennen mit konstanter Geschwindigkeit 1 pro Sekunde.

Wenn das Feuer einen Knoten v erstmals erreicht, zum Zeitpunkt $t = t_v$, fangen alle Zündschnüre, die von v ausgehen, an zu brennen.

Wenn später oder gleichzeitig das Feuer über andere Schnüre nochmals bei v ankommt, passiert nichts weiter.

Zum Zeitpunkt $t_s = 0$ halten wir ein Zündholz an den Knoten s.

Veranschaulichung: Auf späteren Folien.

(Zündende) Idee:

Eine Kante (v, w) wird als „Einbahnstraßen-Zündschnur“ mit *Länge* $c(v, w)$ gedacht.

Die Zündschnüre brennen mit konstanter Geschwindigkeit 1 pro Sekunde.

Wenn das Feuer einen Knoten v erstmals erreicht, zum Zeitpunkt $t = t_v$, fangen alle Zündschnüre, die von v ausgehen, an zu brennen.

Wenn später oder gleichzeitig das Feuer über andere Schnüre nochmals bei v ankommt, passiert nichts weiter.

Zum Zeitpunkt $t_s = 0$ halten wir ein Zündholz an den Knoten s.

Veranschaulichung: Auf späteren Folien. **Orange**: Schon erreichte Knoten.

(Zündende) Idee:

Eine Kante (v, w) wird als „Einbahnstraßen-Zündschnur“ mit *Länge* $c(v, w)$ gedacht.

Die Zündschnüre brennen mit konstanter Geschwindigkeit 1 pro Sekunde.

Wenn das Feuer einen Knoten v erstmals erreicht, zum Zeitpunkt $t = t_v$, fangen alle Zündschnüre, die von v ausgehen, an zu brennen.

Wenn später oder gleichzeitig das Feuer über andere Schnüre nochmals bei v ankommt, passiert nichts weiter.

Zum Zeitpunkt $t_s = 0$ halten wir ein Zündholz an den Knoten s.

Veranschaulichung: Auf späteren Folien. **Orange**: Schon erreichte Knoten.

Zahlen in Knoten: Wann erreicht das Feuer den Knoten, *nach aktuellem Stand*?

(Zündende) Idee:

Eine Kante (v, w) wird als „Einbahnstraßen-Zündschnur“ mit *Länge* $c(v, w)$ gedacht.

Die Zündschnüre brennen mit konstanter Geschwindigkeit 1 pro Sekunde.

Wenn das Feuer einen Knoten v erstmals erreicht, zum Zeitpunkt $t = t_v$, fangen alle Zündschnüre, die von v ausgehen, an zu brennen.

Wenn später oder gleichzeitig das Feuer über andere Schnüre nochmals bei v ankommt, passiert nichts weiter.

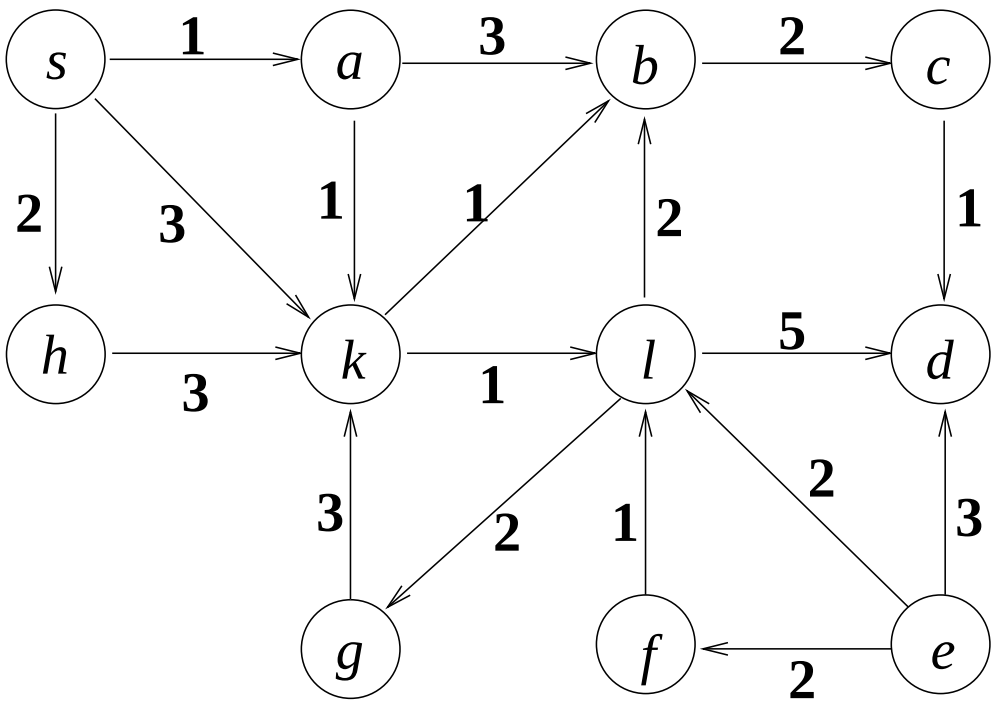
Zum Zeitpunkt $t_s = 0$ halten wir ein Zündholz an den Knoten s.

Veranschaulichung: Auf späteren Folien. **Orange**: Schon erreichte Knoten.

Zahlen in Knoten: Wann erreicht das Feuer den Knoten, *nach aktuellem Stand*?

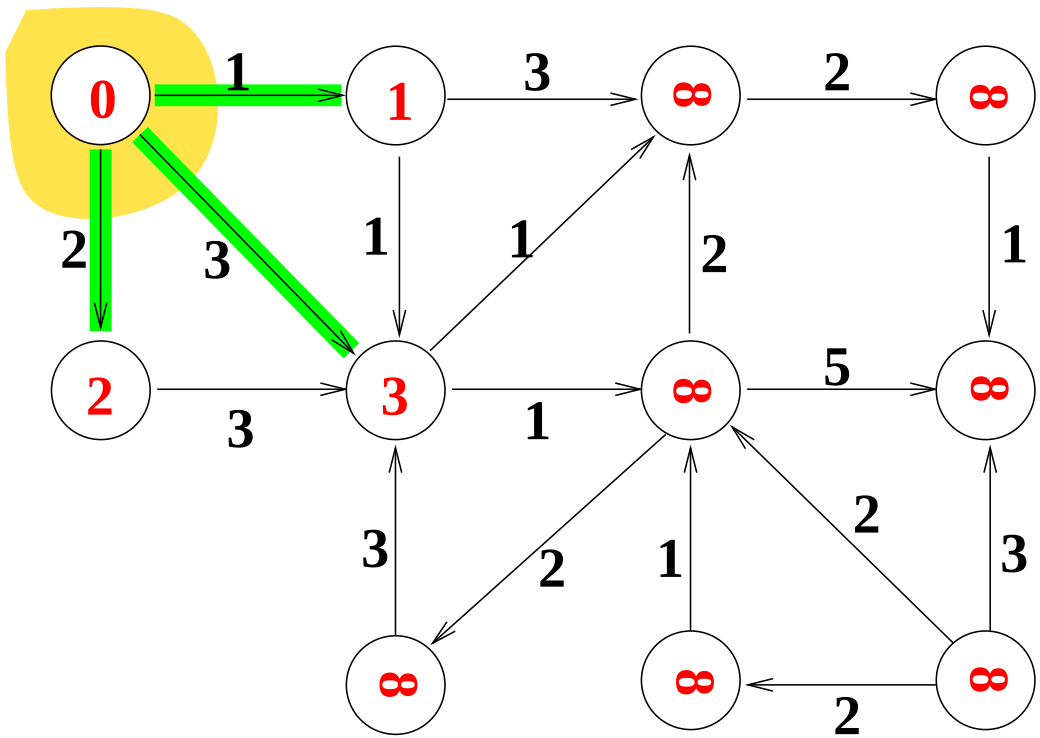
Über **grüne** Kanten werden bisher unerreichte Knoten **nach aktuellem Stand erstmals** erreicht. (Diese Kanten muss man also beobachten.)

Ablauf des Algorithmus von Dijkstra



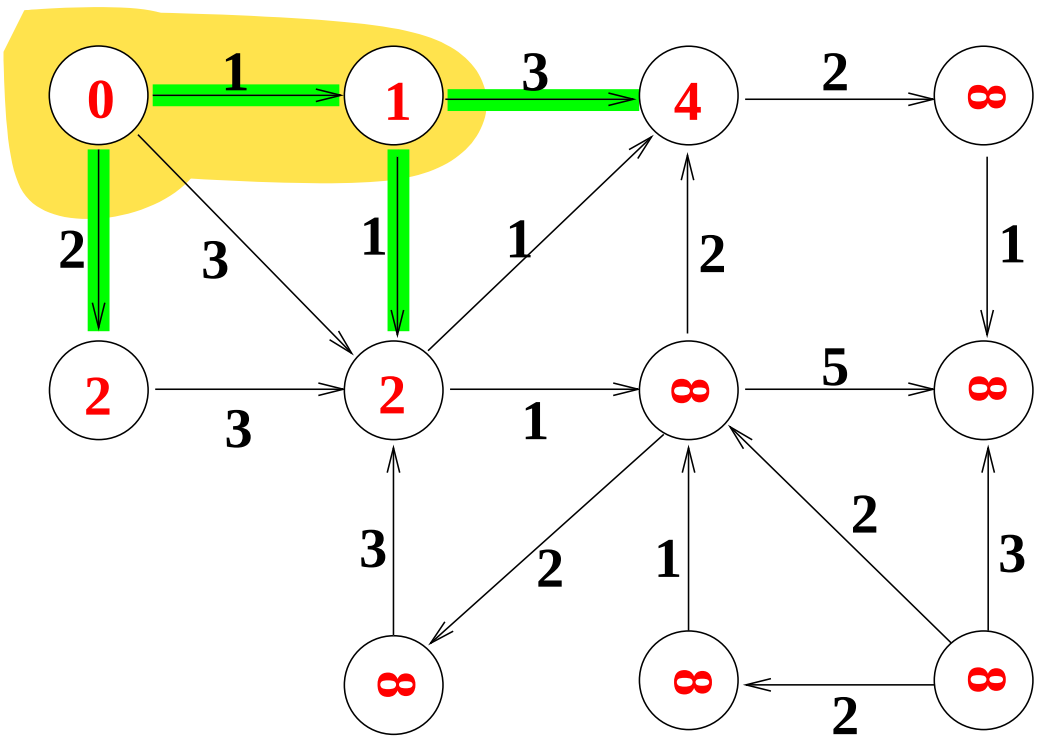
Der Ausgangsgraph.

Ablauf des Algorithmus von Dijkstra



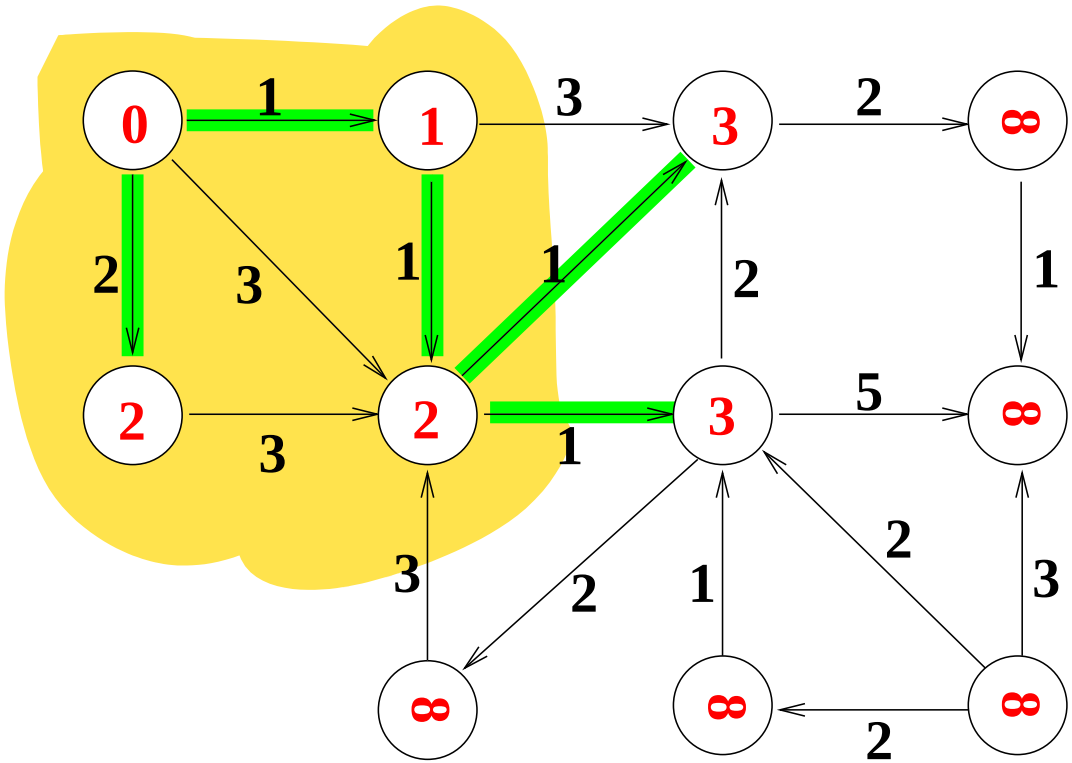
s wird zur Zeit $t_s = 0$ angezündet, **brennende Schnüre** mit Erstankunft.

Ablauf des Algorithmus von Dijkstra



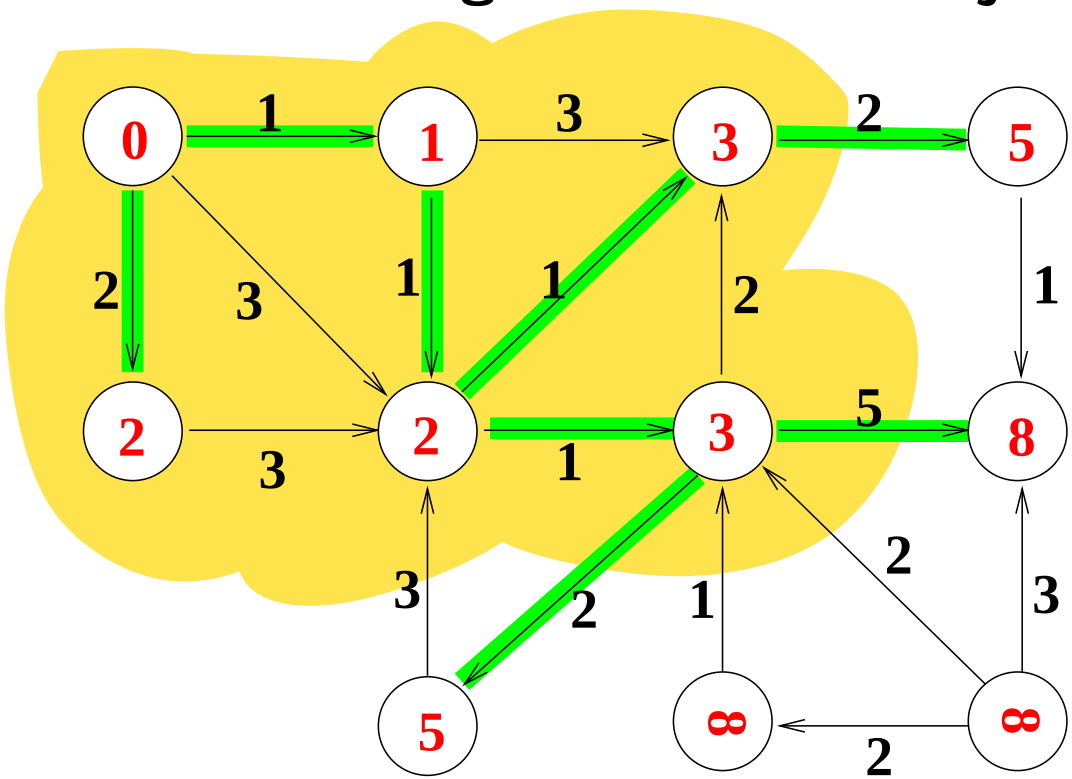
Zeitpunkt 1, neuer Knoten erreicht, neue Schnüre.

Ablauf des Algorithmus von Dijkstra



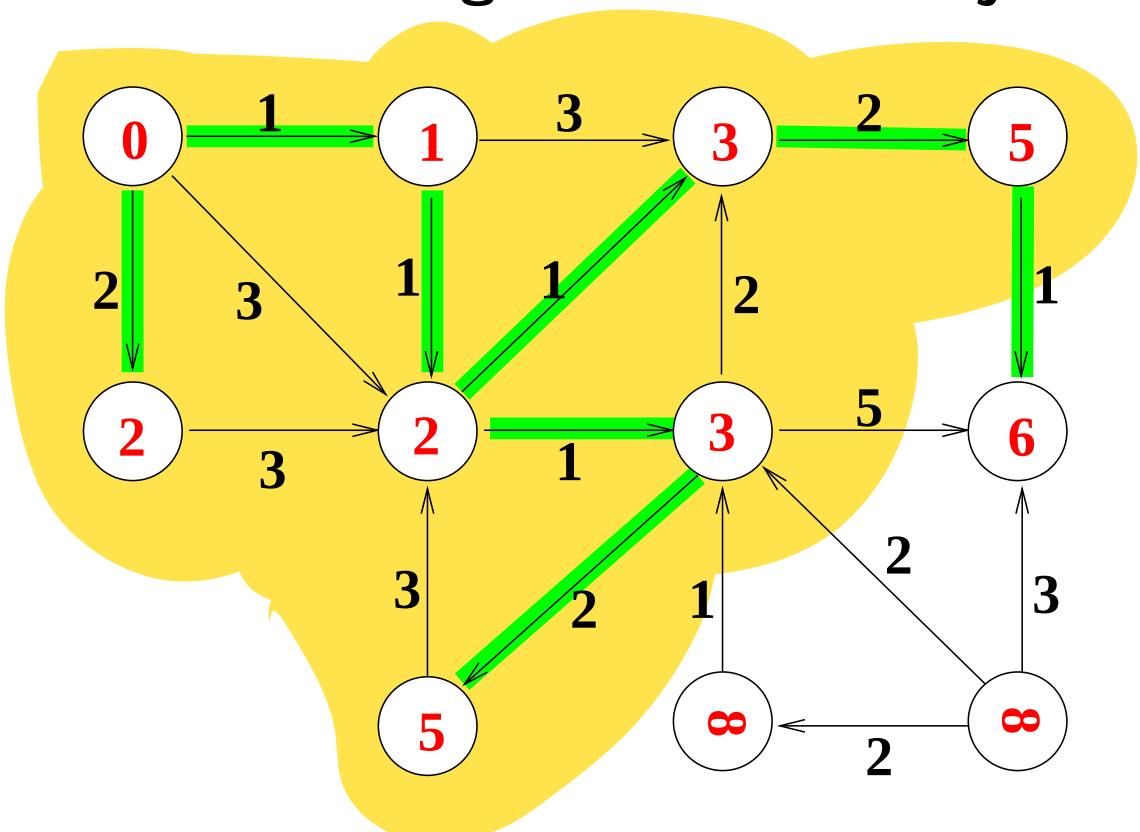
Zeitpunkt 2, zwei neue Knoten erreicht, neue Schnüre.

Ablauf des Algorithmus von Dijkstra



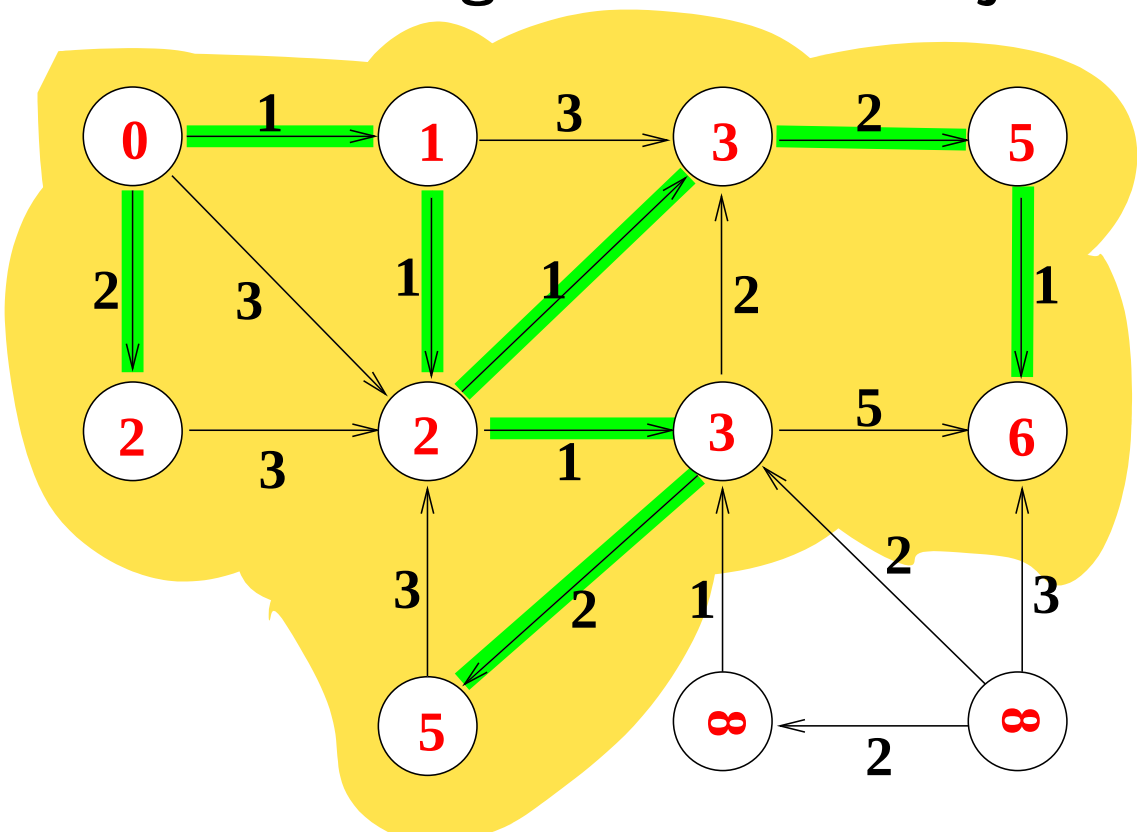
Zeitpunkt 3, zwei neue Knoten erreicht, neue Schnüre.

Ablauf des Algorithmus von Dijkstra



Zeitpunkt 5, zwei neue Knoten erreicht, neue Schnüre.

Ablauf des Algorithmus von Dijkstra



Zeitpunkt 6, neuer Knoten erreicht, keine neue Schnur.

„Klar“ **(??)**:

Das Feuer erreicht den Knoten v genau zum Zeitpunkt $t_v = d(s, v)$.

„Klar“ **(??)**:

Das Feuer erreicht den Knoten v genau zum Zeitpunkt $t_v = d(s, v)$.

Denn: Der Zeitraum $[0, d(s, v)]$ genügt, damit das Feuer einen kürzesten Weg von s nach v durchwandern kann, und es kann nicht schneller gehen.

„Klar“ **(??)**:

Das Feuer erreicht den Knoten v genau zum Zeitpunkt $t_v = d(s, v)$.

Denn: Der Zeitraum $[0, d(s, v)]$ genügt, damit das Feuer einen kürzesten Weg von s nach v durchwandern kann, und es kann nicht schneller gehen.

Wir bilden diese Idee nun algorithmisch nach.

Wichtige Beobachtung: Nur die n Zeitpunkte $t_v = d(s, v)$, $v \in V$, sind interessant.

Dazwischen wandert das Feuer irgendwelche Kanten entlang, ohne dass im Prinzip viel passiert (selbst wenn ein schon erreichter Knoten nochmals erreicht wird).

„Klar“ **(??)**:

Das Feuer erreicht den Knoten v genau zum Zeitpunkt $t_v = d(s, v)$.

Denn: Der Zeitraum $[0, d(s, v)]$ genügt, damit das Feuer einen kürzesten Weg von s nach v durchwandern kann, und es kann nicht schneller gehen.

Wir bilden diese Idee nun algorithmisch nach.

Wichtige Beobachtung: Nur die n Zeitpunkte $t_v = d(s, v)$, $v \in V$, sind interessant.

Dazwischen wandert das Feuer irgendwelche Kanten entlang, ohne dass im Prinzip viel passiert (selbst wenn ein schon erreichter Knoten nochmals erreicht wird).

O.B.d.A.: $V = \{1, \ldots, n\}$.

Der Algorithmus arbeitet in bis zu n Runden, eine für jeden erreichbaren Knoten.

„Klar“ **(??)**:

Das Feuer erreicht den Knoten v genau zum Zeitpunkt $t_v = d(s, v)$.

Denn: Der Zeitraum $[0, d(s, v)]$ genügt, damit das Feuer einen kürzesten Weg von s nach v durchwandern kann, und es kann nicht schneller gehen.

Wir bilden diese Idee nun algorithmisch nach.

Wichtige Beobachtung: Nur die n Zeitpunkte $t_v = d(s, v)$, $v \in V$, sind interessant.

Dazwischen wandert das Feuer irgendwelche Kanten entlang, ohne dass im Prinzip viel passiert (selbst wenn ein schon erreichter Knoten nochmals erreicht wird).

O.B.d.A.: $V = \{1, \ldots, n\}$.

Der Algorithmus arbeitet in bis zu n Runden, eine für jeden erreichbaren Knoten.

V ist stets in zwei disjunkte Mengen S (**orange**) und $V - S$ (weiß) zerlegt.

„Klar“ **(??)**:

Das Feuer erreicht den Knoten v genau zum Zeitpunkt $t_v = d(s, v)$.

Denn: Der Zeitraum $[0, d(s, v)]$ genügt, damit das Feuer einen kürzesten Weg von s nach v durchwandern kann, und es kann nicht schneller gehen.

Wir bilden diese Idee nun algorithmisch nach.

Wichtige Beobachtung: Nur die n Zeitpunkte $t_v = d(s, v)$, $v \in V$, sind interessant.

Dazwischen wandert das Feuer irgendwelche Kanten entlang, ohne dass im Prinzip viel passiert (selbst wenn ein schon erreichter Knoten nochmals erreicht wird).

O.B.d.A.: $V = \{1, \ldots, n\}$.

Der Algorithmus arbeitet in bis zu n Runden, eine für jeden erreichbaren Knoten.

V ist stets in zwei disjunkte Mengen S (**orange**) und $V - S$ (weiß) zerlegt.

In jeder Runde wächst S um einen Knoten.

„Klar“ **(??)**:

Das Feuer erreicht den Knoten v genau zum Zeitpunkt $t_v = d(s, v)$.

Denn: Der Zeitraum $[0, d(s, v)]$ genügt, damit das Feuer einen kürzesten Weg von s nach v durchwandern kann, und es kann nicht schneller gehen.

Wir bilden diese Idee nun algorithmisch nach.

Wichtige Beobachtung: Nur die n Zeitpunkte $t_v = d(s, v)$, $v \in V$, sind interessant.

Dazwischen wandert das Feuer irgendwelche Kanten entlang, ohne dass im Prinzip viel passiert (selbst wenn ein schon erreichter Knoten nochmals erreicht wird).

O.B.d.A.: $V = \{1, \ldots, n\}$.

Der Algorithmus arbeitet in bis zu n Runden, eine für jeden erreichbaren Knoten.

V ist stets in zwei disjunkte Mengen S (**orange**) und $V - S$ (weiß) zerlegt.

In jeder Runde wächst S um einen Knoten.

Anfangs: $S \leftarrow \emptyset$.

Array `dist`$[1..n]$ speichert Zeiten.

Array `dist`$[1..n]$ speichert Zeiten.

Für $v \in S$: `dist`$[v] = d(s, v)$ $\quad (= t_v)$.

Array `dist[1..n]` speichert Zeiten.

Für $v \in S$: `dist[v]` $= d(s, v)$ $\quad (= t_v)$.

Für $w \notin S$: `dist[w]` $= \min\{$`dist[v]` $+ c(v, w) \mid v \in S, (v, w) \in E\}$.

Array $\texttt{dist}[1..n]$ speichert Zeiten.

Für $v \in S$: $\texttt{dist}[v] = d(s,v) \quad (= t_v)$.

Für $w \notin S$: $\texttt{dist}[w] = \min\{\texttt{dist}[v] + c(v,w) \mid v \in S, (v,w) \in E\}$.

(Der Zeitpunkt, zu dem das Feuer *nach gegenwärtigem Stand der Dinge* Knoten w erreichen wird.

Array $\texttt{dist}[1..n]$ speichert Zeiten.

Für $v \in S$: $\texttt{dist}[v] = d(s,v) \quad (= t_v)$.

Für $w \notin S$: $\texttt{dist}[w] = \min\{\texttt{dist}[v] + c(v,w) \mid v \in S, (v,w) \in E\}$.

(Der Zeitpunkt, zu dem das Feuer *nach gegenwärtigem Stand der Dinge* Knoten w erreichen wird. Wenn es keine Kante von S nach w gibt, ist $\texttt{dist}[w] = \infty$.)

Array $\texttt{dist}[1..n]$ speichert Zeiten.

Für $v \in S$: $\texttt{dist}[v] = d(s,v) \quad (= t_v)$.

Für $w \notin S$: $\texttt{dist}[w] = \min\{\texttt{dist}[v] + c(v,w) \mid v \in S, (v,w) \in E\}$.

(Der Zeitpunkt, zu dem das Feuer *nach gegenwärtigem Stand der Dinge* Knoten w erreichen wird. Wenn es keine Kante von S nach w gibt, ist $\texttt{dist}[w] = \infty$.)

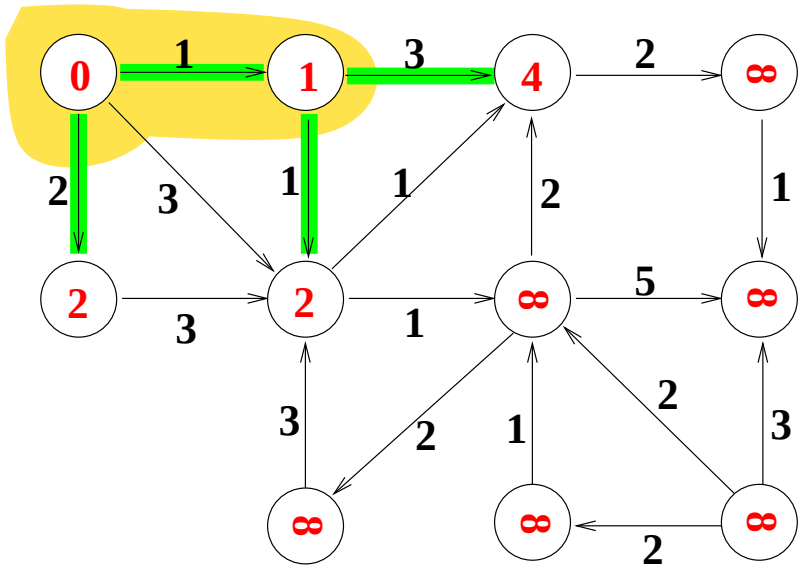
Initialisierung:

$S \leftarrow \emptyset$.

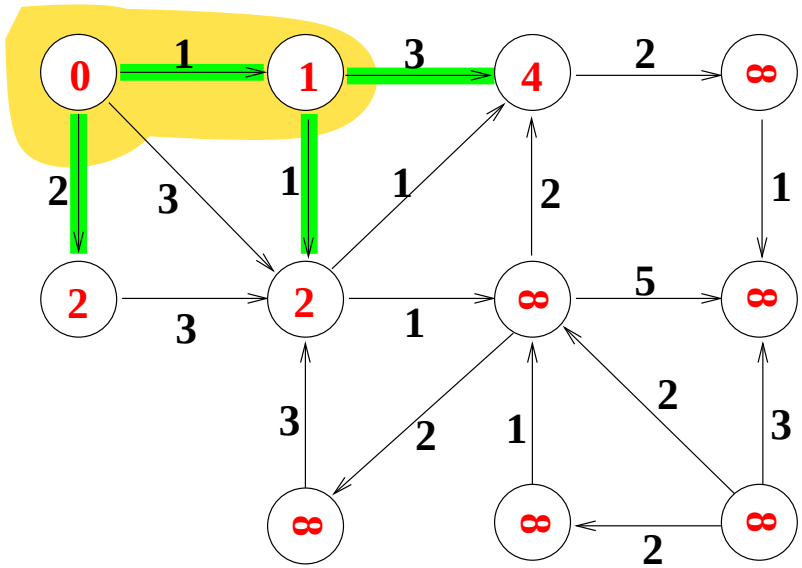
$\texttt{dist}[s] \leftarrow 0$.

Für alle $w \neq s$: $\texttt{dist}[w] \leftarrow \infty$.

Beispiel: **Orange: S.**

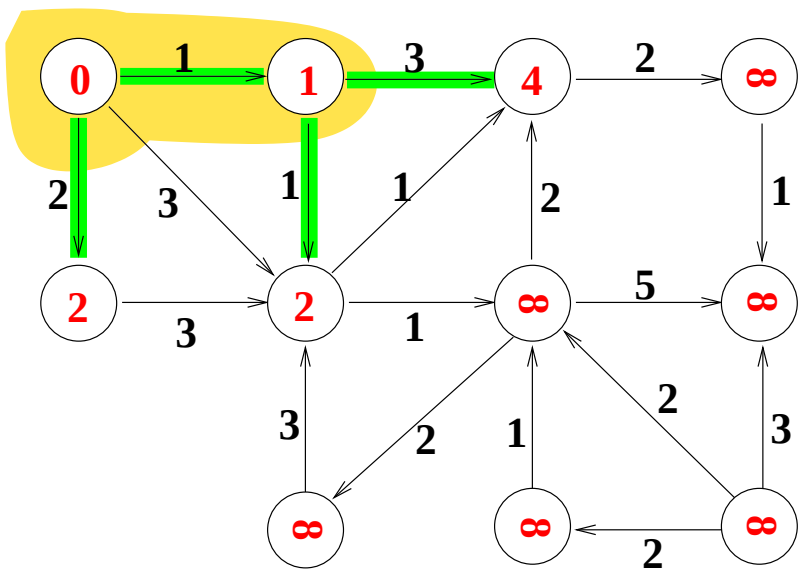


Beispiel: **Orange: S**.



Rote Zahlen in den Knoten sind die `dist[·]`-Werte.

Beispiel: **Orange:** S.

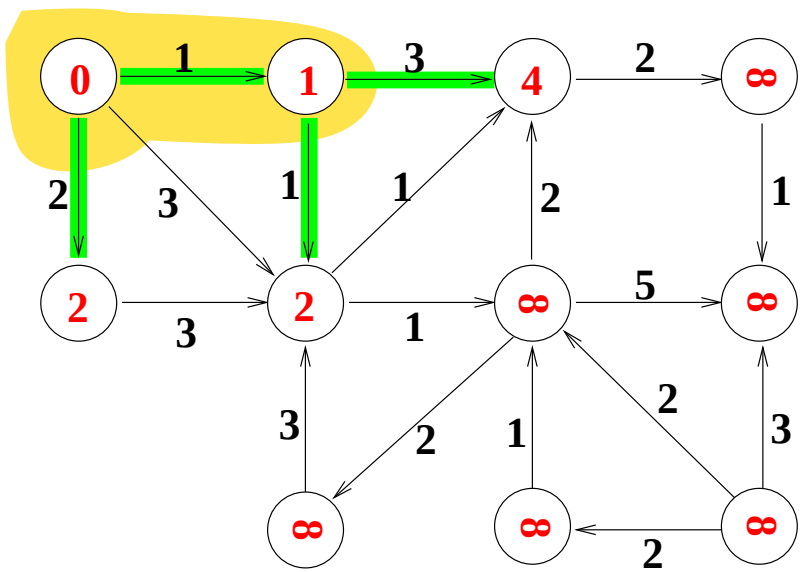


Rote Zahlen in den Knoten sind die `dist[·]`-Werte.

Runde: Bestimme den Knoten $u \in V - S$, der `dist[`w`]`, $w \in V - S$, minimiert.

(Hier: Einer der Knoten mit dist-Wert 2. Wenn es mehrere gibt, wähle einen beliebigen davon.)

Beispiel: **Orange:** S.



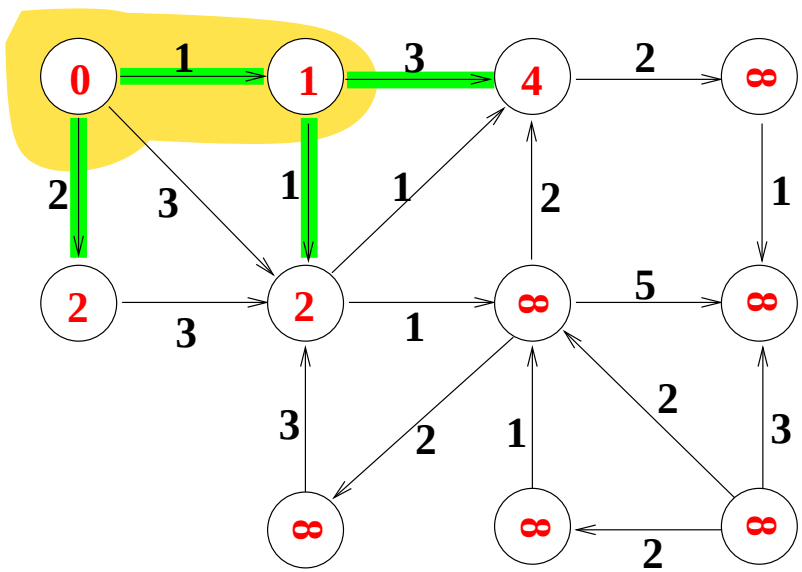
Rote Zahlen in den Knoten sind die `dist[·]`-Werte.

Runde: Bestimme den Knoten $u \in V - S$, der `dist[`w`]`, $w \in V - S$, minimiert.

(Hier: Einer der Knoten mit dist-Wert 2. Wenn es mehrere gibt, wähle einen beliebigen davon.)

Füge Knoten u zu S hinzu.

Beispiel: **Orange:** S.



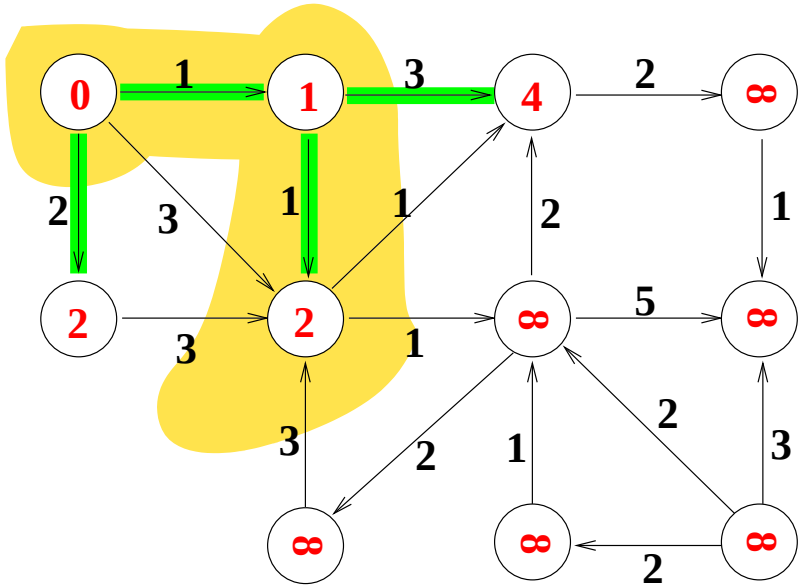
Rote Zahlen in den Knoten sind die `dist[·]`-Werte.

Runde: Bestimme den Knoten $u \in V - S$, der `dist[`w`]`, $w \in V - S$, minimiert.

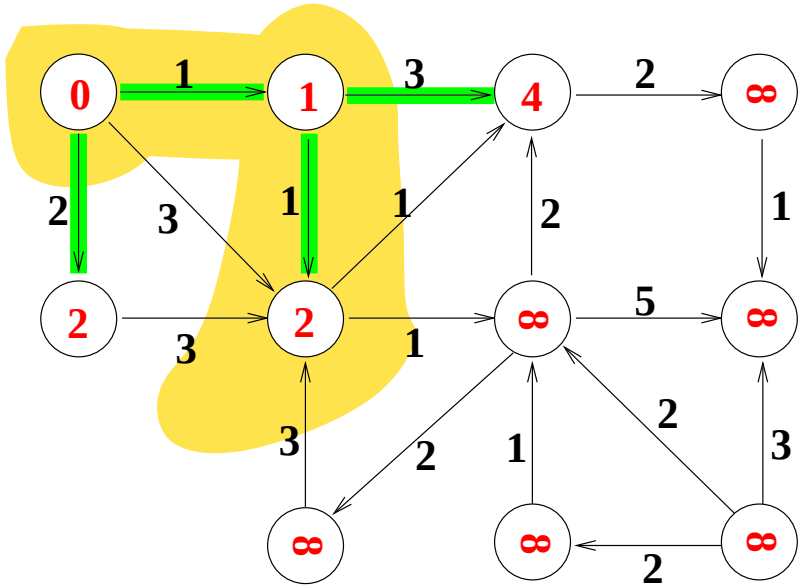
(Hier: Einer der Knoten mit dist-Wert 2. Wenn es mehrere gibt, wähle einen beliebigen davon.)

Füge Knoten u zu S hinzu. (Er „brennt an“. Der aktuelle Wert `dist[`u`]` wird „eingefroren“.)

Beispiel:

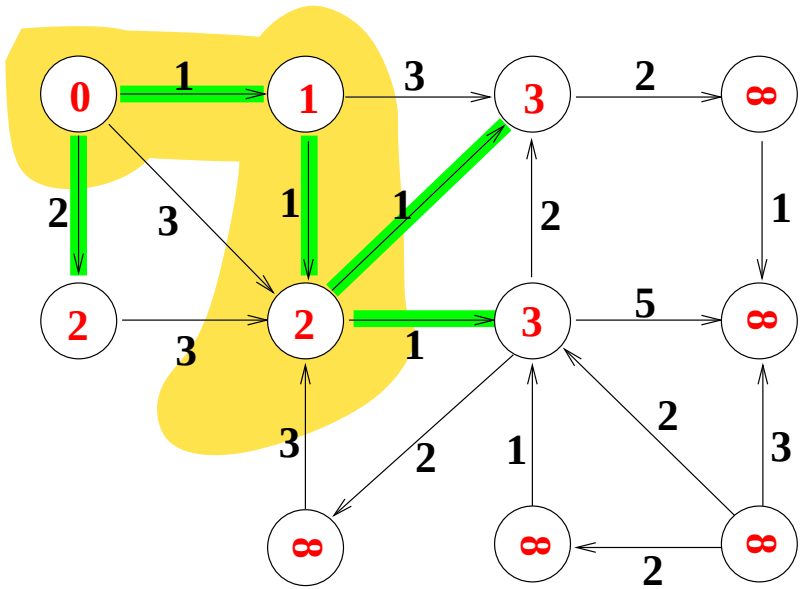


Beispiel:



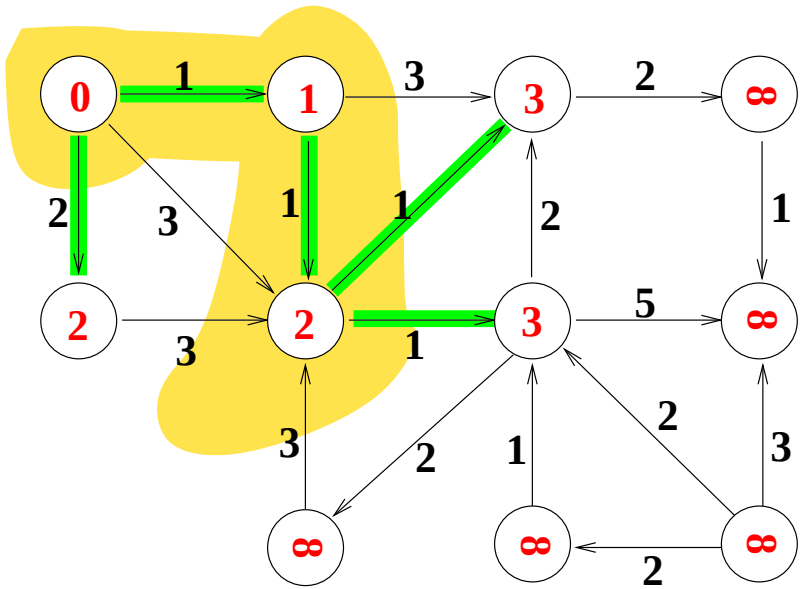
Was ist noch zu tun?

Beispiel:



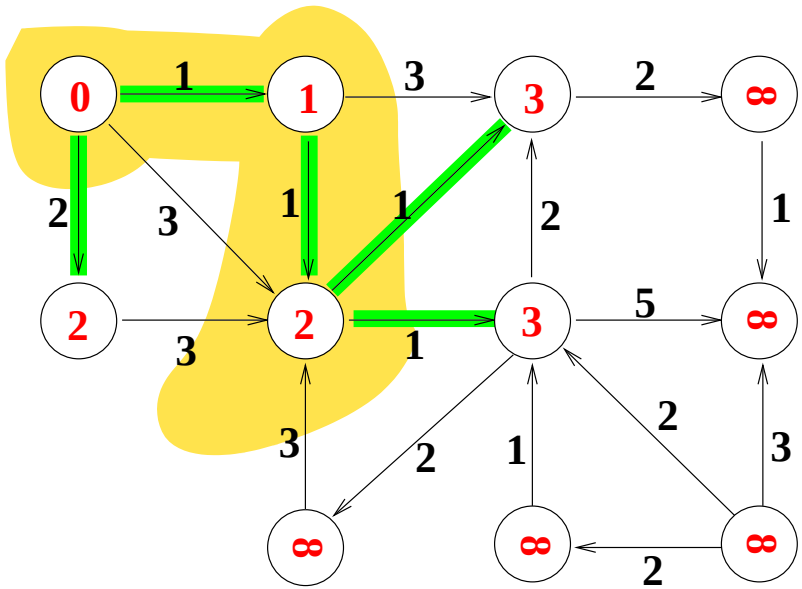
Was ist noch zu tun?

Beispiel:



Was ist noch zu tun? Auf Kante (u, w) kann neuer Knoten $w \in V - S$ erreicht werden, oder $w \in V - S$ kann schneller erreicht werden.

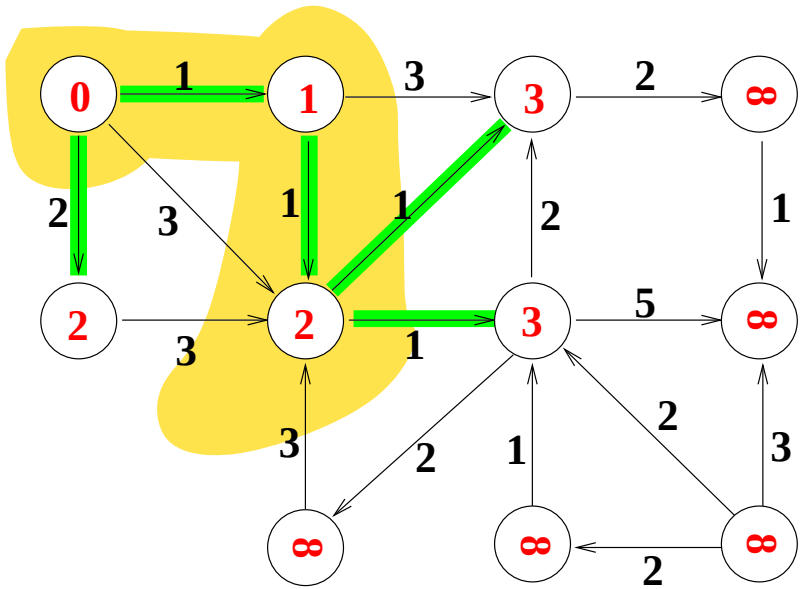
Beispiel:



Was ist noch zu tun? Auf Kante (u, w) kann neuer Knoten $w \in V - S$ erreicht werden, oder $w \in V - S$ kann schneller erreicht werden. Also: Eventuell `dist[`w`]` aktualisieren:

$$\texttt{dist}[w] \leftarrow \min\{\texttt{dist}[w], \texttt{dist}[u] + c(u, w)\}.$$

Beispiel:



Was ist noch zu tun? Auf Kante (u, w) kann neuer Knoten $w \in V - S$ erreicht werden, oder $w \in V - S$ kann schneller erreicht werden. Also: Eventuell $\texttt{dist}[w]$ aktualisieren:

$$\texttt{dist}[w] \leftarrow \min\{\texttt{dist}[w], \texttt{dist}[u] + c(u, w)\}.$$

Der bisher entwickelte Algorithmus berechnet schon die **Länge** der kürzesten Wege (also die **Zeitpunkte**, zu denen das Feuer die Knoten erreicht).

Algorithmus Dijkstra-Distanzen(G, s)

Algorithmus Dijkstra-Distanzen(G, s)

Eingabe: gewichteter Digraph $G = (V, E, c)$, Startknoten $s \in V$

Ausgabe: Länge der kürzesten Wege von s zu den Knoten in G

(1) $\texttt{S} \leftarrow \emptyset$;

(2) $\texttt{dist}[s] \leftarrow 0$;

Algorithmus Dijkstra-Distanzen(G, s)

Eingabe: gewichteter Digraph $G = (V, E, c)$, Startknoten $s \in V$
Ausgabe: Länge der kürzesten Wege von s zu den Knoten in G

(1) $\texttt{S} \leftarrow \emptyset$;
(2) $\texttt{dist[}s\texttt{]} \leftarrow 0$;
(3) **for** $\texttt{w} \in V - \{s\}$ **do** $\texttt{dist[w]} \leftarrow \infty$;

Algorithmus Dijkstra-Distanzen(G, s)

Eingabe: gewichteter Digraph $G = (V, E, c)$, Startknoten $s \in V$
Ausgabe: Länge der kürzesten Wege von s zu den Knoten in G

(1) $\mathtt{S} \leftarrow \emptyset$;
(2) $\mathtt{dist}[s] \leftarrow 0$;
(3) **for** $\mathtt{w} \in V - \{s\}$ **do** $\mathtt{dist}[\mathtt{w}] \leftarrow \infty$;
(4) **while** $\exists u \in V - \mathtt{S}$: $\mathtt{dist}[u] < \infty$ **do** // „Runde“

Algorithmus Dijkstra-Distanzen(G, s)

Eingabe: gewichteter Digraph $G = (V, E, c)$, Startknoten $s \in V$
Ausgabe: Länge der kürzesten Wege von s zu den Knoten in G

(1) $\mathtt{S} \leftarrow \emptyset$;
(2) $\mathtt{dist}[s] \leftarrow 0$;
(3) **for** $\mathtt{w} \in V - \{s\}$ **do** $\mathtt{dist}[\mathtt{w}] \leftarrow \infty$;
(4) **while** $\exists u \in V - \mathtt{S}$: $\mathtt{dist}[u] < \infty$ **do** // „Runde“
(5) $\quad \mathtt{u} \leftarrow$ ein solcher Knoten u mit minimalem $\mathtt{dist}[u]$;
(6) $\quad \mathtt{S} \leftarrow \mathtt{S} \cup \{\mathtt{u}\}$;

Algorithmus Dijkstra-Distanzen(G, s)

Eingabe: gewichteter Digraph $G = (V, E, c)$, Startknoten $s \in V$
Ausgabe: Länge der kürzesten Wege von s zu den Knoten in G

(1) $\mathtt{S} \leftarrow \emptyset$;
(2) $\mathtt{dist}[s] \leftarrow 0$;
(3) **for** $\mathtt{w} \in V - \{s\}$ **do** $\mathtt{dist}[\mathtt{w}] \leftarrow \infty$;
(4) **while** $\exists u \in V - \mathtt{S}$: $\mathtt{dist}[u] < \infty$ **do** // „Runde“
(5) $\quad \mathtt{u} \leftarrow$ ein solcher Knoten u mit minimalem $\mathtt{dist}[u]$;
(6) $\quad \mathtt{S} \leftarrow \mathtt{S} \cup \{\mathtt{u}\}$;
(7) $\quad$ **for** $\mathtt{w}$ mit $(\mathtt{u}, \mathtt{w}) \in E$ und $\mathtt{w} \notin \mathtt{S}$ **do** // Nachfolger von u, nicht bearbeitet

Algorithmus Dijkstra-Distanzen(G, s)

Eingabe: gewichteter Digraph $G = (V, E, c)$, Startknoten $s \in V$
Ausgabe: Länge der kürzesten Wege von s zu den Knoten in G

(1) $\mathtt{S} \leftarrow \emptyset$;
(2) $\mathtt{dist}[s] \leftarrow 0$;
(3) **for** $\mathtt{w} \in V - \{s\}$ **do** $\mathtt{dist}[\mathtt{w}] \leftarrow \infty$;
(4) **while** $\exists u \in V - \mathtt{S}$: $\mathtt{dist}[u] < \infty$ **do** // „Runde“
(5) $\quad \mathtt{u} \leftarrow$ ein solcher Knoten u mit minimalem $\mathtt{dist}[u]$;
(6) $\quad \mathtt{S} \leftarrow \mathtt{S} \cup \{\mathtt{u}\}$;
(7) $\quad$ **for** $\mathtt{w}$ mit $(\mathtt{u}, \mathtt{w}) \in E$ und $\mathtt{w} \notin \mathtt{S}$ **do** // Nachfolger von u, nicht bearbeitet
(8) $\quad\quad \mathtt{dist}[\mathtt{w}] \leftarrow \min\{\mathtt{dist}[\mathtt{w}], \mathtt{dist}[\mathtt{u}] + c(\mathtt{u}, \mathtt{w})\}$;

Algorithmus Dijkstra-Distanzen(G, s)

Eingabe: gewichteter Digraph $G = (V, E, c)$, Startknoten $s \in V$
Ausgabe: Länge der kürzesten Wege von s zu den Knoten in G

(1) $\mathtt{S} \leftarrow \emptyset$;
(2) $\mathtt{dist}[s] \leftarrow 0$;
(3) **for** $\mathtt{w} \in V - \{s\}$ **do** $\mathtt{dist}[\mathtt{w}] \leftarrow \infty$;
(4) **while** $\exists u \in V - \mathtt{S}$: $\mathtt{dist}[u] < \infty$ **do** // „Runde“
(5) $\quad \mathtt{u} \leftarrow$ ein solcher Knoten u mit minimalem $\mathtt{dist}[u]$;
(6) $\quad \mathtt{S} \leftarrow \mathtt{S} \cup \{\mathtt{u}\}$;
(7) $\quad$ **for** $\mathtt{w}$ mit $(\mathtt{u}, \mathtt{w}) \in E$ und $\mathtt{w} \notin \mathtt{S}$ **do** // Nachfolger von u, nicht bearbeitet
(8) $\quad\quad \mathtt{dist}[\mathtt{w}] \leftarrow \min\{\mathtt{dist}[\mathtt{w}], \mathtt{dist}[\mathtt{u}] + c(\mathtt{u}, \mathtt{w})\}$;
(9) **Ausgabe:** das Array $\mathtt{dist}[1..n]$.

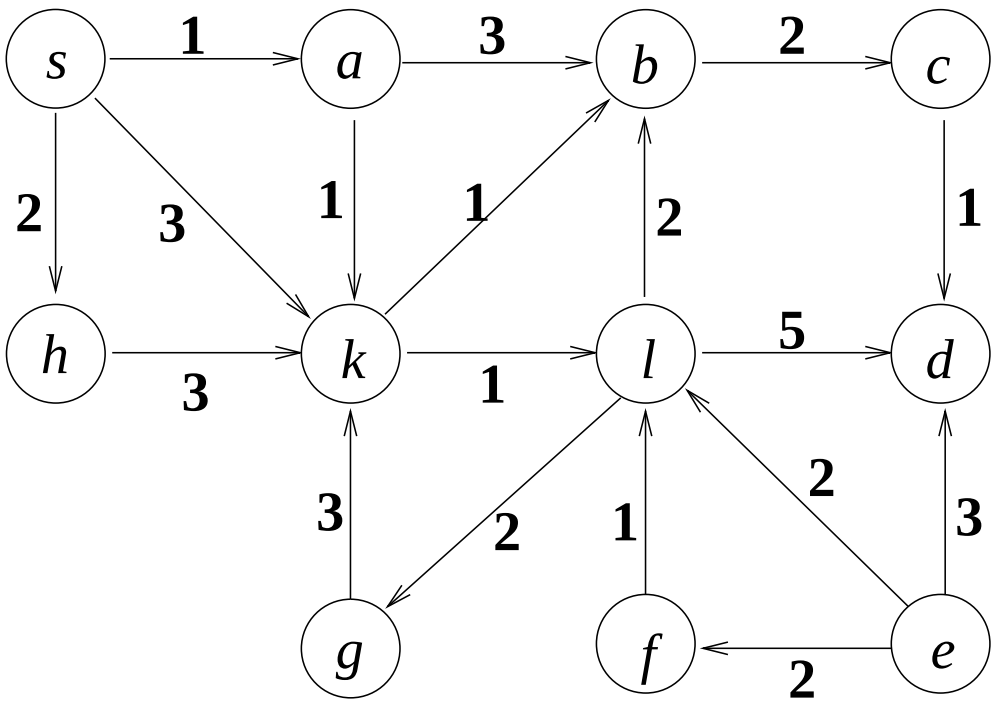
Algorithmus Dijkstra-Distanzen(G, s)

Eingabe: gewichteter Digraph $G = (V, E, c)$, Startknoten $s \in V$
Ausgabe: Länge der kürzesten Wege von s zu den Knoten in G

(1) $\mathtt{S} \leftarrow \emptyset$;
(2) $\mathtt{dist}[s] \leftarrow 0$;
(3) **for** $\mathtt{w} \in V - \{s\}$ **do** $\mathtt{dist}[\mathtt{w}] \leftarrow \infty$;
(4) **while** $\exists u \in V - \mathtt{S}$: $\mathtt{dist}[u] < \infty$ **do** // „Runde“
(5) $\quad \mathtt{u} \leftarrow$ ein solcher Knoten u mit minimalem $\mathtt{dist}[u]$;
(6) $\quad \mathtt{S} \leftarrow \mathtt{S} \cup \{\mathtt{u}\}$;
(7) $\quad$ **for** $\mathtt{w}$ mit $(\mathtt{u}, \mathtt{w}) \in E$ und $\mathtt{w} \notin \mathtt{S}$ **do** // Nachfolger von u, nicht bearbeitet
(8) $\quad\quad \mathtt{dist}[\mathtt{w}] \leftarrow \min\{\mathtt{dist}[\mathtt{w}], \mathtt{dist}[\mathtt{u}] + c(\mathtt{u}, \mathtt{w})\}$;
(9) **Ausgabe:** das Array $\mathtt{dist}[1..n]$.

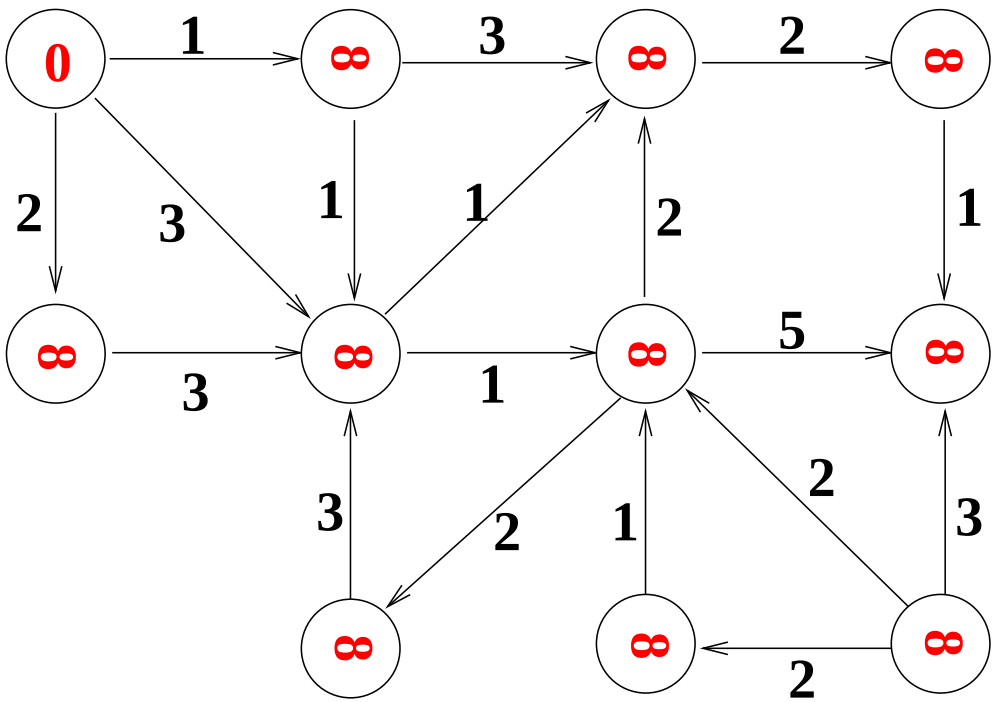
(Technisch wird S durch einen Bitvektor $\mathtt{inS}[1..n]$ realisiert.)

Ablauf des Algorithmus von Dijkstra



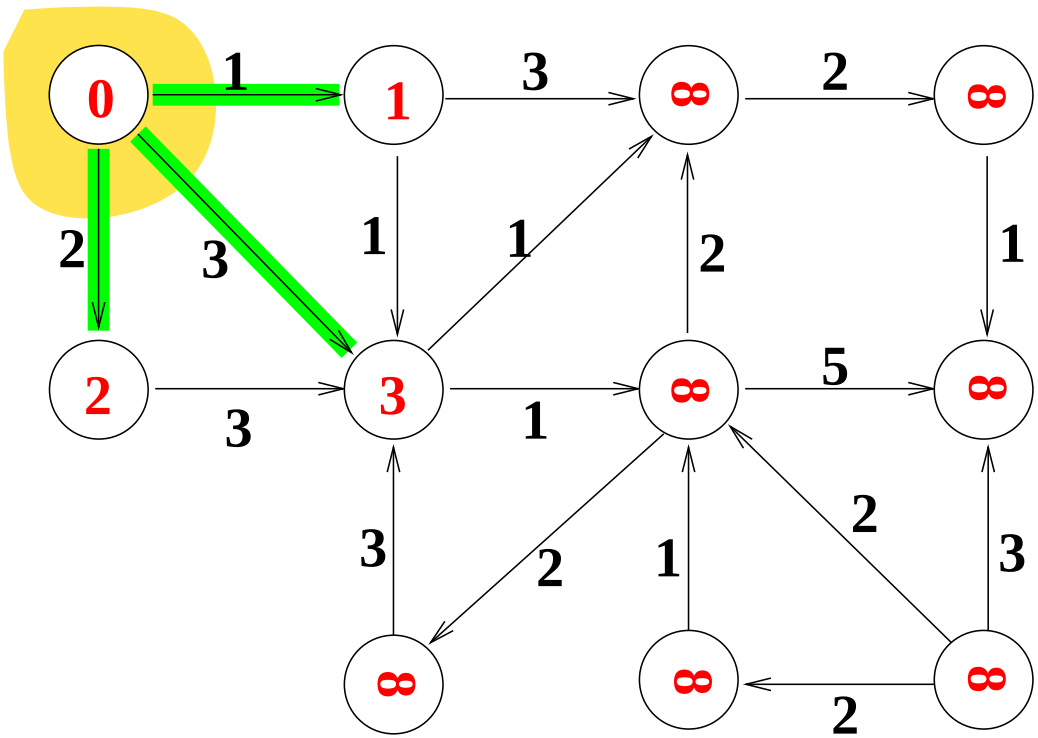
Der Ausgangsgraph.

Ablauf des Algorithmus von Dijkstra



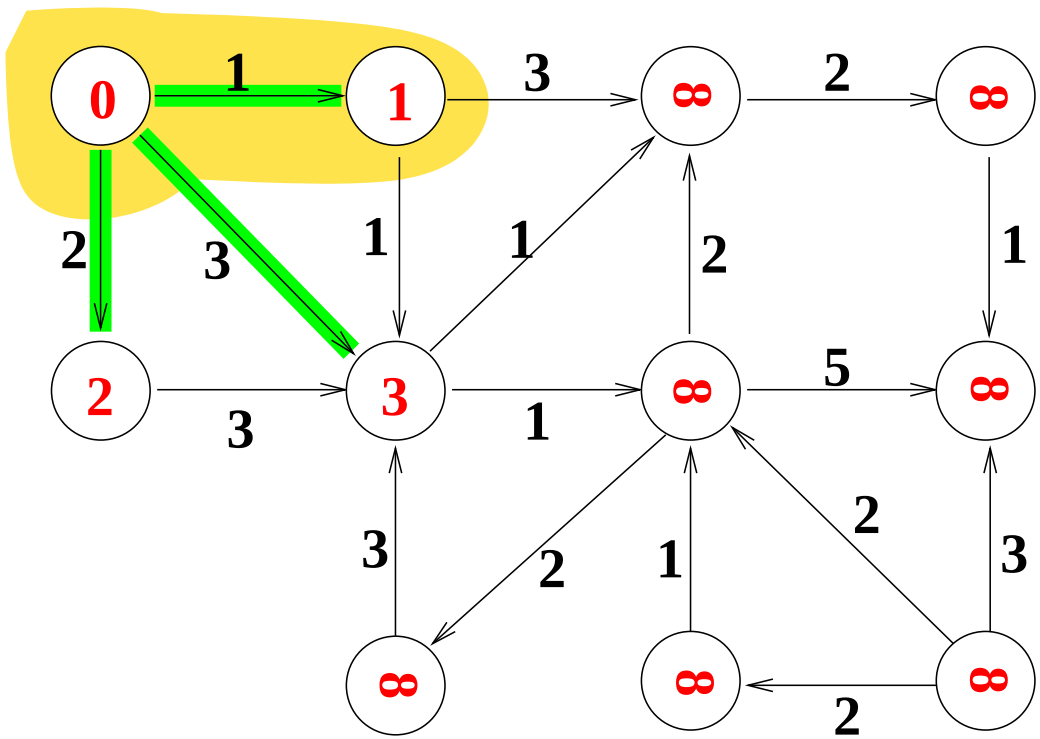
Nach der Initialisierung (Zeilen (1)–(3+)) (Folie 19).

Ablauf des Algorithmus von Dijkstra



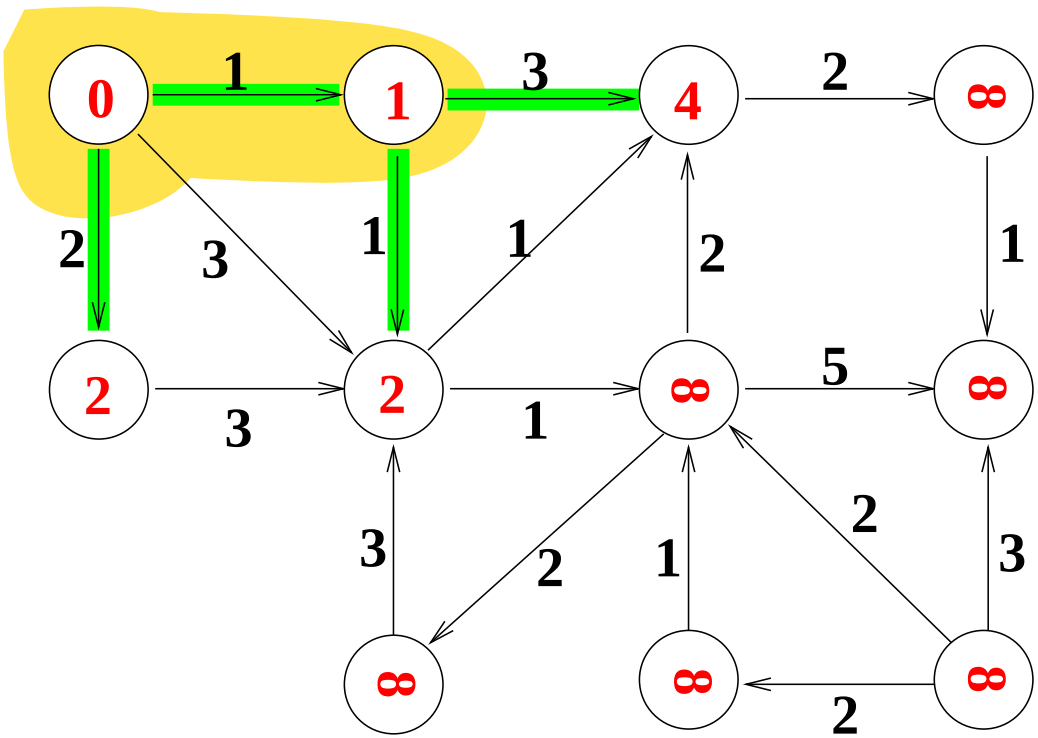
Nach Zeilen (5)–(8d) für $u = s$ (Folie 19).

Ablauf des Algorithmus von Dijkstra



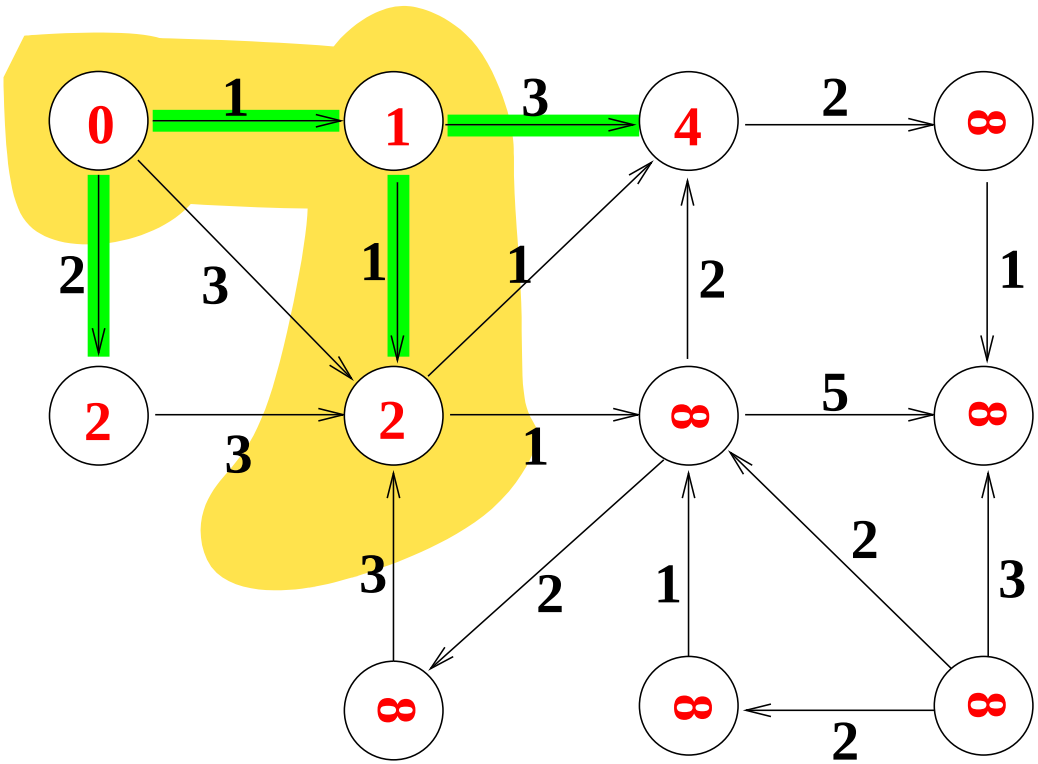
$u = a$, Zeile (6) (Folie 19).

Ablauf des Algorithmus von Dijkstra



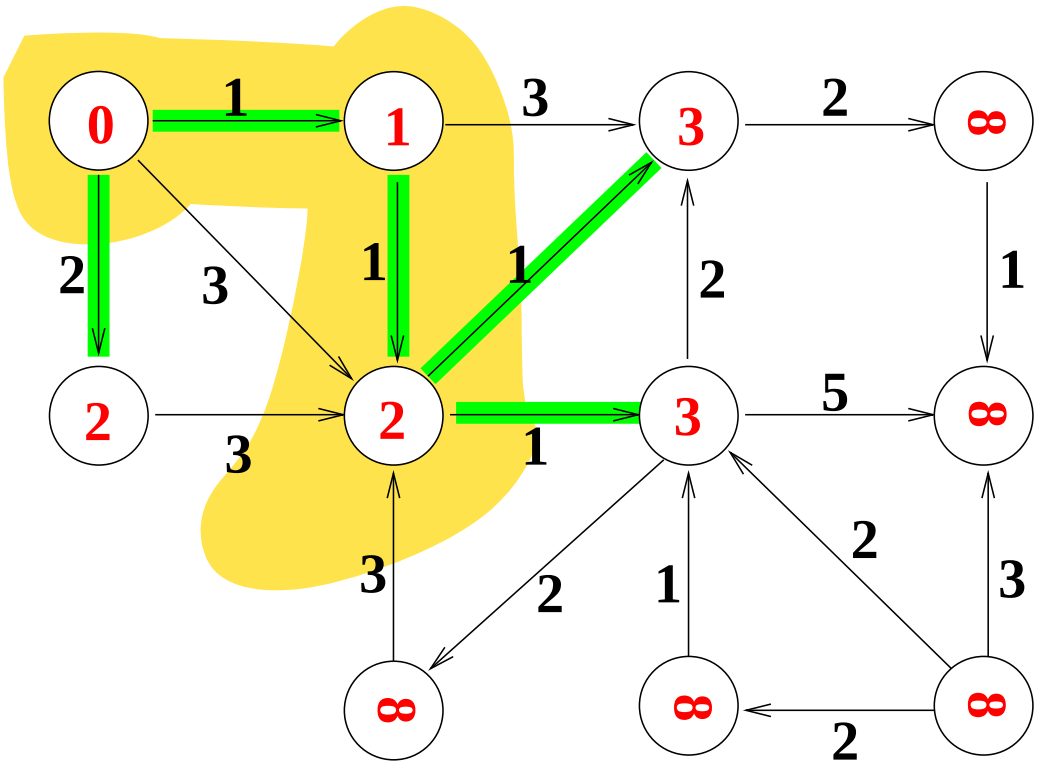
$u = a$, Zeilen (7)–(8d) (Folie 19).

Ablauf des Algorithmus von Dijkstra



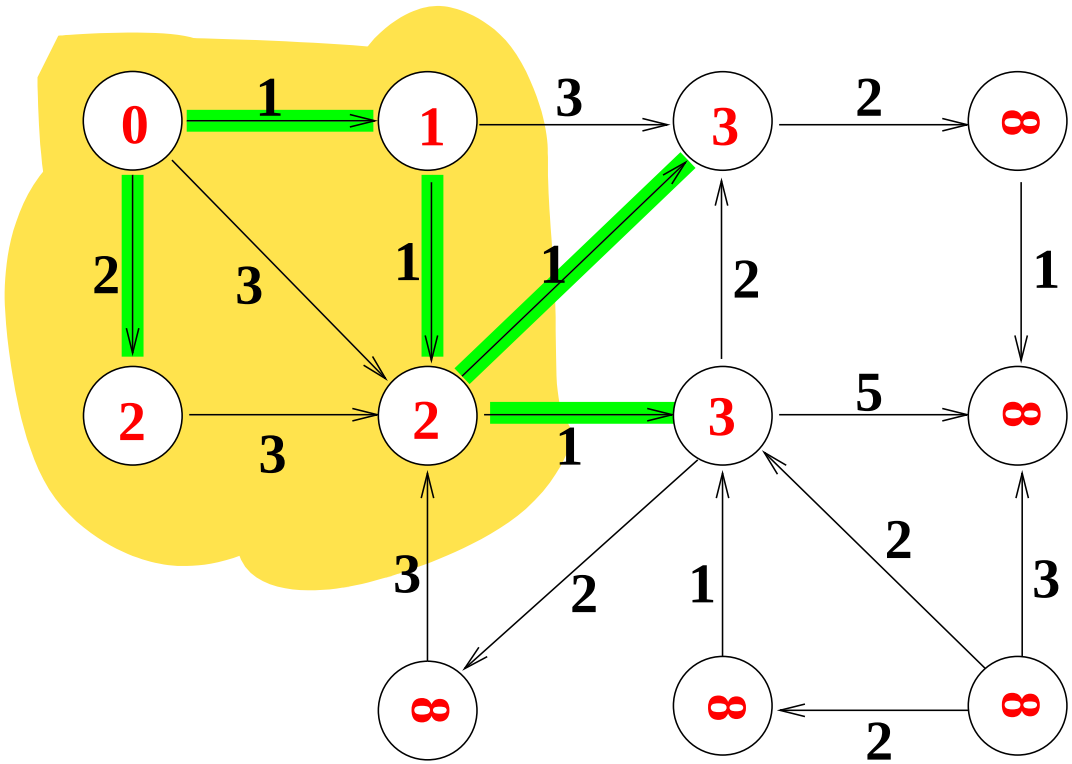
$u = k$, Zeile (6) (Folie 19).

Ablauf des Algorithmus von Dijkstra



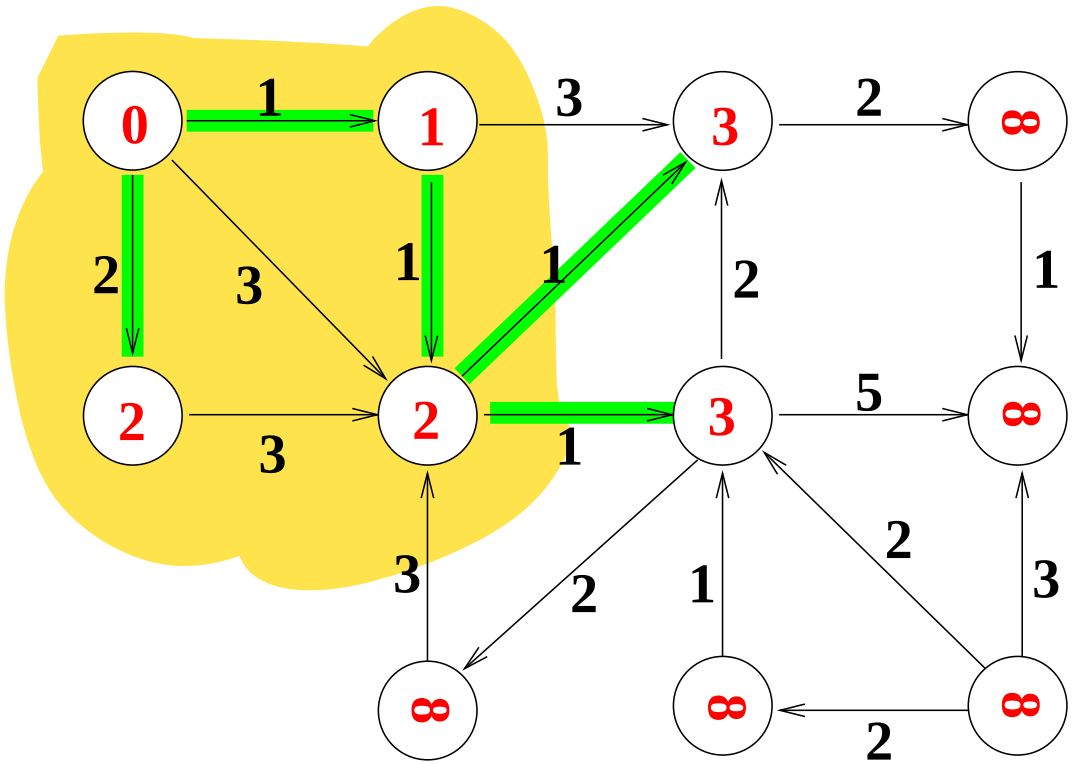
$u = k$, Zeilen (7)–(8d) (Folie 19).

Ablauf des Algorithmus von Dijkstra



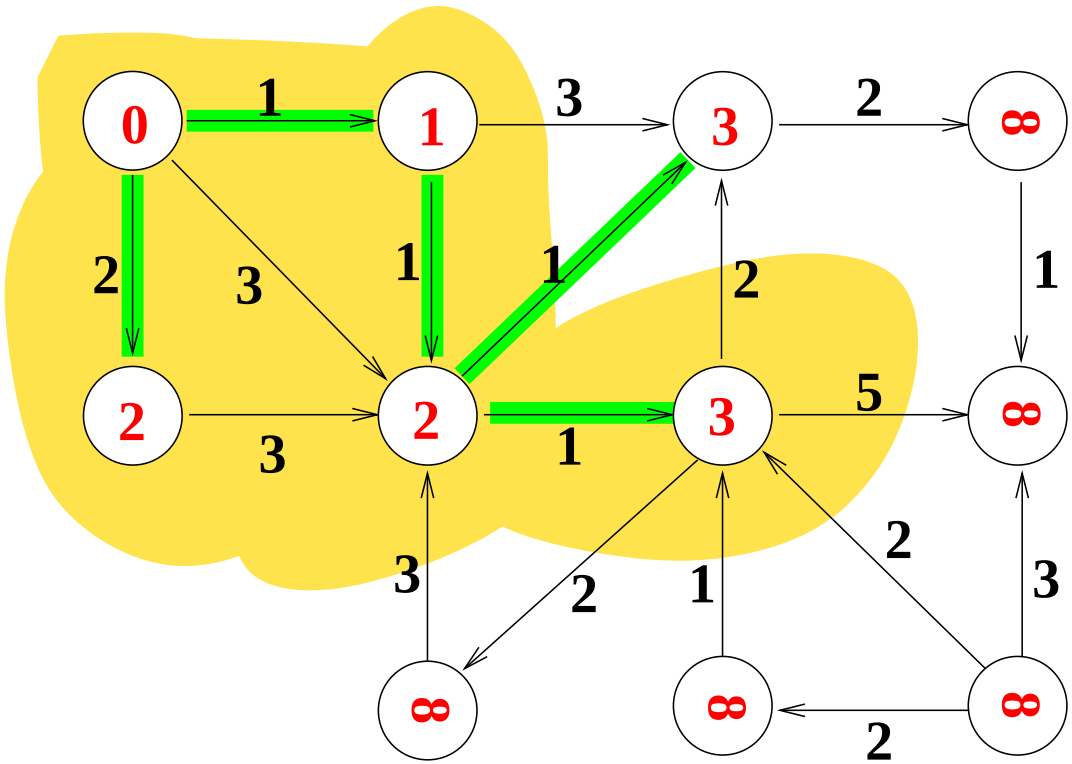
$u = h$, Zeile (6) (Folie 19).

Ablauf des Algorithmus von Dijkstra



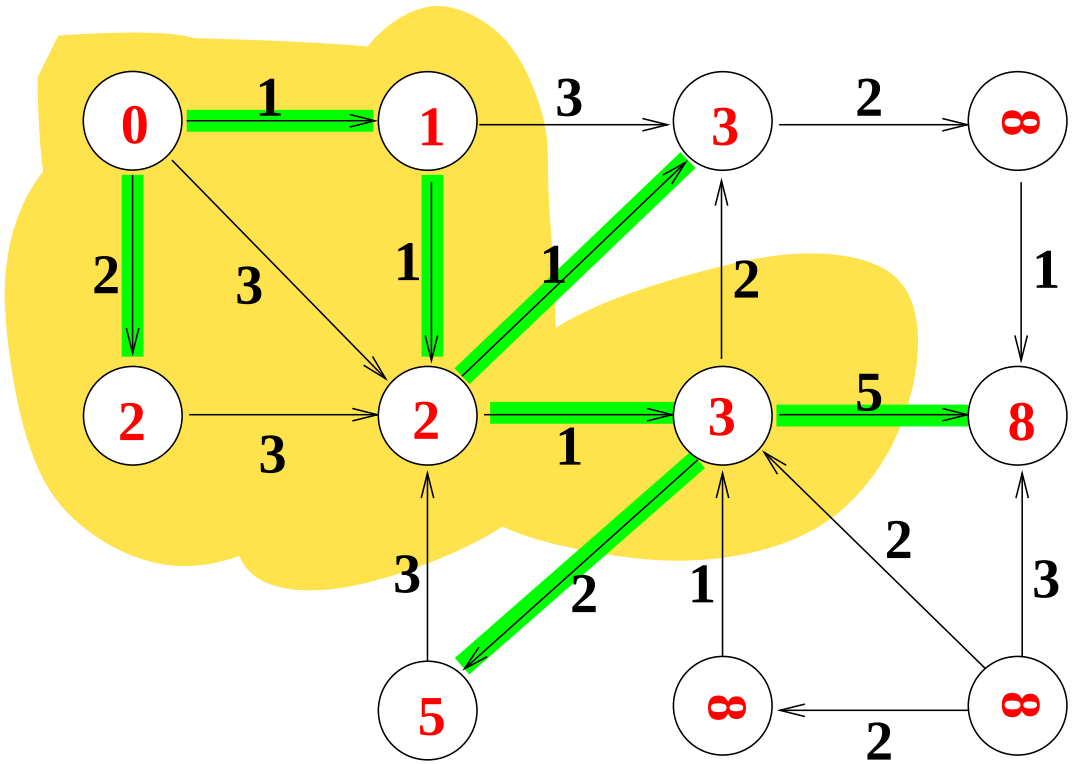
$u = h$, Zeilen (7)–(8d) ändern nichts (Folie 19).

Ablauf des Algorithmus von Dijkstra



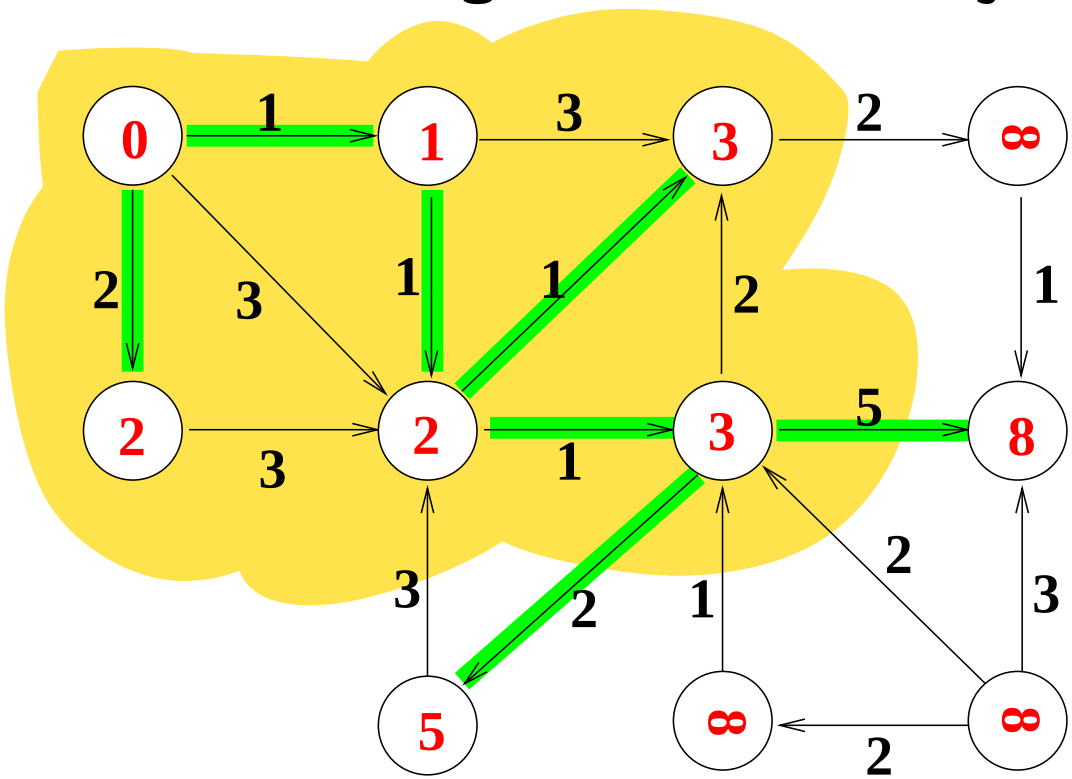
$u = l$, Zeile (6) (Folie 19).

Ablauf des Algorithmus von Dijkstra



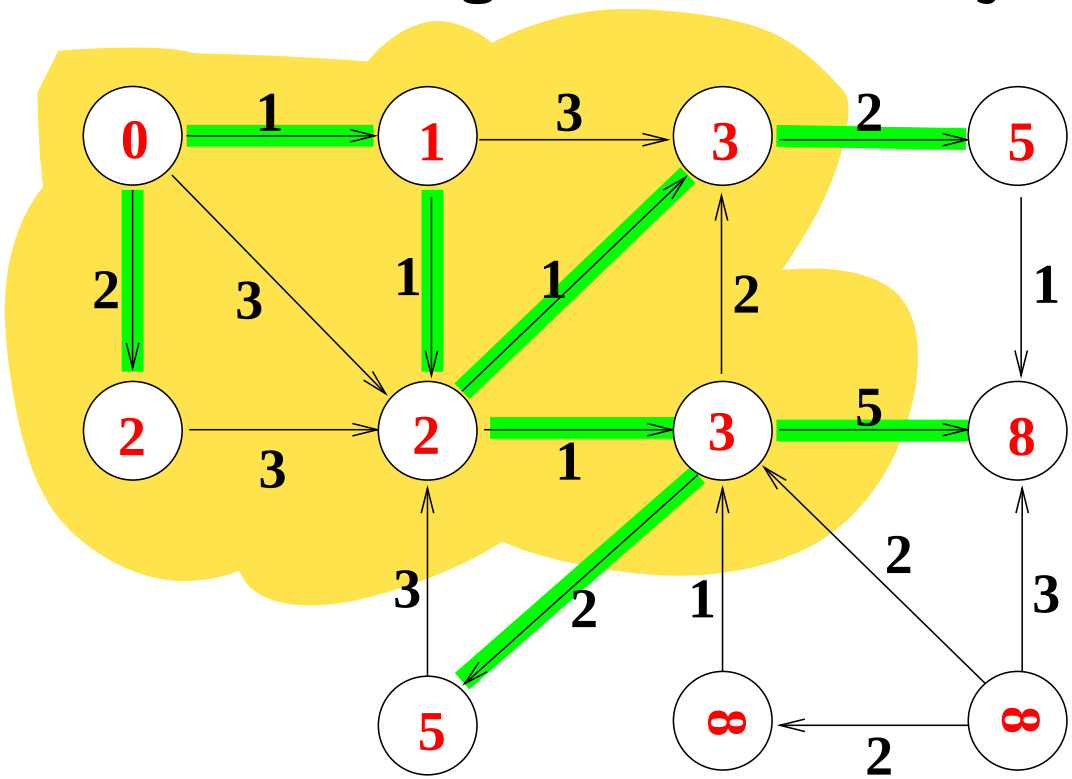
$u = l$, Zeilen (7)–(8d) (Folie 19).

Ablauf des Algorithmus von Dijkstra



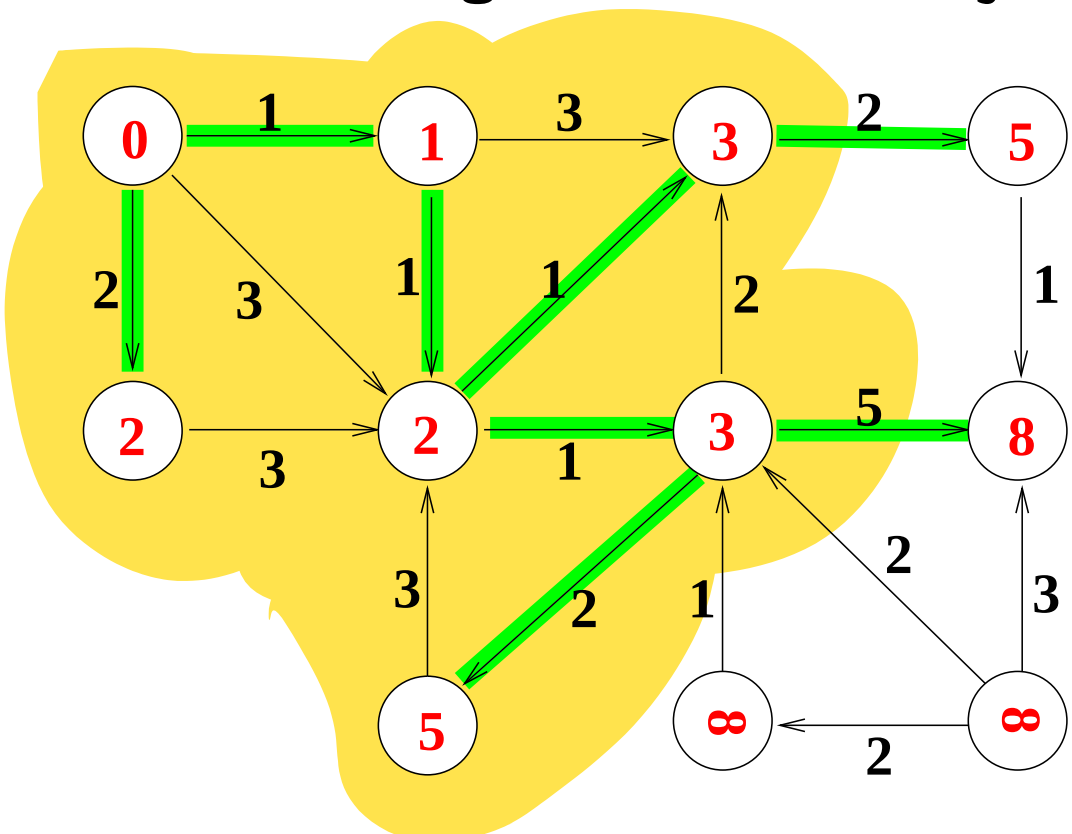
$u = b$, Zeile (6) (Folie 19).

Ablauf des Algorithmus von Dijkstra



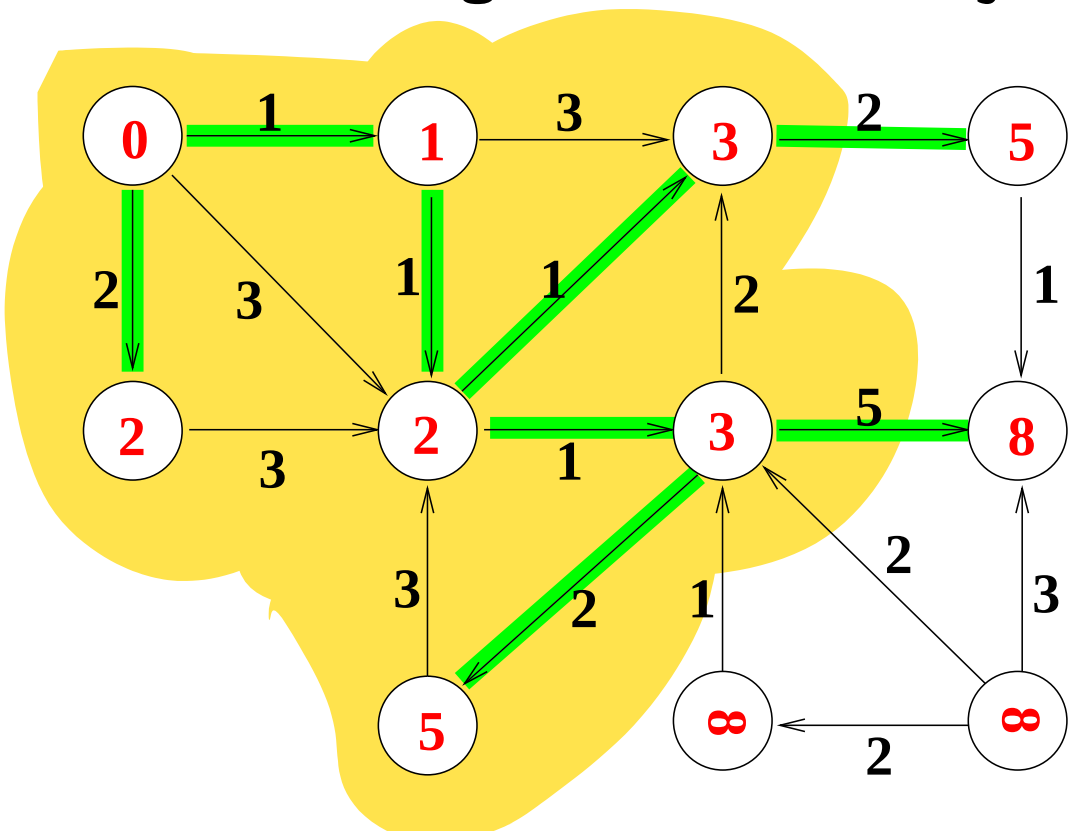
$u = b$, Zeilen (7)–(8d) (Folie 19).

Ablauf des Algorithmus von Dijkstra



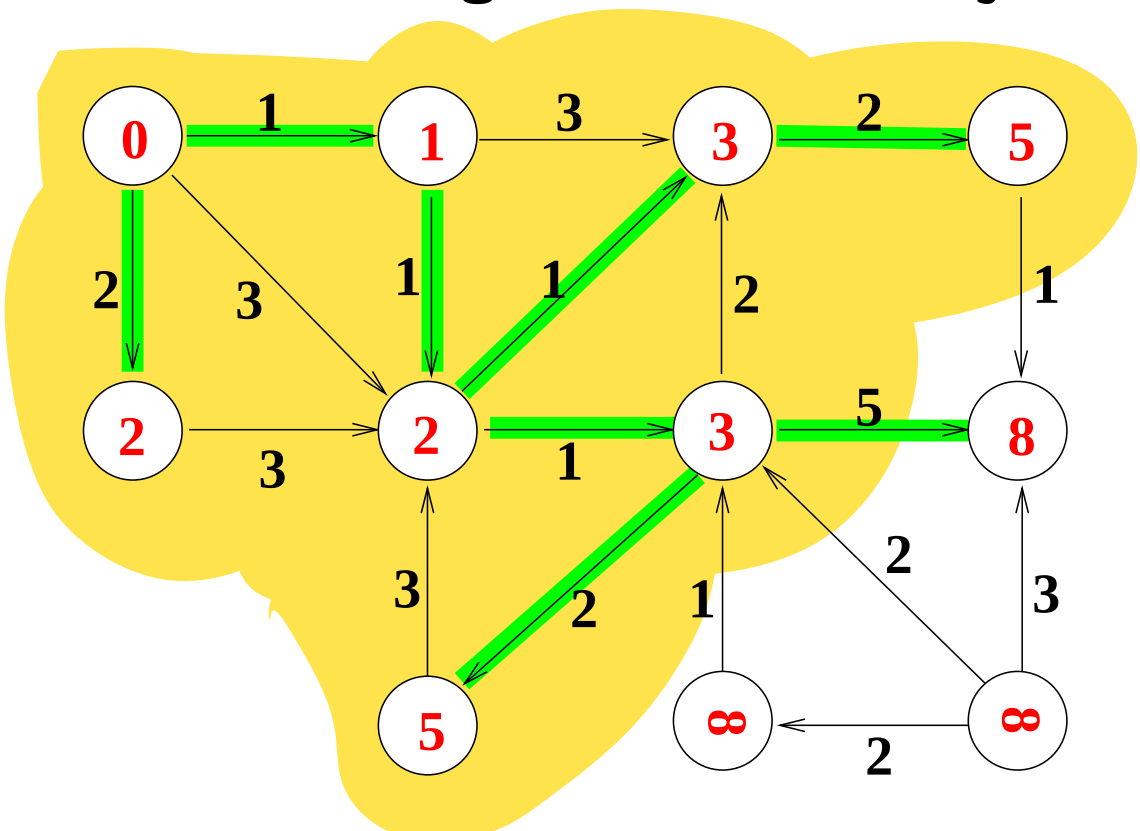
$u = g$, Zeile (6) (Folie 19).

Ablauf des Algorithmus von Dijkstra



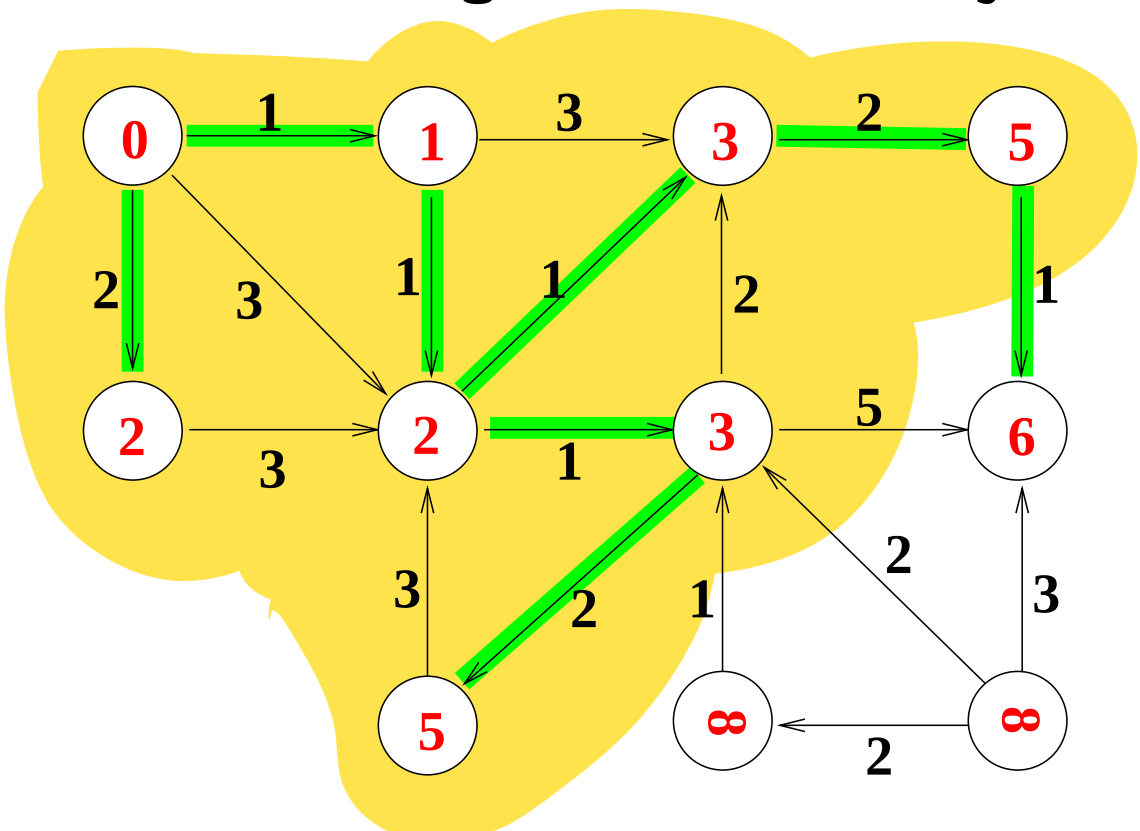
$u = g$, Zeilen (7)–(8d) ändern nichts (Folie 19).

Ablauf des Algorithmus von Dijkstra



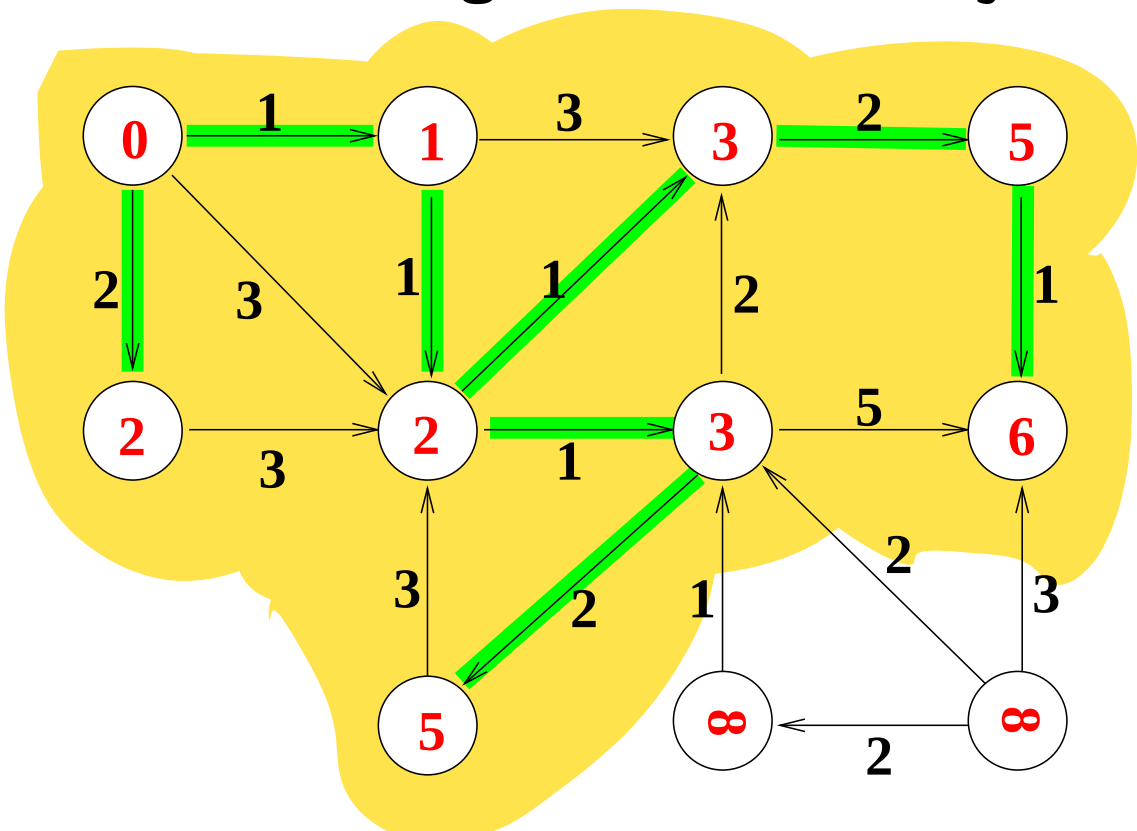
$u = c$, Zeile (6) (Folie 19).

Ablauf des Algorithmus von Dijkstra



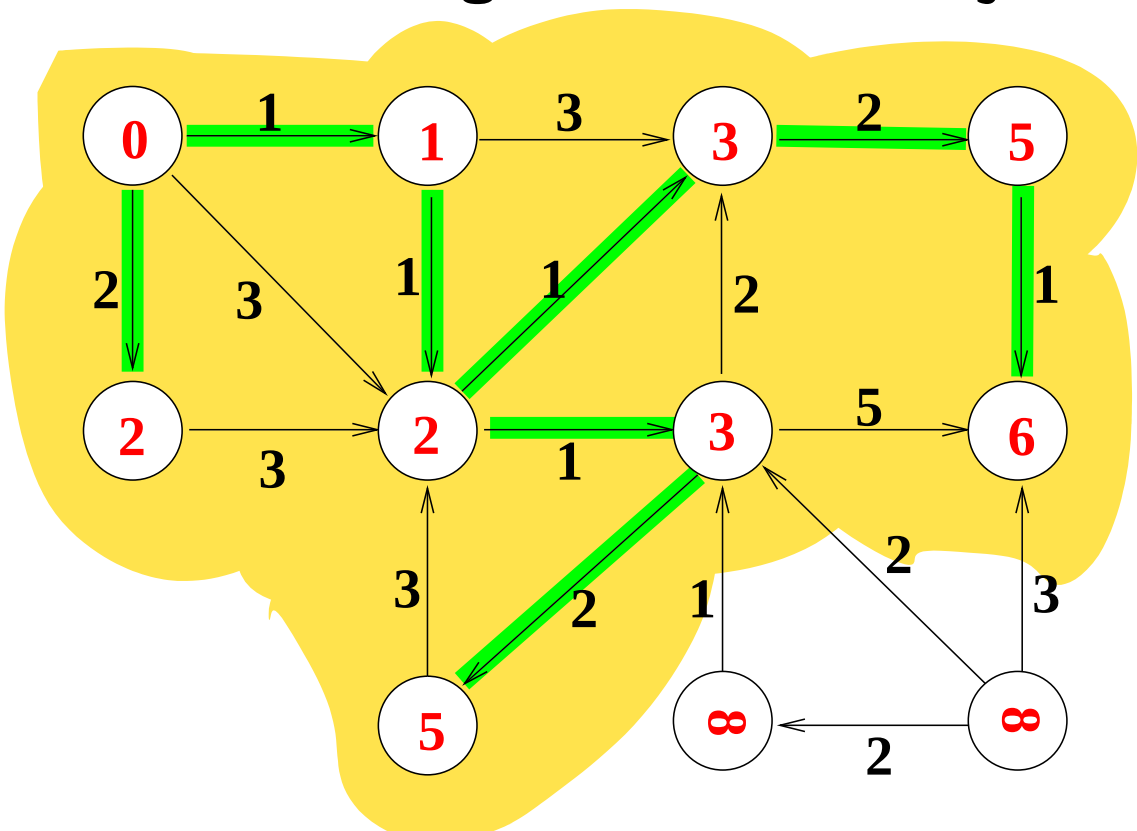
$u = c$, Zeilen (7)–(8d) (Folie 19).

Ablauf des Algorithmus von Dijkstra



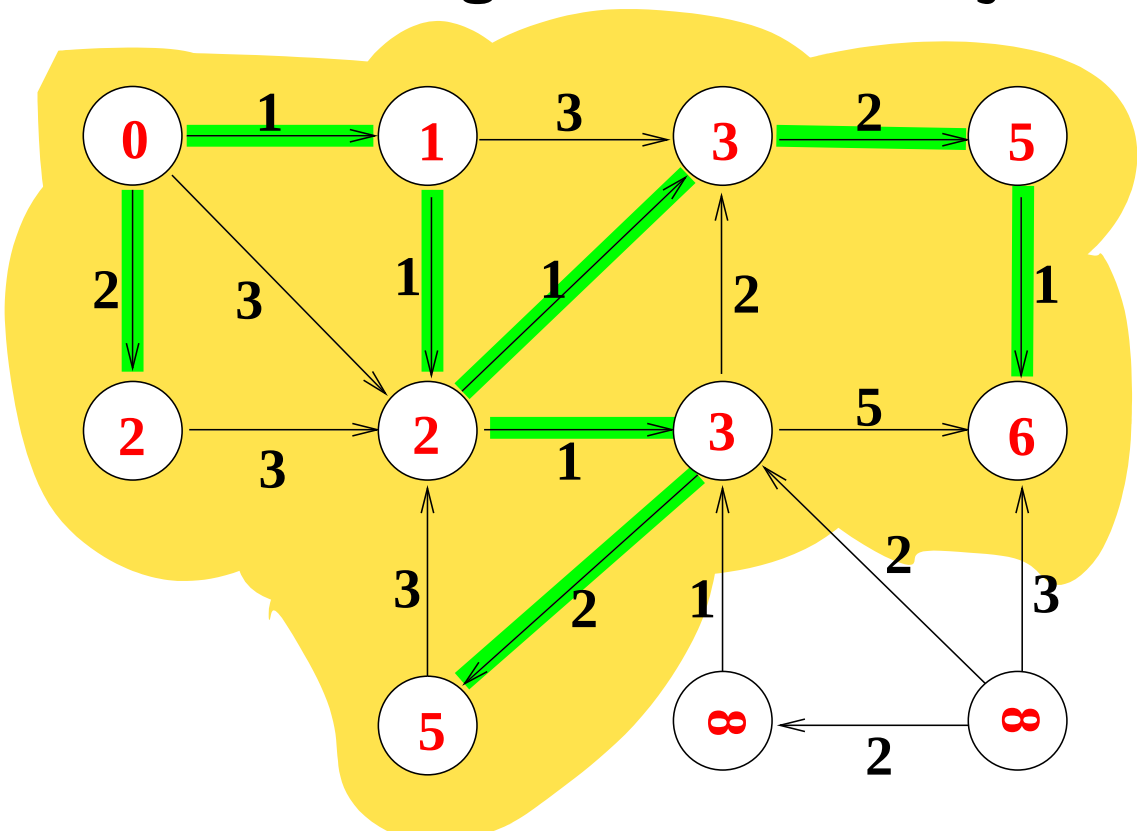
$u = d$, Zeile (6) (Folie 19).

Ablauf des Algorithmus von Dijkstra



$u = d$, Zeilen (7)–(8d) ändern nichts. (Folie 19).

Ablauf des Algorithmus von Dijkstra



Alle $v \in V - \mathtt{S}$ erfüllen $\mathtt{dist}[v] = \infty$: **Ende**.

PAUSE

Es folgt: Korrektheitsbeweis für den Algorithmus von Dijkstra

Lemma 11.1.2

Der Algorithmus **Dijkstra-Distanzen** gibt in `dist`$[v]$ für alle $v \in V$ den Wert $d(s, v)$ aus.

Lemma 11.1.2

Der Algorithmus **Dijkstra-Distanzen** gibt in `dist`$[v]$ für alle $v \in V$ den Wert $d(s, v)$ aus.

Beweis:

Lemma 11.1.2

Der Algorithmus **Dijkstra-Distanzen** gibt in `dist`$[v]$ für alle $v \in V$ den Wert $d(s, v)$ aus.

Beweis:

Wenn Knoten v (mit v in `u`) in Zeilen (5)–(8) betrachtet wird, sagen wir, dass v **bearbeitet** wird.

Lemma 11.1.2

Der Algorithmus **Dijkstra-Distanzen** gibt in `dist[`v`]` für alle $v \in V$ den Wert $d(s, v)$ aus.

Beweis:

Wenn Knoten v (mit v in `u`) in Zeilen (5)–(8) betrachtet wird, sagen wir, dass v **bearbeitet** wird.

Wenn `dist[`v`]` in Zeile (2) oder (8) (mit v in `w`) erstmals auf einen Wert $< \infty$ gesetzt wird, sagen wir, dass Knoten v **gefunden** wird.

Lemma 11.1.2

Der Algorithmus **Dijkstra-Distanzen** gibt in `dist[`v`]` für alle $v \in V$ den Wert $d(s, v)$ aus.

Beweis:

Wenn Knoten v (mit v in `u`) in Zeilen (5)–(8) betrachtet wird, sagen wir, dass v **bearbeitet** wird.

Wenn `dist[`v`]` in Zeile (2) oder (8) (mit v in `w`) erstmals auf einen Wert $< \infty$ gesetzt wird, sagen wir, dass Knoten v **gefunden** wird.

Unerreichbare Knoten:

Lemma 11.1.2

Der Algorithmus **Dijkstra-Distanzen** gibt in `dist[`v`]` für alle $v \in V$ den Wert $d(s, v)$ aus.

Beweis:

Wenn Knoten v (mit v in `u`) in Zeilen (5)–(8) betrachtet wird, sagen wir, dass v **bearbeitet** wird.

Wenn `dist[`v`]` in Zeile (2) oder (8) (mit v in `w`) erstmals auf einen Wert $< \infty$ gesetzt wird, sagen wir, dass Knoten v **gefunden** wird.

Unerreichbare Knoten: Durch eine einfache Induktion (wie bei BFS oder DFS) zeigt man, dass jeder Knoten v, der von s aus erreichbar ist, irgendwann gefunden und irgendwann danach bearbeitet wird.

Lemma 11.1.2

Der Algorithmus **Dijkstra-Distanzen** gibt in `dist[`v`]` für alle $v \in V$ den Wert $d(s, v)$ aus.

Beweis:

Wenn Knoten v (mit v in `u`) in Zeilen (5)–(8) betrachtet wird, sagen wir, dass v **bearbeitet** wird.

Wenn `dist[`v`]` in Zeile (2) oder (8) (mit v in `w`) erstmals auf einen Wert $< \infty$ gesetzt wird, sagen wir, dass Knoten v **gefunden** wird.

Unerreichbare Knoten: Durch eine einfache Induktion (wie bei BFS oder DFS) zeigt man, dass jeder Knoten v, der von s aus erreichbar ist, irgendwann gefunden und irgendwann danach bearbeitet wird.

Genau diejenigen Knoten v, die nicht von s aus erreichbar sind, behalten also den Wert `dist[`v`]` $= \infty$.

Nun zeigen wir durch Induktion über Runden die folgenden **Invarianten**, gültig am Ende jeder Runde:

Nun zeigen wir durch Induktion über Runden die folgenden **Invarianten**, gültig am Ende jeder Runde:

(I1) Für alle $v \in V$ gilt:

$$\texttt{dist}[v] < \infty \Rightarrow \exists \text{ Weg von } s \text{ nach } v \text{ mit Länge} \leq \texttt{dist}[v].$$

Nun zeigen wir durch Induktion über Runden die folgenden **Invarianten**, gültig am Ende jeder Runde:

(I1) Für alle $v \in V$ gilt:

$$\texttt{dist}[v] < \infty \Rightarrow \ \exists \text{ Weg von } s \text{ nach } v \text{ mit Länge} \leq \texttt{dist}[v].$$

(I2) Für alle $v \in \texttt{S}$ gilt $\texttt{dist}[v] = d(s, v)$.

Nun zeigen wir durch Induktion über Runden die folgenden **Invarianten**, gültig am Ende jeder Runde:

(I1) Für alle $v \in V$ gilt:

$$\texttt{dist}[v] < \infty \Rightarrow \ \exists \text{ Weg von } s \text{ nach } v \text{ mit Länge} \leq \texttt{dist}[v].$$

(I2) Für alle $v \in \texttt{S}$ gilt $\texttt{dist}[v] = d(s, v)$.

Beweis von (I1) durch Induktion über Runden:

Nun zeigen wir durch Induktion über Runden die folgenden **Invarianten**, gültig am Ende jeder Runde:

(I1) Für alle $v \in V$ gilt:

$$\texttt{dist}[v] < \infty \Rightarrow \ \exists \text{ Weg von } s \text{ nach } v \text{ mit Länge} \leq \texttt{dist}[v].$$

(I2) Für alle $v \in S$ gilt $\texttt{dist}[v] = d(s, v)$.

Beweis von (I1) durch Induktion über Runden:

Nach der Initialisierung gilt $\texttt{dist}[v] < \infty$ nur für $v = s$;

Nun zeigen wir durch Induktion über Runden die folgenden **Invarianten**, gültig am Ende jeder Runde:

(I1) Für alle $v \in V$ gilt:

$$\texttt{dist}[v] < \infty \Rightarrow \ \exists \text{ Weg von } s \text{ nach } v \text{ mit Länge} \leq \texttt{dist}[v].$$

(I2) Für alle $v \in \texttt{S}$ gilt $\texttt{dist}[v] = d(s, v)$.

Beweis von (I1) durch Induktion über Runden:

Nach der Initialisierung gilt $\texttt{dist}[v] < \infty$ nur für $v = s$; es gibt einen Weg von s nach s der Länge 0.

Nun zeigen wir durch Induktion über Runden die folgenden **Invarianten**, gültig am Ende jeder Runde:

(I1) Für alle $v \in V$ gilt:

$$\texttt{dist}[v] < \infty \Rightarrow \ \exists \text{ Weg von } s \text{ nach } v \text{ mit Länge} \leq \texttt{dist}[v].$$

(I2) Für alle $v \in S$ gilt $\texttt{dist}[v] = d(s, v)$.

Beweis von (I1) durch Induktion über Runden:
Nach der Initialisierung gilt $\texttt{dist}[v] < \infty$ nur für $v = s$; es gibt einen Weg von s nach s der Länge 0.
Betrachte nun eine Runde, in der u bearbeitet wird, und einen Knoten $v \in V - S$.

Nun zeigen wir durch Induktion über Runden die folgenden **Invarianten**, gültig am Ende jeder Runde:

(I1) Für alle $v \in V$ gilt:

$$\texttt{dist}[v] < \infty \Rightarrow \ \exists \text{ Weg von } s \text{ nach } v \text{ mit Länge} \leq \texttt{dist}[v].$$

(I2) Für alle $v \in S$ gilt $\texttt{dist}[v] = d(s, v)$.

Beweis von (I1) durch Induktion über Runden:
Nach der Initialisierung gilt $\texttt{dist}[v] < \infty$ nur für $v = s$; es gibt einen Weg von s nach s der Länge 0.
Betrachte nun eine Runde, in der u bearbeitet wird, und einen Knoten $v \in V - S$.
Wenn sich $\texttt{dist}[v]$ in dieser Runde nicht ändert, ist nichts zu zeigen.

Nun zeigen wir durch Induktion über Runden die folgenden **Invarianten**, gültig am Ende jeder Runde:

(I1) Für alle $v \in V$ gilt:

$$\texttt{dist}[v] < \infty \Rightarrow \ \exists \text{ Weg von } s \text{ nach } v \text{ mit Länge} \leq \texttt{dist}[v].$$

(I2) Für alle $v \in S$ gilt $\texttt{dist}[v] = d(s, v)$.

Beweis von (I1) durch Induktion über Runden:
Nach der Initialisierung gilt $\texttt{dist}[v] < \infty$ nur für $v = s$; es gibt einen Weg von s nach s der Länge 0.
Betrachte nun eine Runde, in der u bearbeitet wird, und einen Knoten $v \in V - S$.
Wenn sich $\texttt{dist}[v]$ in dieser Runde nicht ändert, ist nichts zu zeigen.
Wenn $\texttt{dist}[v]$ in dieser Runde geändert wird, dann auf den Wert $\texttt{dist}[u] + c(u, v)$.

Nun zeigen wir durch Induktion über Runden die folgenden **Invarianten**, gültig am Ende jeder Runde:

(I1) Für alle $v \in V$ gilt:

$$\texttt{dist}[v] < \infty \Rightarrow \ \exists \text{ Weg von } s \text{ nach } v \text{ mit Länge} \leq \texttt{dist}[v].$$

(I2) Für alle $v \in S$ gilt $\texttt{dist}[v] = d(s, v)$.

Beweis von (I1) durch Induktion über Runden:

Nach der Initialisierung gilt $\texttt{dist}[v] < \infty$ nur für $v = s$; es gibt einen Weg von s nach s der Länge 0.

Betrachte nun eine Runde, in der u bearbeitet wird, und einen Knoten $v \in V - S$.

Wenn sich $\texttt{dist}[v]$ in dieser Runde nicht ändert, ist nichts zu zeigen.

Wenn $\texttt{dist}[v]$ in dieser Runde geändert wird, dann auf den Wert $\texttt{dist}[u] + c(u, v)$.
Nach I.V. gibt es einen Weg p_u von s nach u der Länge höchstens $\texttt{dist}[u]$. Wenn wir p_u um die Kante (u, v) verlängern, erhalten wir einen Weg von s nach v der Länge höchstens $\texttt{dist}[u] + c(u, v) = \texttt{dist}[v]$.

Beweis von (I2) durch Induktion über Runden:

Beweis von (I2) durch Induktion über Runden:

I.A.: Am Anfang ist `S` leer. In der ersten Runde wird offensichtlich der Startknoten s in `S` aufgenommen, und $d(s,s) = \texttt{dist}[s] = 0$.

Beweis von (I2) durch Induktion über Runden:

I.A.: Am Anfang ist `S` leer. In der ersten Runde wird offensichtlich der Startknoten s in `S` aufgenommen, und $d(s, s) = \texttt{dist}[s] = 0$.

I.V.: Sei $u \neq s$.

(I2) stimmt für alle Runden vor derjenigen, in der u bearbeitet, also in `S` aufgenommen wird.

Beweis von (I2) durch Induktion über Runden:

I.A.: Am Anfang ist S leer. In der ersten Runde wird offensichtlich der Startknoten s in S aufgenommen, und $d(s, s) = \texttt{dist}[s] = 0$.

I.V.: Sei $u \neq s$.

(I2) stimmt für alle Runden vor derjenigen, in der u bearbeitet, also in S aufgenommen wird.

I.S.: Betrachte Runde, in der $u \neq s$ bearbeitet wird.

Beweis von (I2) durch Induktion über Runden:

I.A.: Am Anfang ist S leer. In der ersten Runde wird offensichtlich der Startknoten s in S aufgenommen, und $d(s, s) = \texttt{dist}[s] = 0$.

I.V.: Sei $u \neq s$.

(I2) stimmt für alle Runden vor derjenigen, in der u bearbeitet, also in S aufgenommen wird.

I.S.: Betrachte Runde, in der $u \neq s$ bearbeitet wird.

Nach (I1) gibt es einen Weg von s nach u der Länge $\leq \texttt{dist}[u]$.

Beweis von (I2) durch Induktion über Runden:

I.A.: Am Anfang ist S leer. In der ersten Runde wird offensichtlich der Startknoten s in S aufgenommen, und $d(s, s) = \texttt{dist}[s] = 0$.

I.V.: Sei $u \neq s$.

(I2) stimmt für alle Runden vor derjenigen, in der u bearbeitet, also in S aufgenommen wird.

I.S.: Betrachte Runde, in der $u \neq s$ bearbeitet wird.

Nach (I1) gibt es einen Weg von s nach u der Länge $\leq \texttt{dist}[u]$.

Zu zeigen bleibt: Es gibt keinen Weg von s nach u, der kürzer als $\texttt{dist}[u]$ ist.

Beweis von (I2) durch Induktion über Runden:

I.A.: Am Anfang ist `S` leer. In der ersten Runde wird offensichtlich der Startknoten s in `S` aufgenommen, und $d(s, s) = \texttt{dist}[s] = 0$.

I.V.: Sei $u \neq s$.

(I2) stimmt für alle Runden vor derjenigen, in der u bearbeitet, also in `S` aufgenommen wird.

I.S.: Betrachte Runde, in der $u \neq s$ bearbeitet wird.

Nach (I1) gibt es einen Weg von s nach u der Länge $\leq \texttt{dist}[u]$.

Zu zeigen bleibt: Es gibt keinen Weg von s nach u, der kürzer als $\texttt{dist}[u]$ ist.

Sei $p = (s = v_0, v_1, \ldots, v_t = u)$ irgendein Weg von s nach u.

Beweis von (I2) durch Induktion über Runden:

I.A.: Am Anfang ist S leer. In der ersten Runde wird offensichtlich der Startknoten s in S aufgenommen, und $d(s, s) = \texttt{dist}[s] = 0$.

I.V.: Sei $u \neq s$.

(I2) stimmt für alle Runden vor derjenigen, in der u bearbeitet, also in S aufgenommen wird.

I.S.: Betrachte Runde, in der $u \neq s$ bearbeitet wird.

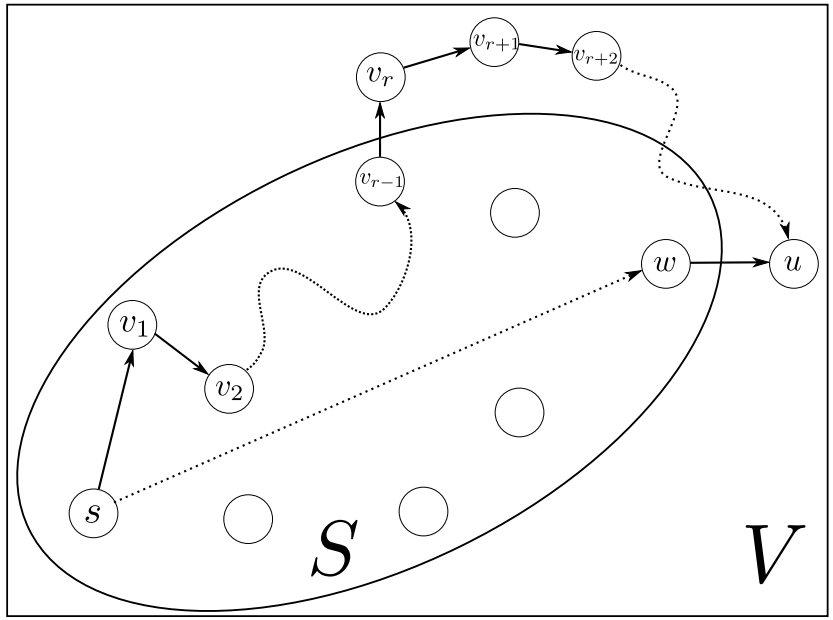
Nach (I1) gibt es einen Weg von s nach u der Länge $\leq \texttt{dist}[u]$.

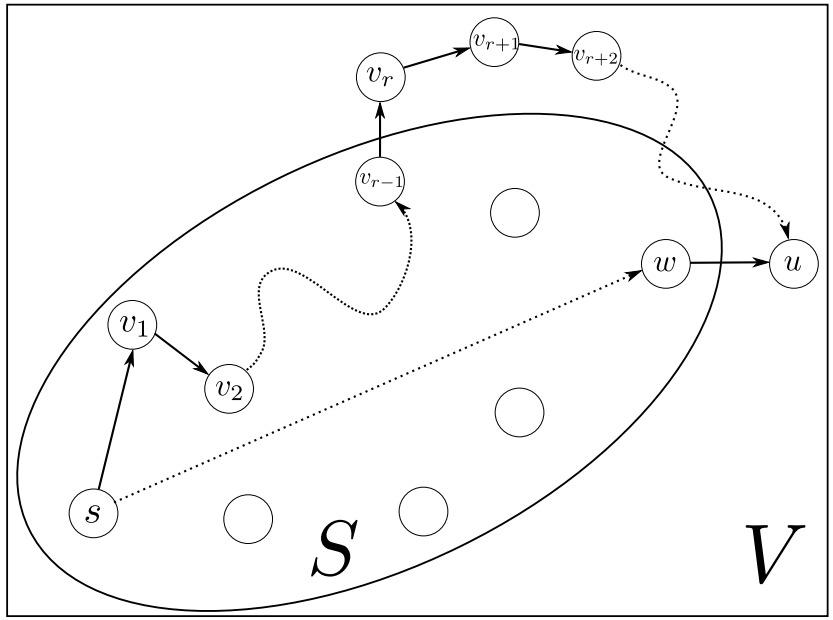
Zu zeigen bleibt: Es gibt keinen Weg von s nach u, der kürzer als $\texttt{dist}[u]$ ist.

Sei $p = (s = v_0, v_1, \ldots, v_t = u)$ irgendein Weg von s nach u.

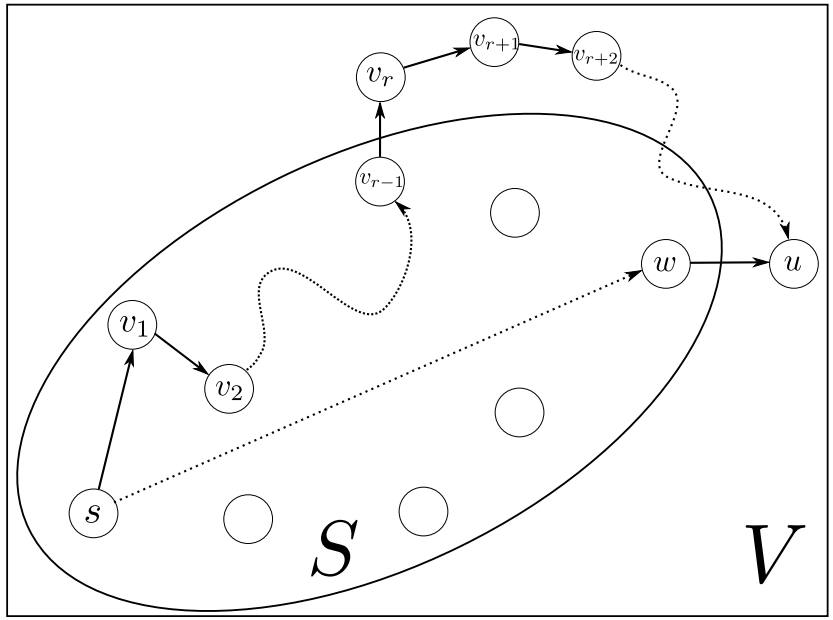
p beginnt in $v_0 = s \in$ S und endet in $v_t = u \in V -$ S.

Also gibt es ein r mit $s = v_0, v_1, \ldots, v_{r-1} \in$ S und $v_r \notin$ S.

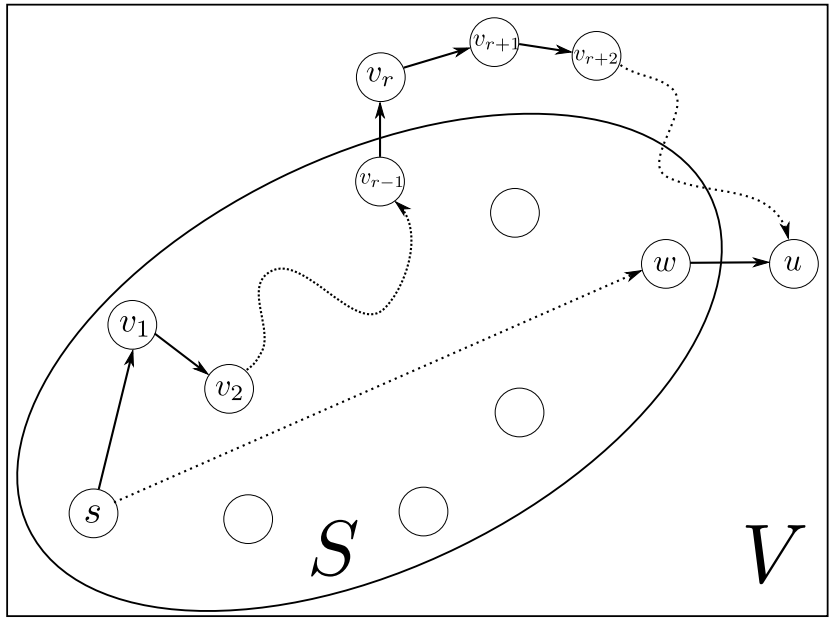

v_{r+1}
v_{r+2}
v_r
v_{r-1}
w
u
v_1
v_2
s
S
V



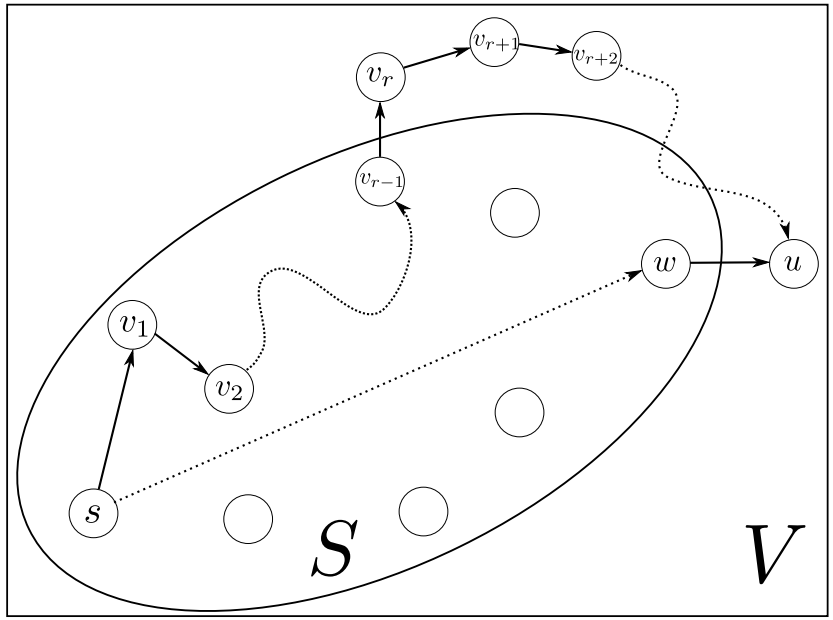
Nach der Definition der Distanzfunktion gilt für das Anfangsstück $p_{r-1} = (v_0, \ldots, v_{r-1})$ von p:



Nach der Definition der Distanzfunktion gilt für das Anfangsstück $p_{r-1} = (v_0, \ldots, v_{r-1})$ von p:
$c(p_{r-1}) \geq d(s, v_{r-1})$.



Nach der Definition der Distanzfunktion gilt für das Anfangsstück $p_{r-1} = (v_0, \ldots, v_{r-1})$ von p:
$c(p_{r-1}) \geq d(s, v_{r-1})$.
Nach I.V. für $v_{r-1} \in$ `S` gilt $d(s, v_{r-1}) =$ `dist`$[v_{r-1}]$.



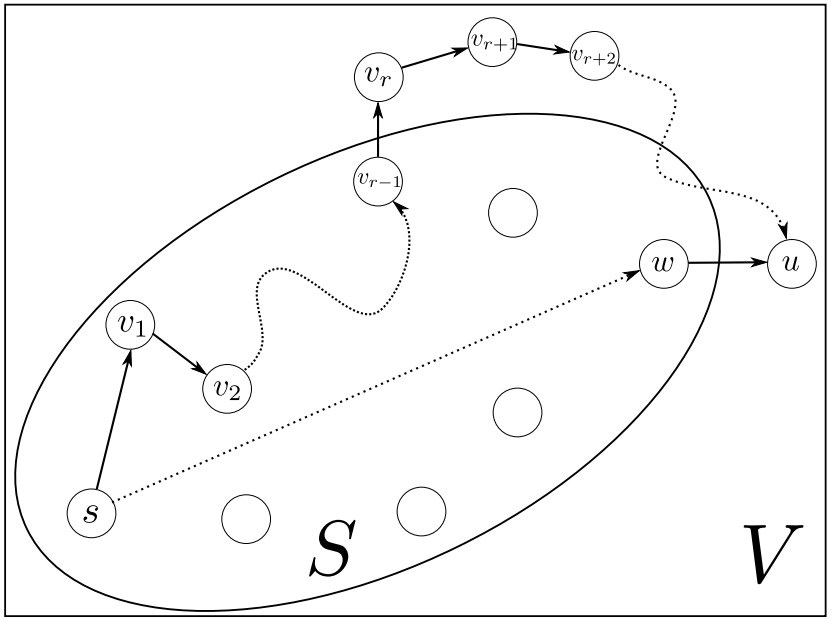
Nach der Definition der Distanzfunktion gilt für das Anfangsstück $p_{r-1} = (v_0, \ldots, v_{r-1})$ von p:

$c(p_{r-1}) \geq d(s, v_{r-1})$.

Nach I.V. für $v_{r-1} \in \texttt{S}$ gilt $d(s, v_{r-1}) = \texttt{dist}[v_{r-1}]$.

Also haben wir $c(p_r) \geq \texttt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$,

für das Anfangsstück $p_r = (v_0, \ldots, v_{r-1}, v_r)$ von p.



Nach der Definition der Distanzfunktion gilt für das Anfangsstück $p_{r-1} = (v_0, \ldots, v_{r-1})$ von p:

$c(p_{r-1}) \geq d(s, v_{r-1})$.

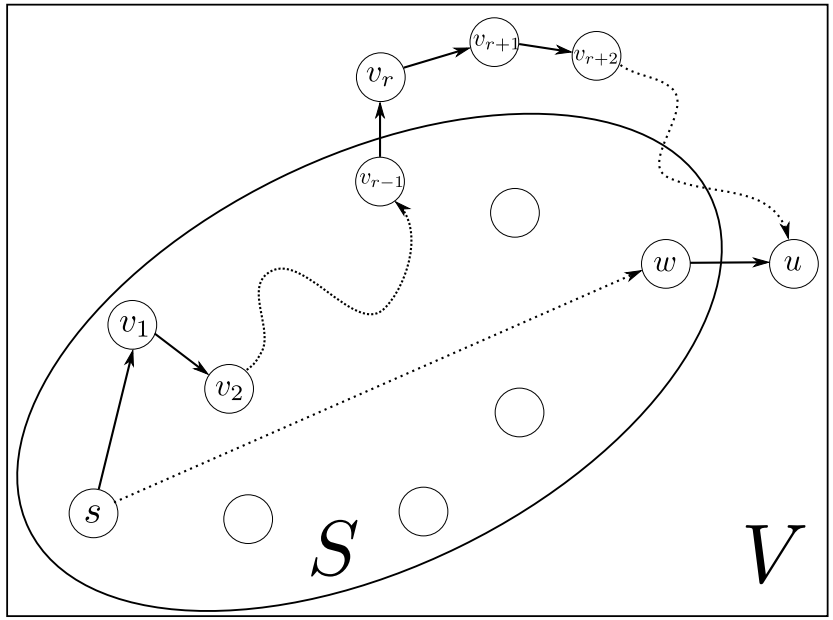
Nach I.V. für $v_{r-1} \in \texttt{S}$ gilt $d(s, v_{r-1}) = \texttt{dist}[v_{r-1}]$.

Also haben wir $c(p_r) \geq \texttt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$,

für das Anfangsstück $p_r = (v_0, \ldots, v_{r-1}, v_r)$ von p.

Weil nach Vor. alle **Kantengewichte nichtnegativ** sind, insbesondere also $c(p) \geq c(p_r)$ gilt, folgt

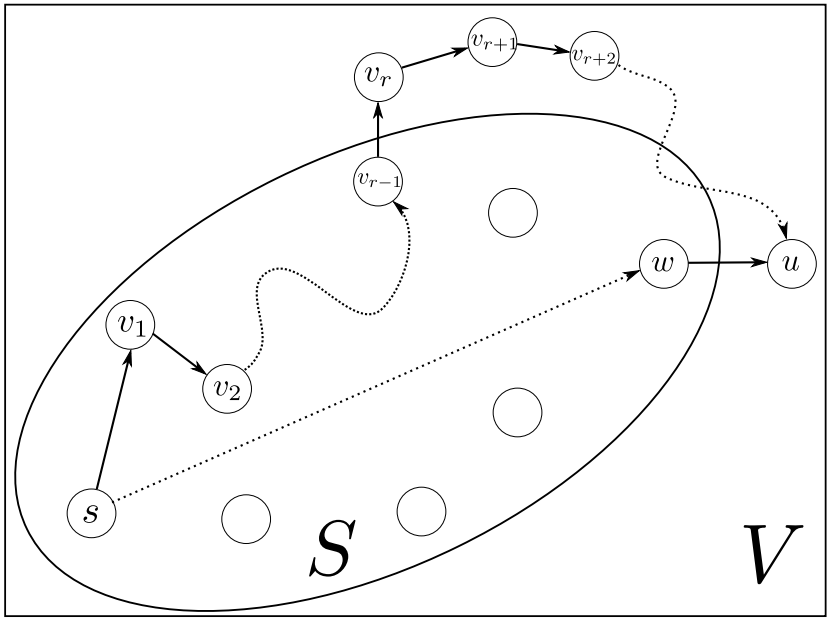
$(*)$ $\quad c(p) \geq \texttt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$.



Nach der Definition der Distanzfunktion gilt für das Anfangsstück $p_{r-1} = (v_0, \ldots, v_{r-1})$ von p:
$c(p_{r-1}) \geq d(s, v_{r-1})$.
Nach I.V. für $v_{r-1} \in \mathtt{S}$ gilt $d(s, v_{r-1}) = \mathtt{dist}[v_{r-1}]$.
Also haben wir $c(p_r) \geq \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$,
für das Anfangsstück $p_r = (v_0, \ldots, v_{r-1}, v_r)$ von p.
Weil nach Vor. alle **Kantengewichte nichtnegativ** sind, insbesondere also $c(p) \geq c(p_r)$ gilt, folgt

$(*)$ $\quad c(p) \geq \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$.

Weiter gilt: $\quad (**) \quad \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r) \geq \mathtt{dist}[v_r]$.

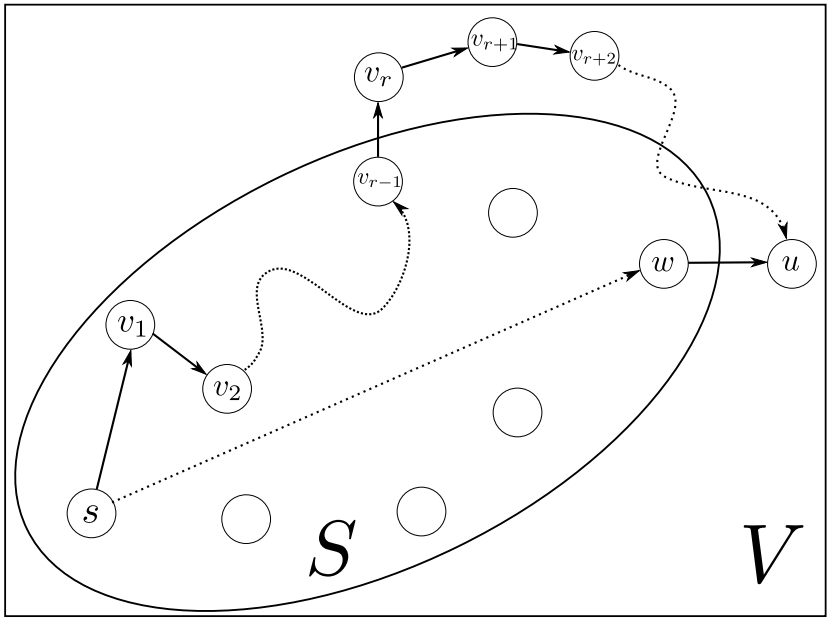


Nach der Definition der Distanzfunktion gilt für das Anfangsstück $p_{r-1} = (v_0, \ldots, v_{r-1})$ von p:
$c(p_{r-1}) \geq d(s, v_{r-1})$.
Nach I.V. für $v_{r-1} \in \mathtt{S}$ gilt $d(s, v_{r-1}) = \mathtt{dist}[v_{r-1}]$.
Also haben wir $c(p_r) \geq \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$,
für das Anfangsstück $p_r = (v_0, \ldots, v_{r-1}, v_r)$ von p.
Weil nach Vor. alle **Kantengewichte nichtnegativ** sind, insbesondere also $c(p) \geq c(p_r)$ gilt, folgt

$(*)$ $\quad c(p) \geq \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$.

Weiter gilt: $\quad (**) \quad \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r) \geq \mathtt{dist}[v_r]$.

(Denn: In der Runde, in der v_{r-1} bearbeitet wurde, wurden $\mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$ und $\mathtt{dist}[v_r]$ verglichen, und $(**)$ wurde erzwungen. Wenn sich $\mathtt{dist}[v_r]$ danach noch geändert hat, ist es nur kleiner geworden.)

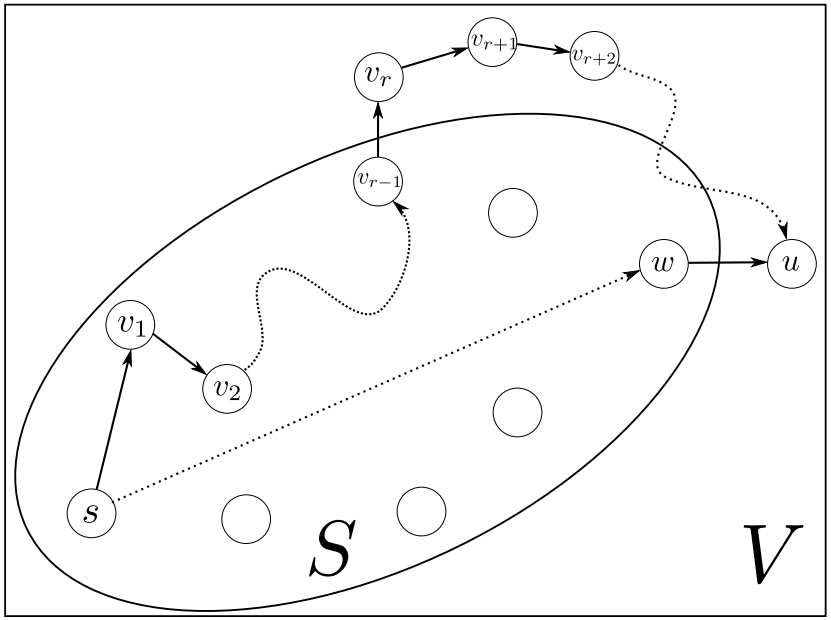


Nach der Definition der Distanzfunktion gilt für das Anfangsstück $p_{r-1} = (v_0, \ldots, v_{r-1})$ von p:
$c(p_{r-1}) \geq d(s, v_{r-1})$.
Nach I.V. für $v_{r-1} \in \mathtt{S}$ gilt $d(s, v_{r-1}) = \mathtt{dist}[v_{r-1}]$.
Also haben wir $c(p_r) \geq \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$,
für das Anfangsstück $p_r = (v_0, \ldots, v_{r-1}, v_r)$ von p.
Weil nach Vor. alle **Kantengewichte nichtnegativ** sind, insbesondere also $c(p) \geq c(p_r)$ gilt, folgt

$(*)$ $\quad c(p) \geq \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$.

Weiter gilt: $(**)$ $\quad \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r) \geq \mathtt{dist}[v_r]$.
(Denn: In der Runde, in der v_{r-1} bearbeitet wurde, wurden $\mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$ und $\mathtt{dist}[v_r]$ verglichen, und $(**)$ wurde erzwungen. Wenn sich $\mathtt{dist}[v_r]$ danach noch geändert hat, ist es nur kleiner geworden.)

Schließlich gilt in der aktuellen Runde: $(***)$ $\quad \mathtt{dist}[v_r] \geq \mathtt{dist}[u]$.



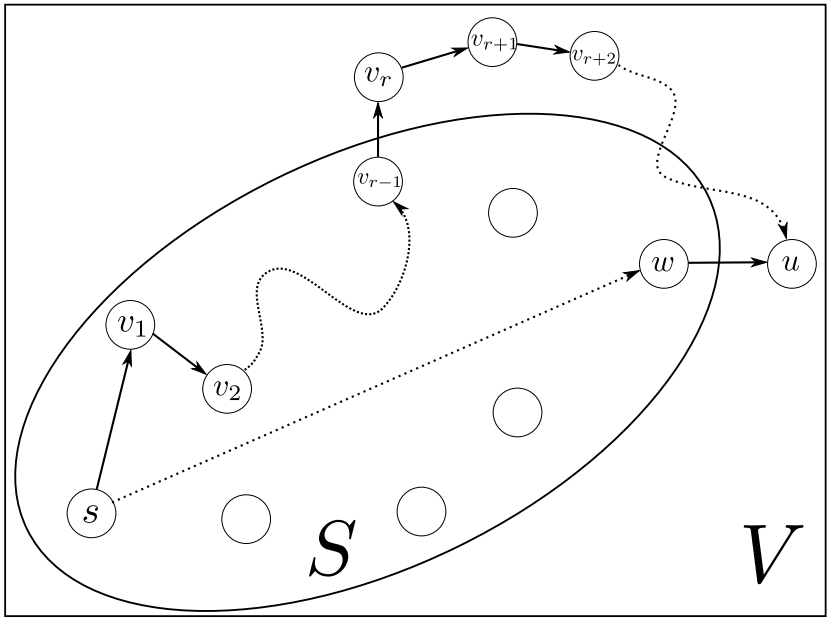
Nach der Definition der Distanzfunktion gilt für das Anfangsstück $p_{r-1} = (v_0, \ldots, v_{r-1})$ von p:
$c(p_{r-1}) \geq d(s, v_{r-1})$.
Nach I.V. für $v_{r-1} \in \mathtt{S}$ gilt $d(s, v_{r-1}) = \mathtt{dist}[v_{r-1}]$.
Also haben wir $c(p_r) \geq \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$,
für das Anfangsstück $p_r = (v_0, \ldots, v_{r-1}, v_r)$ von p.
Weil nach Vor. alle **Kantengewichte nichtnegativ** sind, insbesondere also $c(p) \geq c(p_r)$ gilt, folgt

$(*)$ $\quad c(p) \geq \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$.

Weiter gilt: $(**)$ $\quad \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r) \geq \mathtt{dist}[v_r]$.
(Denn: In der Runde, in der v_{r-1} bearbeitet wurde, wurden $\mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$ und $\mathtt{dist}[v_r]$ verglichen, und $(**)$ wurde erzwungen. Wenn sich $\mathtt{dist}[v_r]$ danach noch geändert hat, ist es nur kleiner geworden.)

Schließlich gilt in der aktuellen Runde: $(***)$ $\quad \mathtt{dist}[v_r] \geq \mathtt{dist}[u]$.

Dies liegt daran, dass der Algorithmus einen Knoten mit kleinstem $\mathtt{dist}[v]$-Wert als u wählt.



Nach der Definition der Distanzfunktion gilt für das Anfangsstück $p_{r-1} = (v_0, \ldots, v_{r-1})$ von p:
$c(p_{r-1}) \geq d(s, v_{r-1})$.
Nach I.V. für $v_{r-1} \in \mathtt{S}$ gilt $d(s, v_{r-1}) = \mathtt{dist}[v_{r-1}]$.
Also haben wir $c(p_r) \geq \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$,
für das Anfangsstück $p_r = (v_0, \ldots, v_{r-1}, v_r)$ von p.
Weil nach Vor. alle **Kantengewichte nichtnegativ** sind, insbesondere also $c(p) \geq c(p_r)$ gilt, folgt

$(*)$ $\quad c(p) \geq \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$.

Weiter gilt: $(**)$ $\quad \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r) \geq \mathtt{dist}[v_r]$.
(Denn: In der Runde, in der v_{r-1} bearbeitet wurde, wurden $\mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$ und $\mathtt{dist}[v_r]$ verglichen, und $(**)$ wurde erzwungen. Wenn sich $\mathtt{dist}[v_r]$ danach noch geändert hat, ist es nur kleiner geworden.)

Schließlich gilt in der aktuellen Runde: $(***)$ $\quad \mathtt{dist}[v_r] \geq \mathtt{dist}[u]$.

Dies liegt daran, dass der Algorithmus einen Knoten mit kleinstem $\mathtt{dist}[v]$-Wert als u wählt.

Kombinieren von $(*)$, $(**)$ und $(***)$ liefert $c(p) \geq \mathtt{dist}[u]$, wie gewünscht. $\square$

Wir wollen aber eigentlich nicht nur die Distanzen $d(s, v)$, sondern **kürzeste Wege** berechnen.

Wir wollen aber eigentlich nicht nur die Distanzen $d(s, v)$, sondern **kürzeste Wege** berechnen.

Idee: Für jeden Knoten v merken wir uns, über welche Kante (u, v) Knoten v „vom Feuer erreicht wurde“.

Wir wollen aber eigentlich nicht nur die Distanzen $d(s, v)$, sondern **kürzeste Wege** berechnen.

Idee: Für jeden Knoten v merken wir uns, über welche Kante (u, v) Knoten v „vom Feuer erreicht wurde“.

Wenn wir diese „Vorgänger-Information“ benutzen, um von v ausgehend immer weiter rückwärts zu laufen, bis wir s erreichen, erhalten wir einen kürzesten Weg.

Wir wollen aber eigentlich nicht nur die Distanzen $d(s, v)$, sondern **kürzeste Wege** berechnen.

Idee: Für jeden Knoten v merken wir uns, über welche Kante (u, v) Knoten v „vom Feuer erreicht wurde“.

Wenn wir diese „Vorgänger-Information“ benutzen, um von v ausgehend immer weiter rückwärts zu laufen, bis wir s erreichen, erhalten wir einen kürzesten Weg.

Technisch: Für jeden gefundenen Knoten $w \notin \mathtt{S}$ notiere $p(w) \in \mathtt{S}$ mit $(p(w), w) \in E$ und $\mathtt{dist}[w] = \mathtt{dist}[p(w)] + c(p(w), w)$ $(= d(s, p(w)) + c(p(w), w))$.

Wir wollen aber eigentlich nicht nur die Distanzen $d(s, v)$, sondern **kürzeste Wege** berechnen.

Idee: Für jeden Knoten v merken wir uns, über welche Kante (u, v) Knoten v „vom Feuer erreicht wurde“.

Wenn wir diese „Vorgänger-Information“ benutzen, um von v ausgehend immer weiter rückwärts zu laufen, bis wir s erreichen, erhalten wir einen kürzesten Weg.

Technisch: Für jeden gefundenen Knoten $w \notin \mathtt{S}$ notiere $p(w) \in \mathtt{S}$ mit $(p(w), w) \in E$ und $\mathtt{dist}[w] = \mathtt{dist}[p(w)] + c(p(w), w)$ $(= d(s, p(w)) + c(p(w), w))$.

Ab dem Moment, in dem w bearbeitet wird, ändert sich $p(w)$ nicht mehr.

Wir wollen aber eigentlich nicht nur die Distanzen $d(s, v)$, sondern **kürzeste Wege** berechnen.

Idee: Für jeden Knoten v merken wir uns, über welche Kante (u, v) Knoten v „vom Feuer erreicht wurde“.

Wenn wir diese „Vorgänger-Information“ benutzen, um von v ausgehend immer weiter rückwärts zu laufen, bis wir s erreichen, erhalten wir einen kürzesten Weg.

Technisch: Für jeden gefundenen Knoten $w \notin \mathtt{S}$ notiere $p(w) \in \mathtt{S}$ mit $(p(w), w) \in E$ und $\mathtt{dist}[w] = \mathtt{dist}[p(w)] + c(p(w), w)$ $(= d(s, p(w)) + c(p(w), w))$.

Ab dem Moment, in dem w bearbeitet wird, ändert sich $p(w)$ nicht mehr.

Datenstruktur für die Vorgänger: $\mathtt{p}[1..n]$.

Notwendige Ergänzungen:

(2+). . . $\texttt{p}[s] \leftarrow -2$; // Sonderfall Wurzel

(3+) **for** $\texttt{w} \in V - \{s\}$ **do** . . . $\texttt{p}[\texttt{w}] \leftarrow -1$; // „undefiniert“

Notwendige Ergänzungen:

(2+). . . $\texttt{p}[s] \leftarrow -2$; // Sonderfall Wurzel

(3+) **for** $\texttt{w} \in V - \{s\}$ **do** . . . $\texttt{p}[\texttt{w}] \leftarrow -1$; // „undefiniert“

Aktualisierung in den späteren Runden:

(7) **for** $(\texttt{u}, \texttt{w}) \in E$ mit $\texttt{w} \notin \texttt{S}$ **do** // Nachfolger von u, nicht bearbeitet: „update(u, w)“

Notwendige Ergänzungen:

(2+). . . $\texttt{p}[s] \leftarrow -2$; // Sonderfall Wurzel

(3+) **for** $\texttt{w} \in V - \{s\}$ **do** . . . $\texttt{p}[\texttt{w}] \leftarrow -1$; // „undefiniert“

Aktualisierung in den späteren Runden:

(7) **for** $(\texttt{u}, \texttt{w}) \in E$ mit $\texttt{w} \notin \texttt{S}$ **do** // Nachfolger von u, nicht bearbeitet: „update(u, w)“
(8a) $\texttt{dd} \leftarrow \texttt{dist}[\texttt{u}] + c(\texttt{u}, \texttt{w})$;
(8b) **if** $\texttt{dd} < \texttt{dist}[\texttt{w}]$ **then**
(8c) $\texttt{dist}[\texttt{w}] \leftarrow \texttt{dd}$;
(8d) $\texttt{p}[\texttt{w}] \leftarrow \texttt{u}$;

Algorithmus Dijkstra(G, s)

Eingabe: gewichteter Digraph $G = (V, E, c)$, Startknoten $s \in V$

Ausgabe: Länge $d(s, v)$ der kürzesten Wege, Vorgängerknoten $p(v)$

(1) $\mathtt{S} \leftarrow \emptyset$;

(2+) $\mathtt{dist}[s] \leftarrow 0$; $\mathtt{p}[s] \leftarrow -2$;

(3+) **for** $\mathtt{w} \in V - \{s\}$ **do** $\mathtt{dist}[\mathtt{w}] \leftarrow \infty$; $\mathtt{p}[\mathtt{w}] \leftarrow -1$;

(4) **while** $\exists u \in V - \mathtt{S}$: $\mathtt{dist}[u] < \infty$ **do**

(5) $\quad \mathtt{u} \leftarrow$ ein solcher Knoten u mit minimalem $\mathtt{dist}[u]$;

(6) $\quad \mathtt{S} \leftarrow \mathtt{S} \cup \{\mathtt{u}\}$;

(7) $\quad$ **for** $(\mathtt{u}, \mathtt{w}) \in E$ mit $\mathtt{w} \notin \mathtt{S}$ **do** // Nachfolger von u, nicht bearbeitet: „update(u, w)“

(8a) $\quad \mathtt{dd} \leftarrow \mathtt{dist}[\mathtt{u}] + c(\mathtt{u}, \mathtt{w})$;

(8b) $\quad$ **if** $\mathtt{dd} < \mathtt{dist}[\mathtt{w}]$ **then**

(8c) $\quad\quad \mathtt{dist}[\mathtt{w}] \leftarrow \mathtt{dd}$;

(8d) $\quad\quad \mathtt{p}[\mathtt{w}] \leftarrow \mathtt{u}$;

(9+) **Ausgabe:** $\mathtt{dist}[1..n]$ und $\mathtt{p}[1..n]$.

Nach dem Algorithmus ist klar, dass $\texttt{p}[v] \neq -1$ („undefiniert“) genau dann gilt, wenn $\texttt{dist}[v] < \infty$.

Nach dem Algorithmus ist klar, dass $\texttt{p}[v] \neq -1$ („undefiniert“) genau dann gilt, wenn $\texttt{dist}[v] < \infty$.

$\texttt{p}[v] = -2$ („Startknoten, hat keinen Vorgänger“) gilt nur für $v = s$.

Nach dem Algorithmus ist klar, dass $\texttt{p}[v] \neq -1$ („undefiniert“) genau dann gilt, wenn $\texttt{dist}[v] < \infty$.

$\texttt{p}[v] = -2$ („Startknoten, hat keinen Vorgänger“) gilt nur für $v = s$.

Definition Ein (einfacher) Weg $(s = v_0, v_1, \ldots, v_t)$ in G heißt ein ***S*-Weg**, wenn alle Knoten außer eventuell v_t in S liegen.

Nach dem Algorithmus ist klar, dass $\texttt{p}[v] \neq -1$ („undefiniert“) genau dann gilt, wenn $\texttt{dist}[v] < \infty$.

$\texttt{p}[v] = -2$ („Startknoten, hat keinen Vorgänger“) gilt nur für $v = s$.

Definition Ein (einfacher) Weg $(s = v_0, v_1, \ldots, v_t)$ in G heißt ein ***S*-Weg**, wenn alle Knoten außer eventuell v_t in S liegen.

Behauptung: Neben (I1) und (I2) gelten nach jeder Runde die folgenden Invarianten:

Nach dem Algorithmus ist klar, dass $\texttt{p}[v] \neq -1$ („undefiniert“) genau dann gilt, wenn $\texttt{dist}[v] < \infty$.

$\texttt{p}[v] = -2$ („Startknoten, hat keinen Vorgänger“) gilt nur für $v = s$.

Definition Ein (einfacher) Weg $(s = v_0, v_1, \ldots, v_t)$ in G heißt ein ***S*-Weg**, wenn alle Knoten außer eventuell v_t in S liegen.

Behauptung: Neben (I1) und (I2) gelten nach jeder Runde die folgenden Invarianten:

(I3) Wenn $v \in \texttt{S}$ und $v \neq s$, dann gilt $\texttt{p}[v] \neq -1$ und

Nach dem Algorithmus ist klar, dass $\texttt{p}[v] \neq -1$ („undefiniert“) genau dann gilt, wenn $\texttt{dist}[v] < \infty$.

$\texttt{p}[v] = -2$ („Startknoten, hat keinen Vorgänger“) gilt nur für $v = s$.

Definition Ein (einfacher) Weg $(s = v_0, v_1, \ldots, v_t)$ in G heißt ein ***S*-Weg**, wenn alle Knoten außer eventuell v_t in S liegen.

Behauptung: Neben (I1) und (I2) gelten nach jeder Runde die folgenden Invarianten:

(I3) Wenn $v \in \texttt{S}$ und $v \neq s$, dann gilt $\texttt{p}[v] \neq -1$ und $\texttt{p}[v]$ ist vorletzter Knoten auf einem S-Weg von s nach v der Länge $d(s, v)$.

Nach dem Algorithmus ist klar, dass $\texttt{p}[v] \neq -1$ („undefiniert“) genau dann gilt, wenn $\texttt{dist}[v] < \infty$.

$\texttt{p}[v] = -2$ („Startknoten, hat keinen Vorgänger“) gilt nur für $v = s$.

Definition Ein (einfacher) Weg $(s = v_0, v_1, \ldots, v_t)$ in G heißt ein ***S*-Weg**, wenn alle Knoten außer eventuell v_t in S liegen.

Behauptung: Neben (I1) und (I2) gelten nach jeder Runde die folgenden Invarianten:

(I3) Wenn $v \in \texttt{S}$ und $v \neq s$, dann gilt $\texttt{p}[v] \neq -1$ und $\texttt{p}[v]$ ist vorletzter Knoten auf einem S-Weg von s nach v der Länge $d(s, v)$.

(I4) Wenn $w \notin \texttt{S}$ und $w \neq s$ und $\texttt{dist}[w] < \infty$, dann gilt:

Nach dem Algorithmus ist klar, dass $\texttt{p}[v] \neq -1$ („undefiniert“) genau dann gilt, wenn $\texttt{dist}[v] < \infty$.

$\texttt{p}[v] = -2$ („Startknoten, hat keinen Vorgänger“) gilt nur für $v = s$.

Definition Ein (einfacher) Weg $(s = v_0, v_1, \ldots, v_t)$ in G heißt ein ***S*-Weg**, wenn alle Knoten außer eventuell v_t in S liegen.

Behauptung: Neben (I1) und (I2) gelten nach jeder Runde die folgenden Invarianten:

(I3) Wenn $v \in \texttt{S}$ und $v \neq s$, dann gilt $\texttt{p}[v] \neq -1$ und $\texttt{p}[v]$ ist vorletzter Knoten auf einem S-Weg von s nach v der Länge $d(s, v)$.

(I4) Wenn $w \notin \texttt{S}$ und $w \neq s$ und $\texttt{dist}[w] < \infty$, dann gilt: $\texttt{p}[w] \in \texttt{S}$ und $\texttt{p}[w]$ ist letzter S-Knoten auf einem S-Weg von s nach w der Länge $\texttt{dist}[w]$.

Nach dem Algorithmus ist klar, dass $\texttt{p}[v] \neq -1$ („undefiniert“) genau dann gilt, wenn $\texttt{dist}[v] < \infty$.

$\texttt{p}[v] = -2$ („Startknoten, hat keinen Vorgänger“) gilt nur für $v = s$.

Definition Ein (einfacher) Weg $(s = v_0, v_1, \ldots, v_t)$ in G heißt ein ***S*-Weg**, wenn alle Knoten außer eventuell v_t in S liegen.

Behauptung: Neben (I1) und (I2) gelten nach jeder Runde die folgenden Invarianten:

(I3) Wenn $v \in \texttt{S}$ und $v \neq s$, dann gilt $\texttt{p}[v] \neq -1$ und $\texttt{p}[v]$ ist vorletzter Knoten auf einem S-Weg von s nach v der Länge $d(s, v)$.

(I4) Wenn $w \notin \texttt{S}$ und $w \neq s$ und $\texttt{dist}[w] < \infty$, dann gilt: $\texttt{p}[w] \in \texttt{S}$ und $\texttt{p}[w]$ ist letzter S-Knoten auf einem S-Weg von s nach w der Länge $\texttt{dist}[w]$.

Beweis von (I3) und (I4): Induktion über Runden. Für Interessierte: Auf Druckfolien.

Ergebnis:

Wenn der Algorithmus von Dijkstra anhält, führen die $\texttt{p}[v]$-Verweise von jedem Knoten v aus entlang eines kürzesten Weges (zurück) zu s.

Ergebnis:

Wenn der Algorithmus von Dijkstra anhält, führen die $\mathtt{p}[v]$-Verweise von jedem Knoten v aus entlang eines kürzesten Weges (zurück) zu s.

Da die $\mathtt{p}[v]$-Verweise keinen Kreis bilden können (mit dem Schritt $\mathtt{S} \leftarrow \mathtt{S} \cup \{u\}$ wird der Verweis vom neuen S-Knoten u auf den S-Knoten $\mathtt{p}[u]$ endgültig fixiert)

Ergebnis:

Wenn der Algorithmus von Dijkstra anhält, führen die $\mathtt{p}[v]$-Verweise von jedem Knoten v aus entlang eines kürzesten Weges (zurück) zu s.

Da die $\mathtt{p}[v]$-Verweise keinen Kreis bilden können (mit dem Schritt $\mathtt{S} \leftarrow \mathtt{S} \cup \{u\}$ wird der Verweis vom neuen S-Knoten u auf den S-Knoten $\mathtt{p}[u]$ endgültig fixiert), bilden die Kanten $(\mathtt{p}[v], v)$ einen **Baum**, den sogenannten

Baum der kürzesten Wege.

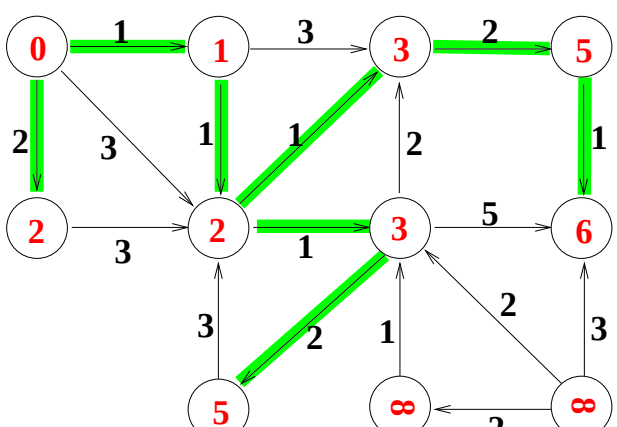
Ergebnis:

Wenn der Algorithmus von Dijkstra anhält, führen die $\mathtt{p}[v]$-Verweise von jedem Knoten v aus entlang eines kürzesten Weges (zurück) zu s.

Da die $\mathtt{p}[v]$-Verweise keinen Kreis bilden können (mit dem Schritt $\mathtt{S} \leftarrow \mathtt{S} \cup \{u\}$ wird der Verweis vom neuen S-Knoten u auf den S-Knoten $\mathtt{p}[u]$ endgültig fixiert), bilden die Kanten $(\mathtt{p}[v], v)$ einen **Baum**, den sogenannten

Baum der kürzesten Wege.

Beispiel:



Wieso steht der Algorithmus von Dijkstra unter der Überschrift „Greedy-Algorithmen“?

Wieso steht der Algorithmus von Dijkstra unter der Überschrift „Greedy-Algorithmen“?
Er hält eine sehr geschickte Datenstruktur bereit, wählt dann aber in jeder Runde nach dem Greedy-Muster als nächsten einzubeziehenden Knoten einen, dessen `dist`$[v]$-Wert minimal ist.

Wieso steht der Algorithmus von Dijkstra unter der Überschrift „Greedy-Algorithmen“?
Er hält eine sehr geschickte Datenstruktur bereit, wählt dann aber in jeder Runde nach dem Greedy-Muster als nächsten einzubeziehenden Knoten einen, dessen `dist`$[v]$-Wert minimal ist.
Weiter unten lernen wir den Algorithmus von Jarník/Prim für das Problem „Minimaler Spannbaum“ kennen, der ein Paradebeispiel für einen „Greedy“-Algorithmus ist und der sich nur minimal vom Algorithmus von Dijkstra unterscheidet.

Wieso steht der Algorithmus von Dijkstra unter der Überschrift „Greedy-Algorithmen“?
Er hält eine sehr geschickte Datenstruktur bereit, wählt dann aber in jeder Runde nach dem Greedy-Muster als nächsten einzubeziehenden Knoten einen, dessen `dist`$[v]$-Wert minimal ist.
Weiter unten lernen wir den Algorithmus von Jarník/Prim für das Problem „Minimaler Spannbaum“ kennen, der ein Paradebeispiel für einen „Greedy“-Algorithmus ist und der sich nur minimal vom Algorithmus von Dijkstra unterscheidet.

Implementierungsdetails:

Noch zu klären:
Wie findet man **effizient** einen Knoten u mit kleinstem Wert `dist`$[u]$?

Wieso steht der Algorithmus von Dijkstra unter der Überschrift „Greedy-Algorithmen“?
Er hält eine sehr geschickte Datenstruktur bereit, wählt dann aber in jeder Runde nach dem Greedy-Muster als nächsten einzubeziehenden Knoten einen, dessen `dist`$[v]$-Wert minimal ist.
Weiter unten lernen wir den Algorithmus von Jarník/Prim für das Problem „Minimaler Spannbaum“ kennen, der ein Paradebeispiel für einen „Greedy“-Algorithmus ist und der sich nur minimal vom Algorithmus von Dijkstra unterscheidet.

Implementierungsdetails:

Noch zu klären:
Wie findet man **effizient** einen Knoten u mit kleinstem Wert `dist`$[u]$?

Einfache Lösung:
Durchsuche in jeder Runde das `dist`-Array, um das Minimum zu finden.

Wieso steht der Algorithmus von Dijkstra unter der Überschrift „Greedy-Algorithmen“?
Er hält eine sehr geschickte Datenstruktur bereit, wählt dann aber in jeder Runde nach dem Greedy-Muster als nächsten einzubeziehenden Knoten einen, dessen `dist`$[v]$-Wert minimal ist.
Weiter unten lernen wir den Algorithmus von Jarník/Prim für das Problem „Minimaler Spannbaum“ kennen, der ein Paradebeispiel für einen „Greedy“-Algorithmus ist und der sich nur minimal vom Algorithmus von Dijkstra unterscheidet.

Implementierungsdetails:

Noch zu klären:
Wie findet man **effizient** einen Knoten u mit kleinstem Wert `dist`$[u]$?

Einfache Lösung:
Durchsuche in jeder Runde das `dist`-Array, um das Minimum zu finden.

Dann benötigt jede Runde Zeit $\Theta(n)$, und die Gesamtrechenzeit des Algorithmus von Dijkstra ergibt sich zu $\Theta(n^2)$.

Wieso steht der Algorithmus von Dijkstra unter der Überschrift „Greedy-Algorithmen“?
Er hält eine sehr geschickte Datenstruktur bereit, wählt dann aber in jeder Runde nach dem Greedy-Muster als nächsten einzubeziehenden Knoten einen, dessen `dist`$[v]$-Wert minimal ist.
Weiter unten lernen wir den Algorithmus von Jarník/Prim für das Problem „Minimaler Spannbaum“ kennen, der ein Paradebeispiel für einen „Greedy“-Algorithmus ist und der sich nur minimal vom Algorithmus von Dijkstra unterscheidet.

Implementierungsdetails:

Noch zu klären:
Wie findet man **effizient** einen Knoten u mit kleinstem Wert `dist`$[u]$?

Einfache Lösung:
Durchsuche in jeder Runde das `dist`-Array, um das Minimum zu finden.

Dann benötigt jede Runde Zeit $\Theta(n)$, und die Gesamtrechenzeit des Algorithmus von Dijkstra ergibt sich zu $\Theta(n^2)$.

Dies ist für „dichte“ Graphen, also Graphen mit sehr vielen Kanten, akzeptabel, **nicht** aber für Graphen mit $|E| \ll |V|^2$ („dünn besetzte“ Graphen).

Effiziente Alternative:

Verwalte die Knoten $w \in V - \mathtt{S}$ mit Werten $\mathtt{dist}[w] < \infty$ mit den $\mathtt{dist}[w]$-Werten als **Schlüssel**

Effiziente Alternative:

Verwalte die Knoten $w \in V - \mathtt{S}$ mit Werten $\mathtt{dist}[w] < \infty$ mit den $\mathtt{dist}[w]$-Werten als **Schlüssel** in einer

Prioritätswarteschlange PQ.

Effiziente Alternative:

Verwalte die Knoten $w \in V - \mathtt{S}$ mit Werten $\mathtt{dist}[w] < \infty$ mit den $\mathtt{dist}[w]$-Werten als **Schlüssel** in einer

Prioritätswarteschlange PQ.

Wenn $\mathtt{dist}[w] = \infty$, ist w (noch) nicht in **PQ**.

Effiziente Alternative:

Verwalte die Knoten $w \in V - \mathtt{S}$ mit Werten $\mathtt{dist}[w] < \infty$ mit den $\mathtt{dist}[w]$-Werten als **Schlüssel** in einer

Prioritätswarteschlange PQ.

Wenn $\mathtt{dist}[w] = \infty$, ist w (noch) nicht in **PQ**.

extractMin liefert den nächsten Knoten u, der zu $\mathtt{S}$ hinzugefügt werden soll.

Effiziente Alternative:

Verwalte die Knoten $w \in V - \mathtt{S}$ mit Werten $\mathtt{dist}[w] < \infty$ mit den $\mathtt{dist}[w]$-Werten als **Schlüssel** in einer

Prioritätswarteschlange PQ.

Wenn $\mathtt{dist}[w] = \infty$, ist w (noch) nicht in **PQ**.

extractMin liefert den nächsten Knoten u, der zu S hinzugefügt werden soll.

`inS: array[1..`n`] of Boolean` (Knoten in S).

Dijkstra(G, s) // (Vollversion mit Prioritätswarteschlange)

Eingabe: gewichteter Digraph $G = (V, E, c)$, $V = \{1, \ldots, n\}$, Startknoten s;
Ausgabe: Länge $d(s, v)$ der kürzesten Wege, Vorgängerknoten $p(v)$
Hilfsdatenstrukturen: **PQ**: eine (anfangs leere) Prioritäts-WS; `inS`, `p`, `dist`: s.o.

```
(1)   for w from 1 to n do
(2)     dist[w] ← ∞; inS[w] ← false; p[w] ← −1;
(3)   dist[s] ← 0; p[s] ← −2; PQ.insert(s);
(4)   while not PQ.isempty do
(5)     u ← PQ.extractMin; inS[u] ← true;   // u wird bearbeitet
(6)     for Knoten w mit (u, w) ∈ E and not inS[w] do
(7)       dd ← dist[u] + c(u, w);
(8)       if p[w] ≥ 0 and dd < dist[w] then
(9)         PQ.decreaseKey(w,dd); p[w] ← u; dist[w] ← dd;
(10)      if p[w] = −1 then   // w wird soeben entdeckt
(11)        dist[w] ← dd; p[w] ← u; PQ.insert(w);
(12)  Ausgabe: dist[1..n] und p[1..n].
```

Aufwandsanalyse:

Aufwandsanalyse:

Die Prioritätswarteschlange realisieren wir als (binären) Heap (s. Abschnitt 6.4).

Aufwandsanalyse:

Die Prioritätswarteschlange realisieren wir als (binären) Heap (s. Abschnitt 6.4).

Maximale Anzahl von Einträgen: n.

Aufwandsanalyse:

Die Prioritätswarteschlange realisieren wir als (binären) Heap (s. Abschnitt 6.4).

Maximale Anzahl von Einträgen: n.

Initialisierung: Zeit $O(1)$ für **PQ**, $O(n)$ für den Rest.

Aufwandsanalyse:

Die Prioritätswarteschlange realisieren wir als (binären) Heap (s. Abschnitt 6.4).

Maximale Anzahl von Einträgen: n.

Initialisierung: Zeit $O(1)$ für **PQ**, $O(n)$ für den Rest.

Es gibt maximal n Durchläufe durch die **while**-Schleife mit Organisationsaufwand jeweils $O(1)$, zusammen also Kosten $O(n)$ für die Schleifenorganisation.

Aufwandsanalyse:

Die Prioritätswarteschlange realisieren wir als (binären) Heap (s. Abschnitt 6.4).

Maximale Anzahl von Einträgen: n.

Initialisierung: Zeit $O(1)$ für **PQ**, $O(n)$ für den Rest.

Es gibt maximal n Durchläufe durch die **while**-Schleife mit Organisationsaufwand jeweils $O(1)$, zusammen also Kosten $O(n)$ für die Schleifenorganisation.

In Schleifendurchlauf Nummer t, in dem u_t bearbeitet wird:

Aufwandsanalyse:

Die Prioritätswarteschlange realisieren wir als (binären) Heap (s. Abschnitt 6.4).

Maximale Anzahl von Einträgen: n.

Initialisierung: Zeit $O(1)$ für **PQ**, $O(n)$ für den Rest.

Es gibt maximal n Durchläufe durch die **while**-Schleife mit Organisationsaufwand jeweils $O(1)$, zusammen also Kosten $O(n)$ für die Schleifenorganisation.

In Schleifendurchlauf Nummer t, in dem u_t bearbeitet wird:
PQ.extractMin kostet Zeit $O(\log n)$.

Aufwandsanalyse:

Die Prioritätswarteschlange realisieren wir als (binären) Heap (s. Abschnitt 6.4).

Maximale Anzahl von Einträgen: n.

Initialisierung: Zeit $O(1)$ für **PQ**, $O(n)$ für den Rest.

Es gibt maximal n Durchläufe durch die **while**-Schleife mit Organisationsaufwand jeweils $O(1)$, zusammen also Kosten $O(n)$ für die Schleifenorganisation.

In Schleifendurchlauf Nummer t, in dem u_t bearbeitet wird:
PQ.extractMin kostet Zeit $O(\log n)$.
Durchmustern der $\deg(u_t)$ Nachbarn von u_t:

Aufwandsanalyse:

Die Prioritätswarteschlange realisieren wir als (binären) Heap (s. Abschnitt 6.4).

Maximale Anzahl von Einträgen: n.

Initialisierung: Zeit $O(1)$ für **PQ**, $O(n)$ für den Rest.

Es gibt maximal n Durchläufe durch die **while**-Schleife mit Organisationsaufwand jeweils $O(1)$, zusammen also Kosten $O(n)$ für die Schleifenorganisation.

In Schleifendurchlauf Nummer t, in dem u_t bearbeitet wird:
PQ.extractMin kostet Zeit $O(\log n)$.
Durchmustern der $\deg(u_t)$ Nachbarn von u_t:
Jedes **PQ.insert** oder **PQ.decreaseKey** kostet Zeit $O(\log n)$.

Insgesamt:

$$n \cdot O(\log n) + \sum_{1 \leq t \leq n} O(\deg(u_t) \cdot \log n)$$

Insgesamt:

$$n \cdot O(\log n) + \sum_{1 \le t \le n} O(\deg(u_t) \cdot \log n)$$

$$= \quad O\Big(n \log n + \log n \cdot \Big(\sum_{1 \le t \le n} \deg(u_t)\Big)\Big) = O(n \log n + m \log n),$$

Insgesamt:

$$n \cdot O(\log n) + \sum_{1 \le t \le n} O(\deg(u_t) \cdot \log n)$$

$$= \quad O\Big(n \log n + \log n \cdot \Big(\sum_{1 \le t \le n} \deg(u_t)\Big)\Big) = O(n \log n + m \log n),$$

wobei $n = |V|$ (Knotenzahl), $m = |E|$ (Kantenzahl).

Insgesamt:

$$n \cdot O(\log n) + \sum_{1 \le t \le n} O(\deg(u_t) \cdot \log n)$$

$$= \quad O\Big(n \log n + \log n \cdot \Big(\sum_{1 \le t \le n} \deg(u_t)\Big)\Big) = O(n \log n + m \log n),$$

wobei $n = |V|$ (Knotenzahl), $m = |E|$ (Kantenzahl).

Satz 11.1.3

Der **Algorithmus von Dijkstra** mit Verwendung einer Prioritätswarteschlange, die als Binärheap realisiert ist, ermittelt kürzeste Wege von Startknoten s aus in einem Digraphen $G = (V, E, c)$ in Zeit $O((n + m) \log n)$.

Mitteilung 11.1.4

Es gibt eine Implementierung der Datenstruktur **Prioritätswarteschlange** mit folgenden Rechenzeiten:

empty und *isempty* (und das Ermitteln des kleinsten Eintrags) kosten konstante Zeit;
extractMin kostet Zeit $O(\log n)$; n Einfügungen kosten zusammen Zeit $O(n \log n)$;
m *decreaseKey*-Operationen benötigen **zusammen** Zeit $O(m)$.

Mitteilung 11.1.4

Es gibt eine Implementierung der Datenstruktur **Prioritätswarteschlange** mit folgenden Rechenzeiten:

empty und *isempty* (und das Ermitteln des kleinsten Eintrags) kosten konstante Zeit;
extractMin kostet Zeit $O(\log n)$; n Einfügungen kosten zusammen Zeit $O(n \log n)$;
m *decreaseKey*-Operationen benötigen **zusammen** Zeit $O(m)$.

„Fibonacci-Heaps“

(Fortgeschritten, siehe Master-Vorlesung „Effiziente Algorithmen“ oder das Buch [Cormen et al., Introduction to Algorithms].)

Mitteilung 11.1.4

Es gibt eine Implementierung der Datenstruktur **Prioritätswarteschlange** mit folgenden Rechenzeiten:

empty und *isempty* (und das Ermitteln des kleinsten Eintrags) kosten konstante Zeit;
extractMin kostet Zeit $O(\log n)$; n Einfügungen kosten zusammen Zeit $O(n \log n)$;
m *decreaseKey*-Operationen benötigen **zusammen** Zeit $O(m)$.

„Fibonacci-Heaps“

(Fortgeschritten, siehe Master-Vorlesung „Effiziente Algorithmen“ oder das Buch [Cormen et al., Introduction to Algorithms].)

Resultierende **Rechenzeit für Algorithmus von Dijkstra:**

$$O(m + n \log n)$$

Mitteilung 11.1.4

Es gibt eine Implementierung der Datenstruktur **Prioritätswarteschlange** mit folgenden Rechenzeiten:

empty und *isempty* (und das Ermitteln des kleinsten Eintrags) kosten konstante Zeit;
extractMin kostet Zeit $O(\log n)$; n Einfügungen kosten zusammen Zeit $O(n \log n)$;
m *decreaseKey*-Operationen benötigen **zusammen** Zeit $O(m)$.

„Fibonacci-Heaps“

(Fortgeschritten, siehe Master-Vorlesung „Effiziente Algorithmen“ oder das Buch [Cormen et al., Introduction to Algorithms].)

Resultierende **Rechenzeit für Algorithmus von Dijkstra:**

$$O(m + n \log n)$$

Lineare Rechenzeit für Graphen mit $m = \Omega(n \log n)$ Kanten.

PAUSE

Als nächstes: Minimale Spannbäume

11.2 Minimale Spannbäume: Der Algorithmus von Jarník+Prim

Erinnerung (a) Ein (ungerichteter) Graph $G = (V, E)$ heißt **kreisfrei**, wenn er **keinen Kreis** besitzt.

11.2 Minimale Spannbäume: Der Algorithmus von Jarník+Prim

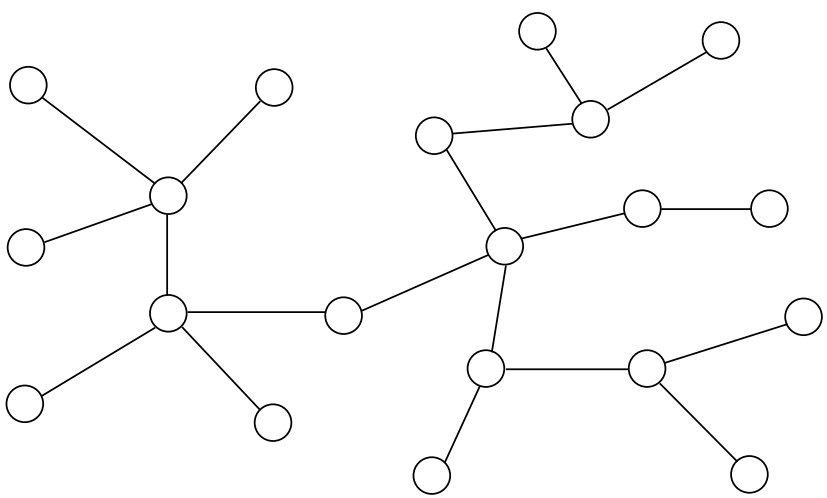
Erinnerung (a) Ein (ungerichteter) Graph $G = (V, E)$ heißt **kreisfrei**, wenn er **keinen Kreis** besitzt.

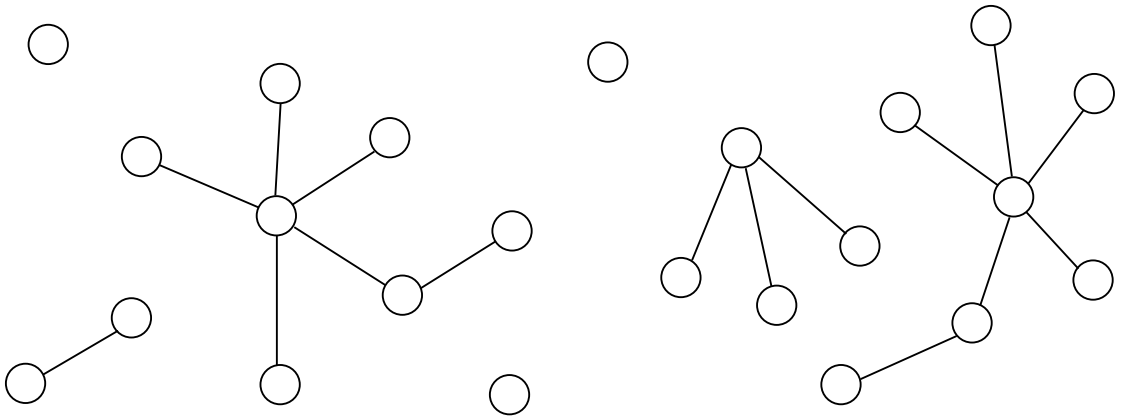
(b) Ein Graph G heißt ein **freier Baum** (oder nur **Baum**), wenn er zusammenhängend und kreisfrei ist.

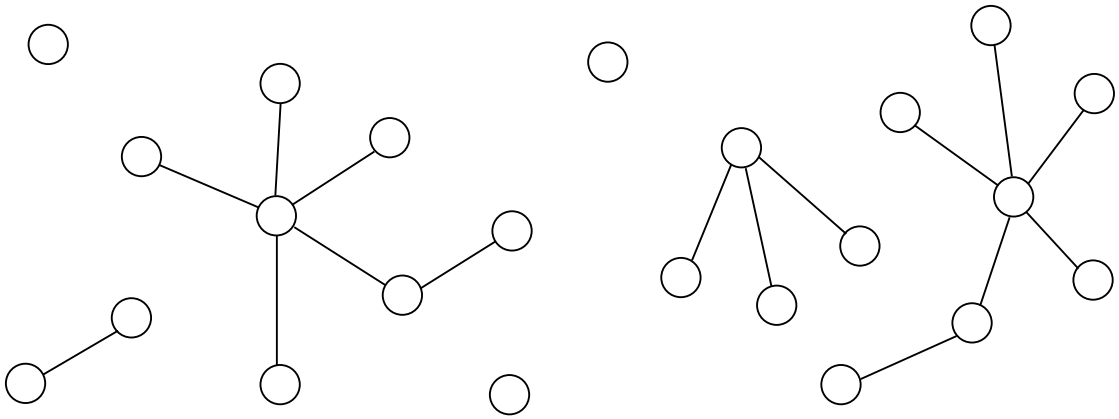
11.2 Minimale Spannbäume: Der Algorithmus von Jarník+Prim

Erinnerung (a) Ein (ungerichteter) Graph $G = (V, E)$ heißt **kreisfrei**, wenn er **keinen Kreis** besitzt.

(b) Ein Graph G heißt ein **freier Baum** (oder nur **Baum**), wenn er zusammenhängend und kreisfrei ist. – *Beispiel*:







Kreisfreie Graphen heißen auch **(freie) Wälder**.

Lemma 7.1.16 über Bäume

Wenn $G = (V, E)$ ein Graph mit n Knoten und m Kanten ist, dann sind folgende Aussagen äquivalent:

Lemma 7.1.16 über Bäume

Wenn $G = (V, E)$ ein Graph mit n Knoten und m Kanten ist, dann sind folgende Aussagen äquivalent:

(a) G ist ein Baum.

Lemma 7.1.16 über Bäume

Wenn $G = (V, E)$ ein Graph mit n Knoten und m Kanten ist, dann sind folgende Aussagen äquivalent:

(a) G ist ein Baum.

(b) G ist kreisfrei und $m \geq n - 1$.

Lemma 7.1.16 über Bäume

Wenn $G = (V, E)$ ein Graph mit n Knoten und m Kanten ist, dann sind folgende Aussagen äquivalent:

(a) G ist ein Baum.

(b) G ist kreisfrei und $m \geq n - 1$.

(c) G ist zusammenhängend und $m \leq n - 1$.

Lemma 7.1.16 über Bäume

Wenn $G = (V, E)$ ein Graph mit n Knoten und m Kanten ist, dann sind folgende Aussagen äquivalent:

(a) G ist ein Baum.

(b) G ist kreisfrei und $m \geq n - 1$.

(c) G ist zusammenhängend und $m \leq n - 1$.

(d) Zu jedem Paar u, v von Knoten gibt es genau einen einfachen Weg von u nach v.

Lemma 7.1.16 über Bäume

Wenn $G = (V, E)$ ein Graph mit n Knoten und m Kanten ist, dann sind folgende Aussagen äquivalent:

(a) G ist ein Baum.

(b) G ist kreisfrei und $m \geq n - 1$.

(c) G ist zusammenhängend und $m \leq n - 1$.

(d) Zu jedem Paar u, v von Knoten gibt es genau einen einfachen Weg von u nach v.

(e) G ist kreisfrei, aber das Hinzufügen einer beliebigen weiteren Kante erzeugt einen Kreis (G ist „maximal kreisfrei“).

Lemma 7.1.16 über Bäume

Wenn $G = (V, E)$ ein Graph mit n Knoten und m Kanten ist, dann sind folgende Aussagen äquivalent:

(a) G ist ein Baum.

(b) G ist kreisfrei und $m \geq n - 1$.

(c) G ist zusammenhängend und $m \leq n - 1$.

(d) Zu jedem Paar u, v von Knoten gibt es genau einen einfachen Weg von u nach v.

(e) G ist kreisfrei, aber das Hinzufügen einer beliebigen weiteren Kante erzeugt einen Kreis (G ist „maximal kreisfrei“).

(f) G ist zusammenhängend, aber das Entfernen einer beliebigen Kante erzeugt einen nicht zusammenhängenden Restgraphen (G ist „minimal zusammenhängend“).

Lemma 7.1.16 über Bäume

Wenn $G = (V, E)$ ein Graph mit n Knoten und m Kanten ist, dann sind folgende Aussagen äquivalent:

(a) G ist ein Baum.

(b) G ist kreisfrei und $m \geq n - 1$.

(c) G ist zusammenhängend und $m \leq n - 1$.

(d) Zu jedem Paar u, v von Knoten gibt es genau einen einfachen Weg von u nach v.

(e) G ist kreisfrei, aber das Hinzufügen einer beliebigen weiteren Kante erzeugt einen Kreis (G ist „maximal kreisfrei“).

(f) G ist zusammenhängend, aber das Entfernen einer beliebigen Kante erzeugt einen nicht zusammenhängenden Restgraphen (G ist „minimal zusammenhängend“).

Aus dem Fundamental-Lemma für Bäume folgt, für einen Baum G mit n Knoten:

Lemma 7.1.16 über Bäume

Wenn $G = (V, E)$ ein Graph mit n Knoten und m Kanten ist, dann sind folgende Aussagen äquivalent:

(a) G ist ein Baum.

(b) G ist kreisfrei und $m \geq n - 1$.

(c) G ist zusammenhängend und $m \leq n - 1$.

(d) Zu jedem Paar u, v von Knoten gibt es genau einen einfachen Weg von u nach v.

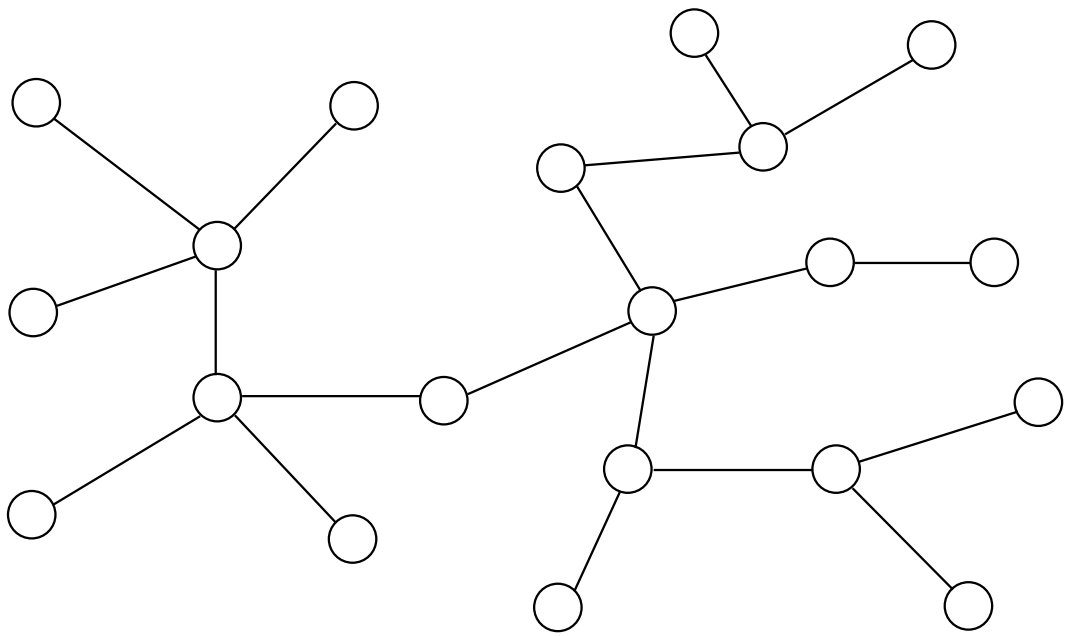
(e) G ist kreisfrei, aber das Hinzufügen einer beliebigen weiteren Kante erzeugt einen Kreis (G ist „maximal kreisfrei“).

(f) G ist zusammenhängend, aber das Entfernen einer beliebigen Kante erzeugt einen nicht zusammenhängenden Restgraphen (G ist „minimal zusammenhängend“).

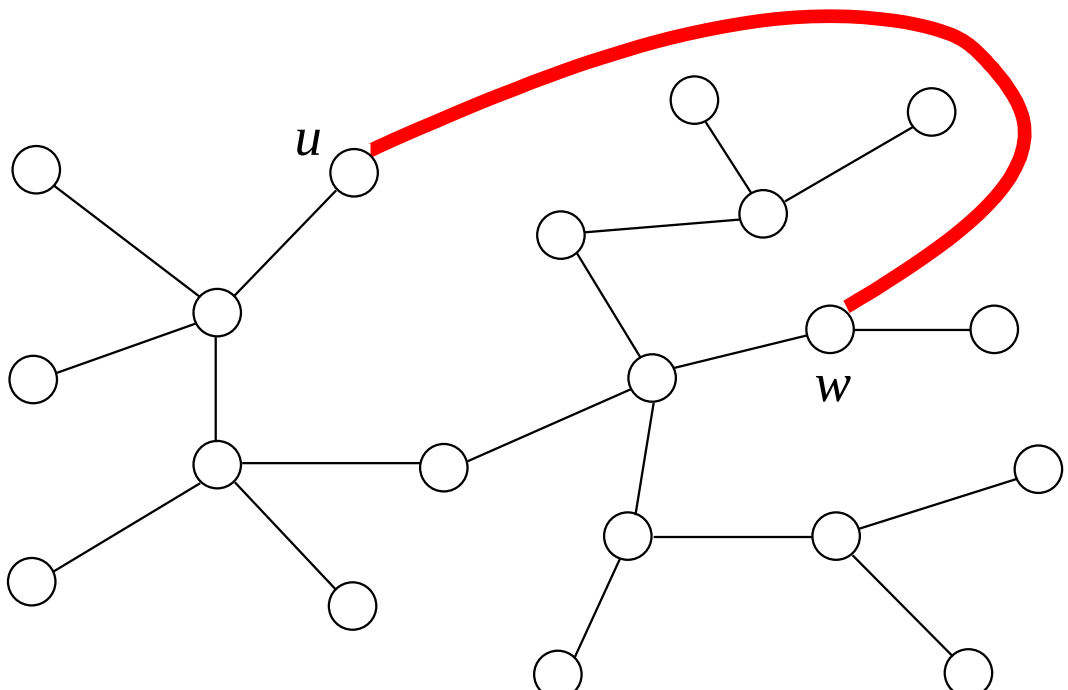
Aus dem Fundamental-Lemma für Bäume folgt, für einen Baum G mit n Knoten:

(1) G hat $n - 1$ Kanten.

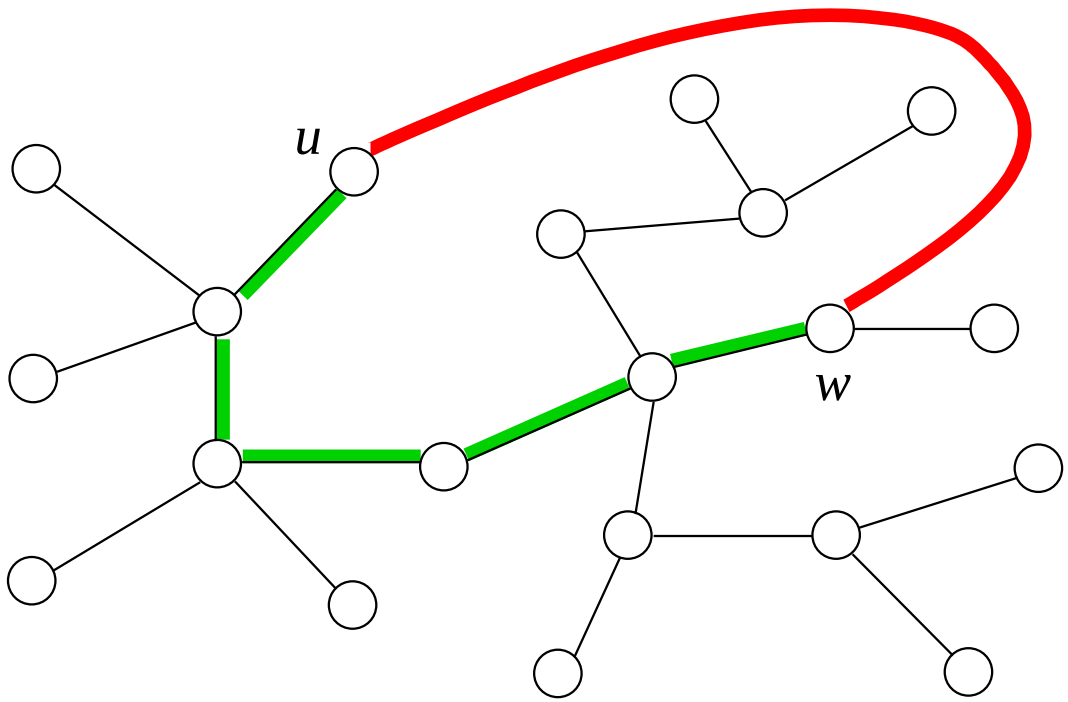
(2) Wenn man zu G eine Kante (u, w) hinzufügt, entsteht genau ein Kreis (aus (u, w) und dem eindeutigen **Weg** von u nach w in G).



(2) Wenn man zu G eine Kante (u, w) hinzufügt, entsteht genau ein Kreis (aus (u, w) und dem eindeutigen **Weg** von u nach w in G).



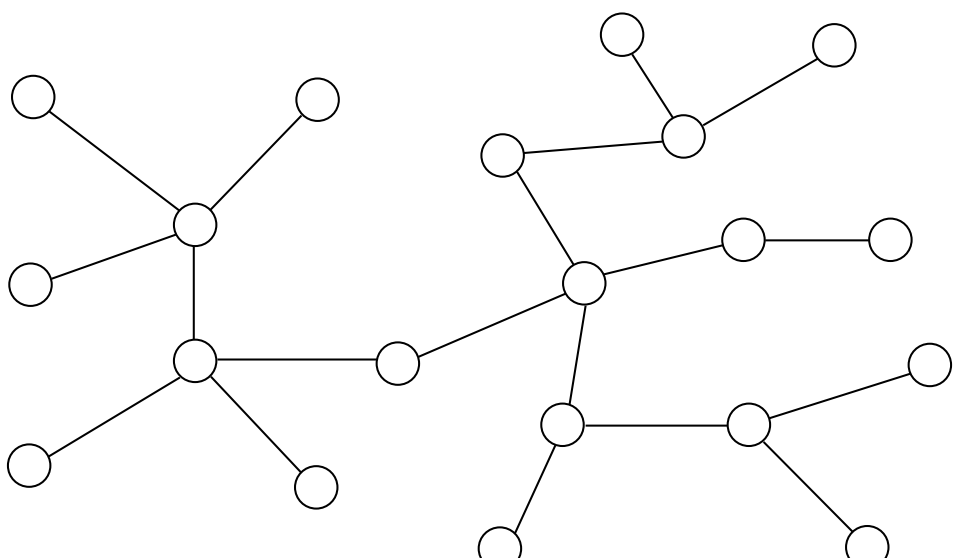
(2) Wenn man zu G eine Kante (u, w) hinzufügt, entsteht genau ein Kreis (aus (u, w) und dem eindeutigen **Weg** von u nach w in G).



(3) Wenn man aus G eine Kante (u, w) streicht, zerfällt der Graph in 2 Komponenten

$U = \{v \in V \mid \ v$ von u aus über Kanten aus $E - \{(u, w)\}$ erreichbar$\}$;

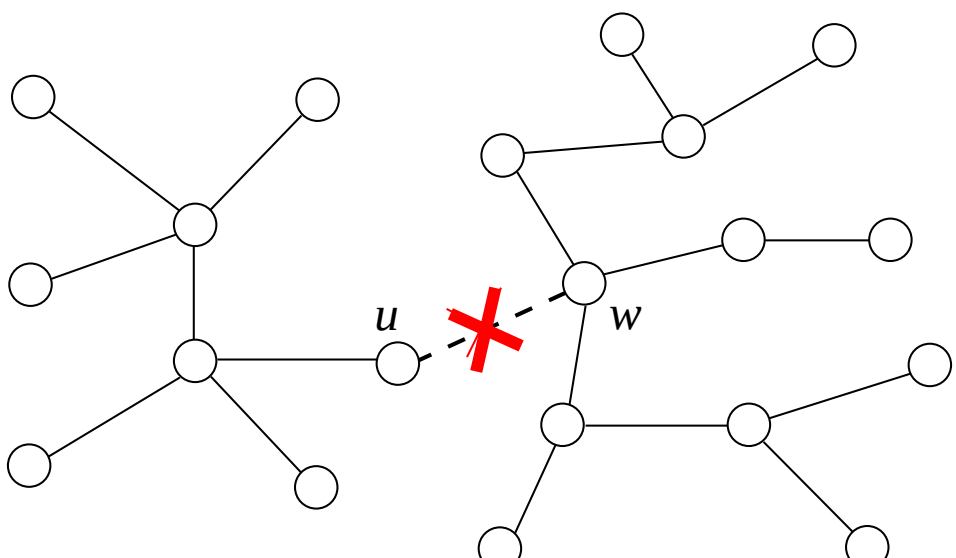
$W = \{v \in V \mid \ v$ von w aus über Kanten aus $E - \{(u, w)\}$ erreichbar$\}$.



(3) Wenn man aus G eine Kante (u, w) streicht, zerfällt der Graph in 2 Komponenten

$U = \{v \in V \mid \ v$ von u aus über Kanten aus $E - \{(u, w)\}$ erreichbar$\}$;

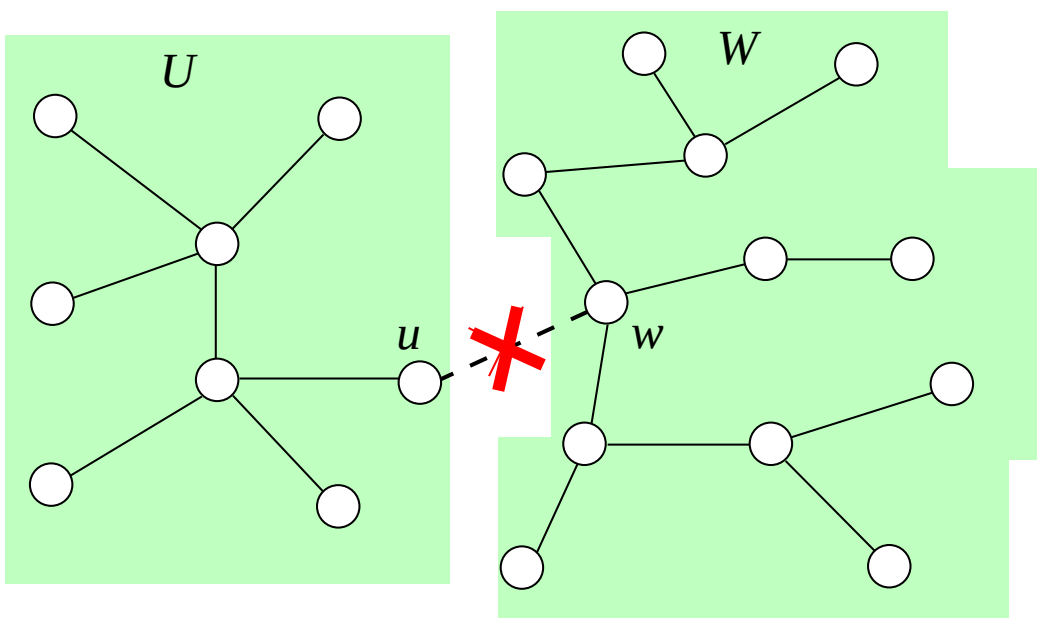
$W = \{v \in V \mid \ v$ von w aus über Kanten aus $E - \{(u, w)\}$ erreichbar$\}$.



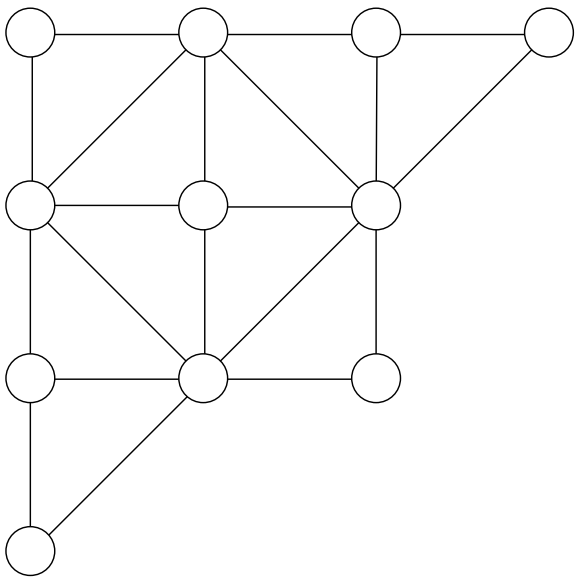
(3) Wenn man aus G eine Kante (u, w) streicht, zerfällt der Graph in 2 Komponenten

$U = \{v \in V \mid \ v$ von u aus über Kanten aus $E - \{(u, w)\}$ erreichbar$\}$;

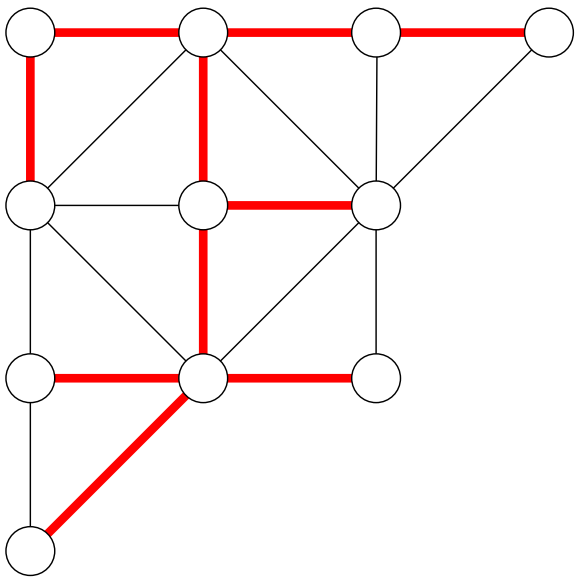
$W = \{v \in V \mid \ v$ von w aus über Kanten aus $E - \{(u, w)\}$ erreichbar$\}$.



Beispiel:



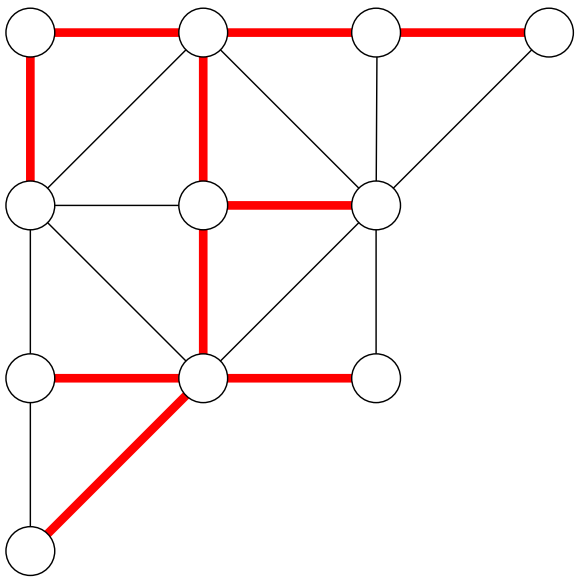
Beispiel:



Definition 11.2.1

Es sei $G = (V, E)$ ein zusammenhängender Graph. Eine Menge $T \subseteq E$ von Kanten

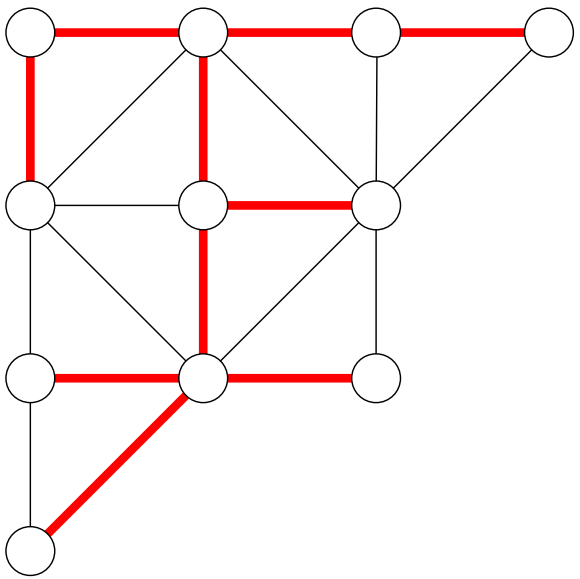
Beispiel:



Definition 11.2.1

Es sei $G = (V, E)$ ein zusammenhängender Graph. Eine Menge $T \subseteq E$ von Kanten heißt ein **Spannbaum** für G, wenn (V, T) ein Baum ist.

Beispiel:

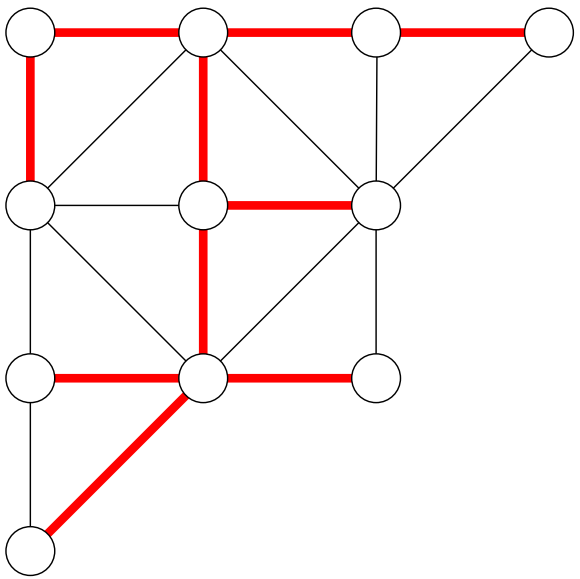


Definition 11.2.1

Es sei $G = (V, E)$ ein zusammenhängender Graph. Eine Menge $T \subseteq E$ von Kanten heißt ein **Spannbaum** für G, wenn (V, T) ein Baum ist.

Klar: Jeder zusammenhängende Graph hat einen Spannbaum.

Beispiel:



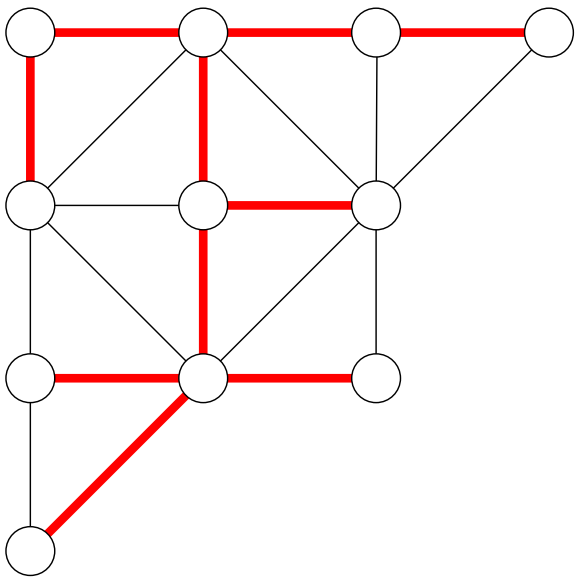
Definition 11.2.1

Es sei $G = (V, E)$ ein zusammenhängender Graph. Eine Menge $T \subseteq E$ von Kanten heißt ein **Spannbaum** für G, wenn (V, T) ein Baum ist.

Klar: Jeder zusammenhängende Graph hat einen Spannbaum.

(Iterativer Prozess: Starte mit E.

Beispiel:



Definition 11.2.1

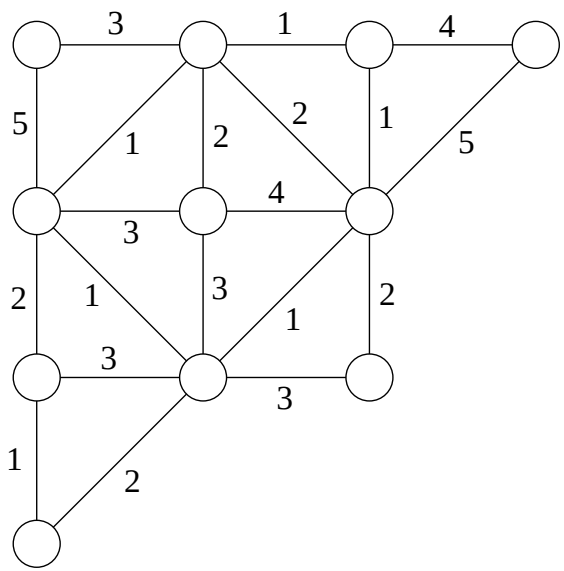
Es sei $G = (V, E)$ ein zusammenhängender Graph. Eine Menge $T \subseteq E$ von Kanten heißt ein **Spannbaum** für G, wenn (V, T) ein Baum ist.

Klar: Jeder zusammenhängende Graph hat einen Spannbaum.

(Iterativer Prozess: Starte mit E. Solange es Kreise gibt, entferne eine beliebige Kante auf einem Kreis. Der Zusammenhang bleibt stets erhalten; der verbleibende Graph muss kreisfrei sein.)

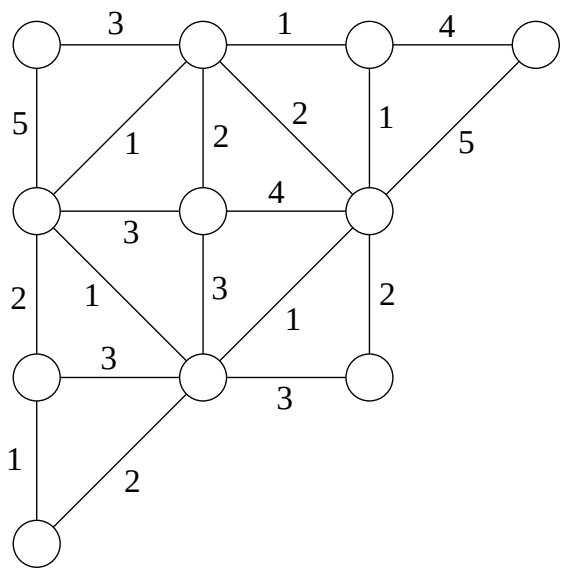
Definition 11.2.2

Es sei $G = (V, E, c)$ ein **gewichteter Graph**, d.h. $c\colon E \to \mathbb{R}$ ist eine „*Gewichtsfunktion*“ oder „*Kostenfunktion*“.



Definition 11.2.2

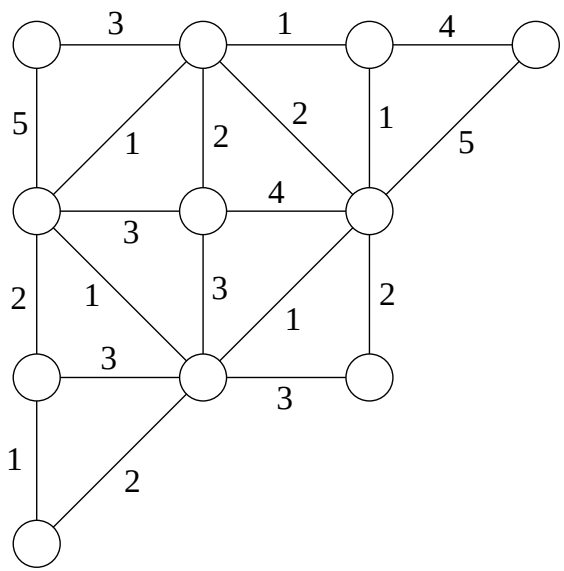
Es sei $G = (V, E, c)$ ein **gewichteter Graph**, d.h. $c\colon E \to \mathbb{R}$ ist eine „*Gewichtsfunktion*“ oder „*Kostenfunktion*“.



Graph modelliert: Straßennetz – Computernetzwerk – Leitungsnetz – . . . **Kantenkosten** modellieren **Herstellungskosten** (Straße, Unterwasserkabel, Pipeline, . . .)

Definition 11.2.2

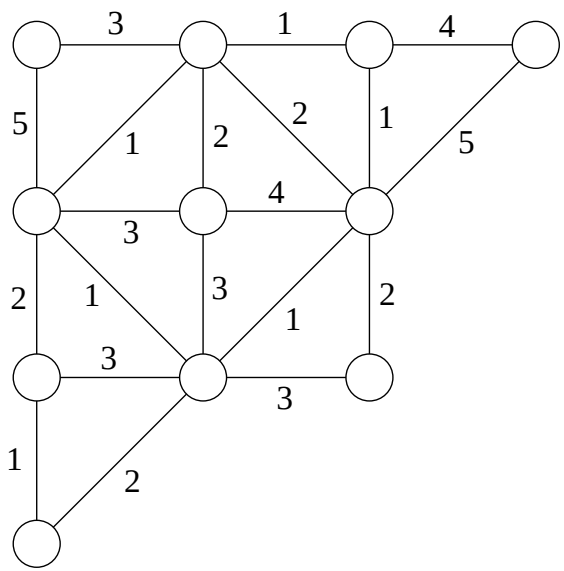
Es sei $G = (V, E, c)$ ein **gewichteter Graph**, d.h. $c\colon E \to \mathbb{R}$ ist eine „*Gewichtsfunktion*“ oder „*Kostenfunktion*“.



Graph modelliert: Straßennetz – Computernetzwerk – Leitungsnetz – . . . **Kantenkosten** modellieren **Herstellungskosten** (Straße, Unterwasserkabel, Pipeline, . . .) oder **Leitungsmiete** oder . . .

Definition 11.2.2

Es sei $G = (V, E, c)$ ein **gewichteter Graph**, d.h. $c\colon E \to \mathbb{R}$ ist eine „*Gewichtsfunktion*“ oder „*Kostenfunktion*“.



Graph modelliert: Straßennetz – Computernetzwerk – Leitungsnetz – . . . **Kantenkosten** modellieren **Herstellungskosten** (Straße, Unterwasserkabel, Pipeline, . . .) oder **Leitungsmiete** oder . . .

Ziel: Finde **Spannbaum** (zus.-hängend, kreisfrei) von G mit möglichst geringen Kosten.

(a) Jeder Kantenmenge $E' \subseteq E$ wird durch

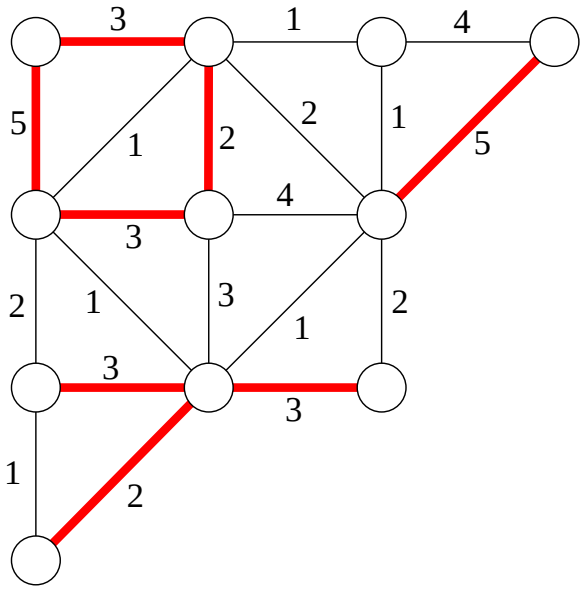
$$c(E') := \sum_{e \in E'} c(e)$$

ein **Gesamtgewicht** zugeordnet.

(a) Jeder Kantenmenge $E' \subseteq E$ wird durch

$$c(E') := \sum_{e \in E'} c(e)$$

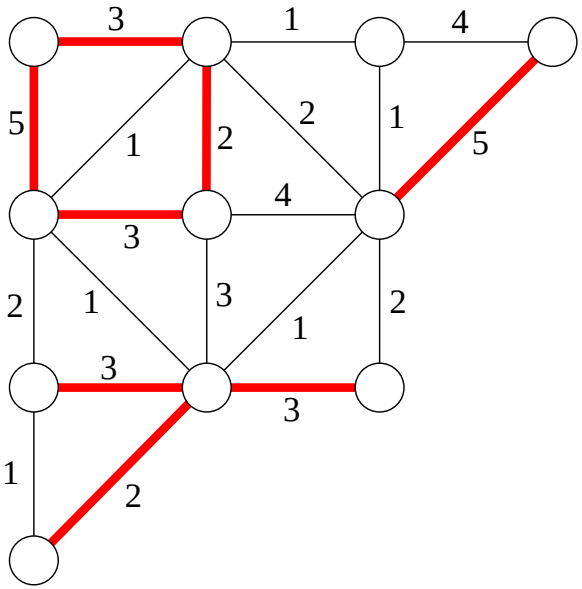
ein **Gesamtgewicht** zugeordnet.



(a) Jeder Kantenmenge $E' \subseteq E$ wird durch

$$c(E') := \sum_{e \in E'} c(e)$$

ein **Gesamtgewicht** zugeordnet.



Gesamtgewicht

$c(\boldsymbol{E'}) = 3+5+2+5+3+3+3+2 = 26.$

(b) Sei G zusammenhängend. Ein Spannbaum $T \subseteq E$ für G heißt ein **minimaler Spannbaum**,

(b) Sei G zusammenhängend. Ein Spannbaum $T \subseteq E$ für G heißt ein **minimaler Spannbaum**, wenn er minimale Kosten unter allen Spannbäumen hat, d. h. wenn

$$c(T) = \min\{c(T') \mid T' \text{ Spannbaum für } G\}.$$

(b) Sei G zusammenhängend. Ein Spannbaum $T \subseteq E$ für G heißt ein **minimaler Spannbaum**, wenn er minimale Kosten unter allen Spannbäumen hat, d. h. wenn

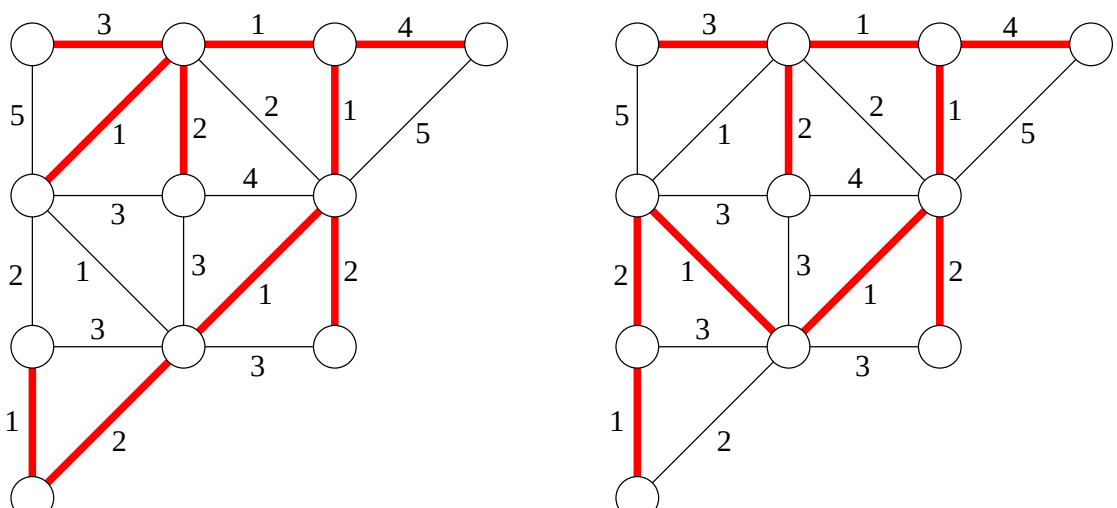
$$c(T) = \min\{c(T') \mid T' \text{ Spannbaum für } G\}.$$

Abkürzung: **MST** („**M**inimum **S**panning **T**ree“).

(b) Sei G zusammenhängend. Ein Spannbaum $T \subseteq E$ für G heißt ein **minimaler Spannbaum**, wenn er minimale Kosten unter allen Spannbäumen hat, d. h. wenn

$$c(T) = \min\{c(T') \mid T' \text{ Spannbaum für } G\}.$$

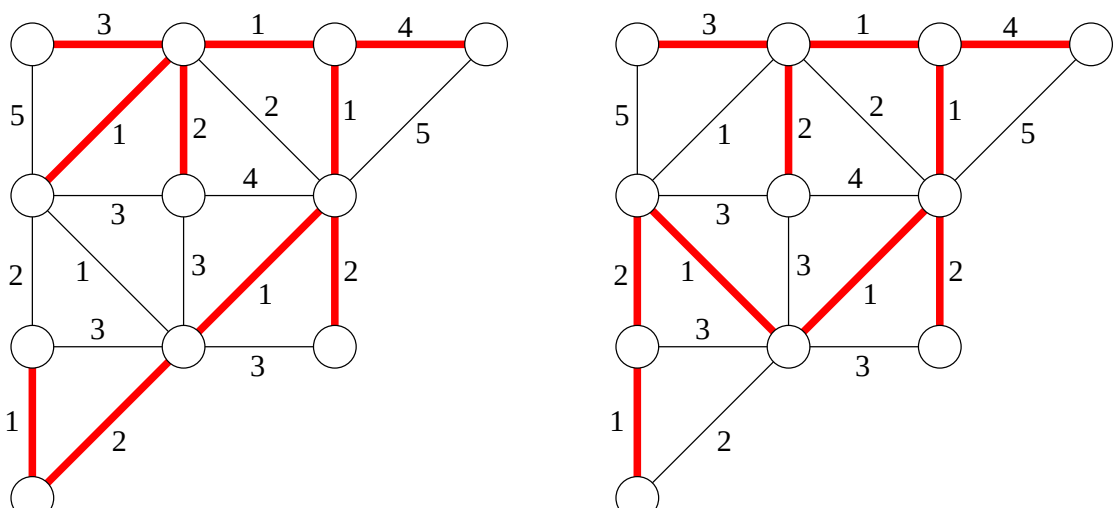
Abkürzung: **MST** („**M**inimum **S**panning **T**ree“).



(b) Sei G zusammenhängend. Ein Spannbaum $T \subseteq E$ für G heißt ein **minimaler Spannbaum**, wenn er minimale Kosten unter allen Spannbäumen hat, d. h. wenn

$$c(T) = \min\{c(T') \mid T' \text{ Spannbaum für } G\}.$$

Abkürzung: **MST** („**M**inimum **S**panning **T**ree“).



Zwei minimale Spannbäume, jeweils mit Gesamtgewicht 18.

Klar: Jeder Graph besitzt einen MST. (Es gibt nur endlich viele Spannbäume.)

Klar: Jeder Graph besitzt einen MST. (Es gibt nur endlich viele Spannbäume.)

Achtung: Es kann mehrere verschiedene MSTs geben (die alle dasselbe Gesamtgewicht haben).

Klar: Jeder Graph besitzt einen MST. (Es gibt nur endlich viele Spannbäume.)

Achtung: Es kann mehrere verschiedene MSTs geben (die alle dasselbe Gesamtgewicht haben).

Aufgabe: Zu gegebenem $G = (V, E, c)$ finde einen MST T.

Klar: Jeder Graph besitzt einen MST. (Es gibt nur endlich viele Spannbäume.)

Achtung: Es kann mehrere verschiedene MSTs geben (die alle dasselbe Gesamtgewicht haben).

Aufgabe: Zu gegebenem $G = (V, E, c)$ finde einen MST T.

Hier: **„Algorithmus von Jarník/Prim“**

Klar: Jeder Graph besitzt einen MST. (Es gibt nur endlich viele Spannbäume.)

Achtung: Es kann mehrere verschiedene MSTs geben (die alle dasselbe Gesamtgewicht haben).

Aufgabe: Zu gegebenem $G = (V, E, c)$ finde einen MST T.

Hier: **„Algorithmus von Jarník/Prim“** und **„Algorithmus von Kruskal“**

Klar: Jeder Graph besitzt einen MST. (Es gibt nur endlich viele Spannbäume.)

Achtung: Es kann mehrere verschiedene MSTs geben (die alle dasselbe Gesamtgewicht haben).

Aufgabe: Zu gegebenem $G = (V, E, c)$ finde einen MST T.

Hier: **„Algorithmus von Jarník/Prim“** und **„Algorithmus von Kruskal“**

Algorithmenparadigma: **„greedy“**.

Klar: Jeder Graph besitzt einen MST. (Es gibt nur endlich viele Spannbäume.)

Achtung: Es kann mehrere verschiedene MSTs geben (die alle dasselbe Gesamtgewicht haben).

Aufgabe: Zu gegebenem $G = (V, E, c)$ finde einen MST T.

Hier: **„Algorithmus von Jarník/Prim“** und **„Algorithmus von Kruskal“**

Algorithmenparadigma: **„greedy“**.

Baue Lösung **Schritt für Schritt** auf.

Klar: Jeder Graph besitzt einen MST. (Es gibt nur endlich viele Spannbäume.)

Achtung: Es kann mehrere verschiedene MSTs geben (die alle dasselbe Gesamtgewicht haben).

Aufgabe: Zu gegebenem $G = (V, E, c)$ finde einen MST T.

Hier: **„Algorithmus von Jarník/Prim“** und **„Algorithmus von Kruskal“**

Algorithmenparadigma: **„greedy“**.

Baue Lösung **Schritt für Schritt** auf.

(Hier: Wähle eine Kante für T nach der anderen.)

Klar: Jeder Graph besitzt einen MST. (Es gibt nur endlich viele Spannbäume.)

Achtung: Es kann mehrere verschiedene MSTs geben (die alle dasselbe Gesamtgewicht haben).

Aufgabe: Zu gegebenem $G = (V, E, c)$ finde einen MST T.

Hier: **„Algorithmus von Jarník/Prim“** und **„Algorithmus von Kruskal“**

Algorithmenparadigma: **„greedy“**.

Baue Lösung **Schritt für Schritt** auf.

(Hier: Wähle eine Kante für T nach der anderen.)

Treffe in jedem Schritt die **(lokal) günstigste Entscheidung**.

Klar: Jeder Graph besitzt einen MST. (Es gibt nur endlich viele Spannbäume.)

Achtung: Es kann mehrere verschiedene MSTs geben (die alle dasselbe Gesamtgewicht haben).

Aufgabe: Zu gegebenem $G = (V, E, c)$ finde einen MST T.

Hier: **„Algorithmus von Jarník/Prim“** und **„Algorithmus von Kruskal“**

Algorithmenparadigma: **„greedy“**.

Baue Lösung **Schritt für Schritt** auf.

(Hier: Wähle eine Kante für T nach der anderen.)

Treffe in jedem Schritt die **(lokal) günstigste Entscheidung**.

Algorithmus von Jarník/Prim ist sehr ähnlich zum Algorithmus von Dijkstra.

Algorithmus von Jarník/Prim:

Algorithmus von Jarník/Prim: (Programmierdetails später)

Algorithmus von Jarník/Prim: (Programmierdetails später)

S: Menge von Knoten. Enthält die „bisher erreichten“ Knoten.

Algorithmus von Jarník/Prim: (Programmierdetails später)

S: Menge von Knoten. Enthält die „bisher erreichten“ Knoten.
R: Menge von Kanten. Enthält die „bisher gewählten“ Kanten.

Algorithmus von Jarník/Prim: (Programmierdetails später)

S: Menge von Knoten. Enthält die „bisher erreichten“ Knoten.
R: Menge von Kanten. Enthält die „bisher gewählten“ Kanten.

(1) Wähle einen beliebigen (Start-)Knoten $s \in V$.

Algorithmus von Jarník/Prim: (Programmierdetails später)

S: Menge von Knoten. Enthält die „bisher erreichten“ Knoten.
R: Menge von Kanten. Enthält die „bisher gewählten“ Kanten.

(1) Wähle einen beliebigen (Start-)Knoten $s \in V$.
$\mathtt{S} \leftarrow \{s\}$;

Algorithmus von Jarník/Prim: (Programmierdetails später)

S: Menge von Knoten. Enthält die „bisher erreichten“ Knoten.
R: Menge von Kanten. Enthält die „bisher gewählten“ Kanten.

(1) Wähle einen beliebigen (Start-)Knoten $s \in V$.
$\mathtt{S} \leftarrow \{s\}$; $\mathtt{R} \leftarrow \emptyset$;

Algorithmus von Jarník/Prim: (Programmierdetails später)

S: Menge von Knoten. Enthält die „bisher erreichten“ Knoten.
R: Menge von Kanten. Enthält die „bisher gewählten“ Kanten.

(1) Wähle einen beliebigen (Start-)Knoten $s \in V$.
$\mathtt{S} \leftarrow \{s\}$; $\mathtt{R} \leftarrow \emptyset$;

(2) Wiederhole $(n-1)$-mal:

Wähle eine billigste Kante (v, u), die ein $v \in \mathtt{S}$ mit einem $u \in V - \mathtt{S}$ verbindet,

d. h. finde $v \in \mathtt{S}$ und $u \in V - \mathtt{S}$, so dass
$c(v, u)$ **minimal** unter allen Werten $c(v', u')$, $v' \in \mathtt{S}$, $u' \in V - \mathtt{S}$, ist.

Algorithmus von Jarník/Prim: (Programmierdetails später)

S: Menge von Knoten. Enthält die „bisher erreichten“ Knoten.
R: Menge von Kanten. Enthält die „bisher gewählten“ Kanten.

(1) Wähle einen beliebigen (Start-)Knoten $s \in V$.
$\mathtt{S} \leftarrow \{s\}$; $\mathtt{R} \leftarrow \emptyset$;

(2) Wiederhole $(n-1)$-mal:

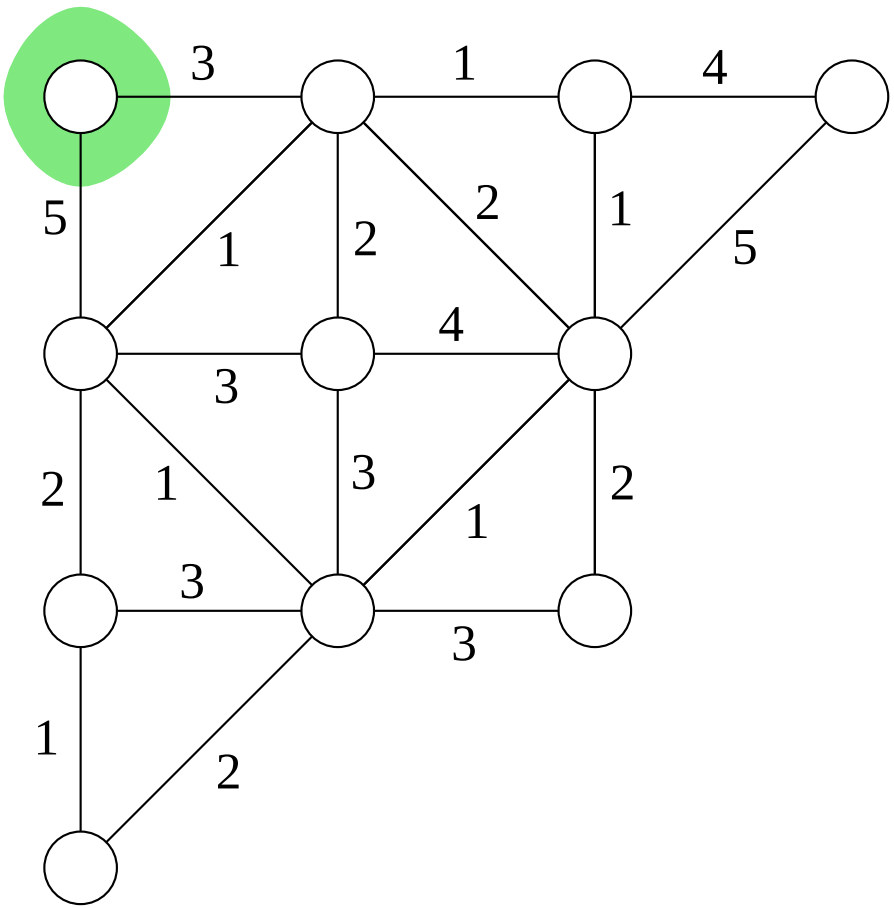
Wähle eine billigste Kante (v,u), die ein $v \in \mathtt{S}$ mit einem $u \in V - \mathtt{S}$ verbindet,

d. h. finde $v \in \mathtt{S}$ und $u \in V - \mathtt{S}$, so dass
$c(v,u)$ **minimal** unter allen Werten $c(v',u')$, $v' \in \mathtt{S}$, $u' \in V - \mathtt{S}$, ist.

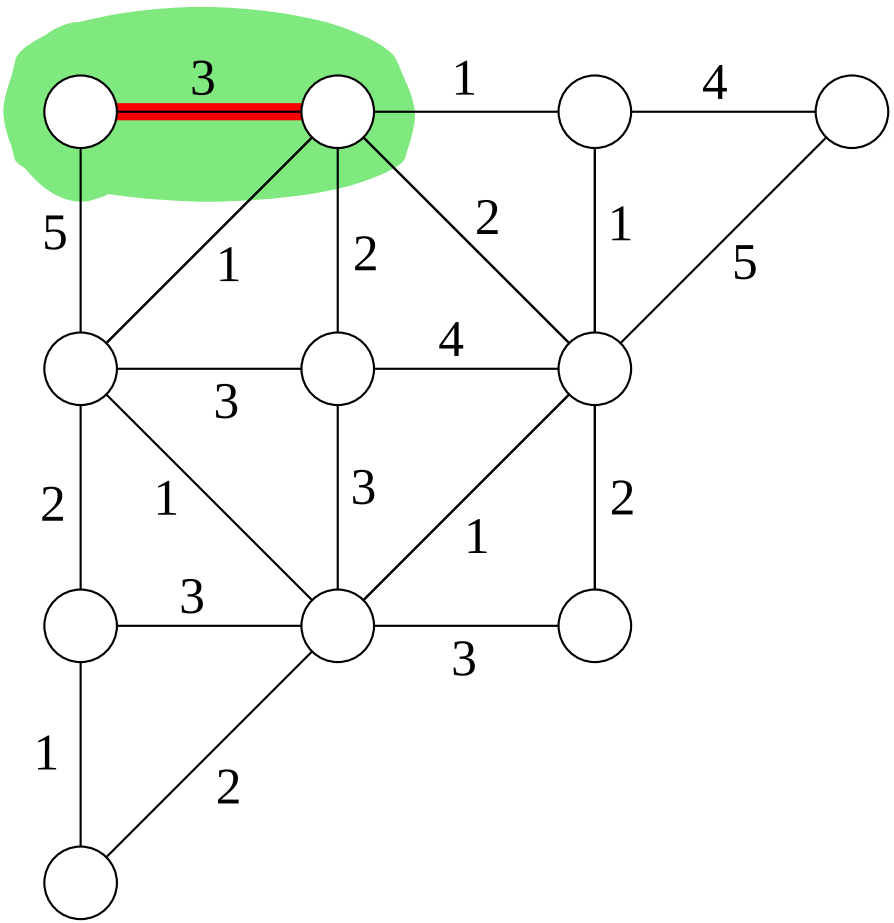
$\mathtt{S} \leftarrow \mathtt{S} \cup \{u\}$;
$\mathtt{R} \leftarrow \mathtt{R} \cup \{(v,u)\}$;

(3) Ausgabe: R.

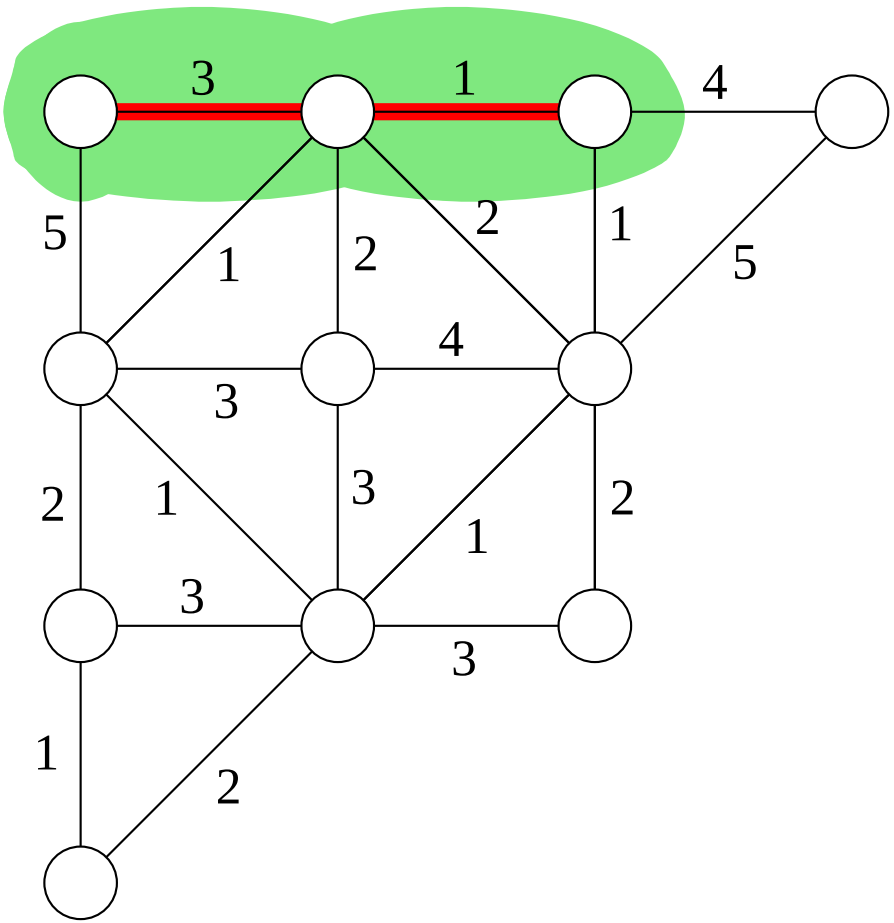
Beispiel (Jarník/Prim):



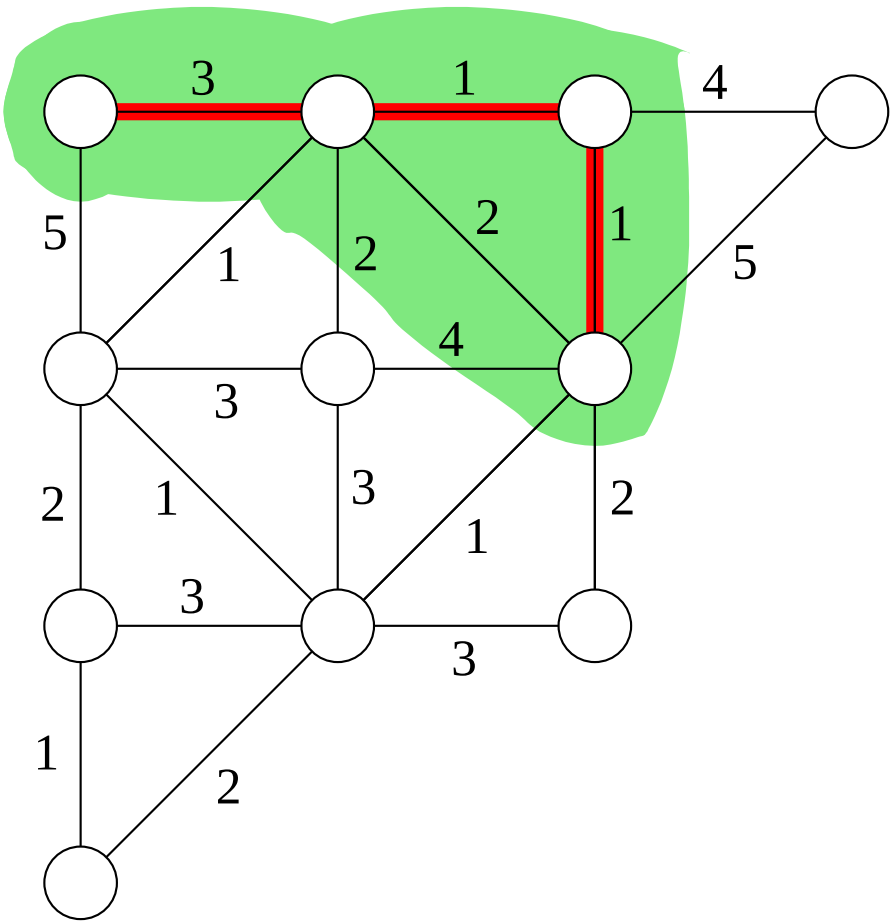
Beispiel (Jarník/Prim):



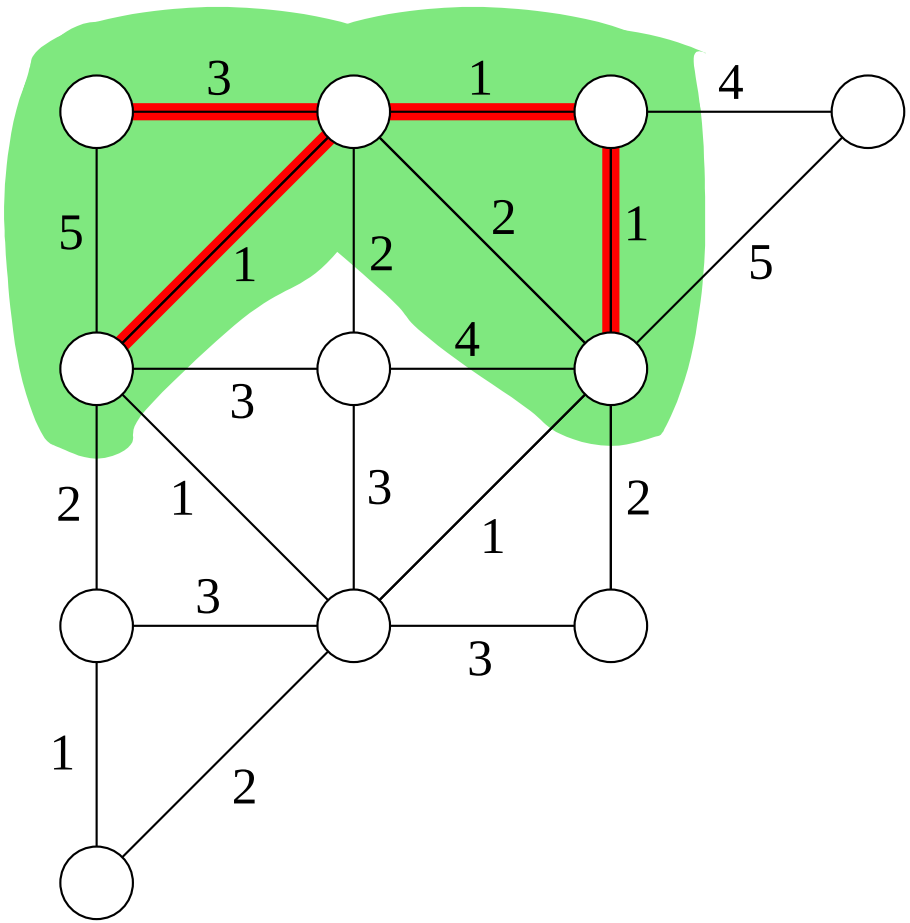
Beispiel (Jarník/Prim):



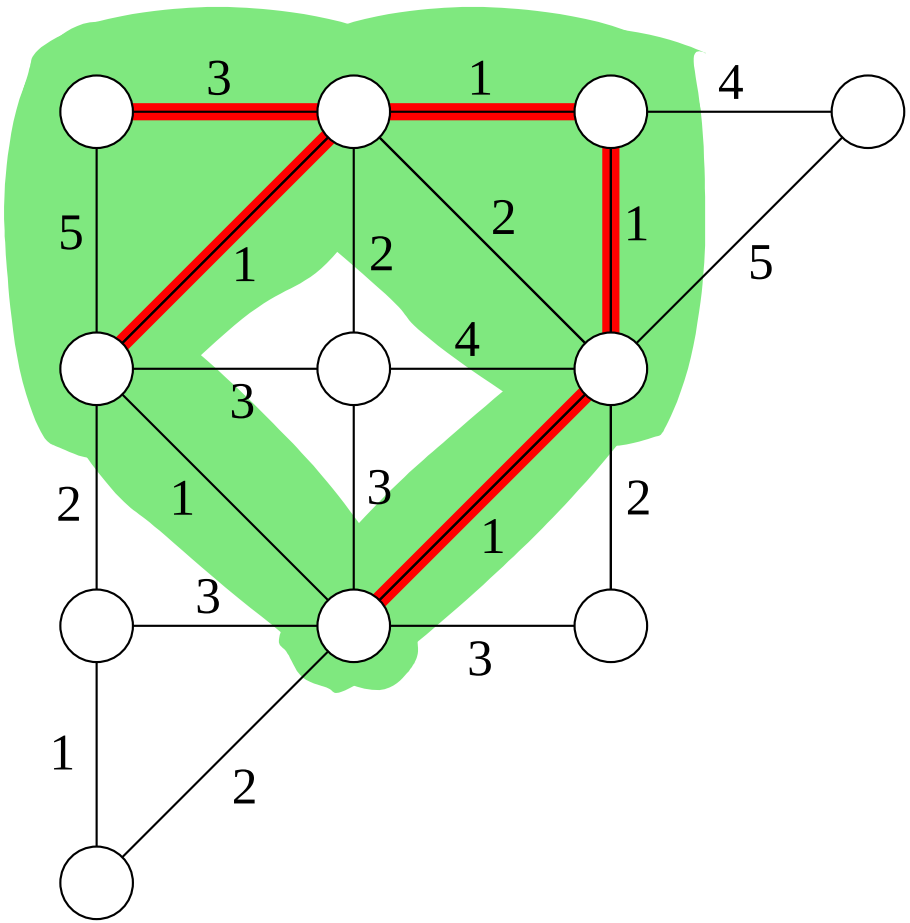
Beispiel (Jarník/Prim):



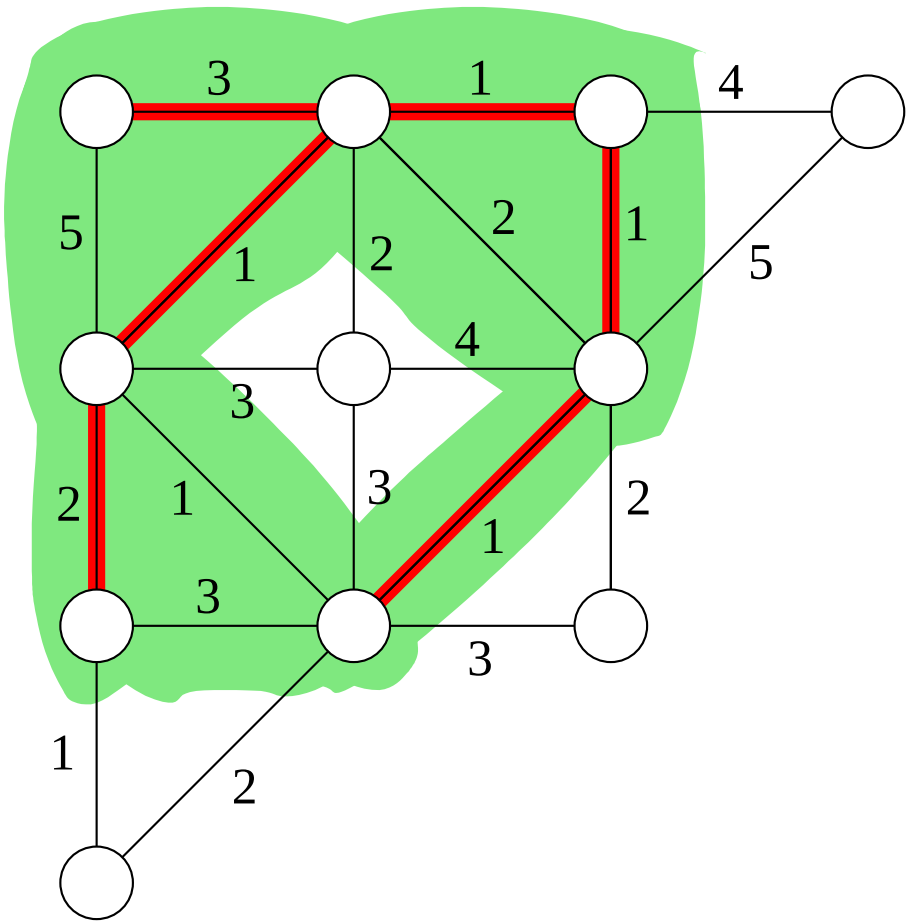
Beispiel (Jarník/Prim):



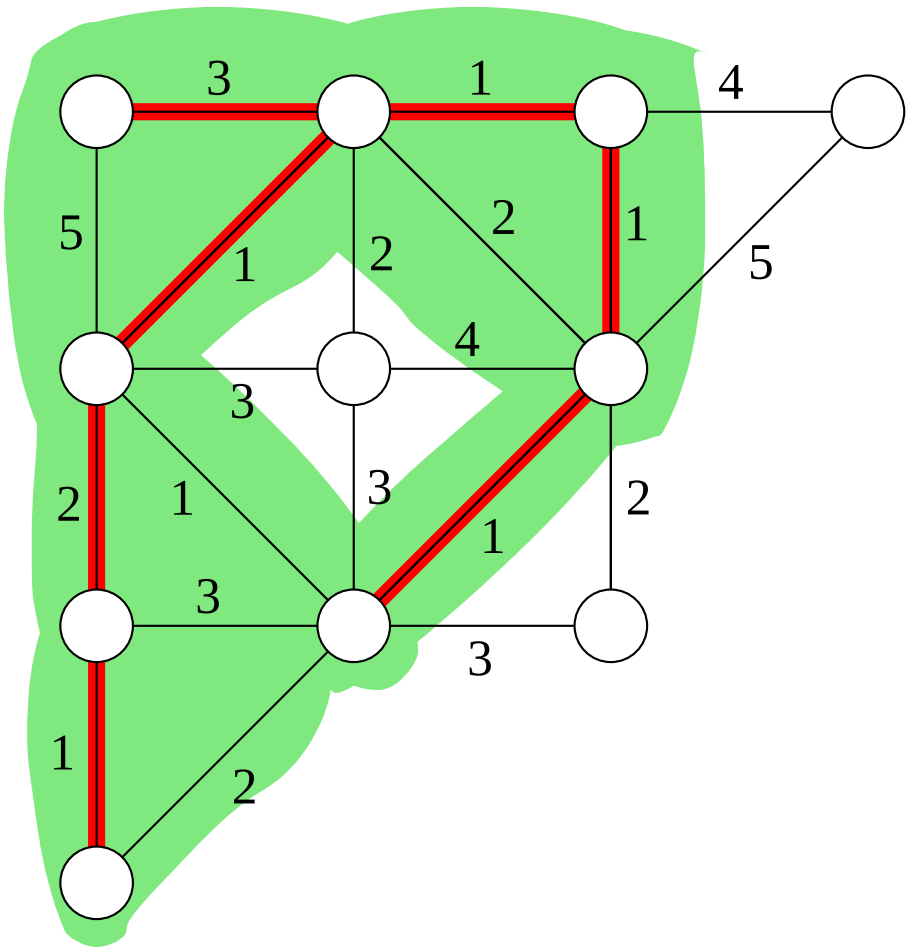
Beispiel (Jarník/Prim):



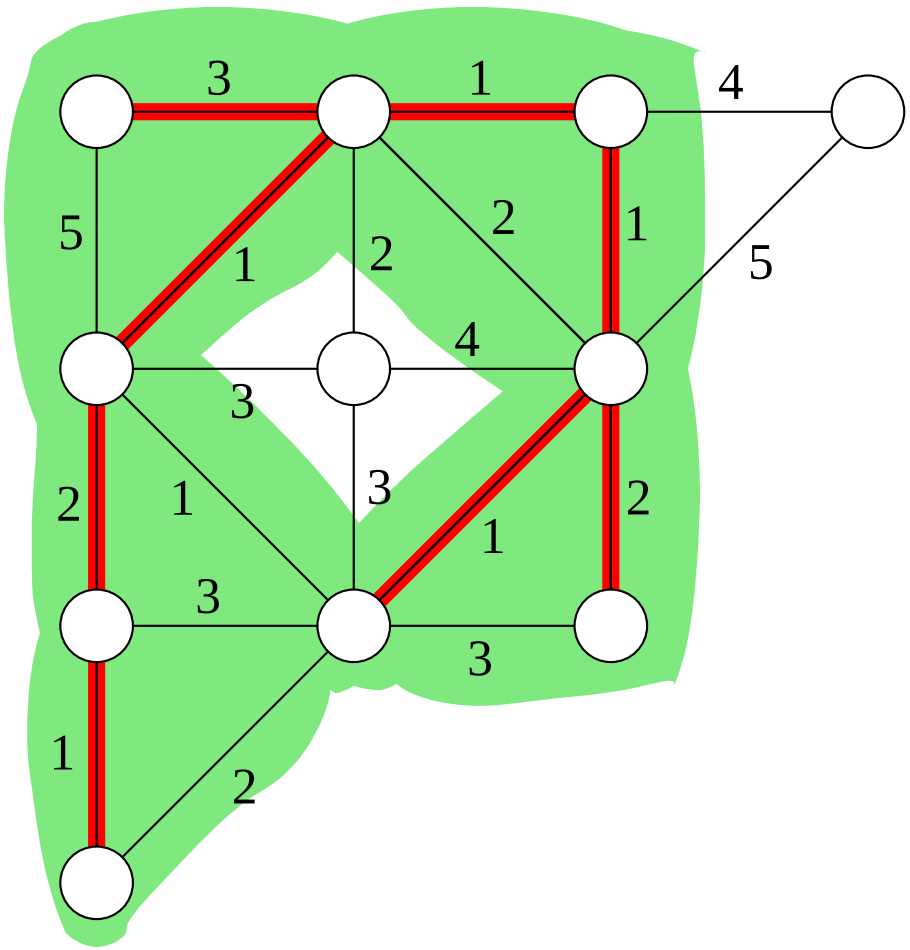
Beispiel (Jarník/Prim):



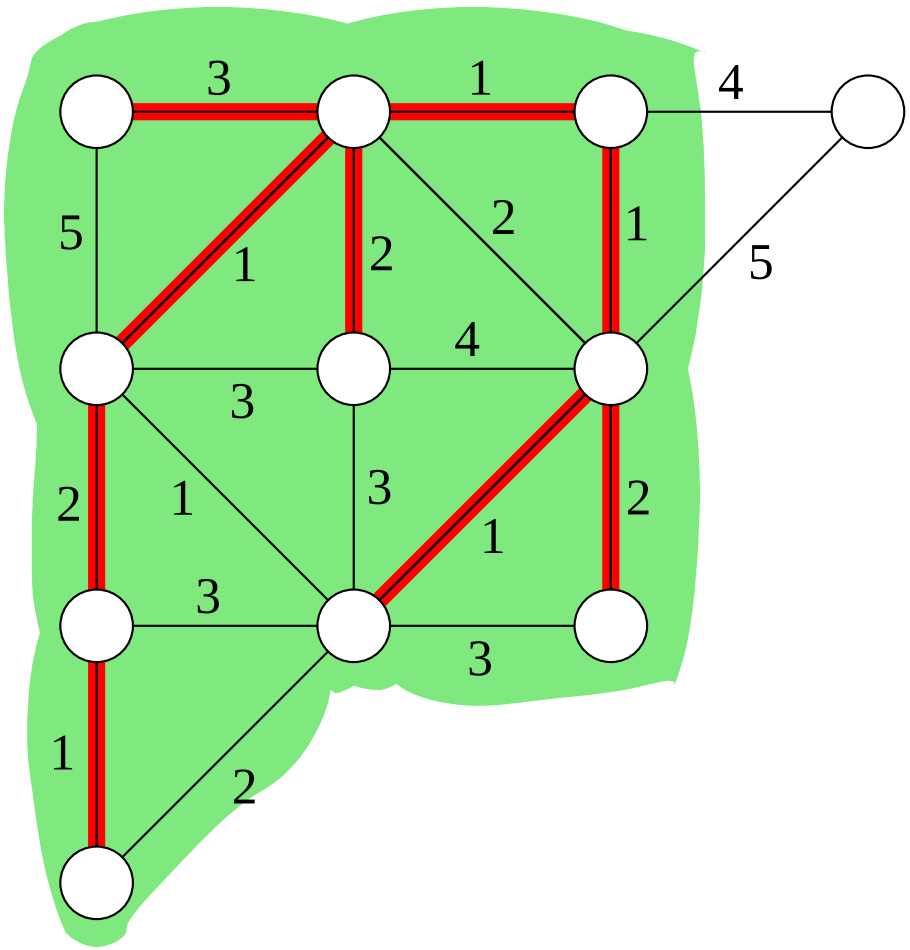
Beispiel (Jarník/Prim):



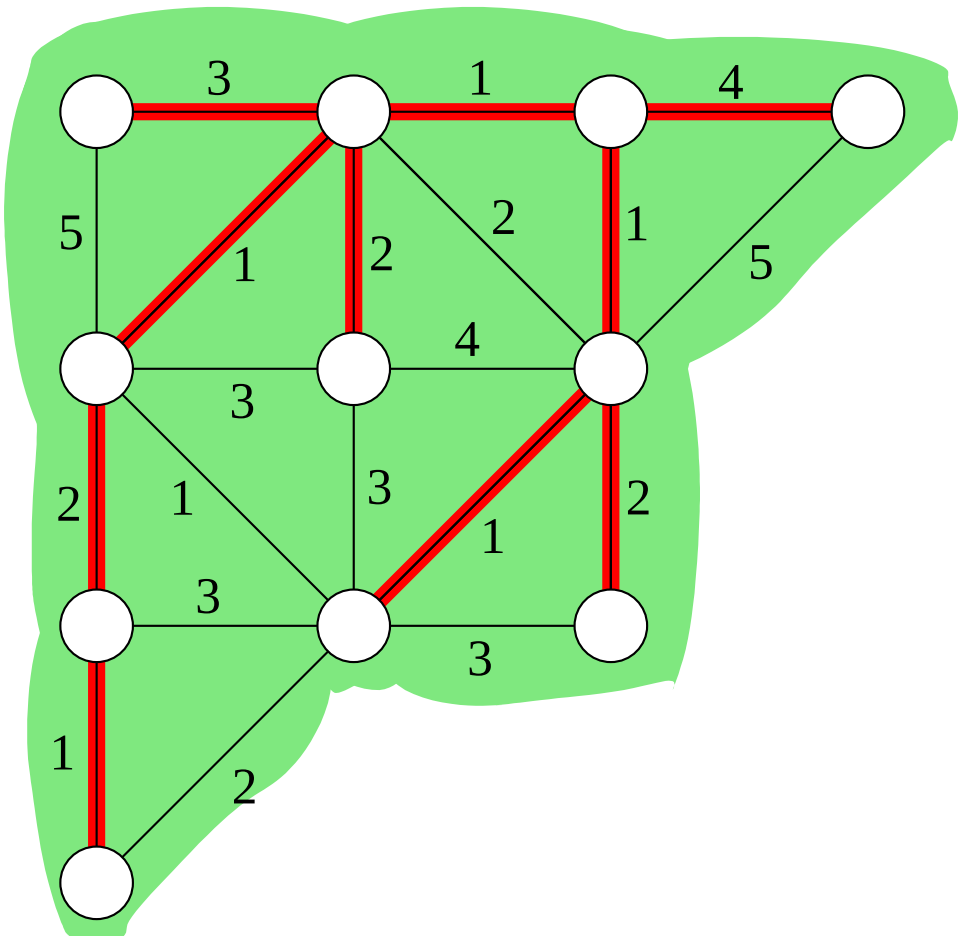
Beispiel (Jarník/Prim):



Beispiel (Jarník/Prim):



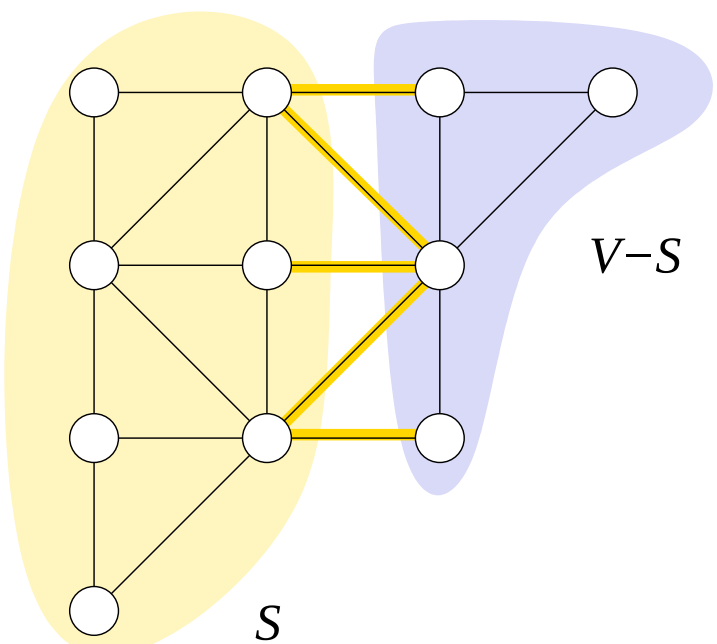
Beispiel (Jarník/Prim):



Die Schnitteigenschaft

Für den Korrektheitsbeweis des Algorithmus von Jarník/Prim:
„Cut property“ – Schnitteigenschaft

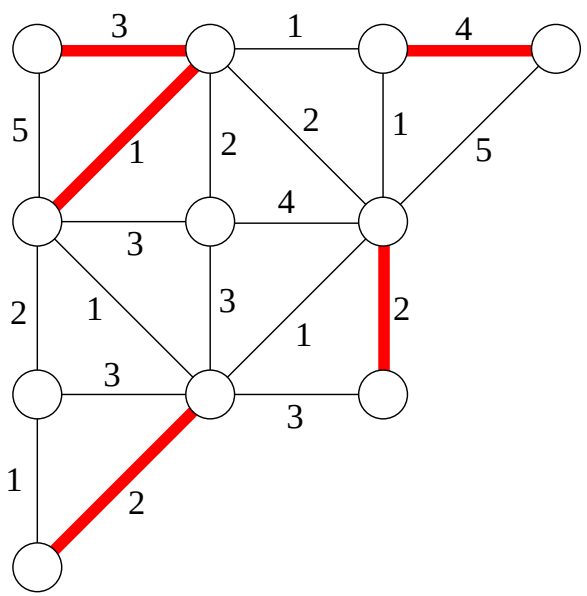
Eine Partition $(S, V - S)$ von V mit $\emptyset \neq S \neq V$ heißt ein **Schnitt**.



Die Schnitteigenschaft

Definition 11.2.3

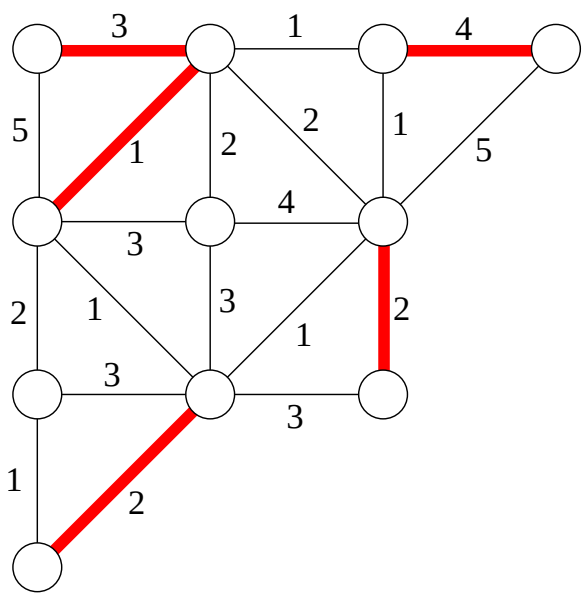
Eine Menge $R \subseteq E$ heißt **erweiterbar** (zu einem MST), wenn es einen MST T mit $R \subseteq T$ gibt.



Die Schnitteigenschaft

Definition 11.2.3

Eine Menge $R \subseteq E$ heißt **erweiterbar** (zu einem MST), wenn es einen MST T mit $R \subseteq T$ gibt.

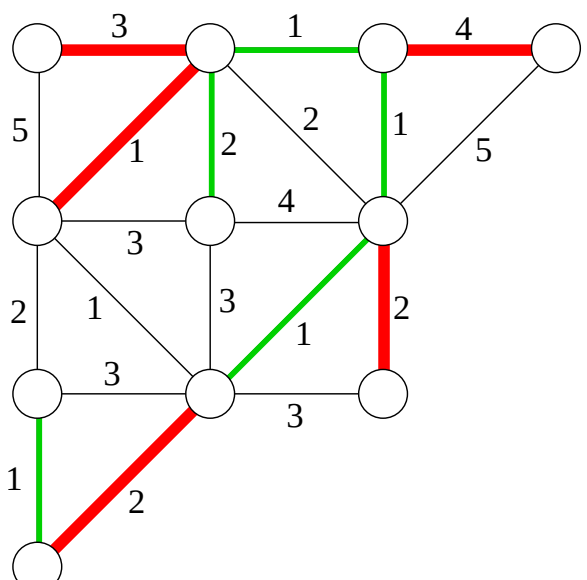


R ist erweiterbar, denn . . .

Die Schnitteigenschaft

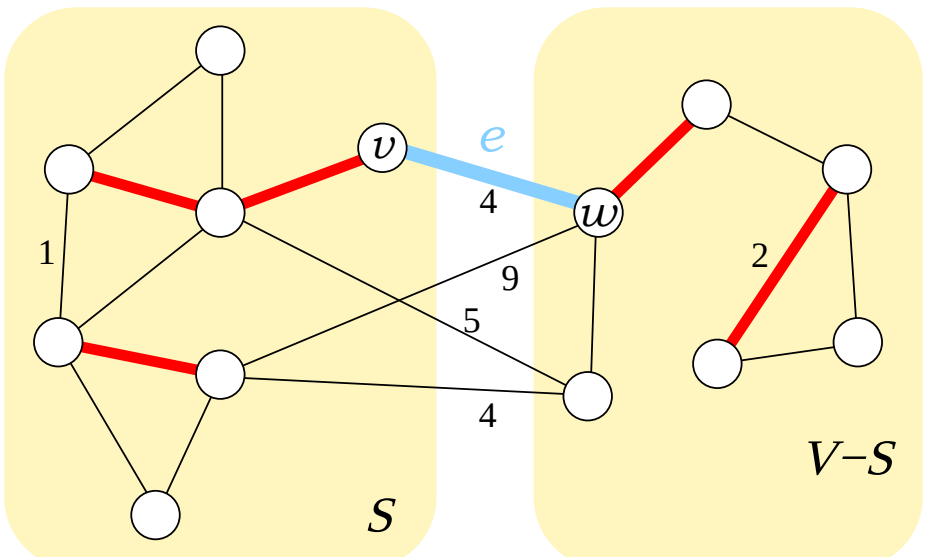
Definition 11.2.4

Eine Menge $R \subseteq E$ heißt **erweiterbar** (zu einem MST), wenn es einen MST T mit $R \subseteq T$ gibt.

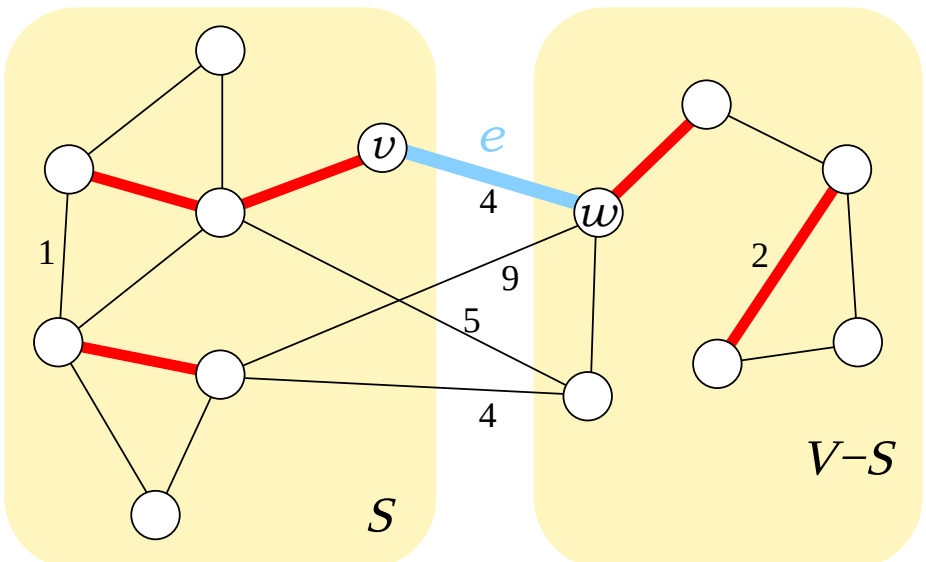


$\boldsymbol{R}$ ist erweiterbar, denn es gibt einen MST $\boldsymbol{T} \supseteq \boldsymbol{R}$.

Die Schnitteigenschaft

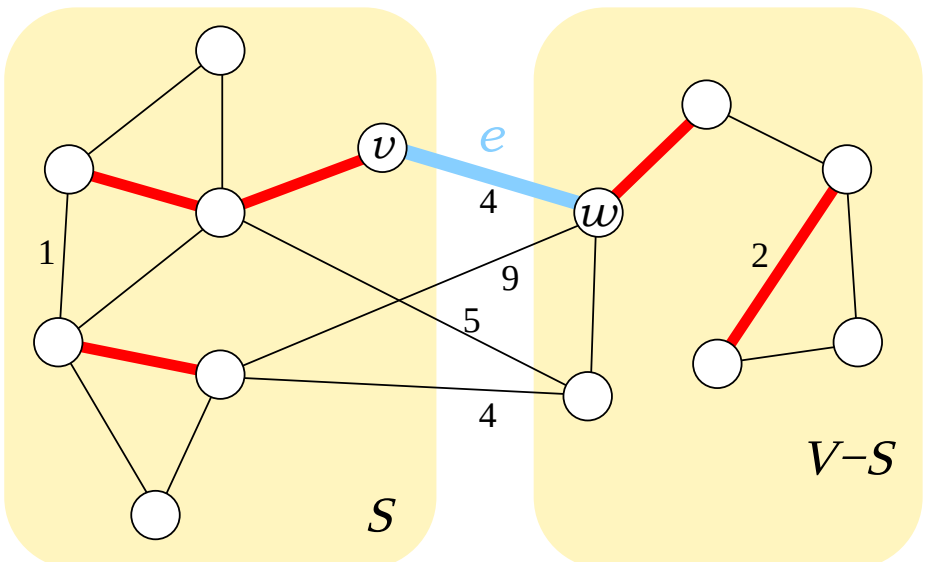


Die Schnitteigenschaft



Behauptung (Schnitteigenschaft):

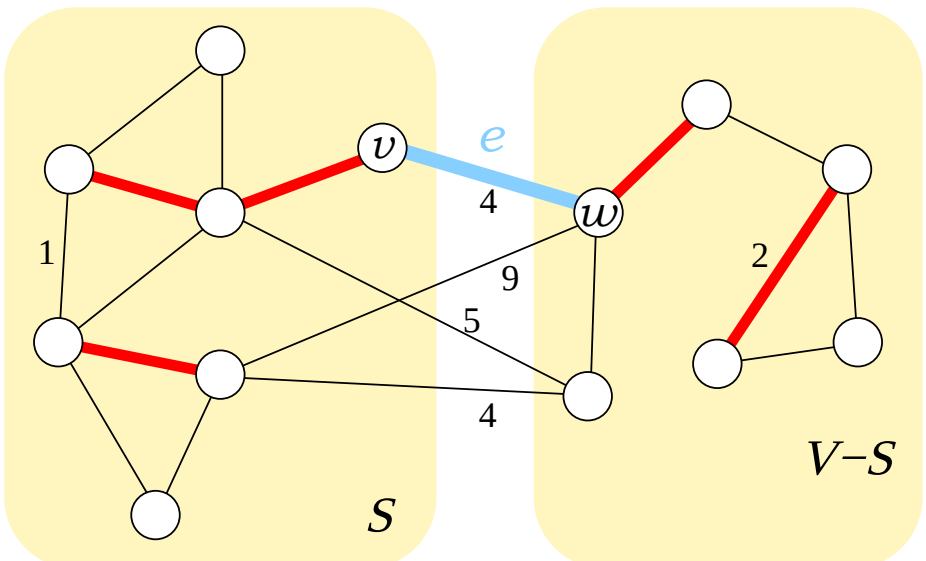
Die Schnitteigenschaft



Behauptung (Schnitteigenschaft):

Sei $\boldsymbol{R} \subseteq E$ erweiterbar **und** sei $(S, V - S)$ ein Schnitt, so dass es keine $\boldsymbol{R}$-Kante von S nach $V - S$ gibt,

Die Schnitteigenschaft

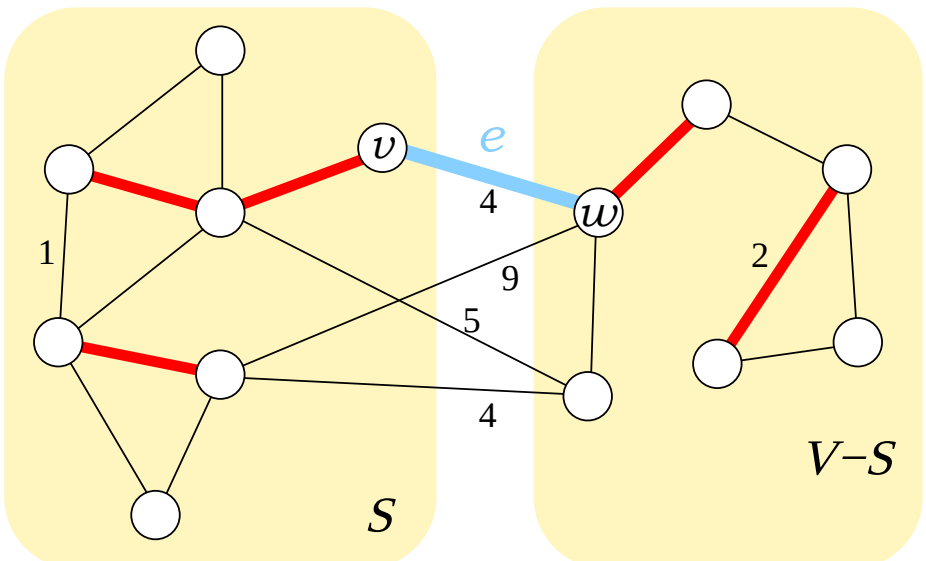


Behauptung (Schnitteigenschaft):

Sei $\boldsymbol{R} \subseteq E$ erweiterbar **und** sei $(S, V - S)$ ein Schnitt, so dass es keine $\boldsymbol{R}$-Kante von S nach $V - S$ gibt,

und sei $\boldsymbol{e} = (v, w)$, $v \in S$, $w \in V - S$ eine Kante, die den Wert $c((v', w'))$, $v' \in S$, $w' \in V - S$, **minimiert**.

Die Schnitteigenschaft



Behauptung (Schnitteigenschaft):

Sei $\boldsymbol{R} \subseteq E$ erweiterbar **und** sei $(S, V - S)$ ein Schnitt, so dass es keine $\boldsymbol{R}$-Kante von S nach $V - S$ gibt,

und sei $\boldsymbol{e} = (v, w)$, $v \in S$, $w \in V - S$ eine Kante, die den Wert $c((v', w'))$, $v' \in S$, $w' \in V - S$, **minimiert**.

Dann ist auch $\boldsymbol{R} \cup \{\boldsymbol{e}\}$ erweiterbar.

Die Schnitteigenschaft

Beweis der Schnitteigenschaft: Sei $R \subseteq E$, sei $T \supseteq R$ ein MST;
sei $(S, V - S)$ ein Schnitt, e wie in der Voraussetzung.

Die Schnitteigenschaft

Beweis der Schnitteigenschaft: Sei $R \subseteq E$, sei $T \supseteq R$ ein MST;
sei $(S, V - S)$ ein Schnitt, e wie in der Voraussetzung.

Wenn $e \in T$, dann gilt $R \cup \{e\} \subseteq T$, also ist $R \cup \{e\}$ erweiterbar.

Die Schnitteigenschaft

Beweis der Schnitteigenschaft: Sei $R \subseteq E$, sei $T \supseteq R$ ein MST;
sei $(S, V - S)$ ein Schnitt, e wie in der Voraussetzung.

Wenn $e \in T$, dann gilt $R \cup \{e\} \subseteq T$, also ist $R \cup \{e\}$ erweiterbar.

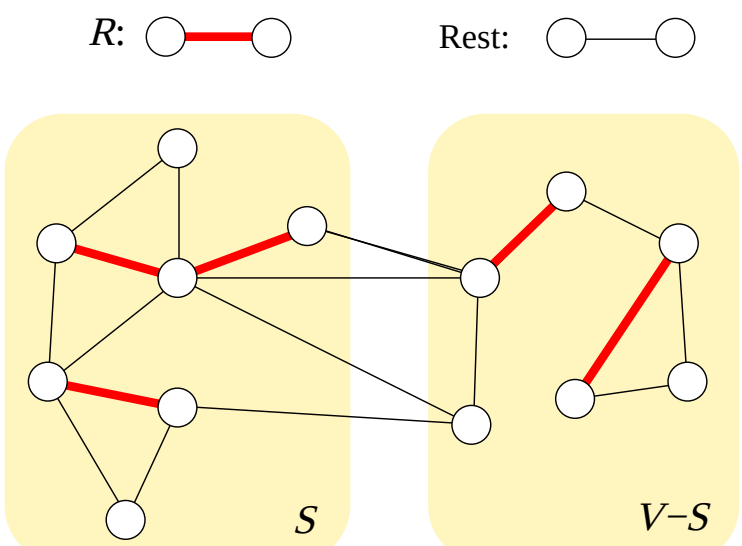
Ab hier: $e \notin T$.

Die Schnitteigenschaft

Beweis der Schnitteigenschaft: Sei $R \subseteq E$, sei $T \supseteq R$ ein MST;
sei $(S, V - S)$ ein Schnitt, e wie in der Voraussetzung.

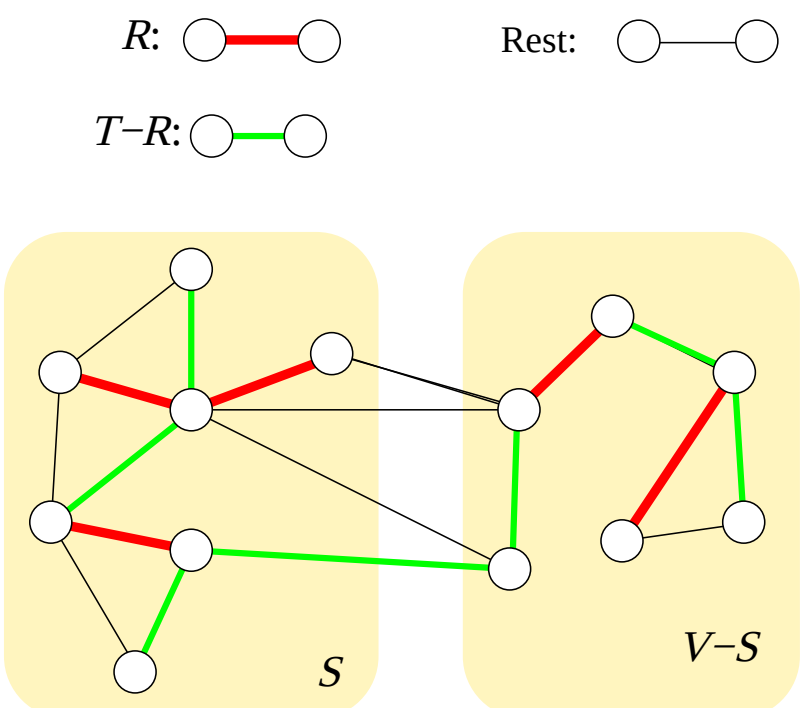
Wenn $e \in T$, dann gilt $R \cup \{e\} \subseteq T$, also ist $R \cup \{e\}$ erweiterbar.

Ab hier: $e \notin T$.



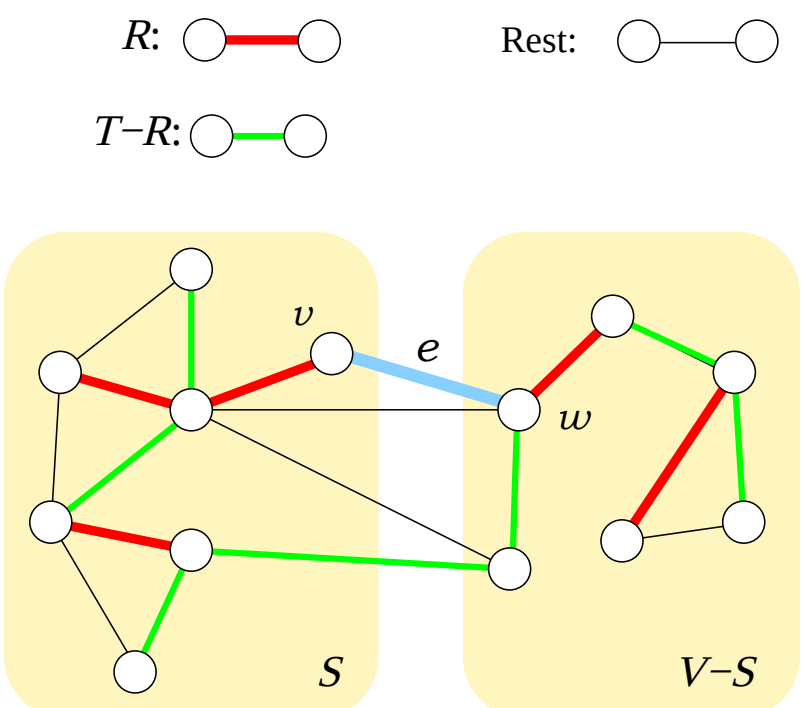
Eine erweiterbare Menge R.

Die Schnitteigenschaft



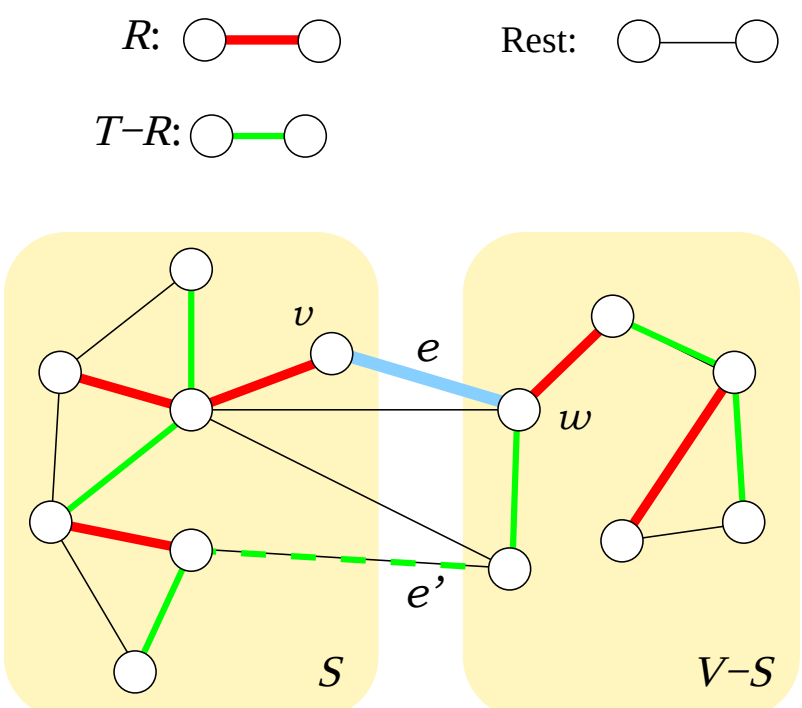
MST T mit $R \subseteq T$.

Die Schnitteigenschaft



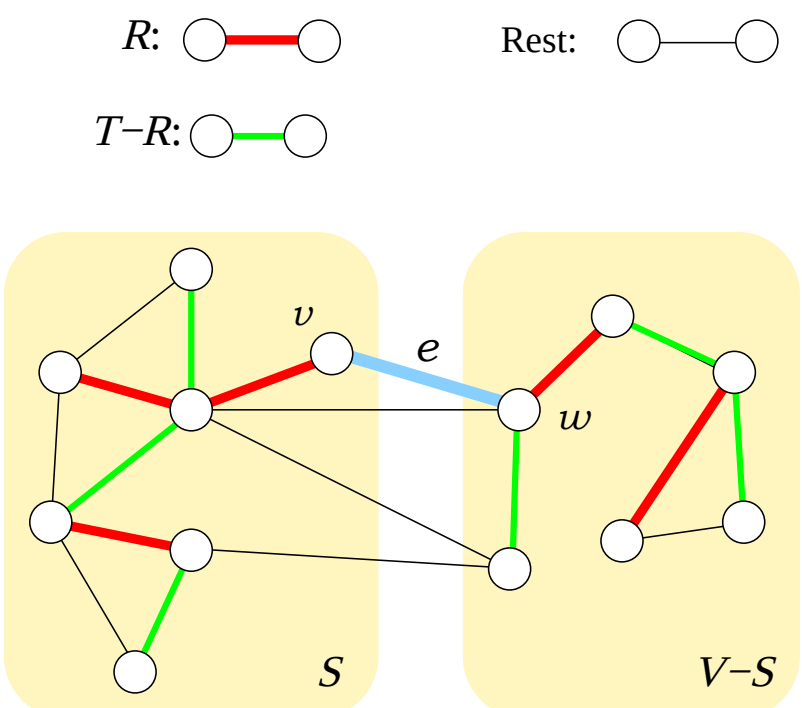
$e = (v, w)$ minimiert $c((v', w'))$, $v' \in S$, $w' \in V - S$.

Die Schnitteigenschaft



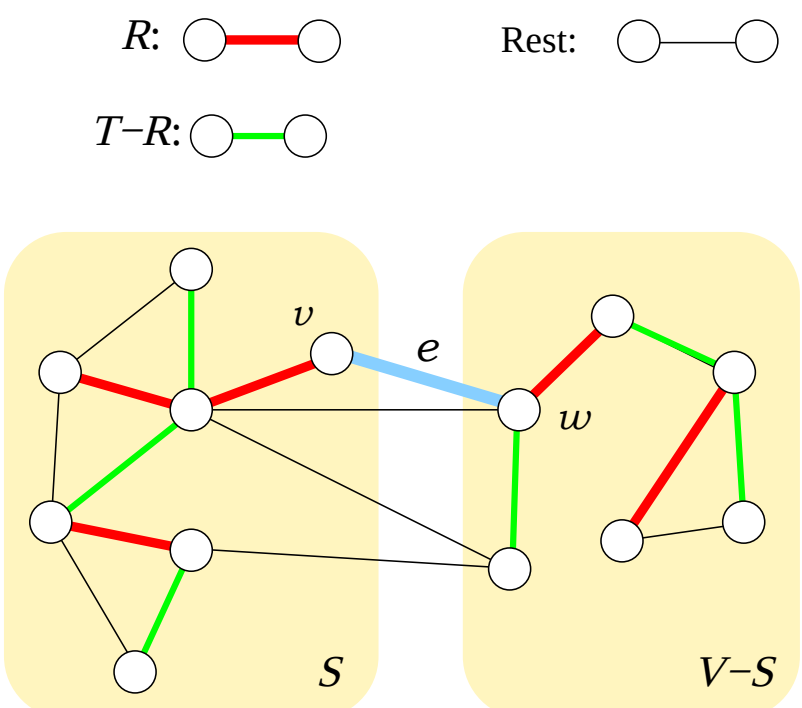
Weg in T von v nach w wechselt von S nach $V - S$ bei Kante e'.
Es entsteht ein Kreis in $T \cup \{e\}$, auf dem e und e' liegen.

Die Schnitteigenschaft



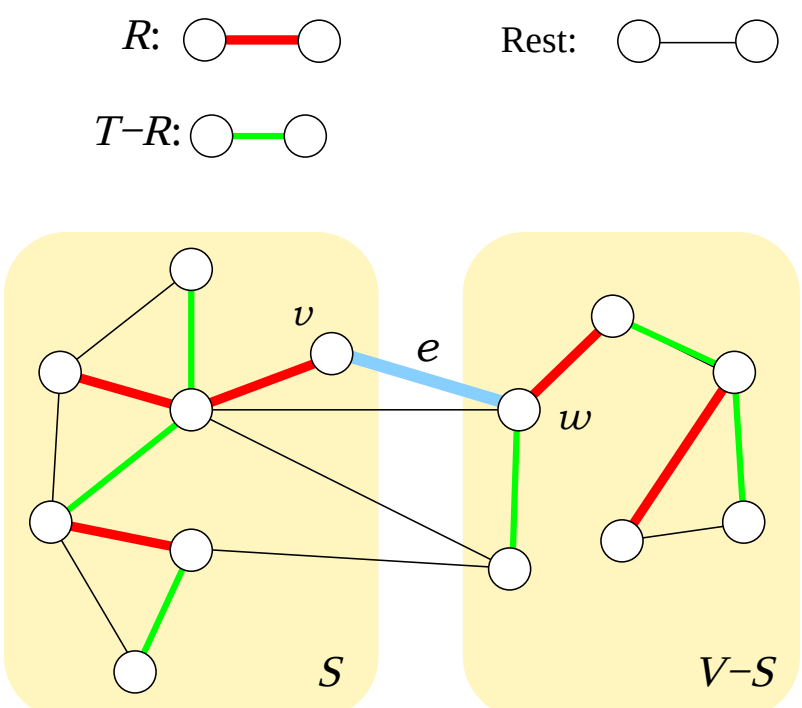
$T_e := (T - \{e'\}) \cup \{e\}$ ist Spannbaum und enthält $R \cup \{e\}$.

Die Schnitteigenschaft



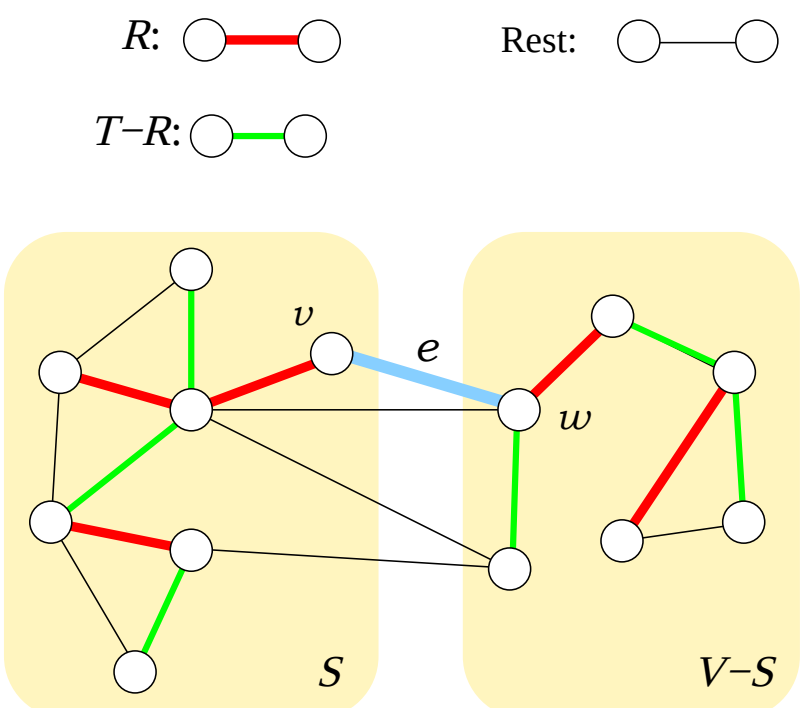
$T_e := (T - \{e'\}) \cup \{e\}$ ist Spannbaum und enthält $R \cup \{e\}$.
$c(T_e) - c(T) = c(e) - c(e') \leq 0$

Die Schnitteigenschaft



$T_e := (T - \{e'\}) \cup \{e\}$ ist Spannbaum und enthält $R \cup \{e\}$.
$c(T_e) - c(T) = c(e) - c(e') \leq 0$, also ist T_e optimal

Die Schnitteigenschaft



$T_e := (T - \{e'\}) \cup \{e\}$ ist Spannbaum und enthält $R \cup \{e\}$.
$c(T_e) - c(T) = c(e) - c(e') \leq 0$, also ist T_e optimal. Also ist $R \cup \{e\}$ erweiterbar.□

Korrektheit des Algorithmus von Jarník/Prim:

R_i: Kantenmenge (Größe i), die nach Runde i in R steht.

Korrektheit des Algorithmus von Jarník/Prim:

R_i: Kantenmenge (Größe i), die nach Runde i in R steht.
S_i: Knotenmenge (Größe $i+1$), die nach Runde i in S steht.

Korrektheit des Algorithmus von Jarník/Prim:

R_i: Kantenmenge (Größe i), die nach Runde i in R steht.
S_i: Knotenmenge (Größe $i+1$), die nach Runde i in S steht.

Weil in jeder Runde i ein neuer Knoten zu S_{i-1} hinzukommt, um S_i zu bilden, und die neue Kante in R_i diesen Knoten an S_{i-1} anschließt, ist jeder Graph (S_i, R_i) zusammenhängend. Da durch keine der neuen Kanten ein Kreis geschlossen wird, ist (S_i, R_i) sogar ein Baum.

Korrektheit des Algorithmus von Jarník/Prim:

R_i: Kantenmenge (Größe i), die nach Runde i in R steht.
S_i: Knotenmenge (Größe $i+1$), die nach Runde i in S steht.

Weil in jeder Runde i ein neuer Knoten zu S_{i-1} hinzukommt, um S_i zu bilden, und die neue Kante in R_i diesen Knoten an S_{i-1} anschließt, ist jeder Graph (S_i, R_i) zusammenhängend. Da durch keine der neuen Kanten ein Kreis geschlossen wird, ist (S_i, R_i) sogar ein Baum.

Der Prozess kann nicht steckenbleiben:
Wenn es keine Kante von S_{i-1} nach $V - S_{i-1}$ gäbe, wäre G nicht zusammenhängend.

Korrektheit des Algorithmus von Jarník/Prim:

R_i: Kantenmenge (Größe i), die nach Runde i in R steht.
S_i: Knotenmenge (Größe $i+1$), die nach Runde i in S steht.

Weil in jeder Runde i ein neuer Knoten zu S_{i-1} hinzukommt, um S_i zu bilden, und die neue Kante in R_i diesen Knoten an S_{i-1} anschließt, ist jeder Graph (S_i, R_i) zusammenhängend. Da durch keine der neuen Kanten ein Kreis geschlossen wird, ist (S_i, R_i) sogar ein Baum.

Der Prozess kann nicht steckenbleiben:
Wenn es keine Kante von S_{i-1} nach $V - S_{i-1}$ gäbe, wäre G nicht zusammenhängend.

Zu zeigen: R_{n-1} ist ein minimaler Spannbaum für G.

Korrektheit des Algorithmus von Jarník/Prim:

R_i: Kantenmenge (Größe i), die nach Runde i in R steht.
S_i: Knotenmenge (Größe $i+1$), die nach Runde i in S steht.

Weil in jeder Runde i ein neuer Knoten zu S_{i-1} hinzukommt, um S_i zu bilden, und die neue Kante in R_i diesen Knoten an S_{i-1} anschließt, ist jeder Graph (S_i, R_i) zusammenhängend. Da durch keine der neuen Kanten ein Kreis geschlossen wird, ist (S_i, R_i) sogar ein Baum.

Der Prozess kann nicht steckenbleiben:
Wenn es keine Kante von S_{i-1} nach $V - S_{i-1}$ gäbe, wäre G nicht zusammenhängend.

Zu zeigen: R_{n-1} ist ein minimaler Spannbaum für G.

Weil R_{n-1} kreisfrei ist und $n-1$ Kanten hat, ist R_{n-1} Spannbaum.

Korrektheit des Algorithmus von Jarník/Prim:

R_i: Kantenmenge (Größe i), die nach Runde i in R steht.
S_i: Knotenmenge (Größe $i+1$), die nach Runde i in S steht.

Weil in jeder Runde i ein neuer Knoten zu S_{i-1} hinzukommt, um S_i zu bilden, und die neue Kante in R_i diesen Knoten an S_{i-1} anschließt, ist jeder Graph (S_i, R_i) zusammenhängend. Da durch keine der neuen Kanten ein Kreis geschlossen wird, ist (S_i, R_i) sogar ein Baum.

Der Prozess kann nicht steckenbleiben:
Wenn es keine Kante von S_{i-1} nach $V - S_{i-1}$ gäbe, wäre G nicht zusammenhängend.

Zu zeigen: R_{n-1} ist ein minimaler Spannbaum für G.

Weil R_{n-1} kreisfrei ist und $n-1$ Kanten hat, ist R_{n-1} Spannbaum.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Korrektheit des Algorithmus von Jarník/Prim:

R_i: Kantenmenge (Größe i), die nach Runde i in R steht.
S_i: Knotenmenge (Größe $i+1$), die nach Runde i in S steht.

Weil in jeder Runde i ein neuer Knoten zu S_{i-1} hinzukommt, um S_i zu bilden, und die neue Kante in R_i diesen Knoten an S_{i-1} anschließt, ist jeder Graph (S_i, R_i) zusammenhängend. Da durch keine der neuen Kanten ein Kreis geschlossen wird, ist (S_i, R_i) sogar ein Baum.

Der Prozess kann nicht steckenbleiben:
Wenn es keine Kante von S_{i-1} nach $V - S_{i-1}$ gäbe, wäre G nicht zusammenhängend.

Zu zeigen: R_{n-1} ist ein minimaler Spannbaum für G.

Weil R_{n-1} kreisfrei ist und $n-1$ Kanten hat, ist R_{n-1} Spannbaum.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

(Dies beweisen wir gleich durch Induktion über $i = 0, 1, \ldots, n-1$.)

Korrektheit des Algorithmus von Jarník/Prim:

R_i: Kantenmenge (Größe i), die nach Runde i in R steht.
S_i: Knotenmenge (Größe $i+1$), die nach Runde i in S steht.

Weil in jeder Runde i ein neuer Knoten zu S_{i-1} hinzukommt, um S_i zu bilden, und die neue Kante in R_i diesen Knoten an S_{i-1} anschließt, ist jeder Graph (S_i, R_i) zusammenhängend. Da durch keine der neuen Kanten ein Kreis geschlossen wird, ist (S_i, R_i) sogar ein Baum.

Der Prozess kann nicht steckenbleiben:
Wenn es keine Kante von S_{i-1} nach $V - S_{i-1}$ gäbe, wäre G nicht zusammenhängend.

Zu zeigen: R_{n-1} ist ein minimaler Spannbaum für G.

Weil R_{n-1} kreisfrei ist und $n-1$ Kanten hat, ist R_{n-1} Spannbaum.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

(Dies beweisen wir gleich durch Induktion über $i = 0, 1, \ldots, n-1$.)

Dann besagt IB$(n-1)$, dass es einen MST T mit $T \supseteq R_{n-1}$ gibt.

Weil T als Spannbaum $n-1$ Kanten hat und R_{n-1} auch $n-1$ Kanten hat, ist $T = R_{n-1}$, also ist die Ausgabe R_{n-1} ein MST.

IB(i): R_i ist erweiterbar.

IB(i): R_i ist erweiterbar.

I.A.: $i = 0$. – Es gibt einen MST T für G, und $R_0 = \emptyset \subseteq T$ gilt trivialerweise.

IB(i): R_i ist erweiterbar.

I.A.: $i = 0$. – Es gibt einen MST T für G, und $R_0 = \emptyset \subseteq T$ gilt trivialerweise.

I.V.: $1 \leq i \leq n - 1$ und R_{i-1} ist erweiterbar.

IB(i): R_i ist erweiterbar.

I.A.: $i = 0$. – Es gibt einen MST T für G, und $R_0 = \emptyset \subseteq T$ gilt trivialerweise.

I.V.: $1 \leq i \leq n - 1$ und R_{i-1} ist erweiterbar.

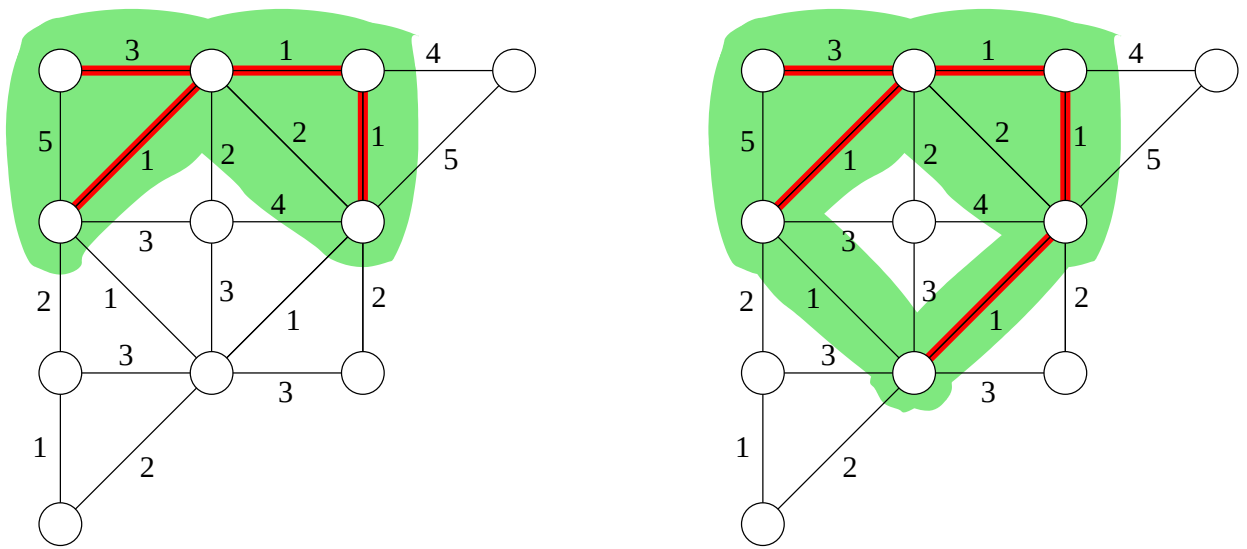
I.S.: $(S_{i-1}, V - S_{i-1})$ ist ein Schnitt, und keine Kante von R_{i-1} überquert diesen Schnitt.

IB(i): R_i ist erweiterbar.

I.A.: $i = 0$. – Es gibt einen MST T für G, und $R_0 = \emptyset \subseteq T$ gilt trivialerweise.

I.V.: $1 \leq i \leq n - 1$ und R_{i-1} ist erweiterbar.

I.S.: $(S_{i-1}, V - S_{i-1})$ ist ein Schnitt, und keine Kante von R_{i-1} überquert diesen Schnitt.



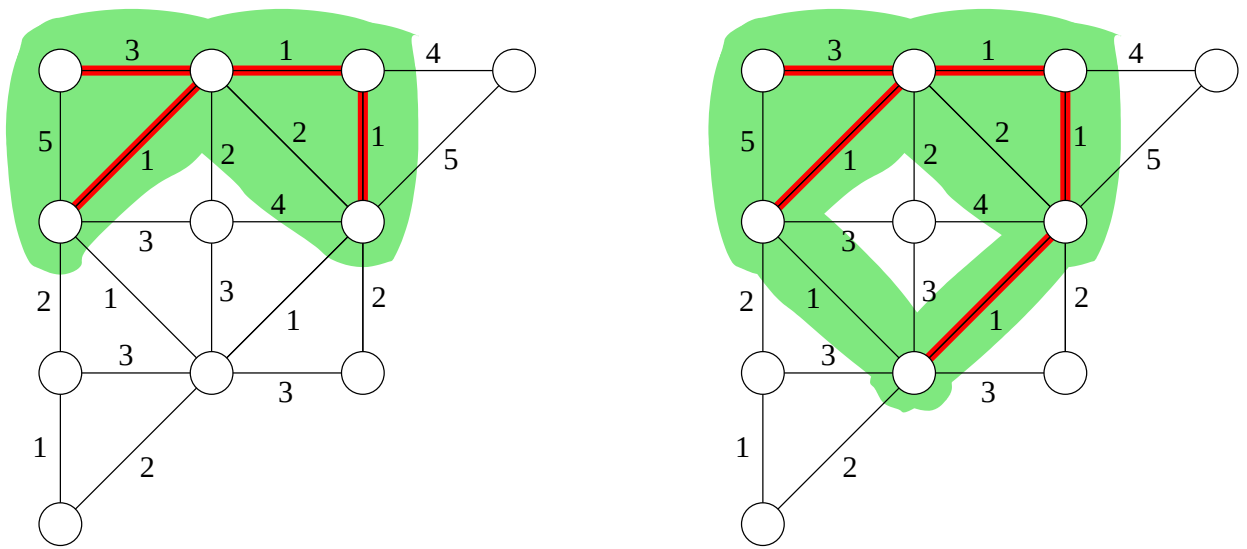
Der Algorithmus von Jarník/Prim wählt unter allen Kanten, die den Schnitt überqueren, eine billigste Kante e, und bildet $R_i := R_{i-1} \cup \{e\}$.

IB(i): R_i ist erweiterbar.

I.A.: $i = 0$. – Es gibt einen MST T für G, und $R_0 = \emptyset \subseteq T$ gilt trivialerweise.

I.V.: $1 \leq i \leq n - 1$ und R_{i-1} ist erweiterbar.

I.S.: $(S_{i-1}, V - S_{i-1})$ ist ein Schnitt, und keine Kante von R_{i-1} überquert diesen Schnitt.



Der Algorithmus von Jarník/Prim wählt unter allen Kanten, die den Schnitt überqueren, eine billigste Kante e, und bildet $R_i := R_{i-1} \cup \{e\}$. Mit der Schnitteigenschaft folgt: R_i ist erweiterbar. □

Implementierungsdetails im Algorithmus von Jarník/Prim:

Implementierungsdetails im Algorithmus von Jarník/Prim:

Wir nehmen an, dass $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$ in Adjazenzlistendarstellung gegeben ist.

Implementierungsdetails im Algorithmus von Jarník/Prim:

Wir nehmen an, dass $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$ in Adjazenzlistendarstellung gegeben ist. Die Kantengewichte $c(e)$ stehen in den Adjazenzlisten bei den Kanten.

Implementierungsdetails im Algorithmus von Jarník/Prim:

Wir nehmen an, dass $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$ in Adjazenzlistendarstellung gegeben ist. Die Kantengewichte $c(e)$ stehen in den Adjazenzlisten bei den Kanten.

Für jeden Knoten $w \in V - \mathtt{S}$ wollen wir immer wissen:

1) die Länge der billigsten Kante (v, w), $v \in \mathtt{S}$, falls es eine gibt:
   in `dist[`w`]` („Abstand von S“), für Array `dist[`$1..n$`]`.

Implementierungsdetails im Algorithmus von Jarník/Prim:

Wir nehmen an, dass $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$ in Adjazenzlistendarstellung gegeben ist. Die Kantengewichte $c(e)$ stehen in den Adjazenzlisten bei den Kanten.

Für jeden Knoten $w \in V - \texttt{S}$ wollen wir immer wissen:

1) die Länge der billigsten Kante (v, w), $v \in \texttt{S}$, falls es eine gibt:
   in $\texttt{dist}[w]$ („Abstand von S“), für Array $\texttt{dist}[1..n]$.
2) den (einen) Knoten $p(w) \in \texttt{S}$ mit $c(p(w), w) = \texttt{dist}[w]$, falls ein solcher existiert:
   in $\texttt{p}[w]$ („Vorgänger in S“), für Array $\texttt{p}[1..n]$.

Implementierungsdetails im Algorithmus von Jarník/Prim:

Wir nehmen an, dass $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$ in Adjazenzlistendarstellung gegeben ist. Die Kantengewichte $c(e)$ stehen in den Adjazenzlisten bei den Kanten.

Für jeden Knoten $w \in V - \mathtt{S}$ wollen wir immer wissen:

1) die Länge der billigsten Kante (v, w), $v \in \mathtt{S}$, falls es eine gibt:
   in $\mathtt{dist}[w]$ („Abstand von S“), für Array $\mathtt{dist}[1..n]$.
2) den (einen) Knoten $p(w) \in \mathtt{S}$ mit $c(p(w), w) = \mathtt{dist}[w]$, falls ein solcher existiert:
   in $\mathtt{p}[w]$ („Vorgänger in S“), für Array $\mathtt{p}[1..n]$.

Solange es von S keine Kante nach w gibt, gilt $\mathtt{dist}[w] = \infty$ und $\mathtt{p}[w] = -1$.

Implementierungsdetails im Algorithmus von Jarník/Prim:

Wir nehmen an, dass $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$ in Adjazenzlistendarstellung gegeben ist. Die Kantengewichte $c(e)$ stehen in den Adjazenzlisten bei den Kanten.

Für jeden Knoten $w \in V - \mathtt{S}$ wollen wir immer wissen:

1) die Länge der billigsten Kante (v, w), $v \in \mathtt{S}$, falls es eine gibt:
   in $\mathtt{dist}[w]$ („Abstand von S“), für Array $\mathtt{dist}[1..n]$.
2) den (einen) Knoten $p(w) \in \mathtt{S}$ mit $c(p(w), w) = \mathtt{dist}[w]$, falls ein solcher existiert:
   in $\mathtt{p}[w]$ („Vorgänger in S“), für Array $\mathtt{p}[1..n]$.

Solange es von S keine Kante nach w gibt, gilt $\mathtt{dist}[w] = \infty$ und $\mathtt{p}[w] = -1$.

Verwalte die Knoten $w \in V - \mathtt{S}$ mit Werten $\mathtt{dist}[w] < \infty$ mit den $\mathtt{dist}[w]$-Werten als **Schlüssel**

Implementierungsdetails im Algorithmus von Jarník/Prim:

Wir nehmen an, dass $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$ in Adjazenzlistendarstellung gegeben ist. Die Kantengewichte $c(e)$ stehen in den Adjazenzlisten bei den Kanten.

Für jeden Knoten $w \in V - \mathtt{S}$ wollen wir immer wissen:

1) die Länge der billigsten Kante (v, w), $v \in \mathtt{S}$, falls es eine gibt:
   in $\mathtt{dist}[w]$ („Abstand von S“), für Array $\mathtt{dist}[1..n]$.
2) den (einen) Knoten $p(w) \in \mathtt{S}$ mit $c(p(w), w) = \mathtt{dist}[w]$, falls ein solcher existiert:
   in $\mathtt{p}[w]$ („Vorgänger in S“), für Array $\mathtt{p}[1..n]$.

Solange es von S keine Kante nach w gibt, gilt $\mathtt{dist}[w] = \infty$ und $\mathtt{p}[w] = -1$.

Verwalte die Knoten $w \in V - \mathtt{S}$ mit Werten $\mathtt{dist}[w] < \infty$ mit den $\mathtt{dist}[w]$-Werten als **Schlüssel** in einer **Prioritätswarteschlange** **PQ**.

Implementierungsdetails im Algorithmus von Jarník/Prim:

Wir nehmen an, dass $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$ in Adjazenzlistendarstellung gegeben ist. Die Kantengewichte $c(e)$ stehen in den Adjazenzlisten bei den Kanten.

Für jeden Knoten $w \in V - \mathtt{S}$ wollen wir immer wissen:

1) die Länge der billigsten Kante (v, w), $v \in \mathtt{S}$, falls es eine gibt:
   in $\mathtt{dist}[w]$ („Abstand von S“), für Array $\mathtt{dist}[1..n]$.
2) den (einen) Knoten $p(w) \in \mathtt{S}$ mit $c(p(w), w) = \mathtt{dist}[w]$, falls ein solcher existiert:
   in $\mathtt{p}[w]$ („Vorgänger in S“), für Array $\mathtt{p}[1..n]$.

Solange es von S keine Kante nach w gibt, gilt $\mathtt{dist}[w] = \infty$ und $\mathtt{p}[w] = -1$.

Verwalte die Knoten $w \in V - \mathtt{S}$ mit Werten $\mathtt{dist}[w] < \infty$ mit den $\mathtt{dist}[w]$-Werten als **Schlüssel** in einer **Prioritätswarteschlange** **PQ**.

Wenn $\mathtt{dist}[w] = \infty$, ist w (noch) nicht in der **PQ**.

Jarnik/Prim(G, s) // (Vollversion mit Prioritätswarteschlange)

Eingabe: gewichteter **Graph** $G = (V, E, c)$, $V = \{1, \ldots, n\}$, Startknoten $s \in V$ (ist beliebig);

Ausgabe: Ein MST für G.

Hilfsstrukturen: PQ: eine (anfangs leere) Prioritäts-WS; `inS`, `p`: wie oben

```
(1)   for w from 1 to n do
(2)     dist[w] ← ∞; inS[w] ← false; p[w] ← −1;
(3)   dist[s] ← 0; p[s] ← −2; PQ.insert(s);
(4)   while not PQ.isempty do
(5)     u ← PQ.extractMin; inS[u] ← true;
(6)     for Knoten w mit (u, w) ∈ E and not inS[w] do
(7)       dd ← c(u, w);    // einziger Unterschied zu Dijkstra!
(8)       if p[w] ≥ 0 and dd < dist[w] then
(9)         PQ.decreaseKey(w,dd); p[w] ← u; dist[w] ← dd;
(10)      if p[w] = −1 then   // w vorher nicht zu S benachbart
(11)        dist[w] ← dd; p[w] ← u; PQ.insert(w);
(12)  Ausgabe: T = {(w, p[w]) | inS[w] = true, w ≠ s}.   // Menge der gewählten Kanten
```

Zeitanalyse: Exakt wie für den Dijkstra-Algorithmus.

Zeitanalyse: Exakt wie für den Dijkstra-Algorithmus.

Satz 11.2.5

Der Algorithmus von Jarník/Prim mit Verwendung einer Prioritätswarteschlange, die als Binärheap realisiert ist, ermittelt einen minimalen Spannbaum für $G = (V, E, c)$ in Zeit $\boldsymbol{O((n+m)\log n)}$.

Zeitanalyse: Exakt wie für den Dijkstra-Algorithmus.

Satz 11.2.5

Der Algorithmus von Jarník/Prim mit Verwendung einer Prioritätswarteschlange, die als Binärheap realisiert ist, ermittelt einen minimalen Spannbaum für $G = (V, E, c)$ in Zeit $\boldsymbol{O((n+m)\log n)}$.

Wie beim Algorithmus von Dijkstra: Wenn man Fibonacci-Heaps o. ä. verwendet, bei denen m decreaseKey-Operationen Zeit $O(m)$ benötigen:
Rechenzeit für Jarník/Prim: $\boldsymbol{O(m + n\log n)}$.

PAUSE

Als nächstes: Algorithmus von Kruskal für minimale Spannbäume

11.3 Der Algorithmus von Kruskal

Auch dieser Algorithmus löst das **MST-Problem**.

11.3 Der Algorithmus von Kruskal

Auch dieser Algorithmus löst das **MST-Problem**.

Anderer Ansatz als bei Jarník/Prim, aber auch „greedy“:
Starte mit $\boldsymbol{R} = \emptyset$. Dann folgen $n - 1$ Runden.

11.3 Der Algorithmus von Kruskal

Auch dieser Algorithmus löst das **MST-Problem**.

Anderer Ansatz als bei Jarník/Prim, aber auch „greedy“:
Starte mit $\boldsymbol{R} = \emptyset$. Dann folgen $n - 1$ Runden.

In jeder Runde:

> Wähle eine Kante $\boldsymbol{e} \in E - \boldsymbol{R}$ von kleinstem Gewicht,
> die mit $(V, \boldsymbol{R})$ keinen Kreis schließt, und füge $\boldsymbol{e}$ zu $\boldsymbol{R}$ hinzu.

11.3 Der Algorithmus von Kruskal

Auch dieser Algorithmus löst das **MST-Problem**.

Anderer Ansatz als bei Jarník/Prim, aber auch „greedy“:
Starte mit $\boldsymbol{R} = \emptyset$. Dann folgen $n - 1$ Runden.

In jeder Runde:

> Wähle eine Kante $\boldsymbol{e} \in E - \boldsymbol{R}$ von kleinstem Gewicht,
> die mit $(V, \boldsymbol{R})$ keinen Kreis schließt, und füge $\boldsymbol{e}$ zu $\boldsymbol{R}$ hinzu.

Eine offensichtlich korrekte Methode, dies zu organisieren:

> Durchmustere Kanten in **aufsteigender Reihenfolge** des Kantengewichts, und nimm eine Kante genau dann in $\boldsymbol{R}$ auf, wenn sie mit $\boldsymbol{R}$ keinen Kreis bildet.

Algorithmus von Kruskal, informal

Algorithmus von Kruskal, informal

1. Schritt:

Sortiere die Kanten $e_1, \ldots, e_m$ nach den Gewichten $c(e_1), \ldots, c(e_m)$ aufsteigend.
Also o.B.d.A.: $c(e_1) \leq \ldots \leq c(e_m)$.

Algorithmus von Kruskal, informal

1. Schritt:
Sortiere die Kanten $e_1, \ldots, e_m$ nach den Gewichten $c(e_1), \ldots, c(e_m)$ aufsteigend.
Also o.B.d.A.: $c(e_1) \leq \ldots \leq c(e_m)$.

2. Schritt: Setze $\mathtt{R} \leftarrow \emptyset$.

Algorithmus von Kruskal, informal

1. Schritt:
Sortiere die Kanten $e_1, \ldots, e_m$ nach den Gewichten $c(e_1), \ldots, c(e_m)$ aufsteigend.
Also o.B.d.A.: $c(e_1) \leq \ldots \leq c(e_m)$.

2. Schritt: Setze $\mathtt{R} \leftarrow \emptyset$.

3. Schritt: Für $i = 1, 2, \ldots, m$ tue folgendes:
Falls $\mathtt{R} \cup \{e_i\}$ kreisfrei ist, setze $\mathtt{R} \leftarrow \mathtt{R} \cup \{e_i\}$
// sonst, d.h. wenn e_i einen Kreis schließt, bleibt $\mathtt{R}$ unverändert

Algorithmus von Kruskal, informal

1. Schritt:
Sortiere die Kanten $e_1, \ldots, e_m$ nach den Gewichten $c(e_1), \ldots, c(e_m)$ aufsteigend.
Also o.B.d.A.: $c(e_1) \leq \ldots \leq c(e_m)$.

2. Schritt: Setze $\mathtt{R} \leftarrow \emptyset$.

3. Schritt: Für $i = 1, 2, \ldots, m$ tue folgendes:
Falls $\mathtt{R} \cup \{e_i\}$ kreisfrei ist, setze $\mathtt{R} \leftarrow \mathtt{R} \cup \{e_i\}$
// sonst, d.h. wenn e_i einen Kreis schließt, bleibt $\mathtt{R}$ unverändert

// Optional: Beende Schleife, wenn $|\mathtt{R}| = n - 1$.

Algorithmus von Kruskal, informal

1. Schritt:
Sortiere die Kanten $e_1, \ldots, e_m$ nach den Gewichten $c(e_1), \ldots, c(e_m)$ aufsteigend.
Also o.B.d.A.: $c(e_1) \leq \ldots \leq c(e_m)$.

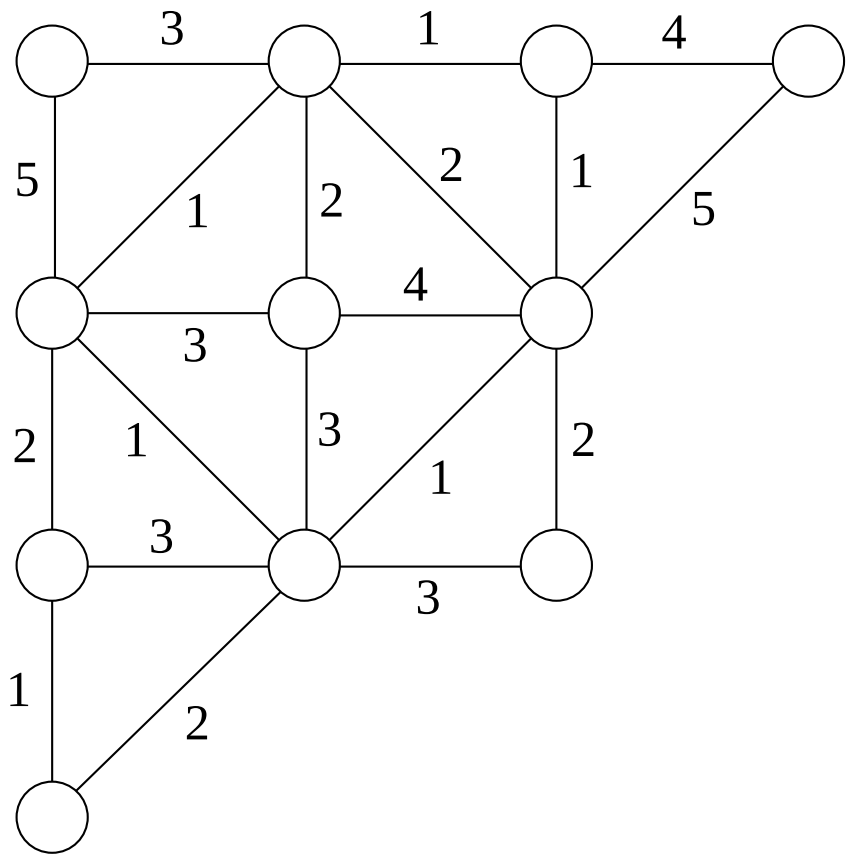
2. Schritt: Setze $\mathtt{R} \leftarrow \emptyset$.

3. Schritt: Für $i = 1, 2, \ldots, m$ tue folgendes:
Falls $\mathtt{R} \cup \{e_i\}$ kreisfrei ist, setze $\mathtt{R} \leftarrow \mathtt{R} \cup \{e_i\}$
// sonst, d.h. wenn e_i einen Kreis schließt, bleibt $\mathtt{R}$ unverändert

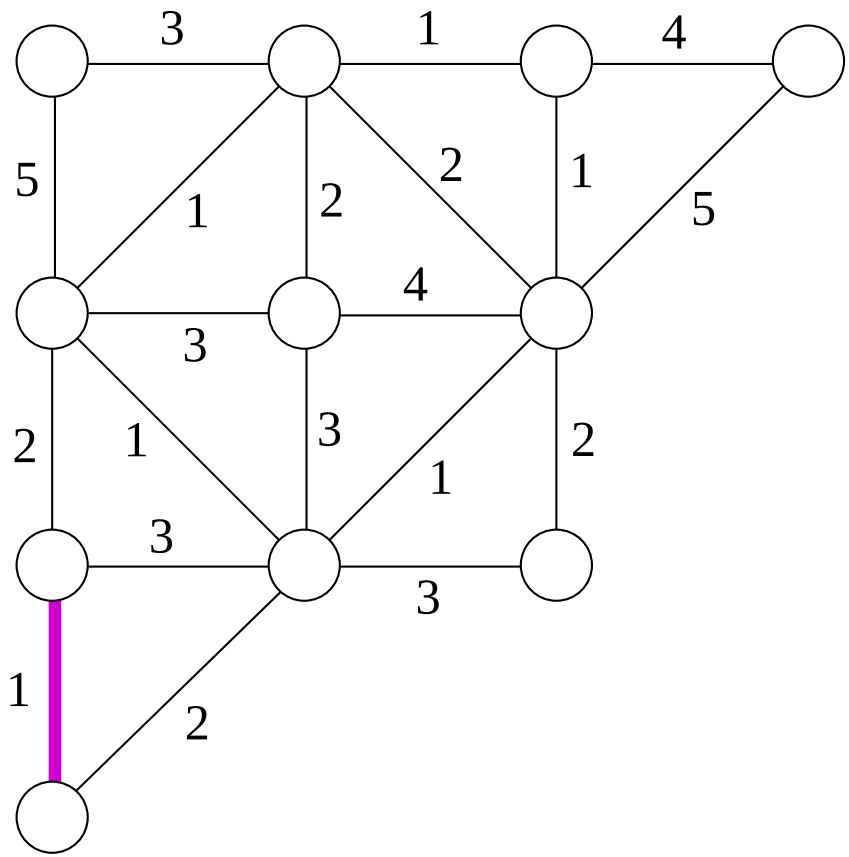
// Optional: Beende Schleife, wenn $|\mathtt{R}| = n - 1$.

4. Schritt: Die Ausgabe ist $\mathtt{R}$.

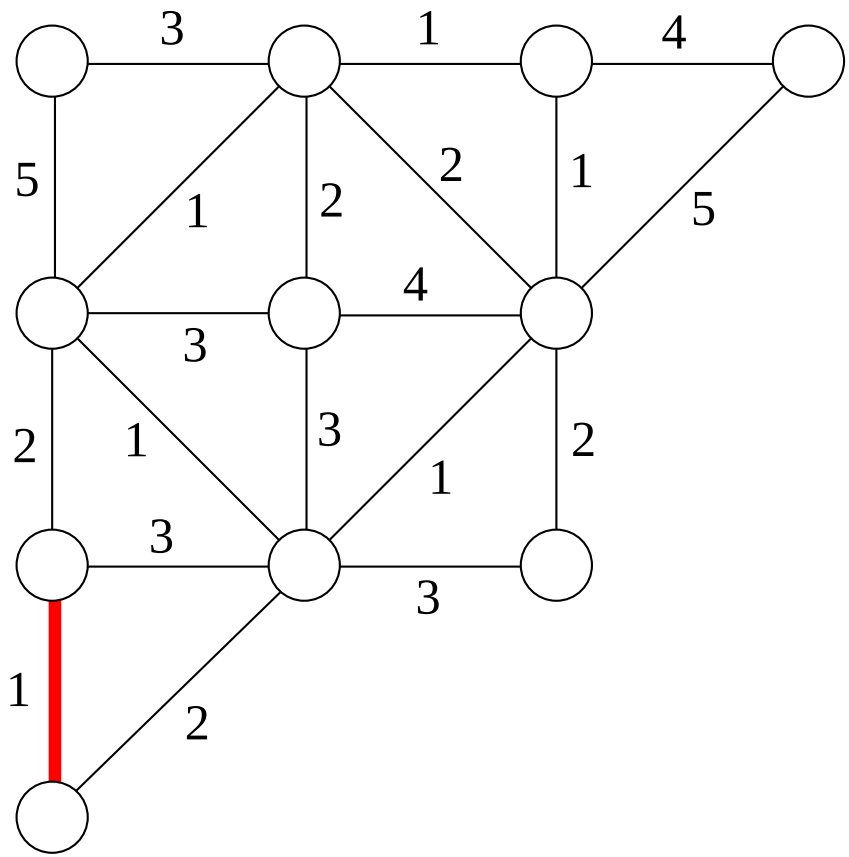
Beispiel (Kruskal):



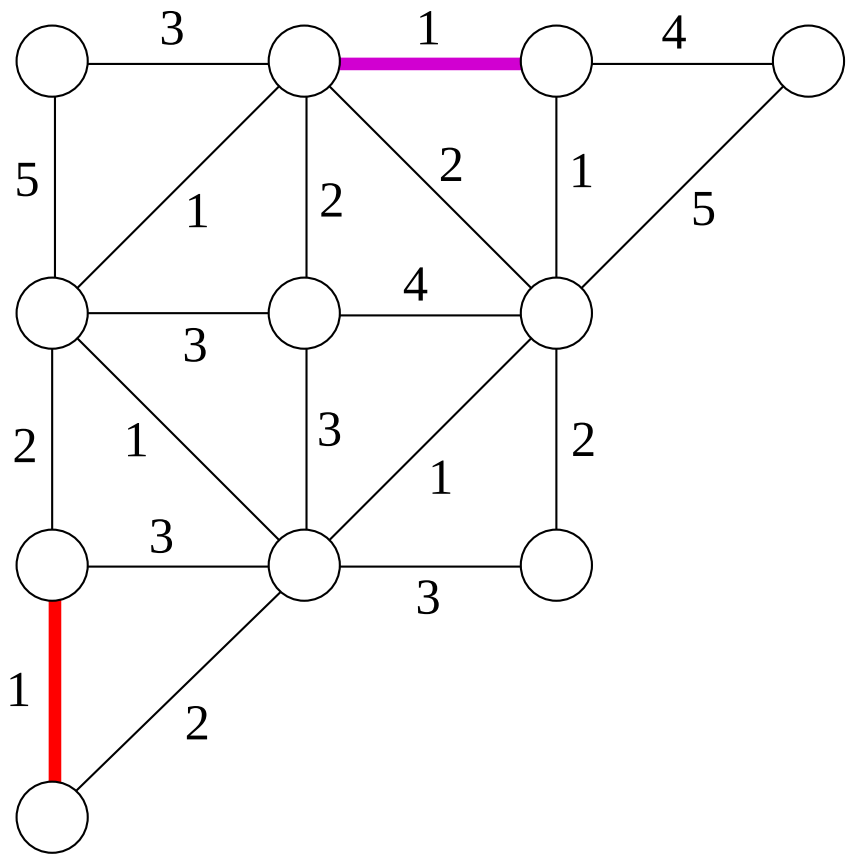
Beispiel (Kruskal):



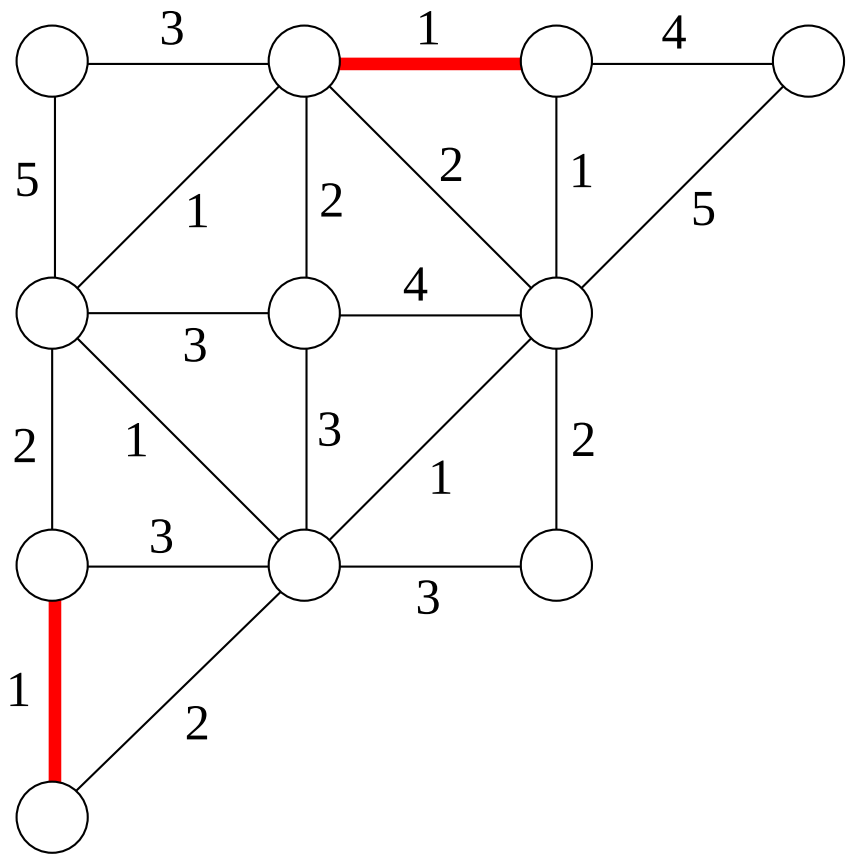
Beispiel (Kruskal):



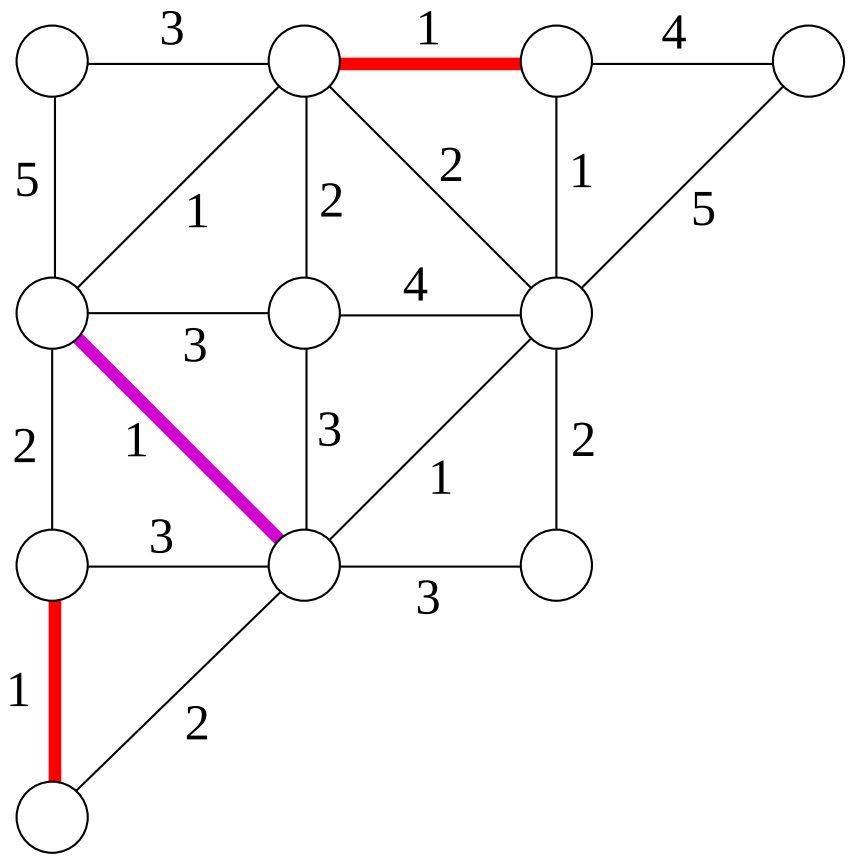
Beispiel (Kruskal):



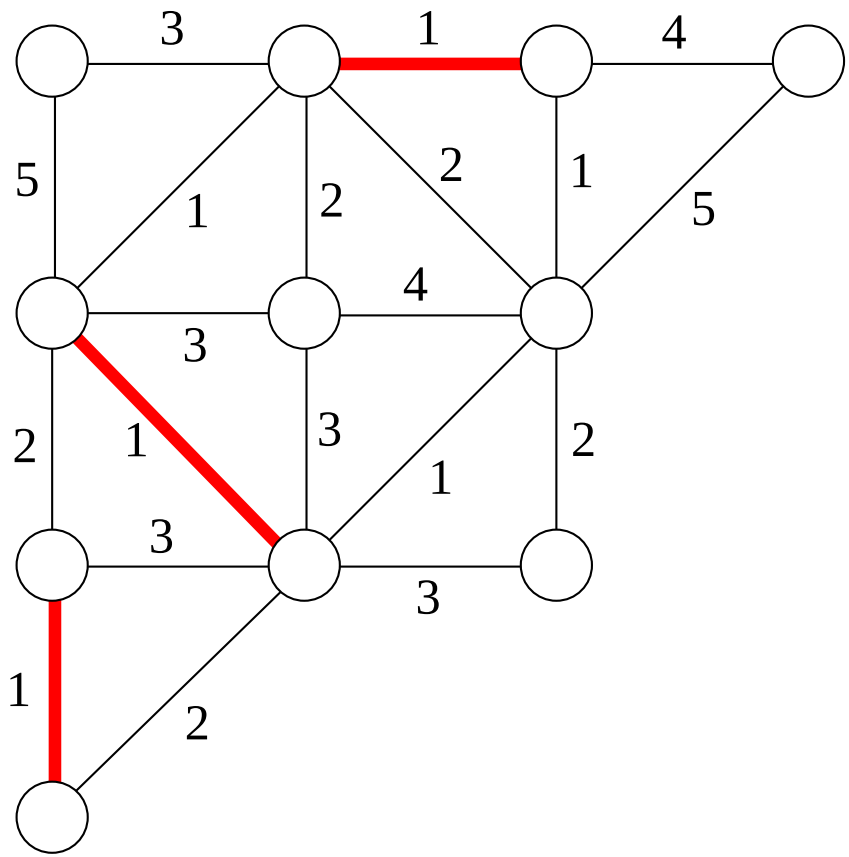
Beispiel (Kruskal):



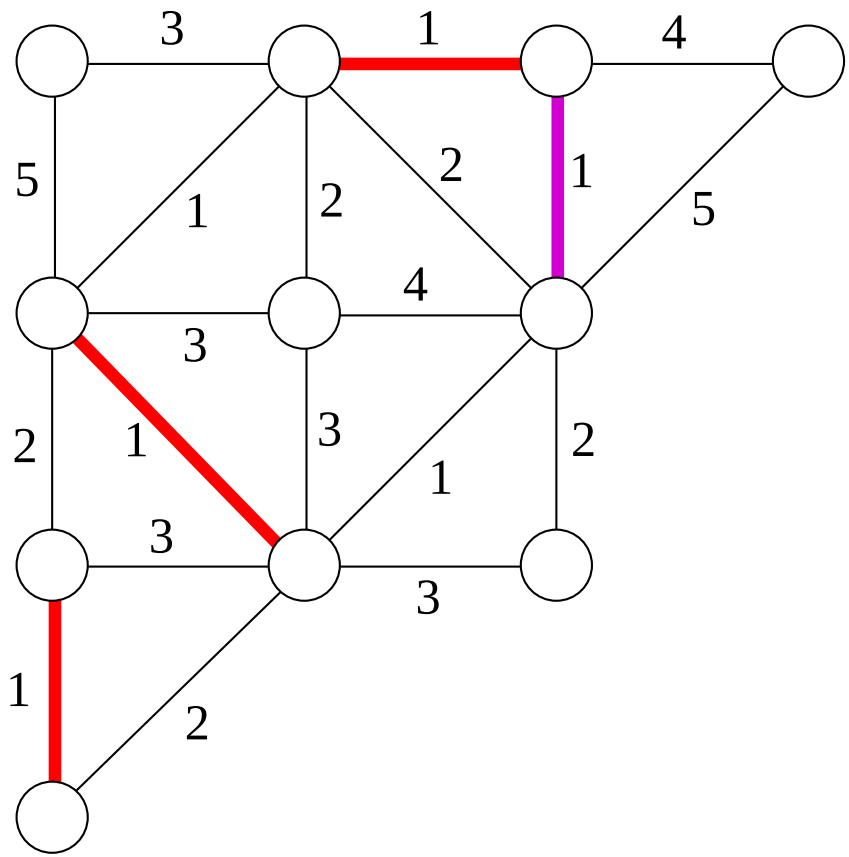
Beispiel (Kruskal):



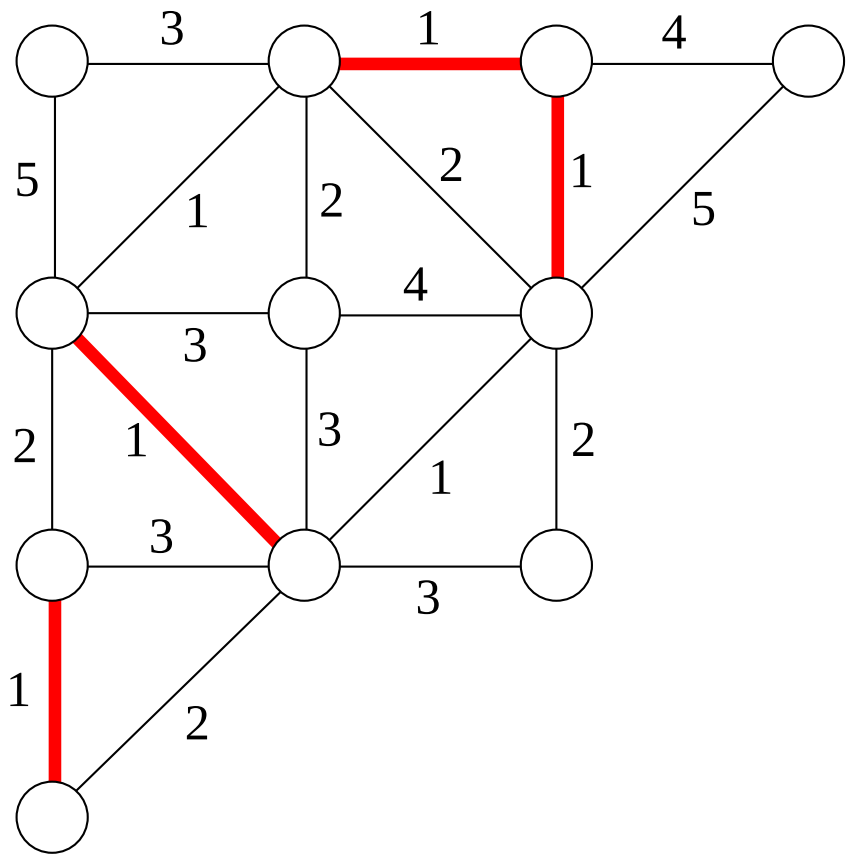
Beispiel (Kruskal):



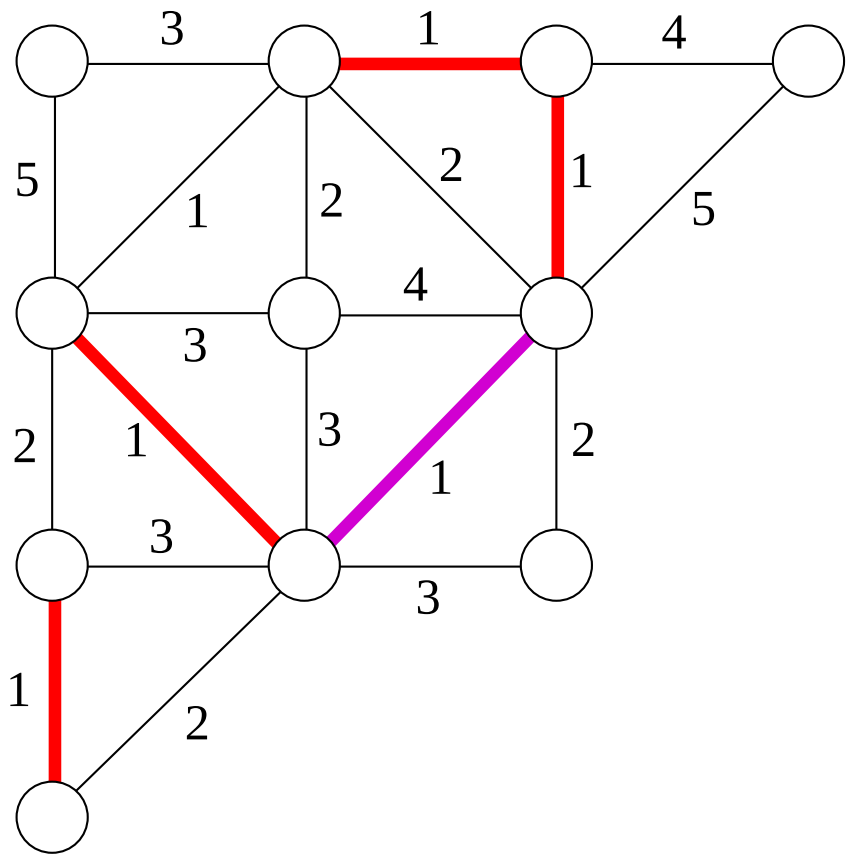
Beispiel (Kruskal):



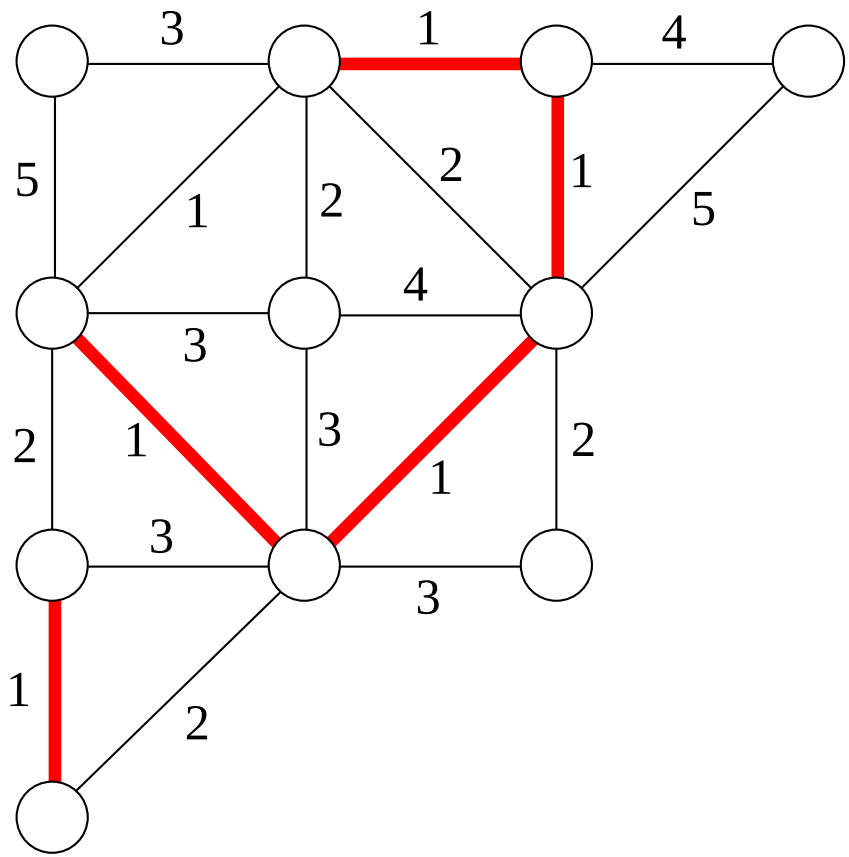
Beispiel (Kruskal):



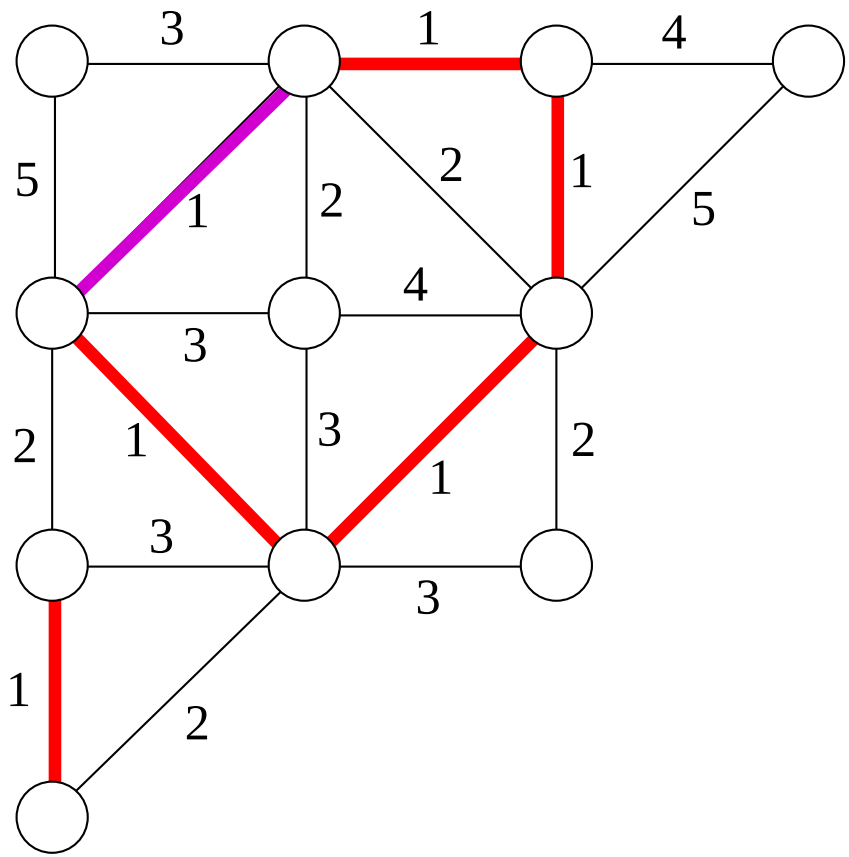
Beispiel (Kruskal):



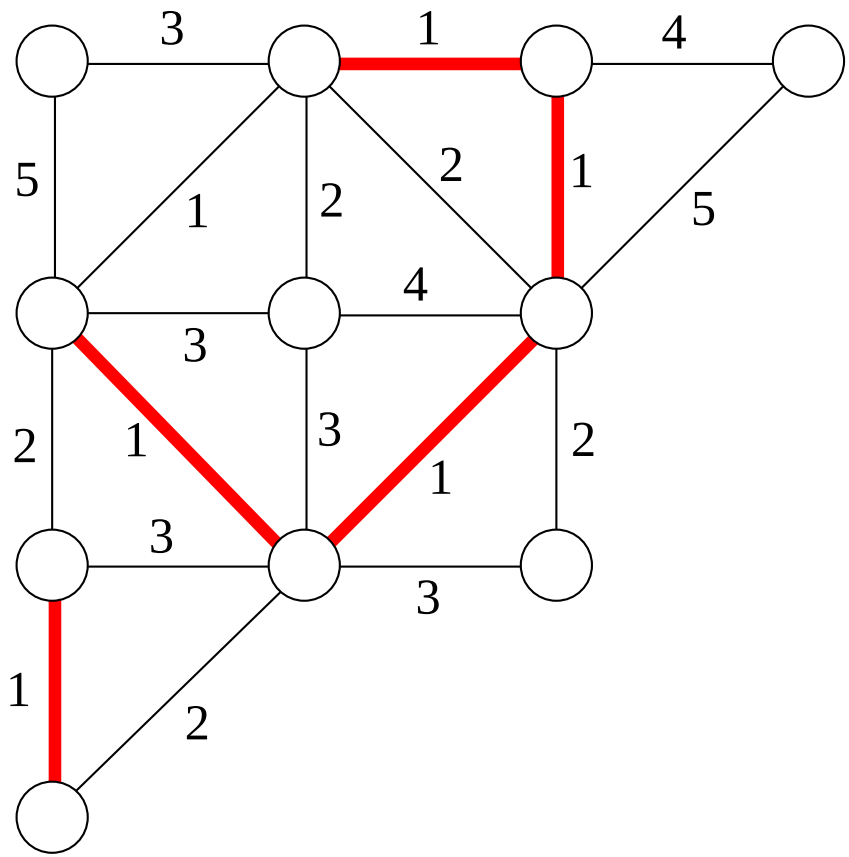
Beispiel (Kruskal):



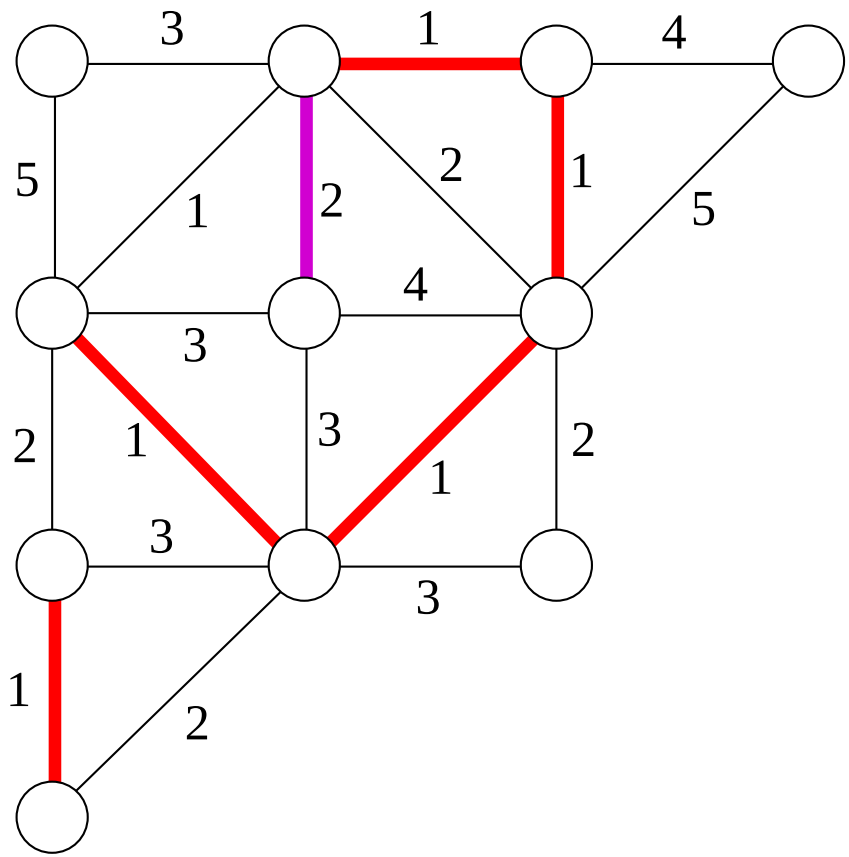
Beispiel (Kruskal):



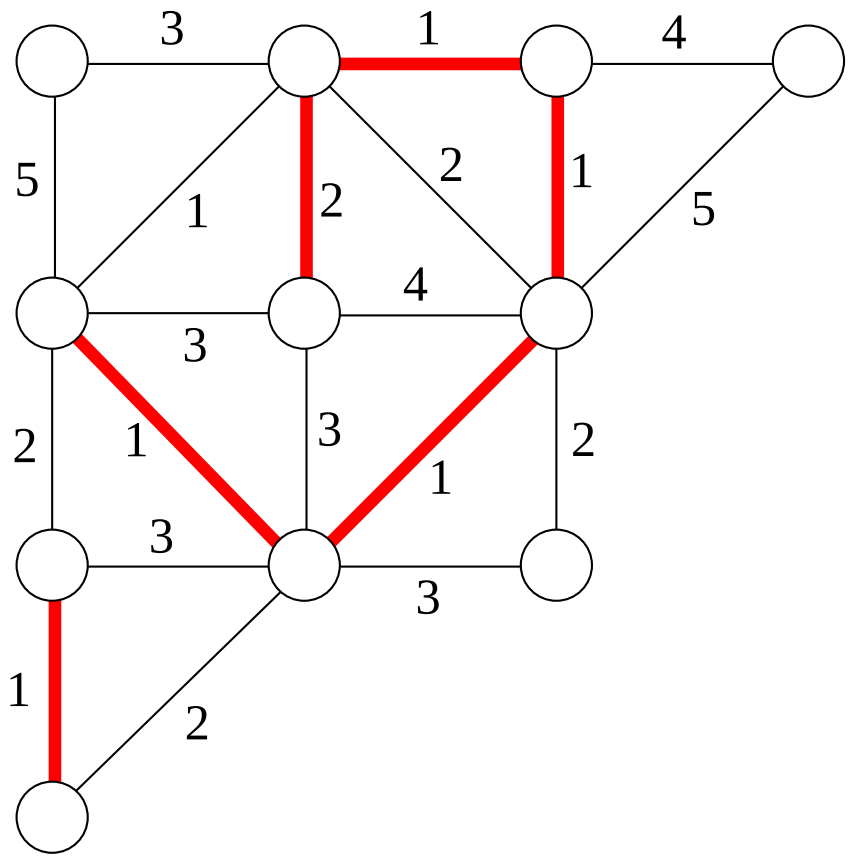
Beispiel (Kruskal):



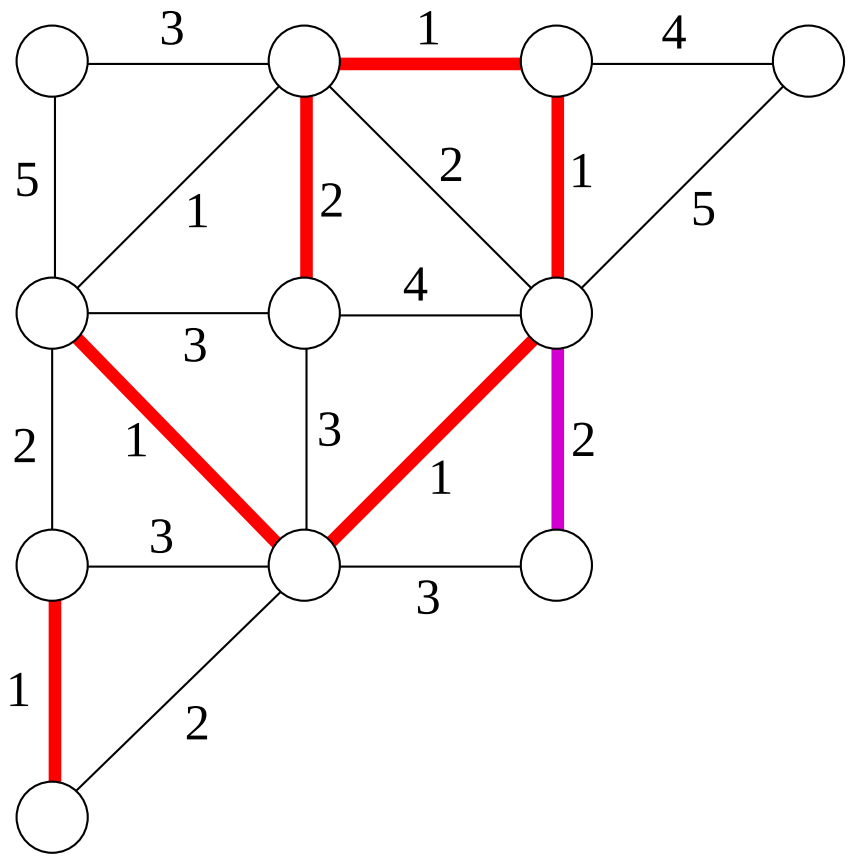
Beispiel (Kruskal):



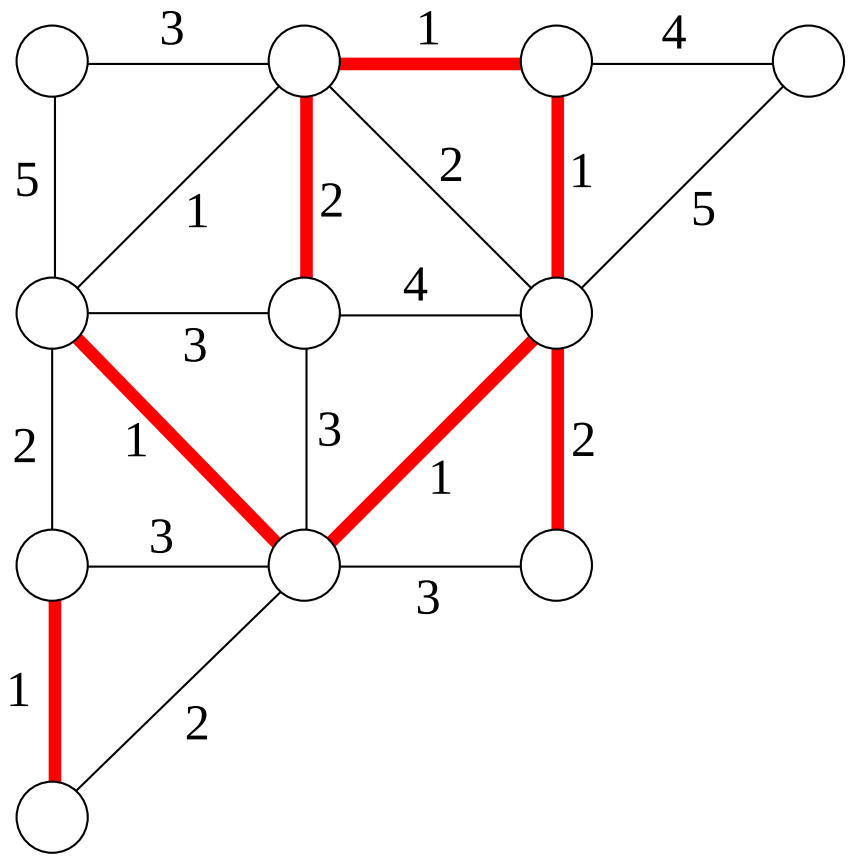
Beispiel (Kruskal):



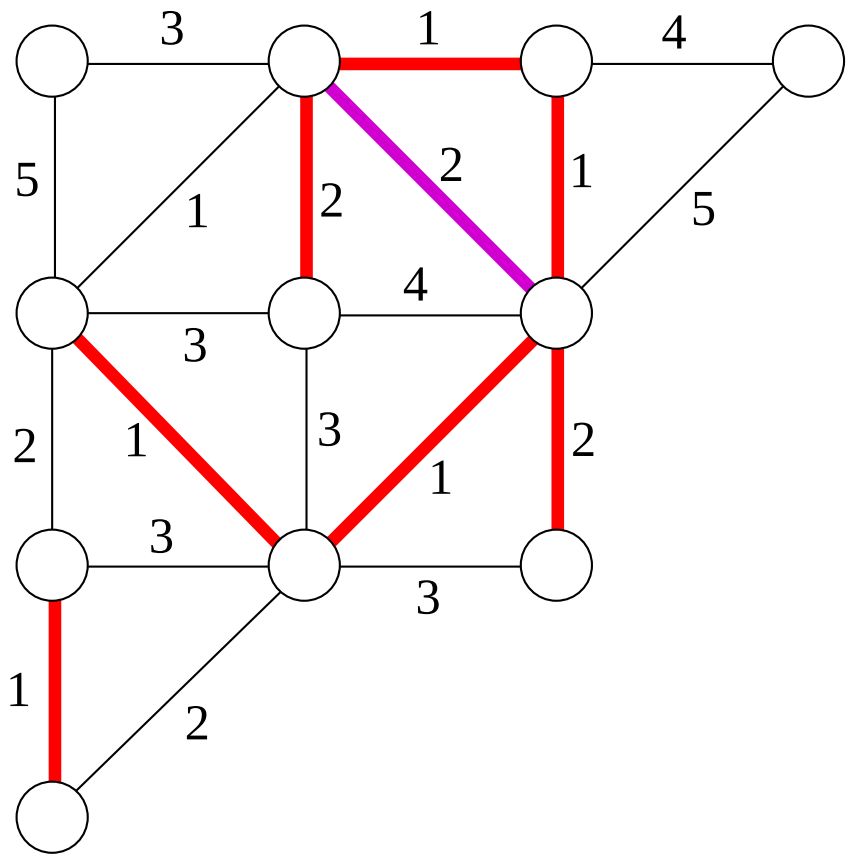
Beispiel (Kruskal):



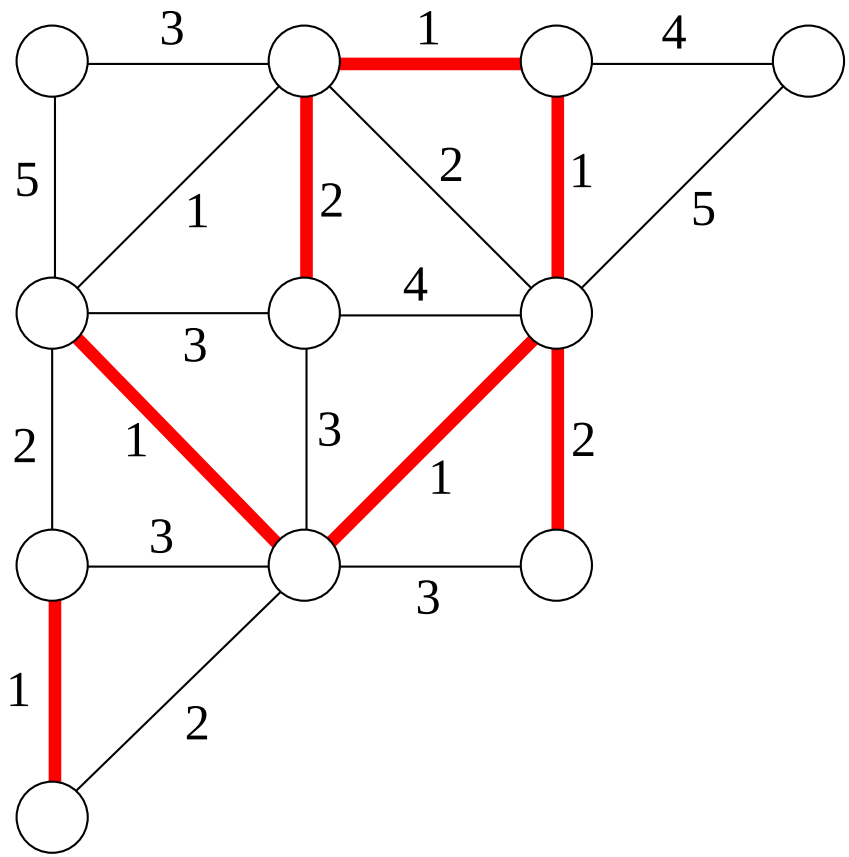
Beispiel (Kruskal):



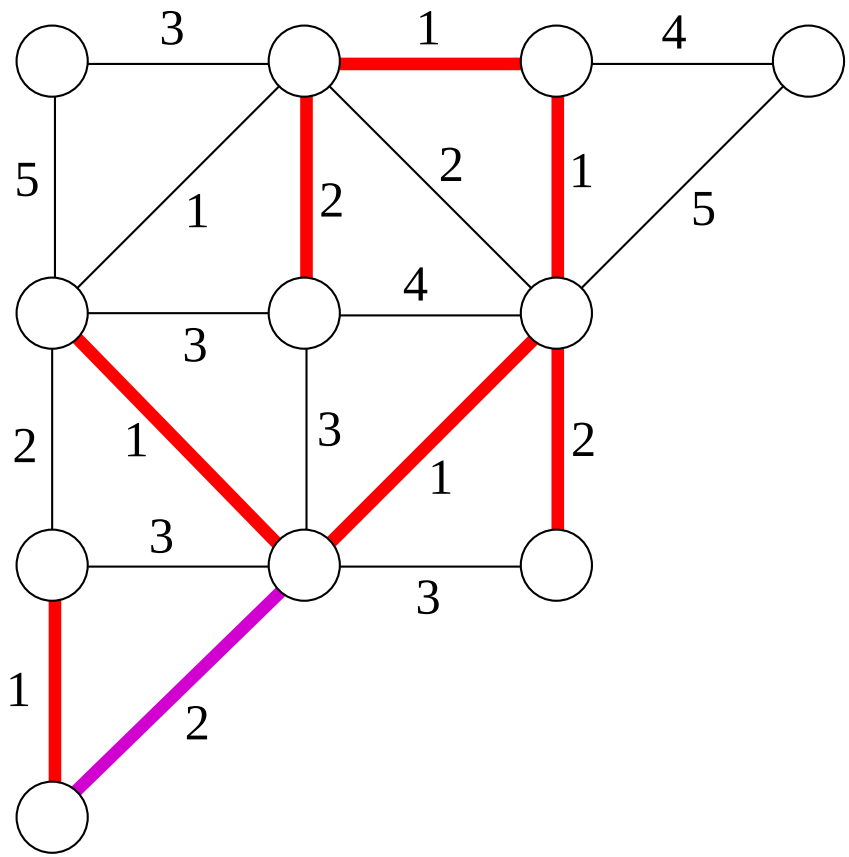
Beispiel (Kruskal):



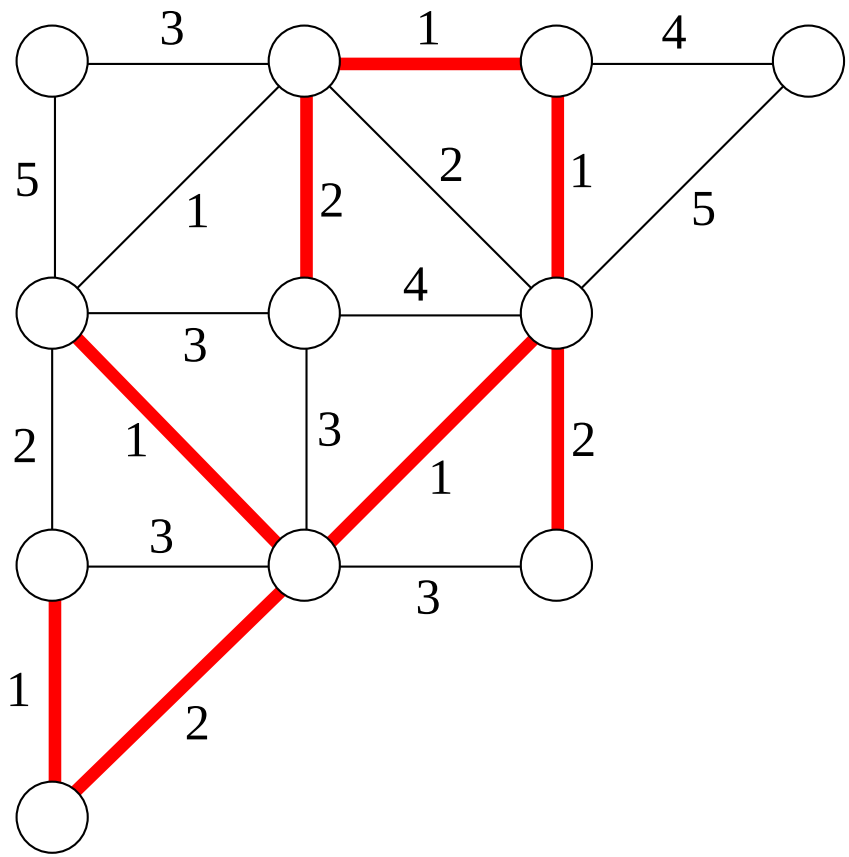
Beispiel (Kruskal):



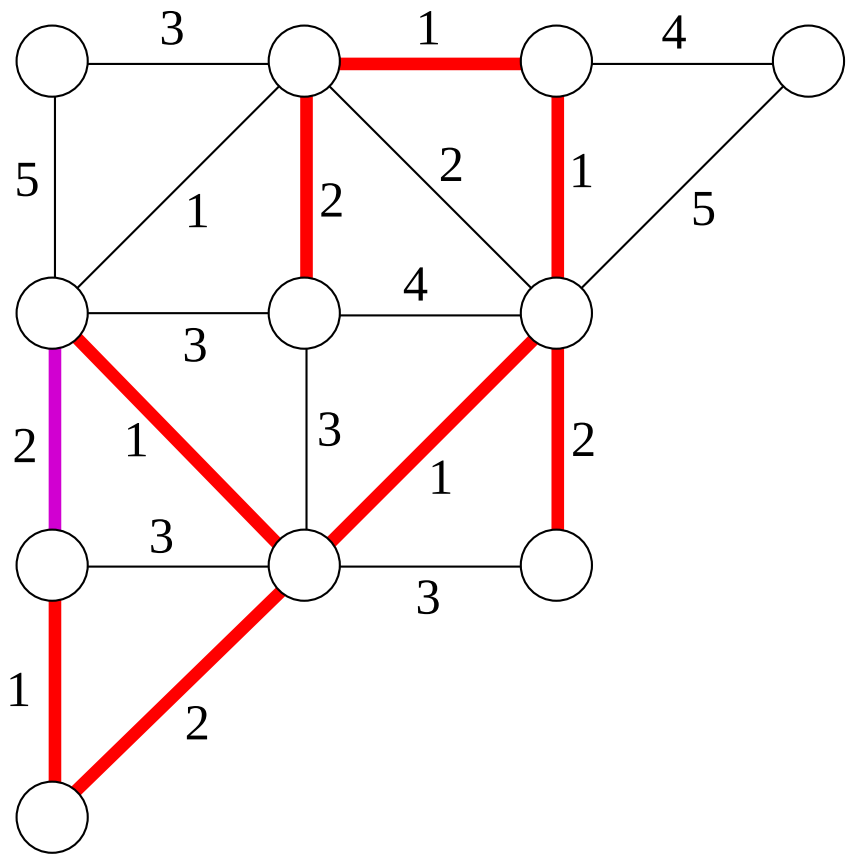
Beispiel (Kruskal):



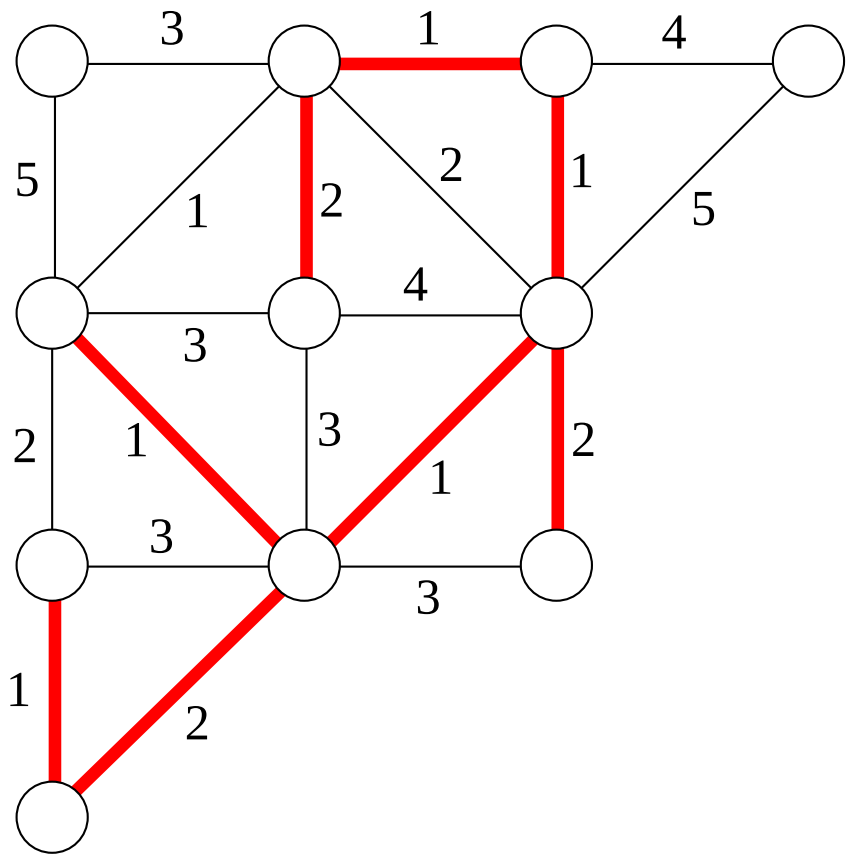
Beispiel (Kruskal):



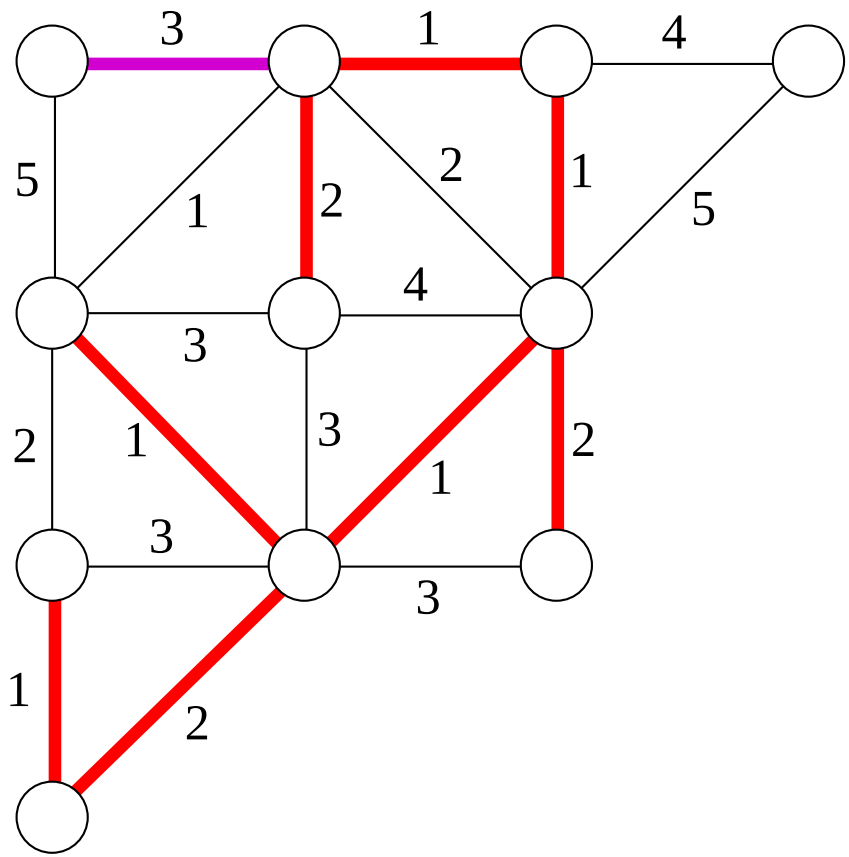
Beispiel (Kruskal):



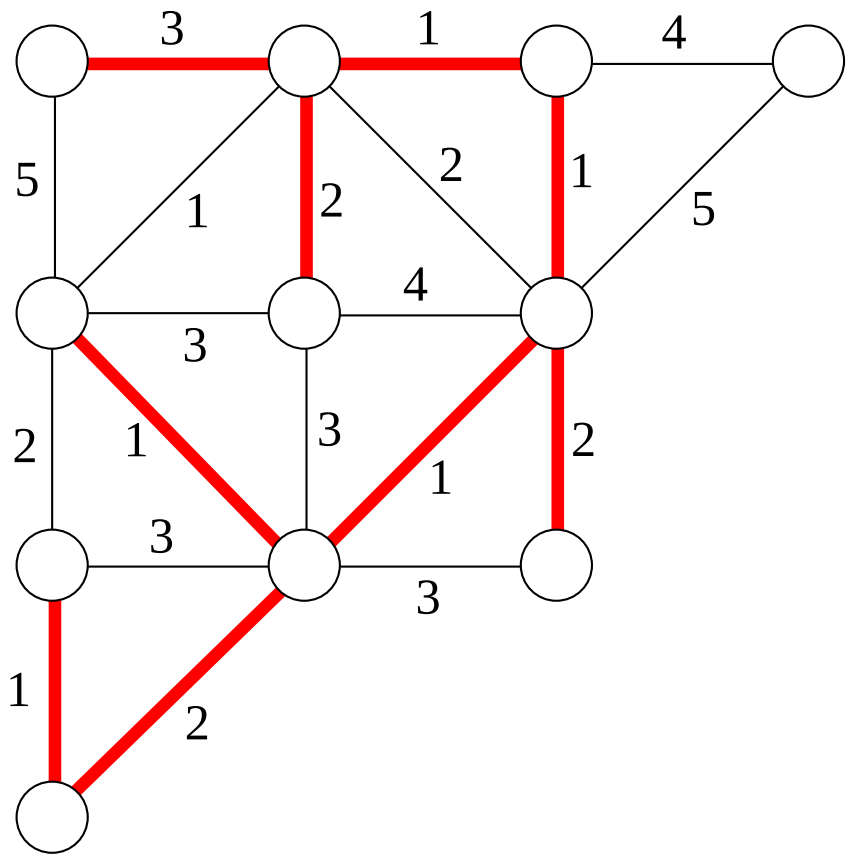
Beispiel (Kruskal):



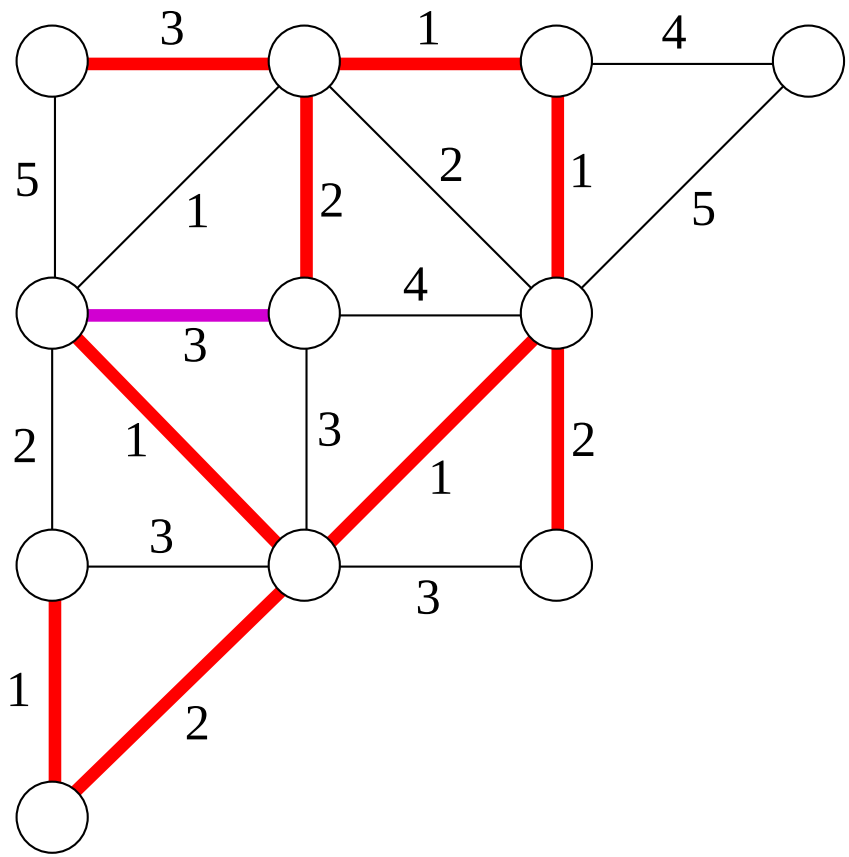
Beispiel (Kruskal):



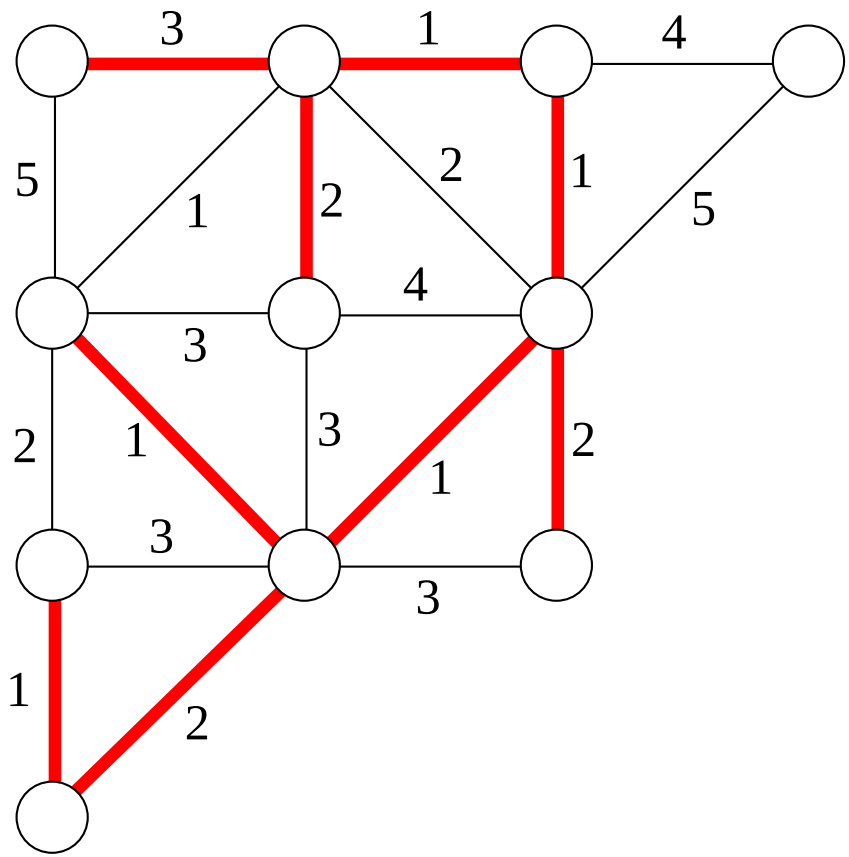
Beispiel (Kruskal):



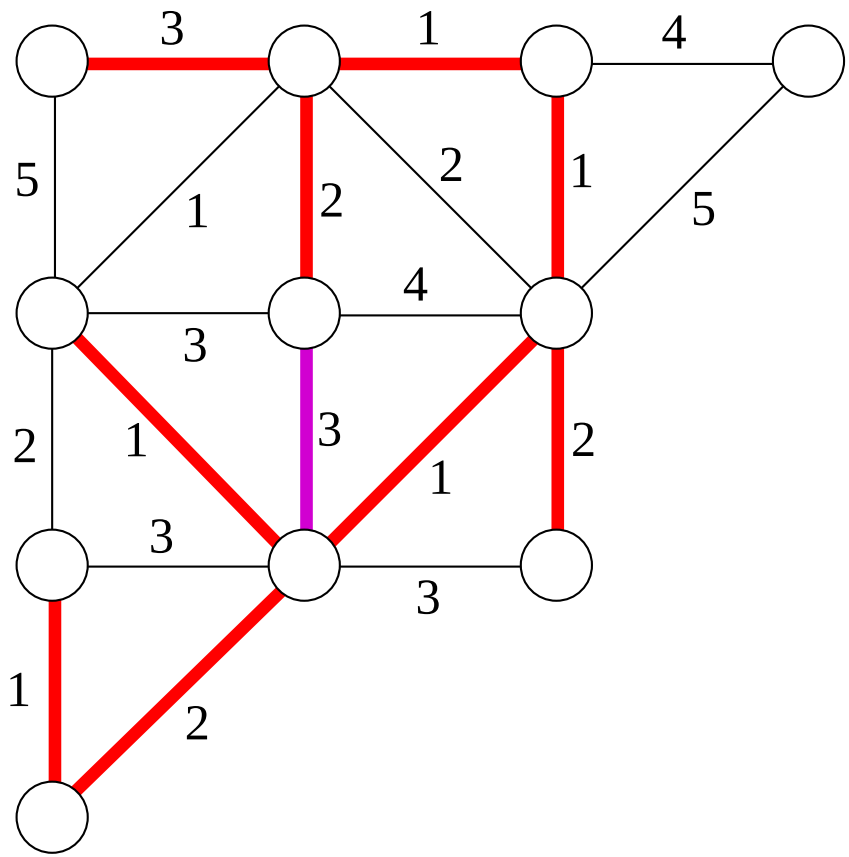
Beispiel (Kruskal):



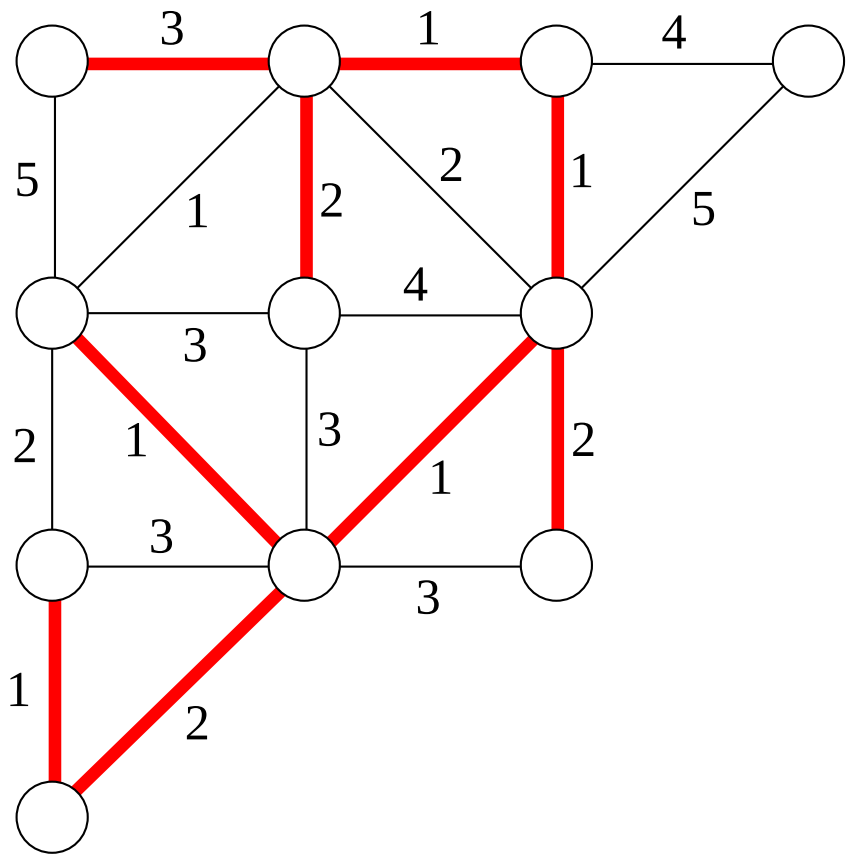
Beispiel (Kruskal):



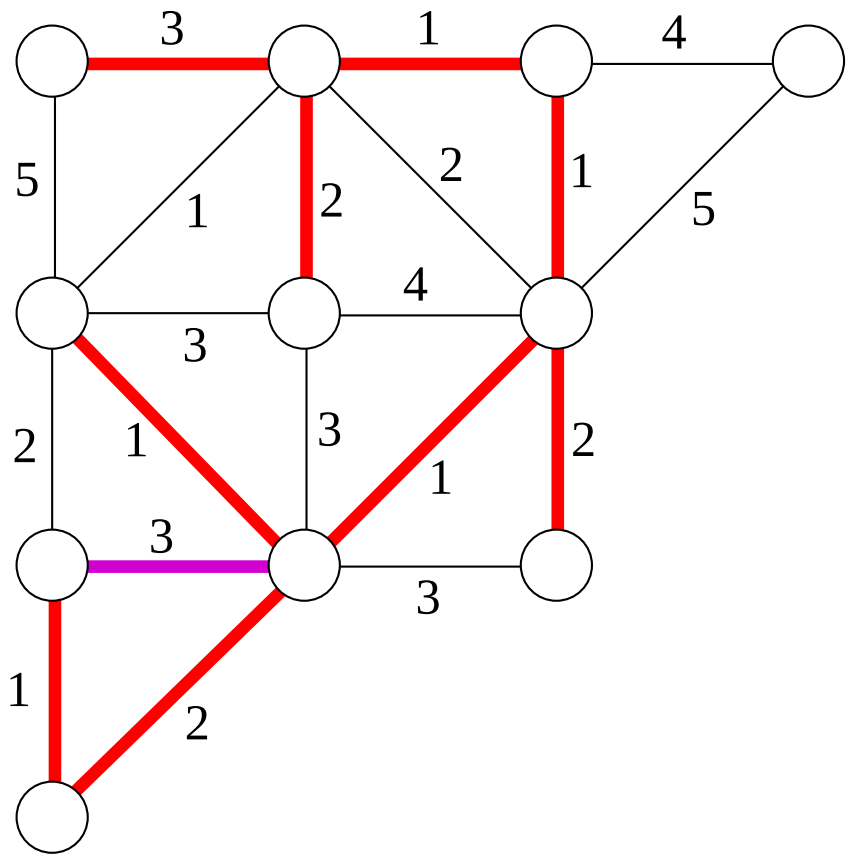
Beispiel (Kruskal):



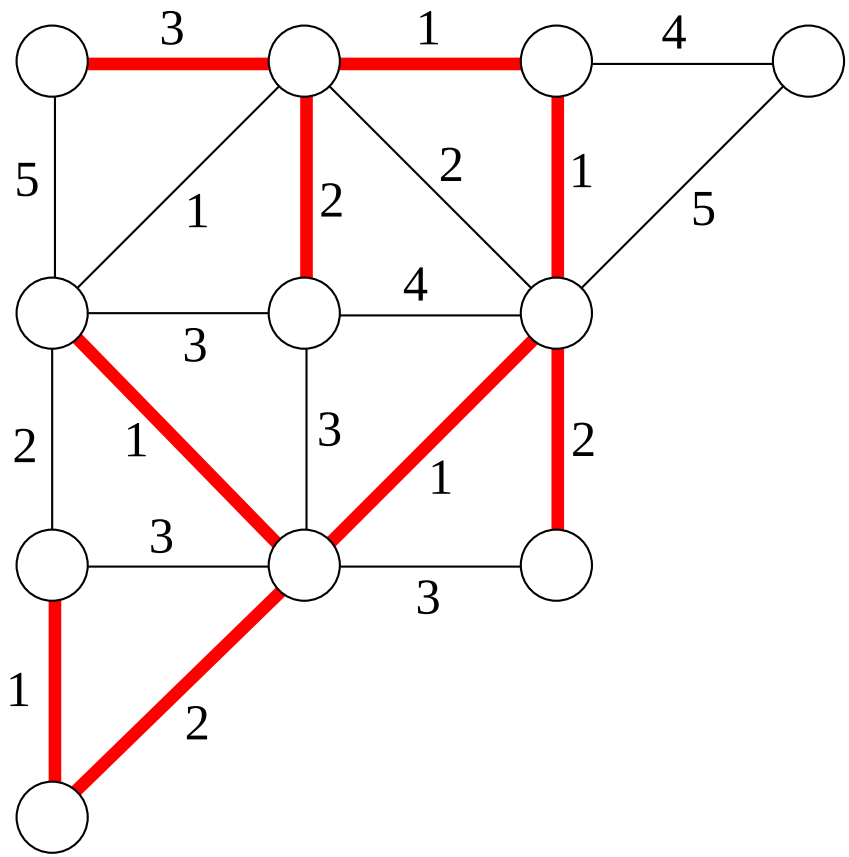
Beispiel (Kruskal):



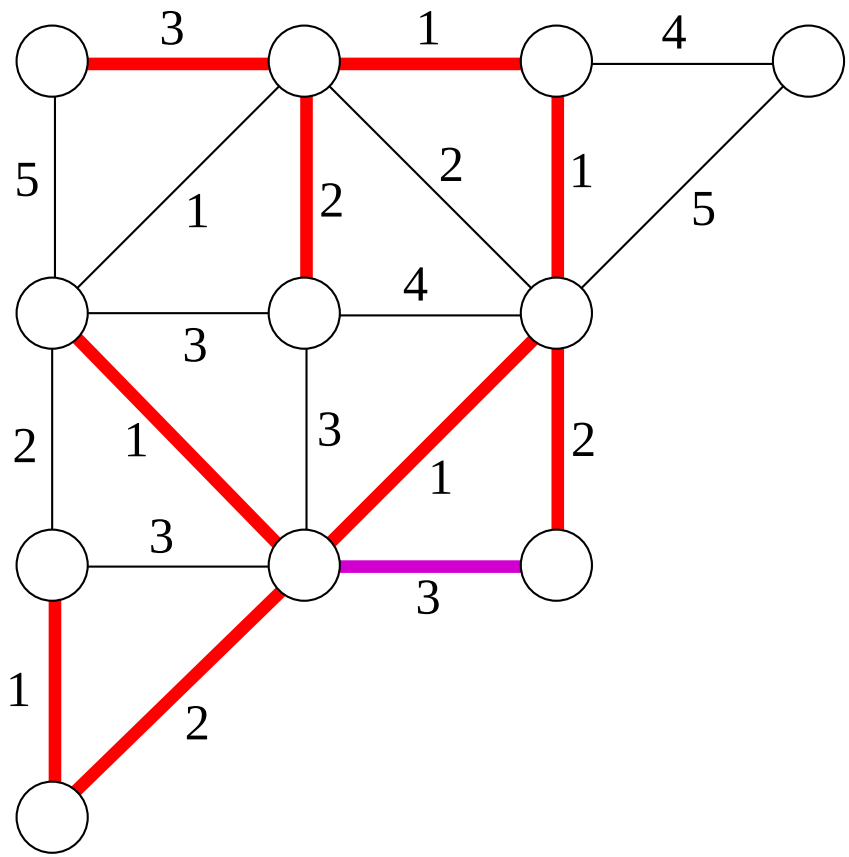
Beispiel (Kruskal):



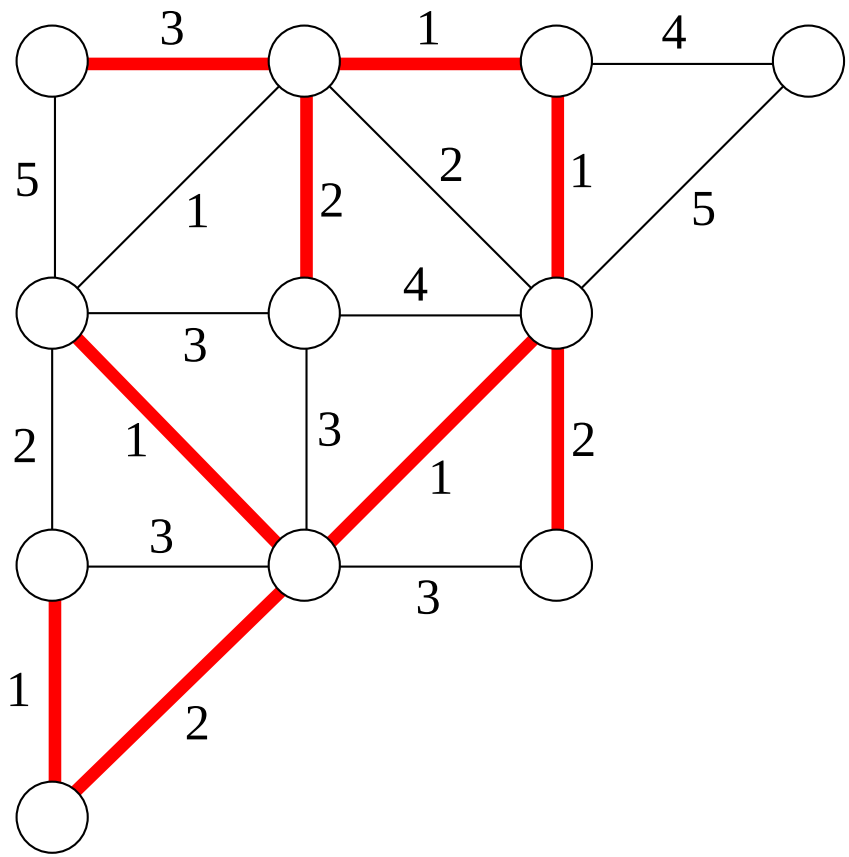
Beispiel (Kruskal):



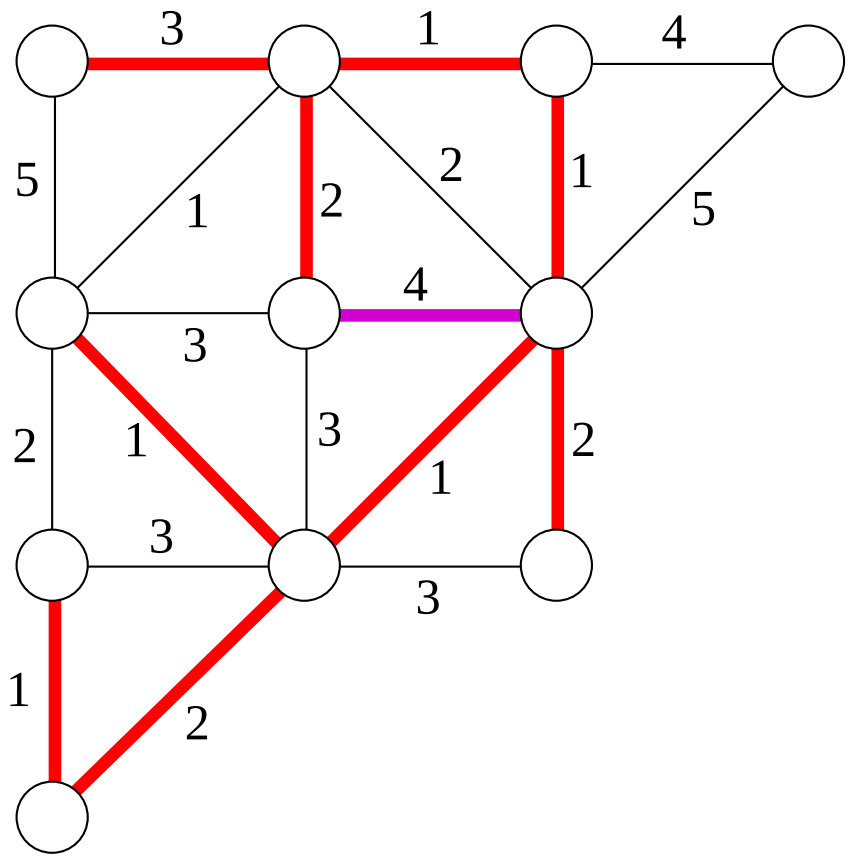
Beispiel (Kruskal):



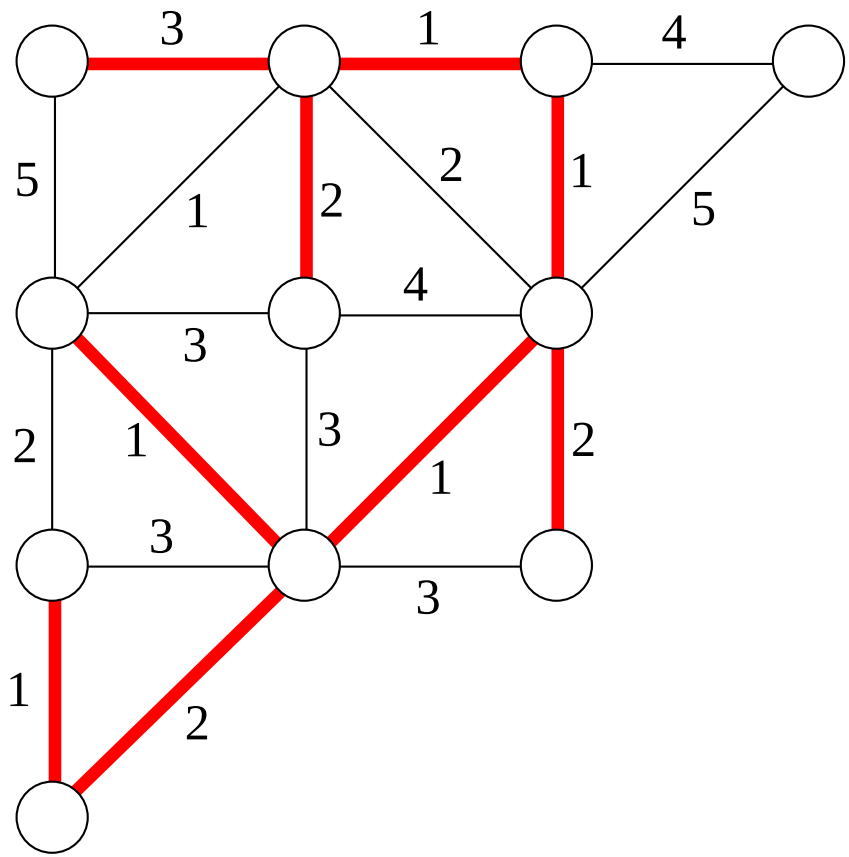
Beispiel (Kruskal):



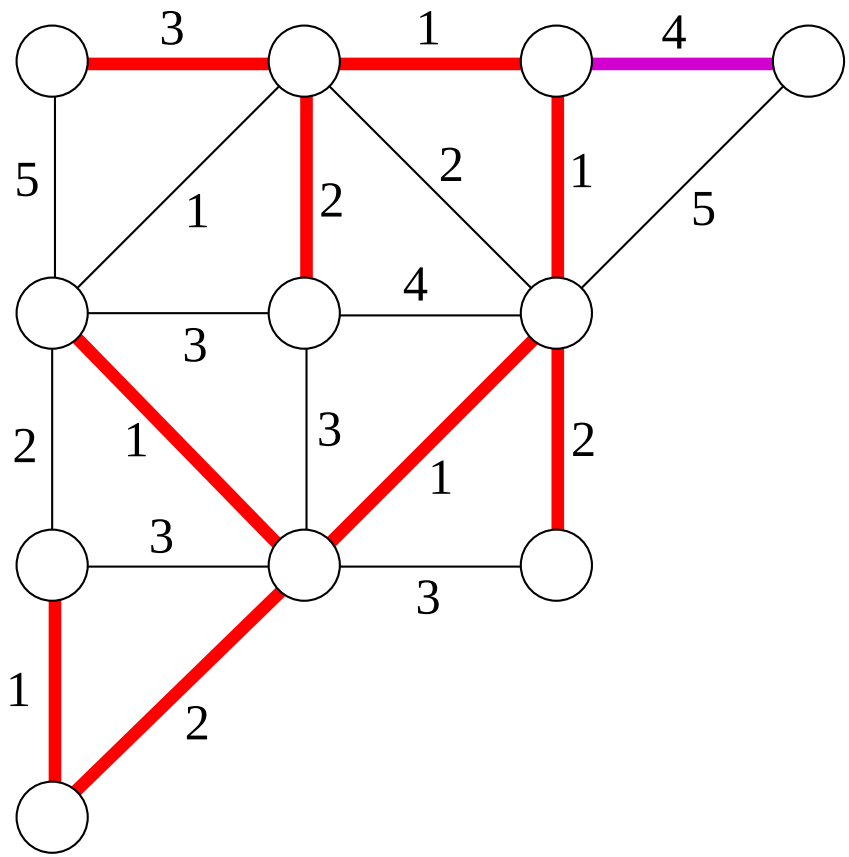
Beispiel (Kruskal):



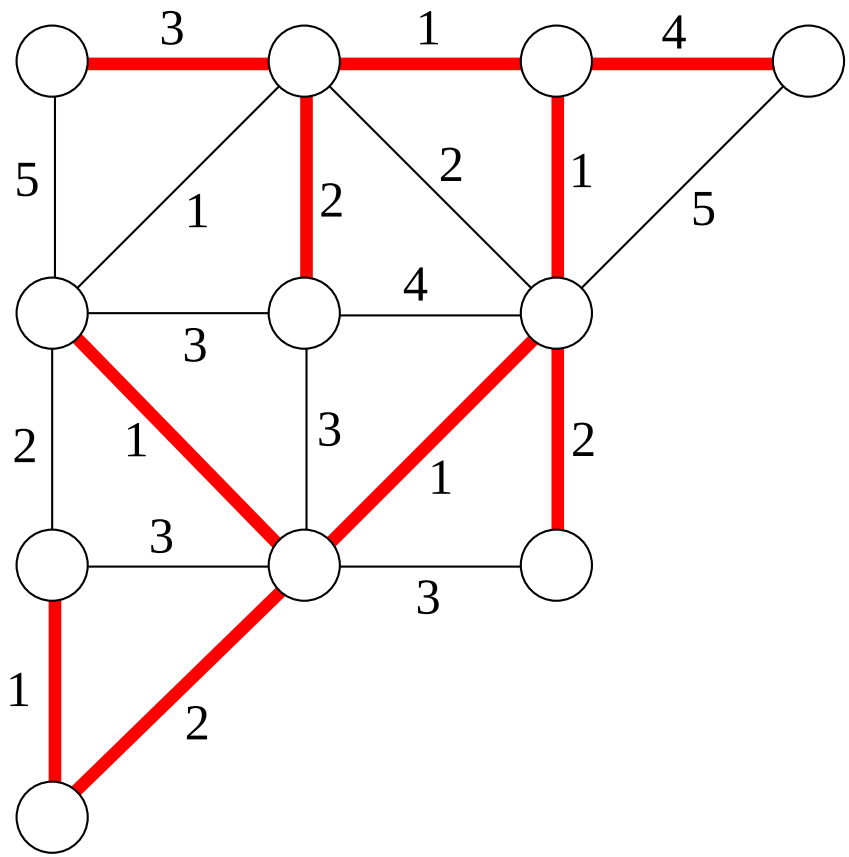
Beispiel (Kruskal):



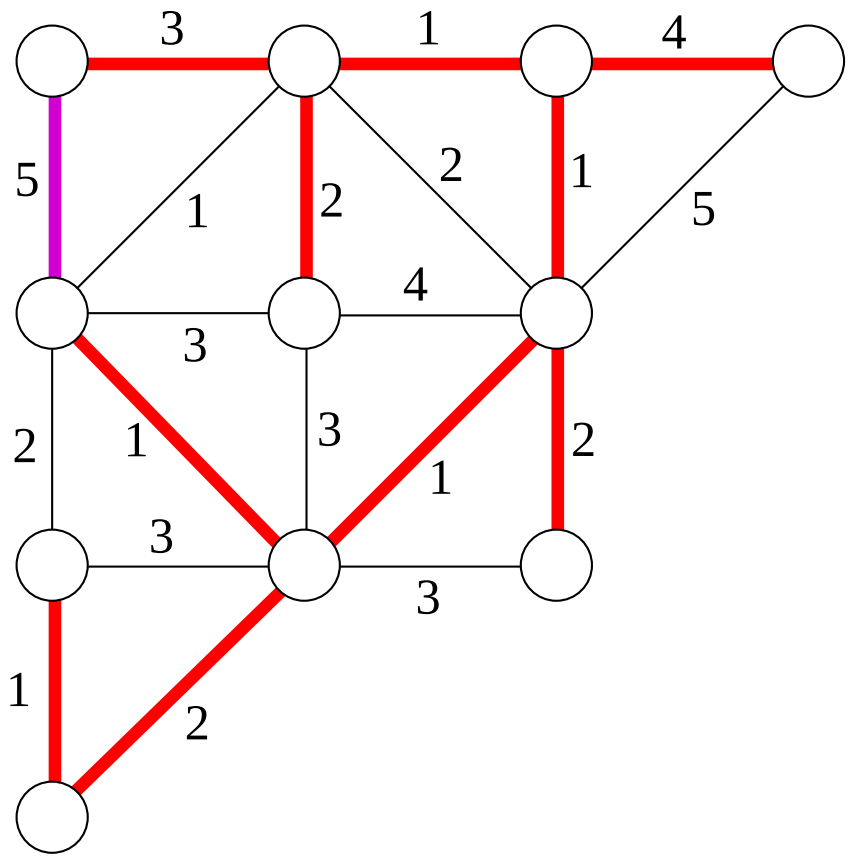
Beispiel (Kruskal):



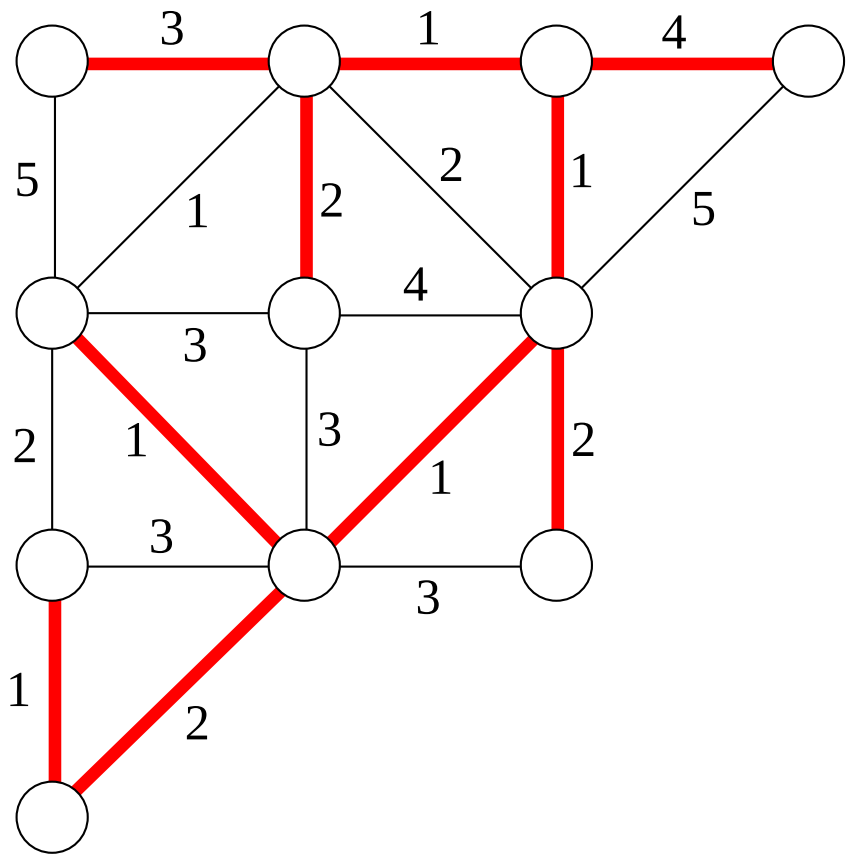
Beispiel (Kruskal):



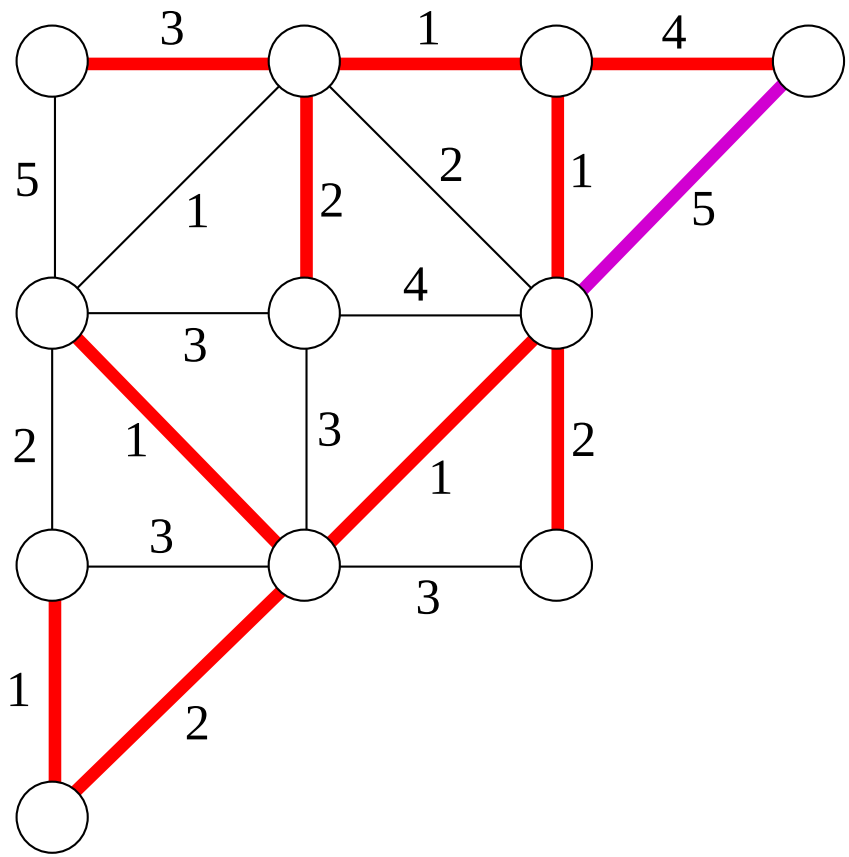
Beispiel (Kruskal):



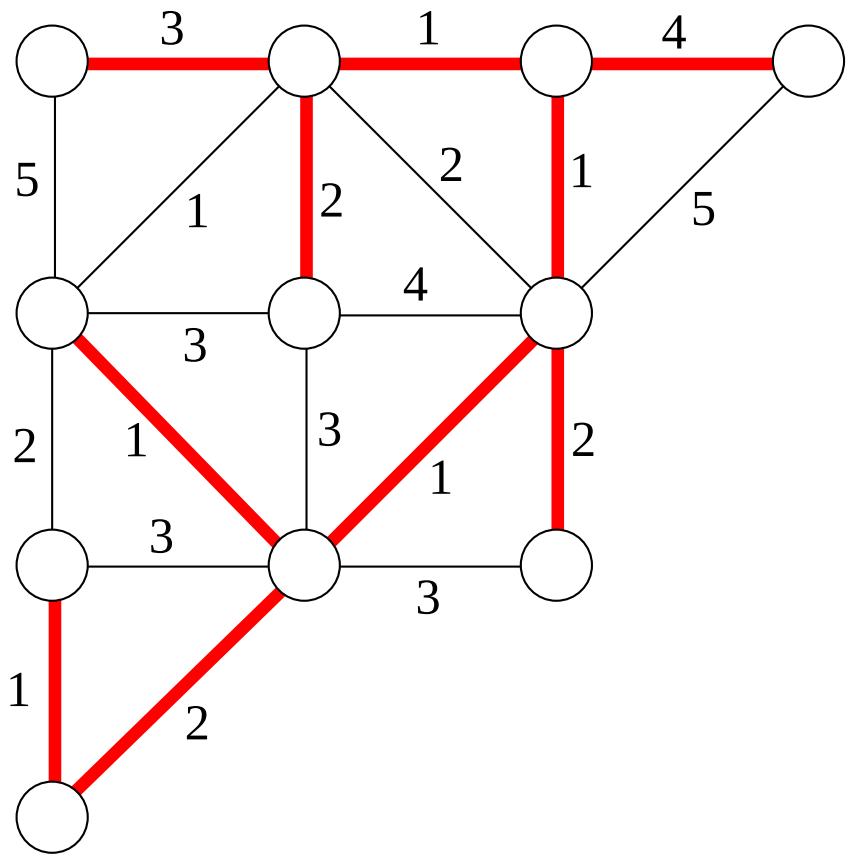
Beispiel (Kruskal):



Beispiel (Kruskal):



Beispiel (Kruskal):



Uns interessieren: 1) Korrektheit; 2) Rechenzeit (später).

Uns interessieren: 1) Korrektheit; 2) Rechenzeit (später).

Korrektheit des Algorithmus von Kruskal:

R_i sei die Kantenmenge, die nach Runde i, der Bearbeitung von e_i, in `R` steht.

Uns interessieren: 1) Korrektheit; 2) Rechenzeit (später).

Korrektheit des Algorithmus von Kruskal:

R_i sei die Kantenmenge, die nach Runde i, der Bearbeitung von e_i, in `R` steht.

Zu zeigen ist: R_m ist ein minimaler Spannbaum für G.

Uns interessieren: 1) Korrektheit; 2) Rechenzeit (später).

Korrektheit des Algorithmus von Kruskal:

R_i sei die Kantenmenge, die nach Runde i, der Bearbeitung von e_i, in `R` steht.

Zu zeigen ist: R_m ist ein minimaler Spannbaum für G.

Erinnerung: $R \subseteq E$ heißt **erweiterbar**, wenn $R \subseteq T$ gilt, für einen MST T.

Uns interessieren: 1) Korrektheit; 2) Rechenzeit (später).

Korrektheit des Algorithmus von Kruskal:

R_i sei die Kantenmenge, die nach Runde i, der Bearbeitung von e_i, in `R` steht.

Zu zeigen ist: R_m ist ein minimaler Spannbaum für G.

Erinnerung: $R \subseteq E$ heißt **erweiterbar**, wenn $R \subseteq T$ gilt, für einen MST T.

Ind.-Behauptung IB(i), $0 \leq i \leq m$: R_i ist erweiterbar.

Uns interessieren: 1) Korrektheit; 2) Rechenzeit (später).

Korrektheit des Algorithmus von Kruskal:

R_i sei die Kantenmenge, die nach Runde i, der Bearbeitung von e_i, in `R` steht.

Zu zeigen ist: R_m ist ein minimaler Spannbaum für G.

Erinnerung: $R \subseteq E$ heißt **erweiterbar**, wenn $R \subseteq T$ gilt, für einen MST T.

Ind.-Behauptung IB(i), $0 \leq i \leq m$: R_i ist erweiterbar. (Beweis per Induktion über i.)

Uns interessieren: 1) Korrektheit; 2) Rechenzeit (später).

Korrektheit des Algorithmus von Kruskal:

R_i sei die Kantenmenge, die nach Runde i, der Bearbeitung von e_i, in `R` steht.

Zu zeigen ist: R_m ist ein minimaler Spannbaum für G.

Erinnerung: $R \subseteq E$ heißt **erweiterbar**, wenn $R \subseteq T$ gilt, für einen MST T.

Ind.-Behauptung IB(i), $0 \leq i \leq m$: R_i ist erweiterbar. (Beweis per Induktion über i.)

Dann besagt IB(m), dass $R_m \subseteq T$ ist, für einen MST T.

Uns interessieren: 1) Korrektheit; 2) Rechenzeit (später).

Korrektheit des Algorithmus von Kruskal:

R_i sei die Kantenmenge, die nach Runde i, der Bearbeitung von e_i, in `R` steht.

Zu zeigen ist: R_m ist ein minimaler Spannbaum für G.

Erinnerung: $R \subseteq E$ heißt **erweiterbar**, wenn $R \subseteq T$ gilt, für einen MST T.

Ind.-Behauptung IB(i), $0 \leq i \leq m$: R_i ist erweiterbar. (Beweis per Induktion über i.)

Dann besagt IB(m), dass $R_m \subseteq T$ ist, für einen MST T.

Beh.: Es gilt auch $T \subseteq R_m$. (Also gilt Gleichheit, und R_m ist ein MST.)

Uns interessieren: 1) Korrektheit; 2) Rechenzeit (später).

Korrektheit des Algorithmus von Kruskal:

R_i sei die Kantenmenge, die nach Runde i, der Bearbeitung von e_i, in `R` steht.

Zu zeigen ist: R_m ist ein minimaler Spannbaum für G.

Erinnerung: $R \subseteq E$ heißt **erweiterbar**, wenn $R \subseteq T$ gilt, für einen MST T.

Ind.-Behauptung IB(i), $0 \leq i \leq m$: R_i ist erweiterbar. (Beweis per Induktion über i.)

Dann besagt IB(m), dass $R_m \subseteq T$ ist, für einen MST T.

Beh.: Es gilt auch $T \subseteq R_m$. (Also gilt Gleichheit, und R_m ist ein MST.)

(*Beweis* der Beh.: Sei $e \in T$.

Uns interessieren: 1) Korrektheit; 2) Rechenzeit (später).

Korrektheit des Algorithmus von Kruskal:

R_i sei die Kantenmenge, die nach Runde i, der Bearbeitung von e_i, in `R` steht.

Zu zeigen ist: R_m ist ein minimaler Spannbaum für G.

Erinnerung: $R \subseteq E$ heißt **erweiterbar**, wenn $R \subseteq T$ gilt, für einen MST T.

Ind.-Behauptung IB(i), $0 \leq i \leq m$**:** R_i ist erweiterbar. (Beweis per Induktion über i.)

Dann besagt IB(m), dass $R_m \subseteq T$ ist, für einen MST T.

Beh.: Es gilt auch $T \subseteq R_m$. (Also gilt Gleichheit, und R_m ist ein MST.)

(*Beweis* der Beh.: Sei $e \in T$. Dann ist $e = e_i$ für ein i, und e_i wurde in Runde i getestet.

Uns interessieren: 1) Korrektheit; 2) Rechenzeit (später).

Korrektheit des Algorithmus von Kruskal:

R_i sei die Kantenmenge, die nach Runde i, der Bearbeitung von e_i, in `R` steht.

Zu zeigen ist: R_m ist ein minimaler Spannbaum für G.

Erinnerung: $R \subseteq E$ heißt **erweiterbar**, wenn $R \subseteq T$ gilt, für einen MST T.

Ind.-Behauptung IB(i), $0 \leq i \leq m$: R_i ist erweiterbar. (Beweis per Induktion über i.)

Dann besagt IB(m), dass $R_m \subseteq T$ ist, für einen MST T.

Beh.: Es gilt auch $T \subseteq R_m$. (Also gilt Gleichheit, und R_m ist ein MST.)

(*Beweis* der Beh.: Sei $e \in T$. Dann ist $e = e_i$ für ein i, und e_i wurde in Runde i getestet. Weil $R_{i-1} \subseteq R_m \subseteq T$ und $e_i \in T$, ist $R_{i-1} \cup \{e_i\} \subseteq T$,

Uns interessieren: 1) Korrektheit; 2) Rechenzeit (später).

Korrektheit des Algorithmus von Kruskal:

R_i sei die Kantenmenge, die nach Runde i, der Bearbeitung von e_i, in `R` steht.

Zu zeigen ist: R_m ist ein minimaler Spannbaum für G.

Erinnerung: $R \subseteq E$ heißt **erweiterbar**, wenn $R \subseteq T$ gilt, für einen MST T.

Ind.-Behauptung IB(i), $0 \leq i \leq m$**:** R_i ist erweiterbar. (Beweis per Induktion über i.)

Dann besagt IB(m), dass $R_m \subseteq T$ ist, für einen MST T.

Beh.: Es gilt auch $T \subseteq R_m$. (Also gilt Gleichheit, und R_m ist ein MST.)

(*Beweis* der Beh.: Sei $e \in T$. Dann ist $e = e_i$ für ein i, und e_i wurde in Runde i getestet. Weil $R_{i-1} \subseteq R_m \subseteq T$ und $e_i \in T$, ist $R_{i-1} \cup \{e_i\} \subseteq T$, also ist $R_{i-1} \cup \{e_i\}$ kreisfrei,

Uns interessieren: 1) Korrektheit; 2) Rechenzeit (später).

Korrektheit des Algorithmus von Kruskal:

R_i sei die Kantenmenge, die nach Runde i, der Bearbeitung von e_i, in R steht.

Zu zeigen ist: R_m ist ein minimaler Spannbaum für G.

Erinnerung: $R \subseteq E$ heißt **erweiterbar**, wenn $R \subseteq T$ gilt, für einen MST T.

Ind.-Behauptung IB(i), $0 \leq i \leq m$**:** R_i ist erweiterbar. (Beweis per Induktion über i.)

Dann besagt IB(m), dass $R_m \subseteq T$ ist, für einen MST T.

Beh.: Es gilt auch $T \subseteq R_m$. (Also gilt Gleichheit, und R_m ist ein MST.)

(*Beweis* der Beh.: Sei $e \in T$. Dann ist $e = e_i$ für ein i, und e_i wurde in Runde i getestet. Weil $R_{i-1} \subseteq R_m \subseteq T$ und $e_i \in T$, ist $R_{i-1} \cup \{e_i\} \subseteq T$, also ist $R_{i-1} \cup \{e_i\}$ kreisfrei, also fügt der Algorithmus die Kante e_i in Runde i zu R hinzu,

Uns interessieren: 1) Korrektheit; 2) Rechenzeit (später).

Korrektheit des Algorithmus von Kruskal:

R_i sei die Kantenmenge, die nach Runde i, der Bearbeitung von e_i, in R steht.

Zu zeigen ist: R_m ist ein minimaler Spannbaum für G.

Erinnerung: $R \subseteq E$ heißt **erweiterbar**, wenn $R \subseteq T$ gilt, für einen MST T.

Ind.-Behauptung IB(i), $0 \leq i \leq m$**:** R_i ist erweiterbar. (Beweis per Induktion über i.)

Dann besagt IB(m), dass $R_m \subseteq T$ ist, für einen MST T.

Beh.: Es gilt auch $T \subseteq R_m$. (Also gilt Gleichheit, und R_m ist ein MST.)

(*Beweis* der Beh.: Sei $e \in T$. Dann ist $e = e_i$ für ein i, und e_i wurde in Runde i getestet. Weil $R_{i-1} \subseteq R_m \subseteq T$ und $e_i \in T$, ist $R_{i-1} \cup \{e_i\} \subseteq T$, also ist $R_{i-1} \cup \{e_i\}$ kreisfrei, also fügt der Algorithmus die Kante e_i in Runde i zu R hinzu, also gilt $e_i \in R_m$.)

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei. – Sei $e_i = (v, w)$.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei. – Sei $e_i = (v, w)$. Definiere

$S := \{u \in V \mid$ im Wald (V, R_{i-1}) ist u von v aus erreichbar$\}$.

(S ist die Zusammenhangskomponente von v in (V, R_{i-1}).)

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei. – Sei $e_i = (v, w)$. Definiere
$S := \{u \in V \mid \text{im Wald } (V, R_{i-1}) \text{ ist } u \text{ von } v \text{ aus erreichbar}\}$.
(S ist die Zusammenhangskomponente von v in (V, R_{i-1}).)
Weil $R_{i-1} \cup \{(v, w)\}$ kreisfrei ist, folgt $w \in V - S$.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei. – Sei $e_i = (v, w)$. Definiere

$S := \{u \in V \mid$ im Wald (V, R_{i-1}) ist u von v aus erreichbar$\}$.

(S ist die Zusammenhangskomponente von v in (V, R_{i-1}).)

Weil $R_{i-1} \cup \{(v, w)\}$ kreisfrei ist, folgt $w \in V - S$.

Nach Definition von S ist klar, dass keine R_{i-1}-Kante S und $V - S$ verbindet.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei. – Sei $e_i = (v, w)$. Definiere

$S := \{u \in V \mid$ im Wald (V, R_{i-1}) ist u von v aus erreichbar$\}$.

(S ist die Zusammenhangskomponente von v in (V, R_{i-1}).)

Weil $R_{i-1} \cup \{(v, w)\}$ kreisfrei ist, folgt $w \in V - S$.

Nach Definition von S ist klar, dass keine R_{i-1}-Kante S und $V - S$ verbindet.

Betrachte nun eine beliebige Kante $e = e_j$, die zwischen S und $V - S$ verläuft.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei. – Sei $e_i = (v, w)$. Definiere
$S := \{u \in V \mid \text{im Wald } (V, R_{i-1}) \text{ ist } u \text{ von } v \text{ aus erreichbar}\}$.
(S ist die Zusammenhangskomponente von v in (V, R_{i-1}).)
Weil $R_{i-1} \cup \{(v, w)\}$ kreisfrei ist, folgt $w \in V - S$.
Nach Definition von S ist klar, dass keine R_{i-1}-Kante S und $V - S$ verbindet.
Betrachte nun eine beliebige Kante $e = e_j$, die zwischen S und $V - S$ verläuft. Wie eben gesagt, ist dann $e_j \notin R_{i-1}$.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei. – Sei $e_i = (v, w)$. Definiere
$S := \{u \in V \mid$ im Wald (V, R_{i-1}) ist u von v aus erreichbar$\}$.
(S ist die Zusammenhangskomponente von v in (V, R_{i-1}).)
Weil $R_{i-1} \cup \{(v, w)\}$ kreisfrei ist, folgt $w \in V - S$.
Nach Definition von S ist klar, dass keine R_{i-1}-Kante S und $V - S$ verbindet.
Betrachte nun eine beliebige Kante $e = e_j$, die zwischen S und $V - S$ verläuft. Wie eben gesagt, ist dann $e_j \notin R_{i-1}$. Es gilt sogar $j \geq i$.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei. – Sei $e_i = (v, w)$. Definiere
$S := \{u \in V \mid \text{im Wald } (V, R_{i-1}) \text{ ist } u \text{ von } v \text{ aus erreichbar}\}$.
(S ist die Zusammenhangskomponente von v in (V, R_{i-1}).)
Weil $R_{i-1} \cup \{(v, w)\}$ kreisfrei ist, folgt $w \in V - S$.
Nach Definition von S ist klar, dass keine R_{i-1}-Kante S und $V - S$ verbindet.
Betrachte nun eine beliebige Kante $e = e_j$, die zwischen S und $V - S$ verläuft. Wie eben gesagt, ist dann $e_j \notin R_{i-1}$. Es gilt sogar $j \geq i$. (e_j schließt mit R_{i-1} keinen Kreis, also erst recht nicht mit $R_{j-1} \subseteq R_{i-1}$. Wäre $j < i$, hätte der Algorithmus e_j in Runde j zu R hinzugefügt.)

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei. – Sei $e_i = (v, w)$. Definiere
$S := \{u \in V \mid \text{im Wald } (V, R_{i-1}) \text{ ist } u \text{ von } v \text{ aus erreichbar}\}$.
(S ist die Zusammenhangskomponente von v in (V, R_{i-1}).)
Weil $R_{i-1} \cup \{(v, w)\}$ kreisfrei ist, folgt $w \in V - S$.
Nach Definition von S ist klar, dass keine R_{i-1}-Kante S und $V - S$ verbindet.
Betrachte nun eine beliebige Kante $e = e_j$, die zwischen S und $V - S$ verläuft. Wie eben gesagt, ist dann $e_j \notin R_{i-1}$. Es gilt sogar $j \geq i$. (e_j schließt mit R_{i-1} keinen Kreis, also erst recht nicht mit $R_{j-1} \subseteq R_{i-1}$. Wäre $j < i$, hätte der Algorithmus e_j in Runde j zu R hinzugefügt.)
Weil wir die Kanten anfangs nach Gewicht sortiert haben, folgt $c(e_j) \geq c(e_i)$.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei. – Sei $e_i = (v, w)$. Definiere
$S := \{u \in V \mid \text{im Wald } (V, R_{i-1}) \text{ ist } u \text{ von } v \text{ aus erreichbar}\}$.
(S ist die Zusammenhangskomponente von v in (V, R_{i-1}).)
Weil $R_{i-1} \cup \{(v, w)\}$ kreisfrei ist, folgt $w \in V - S$.
Nach Definition von S ist klar, dass keine R_{i-1}-Kante S und $V - S$ verbindet.
Betrachte nun eine beliebige Kante $e = e_j$, die zwischen S und $V - S$ verläuft. Wie eben gesagt, ist dann $e_j \notin R_{i-1}$. Es gilt sogar $j \geq i$. (e_j schließt mit R_{i-1} keinen Kreis, also erst recht nicht mit $R_{j-1} \subseteq R_{i-1}$. Wäre $j < i$, hätte der Algorithmus e_j in Runde j zu R hinzugefügt.)
Weil wir die Kanten anfangs nach Gewicht sortiert haben, folgt $c(e_j) \geq c(e_i)$.
Also ist $c(e_i)$ minimal unter allen $c((v', w'))$ mit $(v', w') \in E$, $v' \in S$, $w' \in V - S$.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei. – Sei $e_i = (v, w)$. Definiere
$S := \{u \in V \mid \text{im Wald } (V, R_{i-1}) \text{ ist } u \text{ von } v \text{ aus erreichbar}\}$.
(S ist die Zusammenhangskomponente von v in (V, R_{i-1}).)
Weil $R_{i-1} \cup \{(v, w)\}$ kreisfrei ist, folgt $w \in V - S$.
Nach Definition von S ist klar, dass keine R_{i-1}-Kante S und $V - S$ verbindet.
Betrachte nun eine beliebige Kante $e = e_j$, die zwischen S und $V - S$ verläuft. Wie eben gesagt, ist dann $e_j \notin R_{i-1}$. Es gilt sogar $j \geq i$. (e_j schließt mit R_{i-1} keinen Kreis, also erst recht nicht mit $R_{j-1} \subseteq R_{i-1}$. Wäre $j < i$, hätte der Algorithmus e_j in Runde j zu R hinzugefügt.)
Weil wir die Kanten anfangs nach Gewicht sortiert haben, folgt $c(e_j) \geq c(e_i)$.
Also ist $c(e_i)$ minimal unter allen $c((v', w'))$ mit $(v', w') \in E$, $v' \in S$, $w' \in V - S$.
Nach der **Schnitteigenschaft** folgt: $R_i = R_{i-1} \cup \{e_i\}$ ist erweiterbar,

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei. – Sei $e_i = (v, w)$. Definiere
$S := \{u \in V \mid \text{im Wald } (V, R_{i-1}) \text{ ist } u \text{ von } v \text{ aus erreichbar}\}$.
(S ist die Zusammenhangskomponente von v in (V, R_{i-1}).)
Weil $R_{i-1} \cup \{(v, w)\}$ kreisfrei ist, folgt $w \in V - S$.
Nach Definition von S ist klar, dass keine R_{i-1}-Kante S und $V - S$ verbindet.
Betrachte nun eine beliebige Kante $e = e_j$, die zwischen S und $V - S$ verläuft. Wie eben gesagt, ist dann $e_j \notin R_{i-1}$. Es gilt sogar $j \geq i$. (e_j schließt mit R_{i-1} keinen Kreis, also erst recht nicht mit $R_{j-1} \subseteq R_{i-1}$. Wäre $j < i$, hätte der Algorithmus e_j in Runde j zu R hinzugefügt.)
Weil wir die Kanten anfangs nach Gewicht sortiert haben, folgt $c(e_j) \geq c(e_i)$.
Also ist $c(e_i)$ minimal unter allen $c((v', w'))$ mit $(v', w') \in E$, $v' \in S$, $w' \in V - S$.
Nach der **Schnitteigenschaft** folgt: $R_i = R_{i-1} \cup \{e_i\}$ ist erweiterbar, d. h. die Induktionsbehauptung.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei. – Sei $e_i = (v, w)$. Definiere
$S := \{u \in V \mid \text{im Wald } (V, R_{i-1}) \text{ ist } u \text{ von } v \text{ aus erreichbar}\}$.
(S ist die Zusammenhangskomponente von v in (V, R_{i-1}).)
Weil $R_{i-1} \cup \{(v, w)\}$ kreisfrei ist, folgt $w \in V - S$.
Nach Definition von S ist klar, dass keine R_{i-1}-Kante S und $V - S$ verbindet.
Betrachte nun eine beliebige Kante $e = e_j$, die zwischen S und $V - S$ verläuft. Wie eben gesagt, ist dann $e_j \notin R_{i-1}$. Es gilt sogar $j \geq i$. (e_j schließt mit R_{i-1} keinen Kreis, also erst recht nicht mit $R_{j-1} \subseteq R_{i-1}$. Wäre $j < i$, hätte der Algorithmus e_j in Runde j zu R hinzugefügt.)
Weil wir die Kanten anfangs nach Gewicht sortiert haben, folgt $c(e_j) \geq c(e_i)$.
Also ist $c(e_i)$ minimal unter allen $c((v', w'))$ mit $(v', w') \in E$, $v' \in S$, $w' \in V - S$.
Nach der **Schnitteigenschaft** folgt: $R_i = R_{i-1} \cup \{e_i\}$ ist erweiterbar, d. h. die Induktionsbehauptung.

Damit ist der Korrektheitsbeweis für den Algorithmus von Kruskal vollständig. □

PAUSE

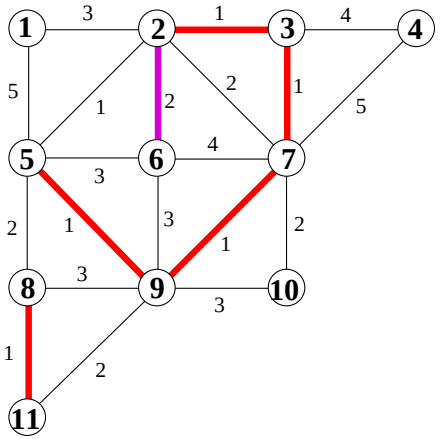
Es folgt: Hilfsstruktur Union-Find.

11.4 Hilfsstruktur: Union-Find

Union-Find-Datenstrukturen dienen als **Hilfsstruktur** für verschiedene Algorithmen, insbesondere für den Algorithmus von Kruskal.

Zwischenkonfiguration im Algorithmus von Kruskal: Menge $R_{i-1} \subseteq E$, die Wald bilden, und Folge $e_i, \ldots, e_m$ von noch zu verarbeitenden Kanten.

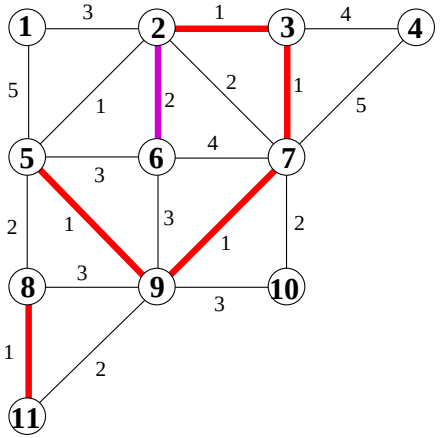
11.4.1 Algorithmus von Kruskal mit Union-Find



Die im Algorithmus von Kruskal zu lösende Aufgabe:
Zu zwei Knoten v und w entscheide, ob es in (V, R_{i-1}) einen Weg von v nach w gibt.

(Hier: zwischen den Endpunkten 2 und 6 der violetten Kante.)

11.4.1 Algorithmus von Kruskal mit Union-Find

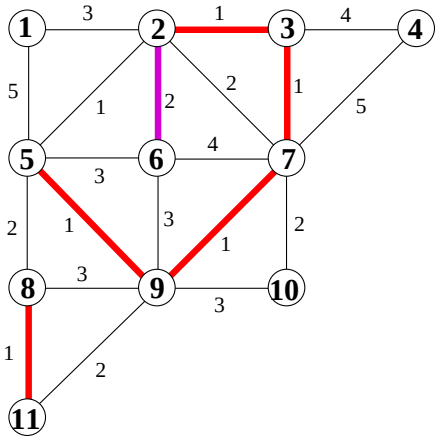


Die im Algorithmus von Kruskal zu lösende Aufgabe:
Zu zwei Knoten v und w entscheide, ob es in (V, R_{i-1}) einen Weg von v nach w gibt.

(Hier: zwischen den Endpunkten 2 und 6 der violetten Kante.)

Möglich, aber ungeschickt: Jedesmal Tiefensuche o. ä.

11.4.1 Algorithmus von Kruskal mit Union-Find



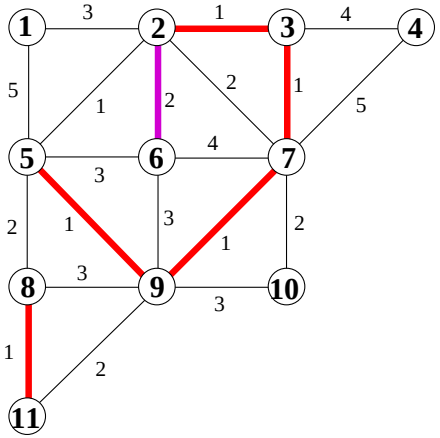
Die im Algorithmus von Kruskal zu lösende Aufgabe:
Zu zwei Knoten v und w entscheide, ob es in (V, R_{i-1}) einen Weg von v nach w gibt.

(Hier: zwischen den Endpunkten 2 und 6 der violetten Kante.)

Möglich, aber ungeschickt: Jedesmal Tiefensuche o. ä.

Ansatz: Repräsentiere die **Knotenmengen**, die den Zusammenhangskomponenten von $(V, \boldsymbol{R})$ entsprechen, in einer Datenstruktur. – Im Beispielbild: $\{1\}, \{2, 3, 5, 7, 9\}, \{4\}, \{6\}, \{8, 11\}, \{10\}$.

11.4.1 Algorithmus von Kruskal mit Union-Find



Die im Algorithmus von Kruskal zu lösende Aufgabe:
Zu zwei Knoten v und w entscheide, ob es in (V, R_{i-1}) einen Weg von v nach w gibt.

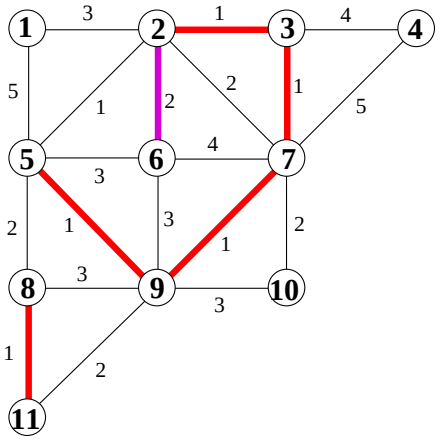
(Hier: zwischen den Endpunkten 2 und 6 der violetten Kante.)

Möglich, aber ungeschickt: Jedesmal Tiefensuche o. ä.

Ansatz: Repräsentiere die **Knotenmengen**, die den Zusammenhangskomponenten von $(V, \boldsymbol{R})$ entsprechen, in einer Datenstruktur. – Im Beispielbild: $\{1\}, \{2, 3, 5, 7, 9\}, \{4\}, \{6\}, \{8, 11\}, \{10\}$.

Es soll **schnell zu ermitteln** sein, ob **zwei Knoten in derselben Komponente/Menge liegen**.

11.4.1 Algorithmus von Kruskal mit Union-Find



Die im Algorithmus von Kruskal zu lösende Aufgabe:
Zu zwei Knoten v und w entscheide, ob es in (V, R_{i-1}) einen Weg von v nach w gibt.

(Hier: zwischen den Endpunkten 2 und 6 der violetten Kante.)

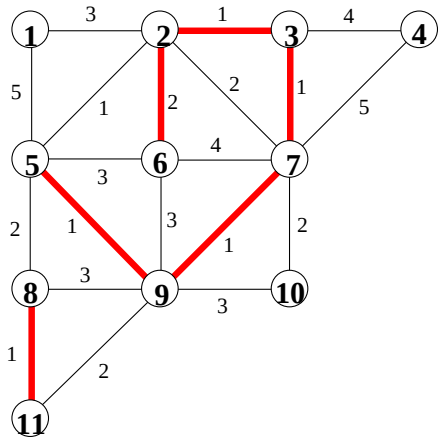
Möglich, aber ungeschickt: Jedesmal Tiefensuche o. ä.

Ansatz: Repräsentiere die **Knotenmengen**, die den Zusammenhangskomponenten von $(V, \boldsymbol{R})$ entsprechen, in einer Datenstruktur. – Im Beispielbild: $\{1\}, \{2, 3, 5, 7, 9\}, \{4\}, \{6\}, \{8, 11\}, \{10\}$.

Es soll **schnell zu ermitteln** sein, ob **zwei Knoten in derselben Komponente/Menge liegen**.

Wenn wir einen Schritt des Kruskal-Algorithmus ausführen, bei dem eine Kante akzeptiert, also in R aufgenommen wird, müssen wir zwei der disjunkten Mengen **vereinigen**.

11.4.1 Algorithmus von Kruskal mit Union-Find



Die im Algorithmus von Kruskal zu lösende Aufgabe:
Zu zwei Knoten v und w entscheide, ob es in (V, R_{i-1}) einen Weg von v nach w gibt.

(Hier: zwischen den Endpunkten 2 und 6 der violetten Kante.)

Möglich, aber ungeschickt: Jedesmal Tiefensuche o. ä.

Ansatz: Repräsentiere die **Knotenmengen**, die den Zusammenhangskomponenten von (V, R) entsprechen, in einer Datenstruktur. – Im Beispielbild: $\{1\}, \{2, 3, 5, 7, 9\}, \{4\}, \{6\}, \{8, 11\}, \{10\}$.

Es soll **schnell zu ermitteln** sein, ob **zwei Knoten in derselben Komponente/Menge liegen**.

Wenn wir einen Schritt des Kruskal-Algorithmus ausführen, bei dem eine Kante akzeptiert, also in R aufgenommen wird, müssen wir zwei der disjunkten Mengen **vereinigen**.

Neue Mengen: $\{1\}, \{2, 3, 5, 6, 7, 9\}, \{4\}, \{8, 11\}, \{10\}$.
Auch diese Operation sollte schnell durchführbar sein.

Eine **Partition*** von $\{1, 2, \ldots, n\}$ ist eine **Zerlegung** in Mengen (hier: „Klassen“)

$$\{1, 2, \ldots, n\} = S_1 \cup S_2 \cup \cdots \cup S_\ell,$$

wobei $S_1, S_2, \ldots, S_\ell$ **disjunkt** sind.

* In der Mathematik heißt die gesamte Aufteilung Partition, die Teile Klassen.
Bei der Speicheraufteilung in Computern bedeutet „Partition“ einen Teil. Nicht verwechseln!

Eine **Partition*** von $\{1, 2, \ldots, n\}$ ist eine **Zerlegung** in Mengen (hier: „Klassen“)

$$\{1, 2, \ldots, n\} = S_1 \cup S_2 \cup \cdots \cup S_\ell,$$

wobei $S_1, S_2, \ldots, S_\ell$ **disjunkt** sind.

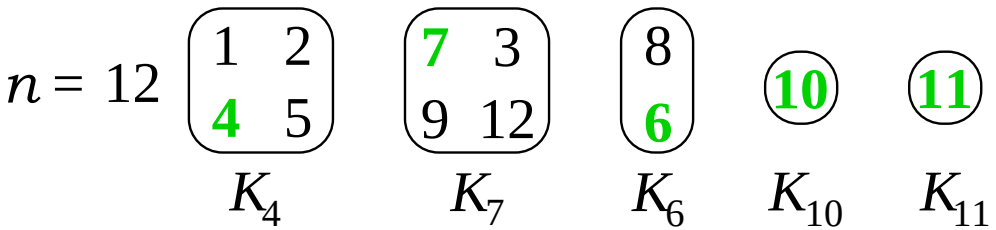
Abstrakte Aufgabe:

Verwalte eine **dynamische** (d.h. veränderliche) **Partition** der Menge $\{1, 2, \ldots, n\}$ unter Operationen

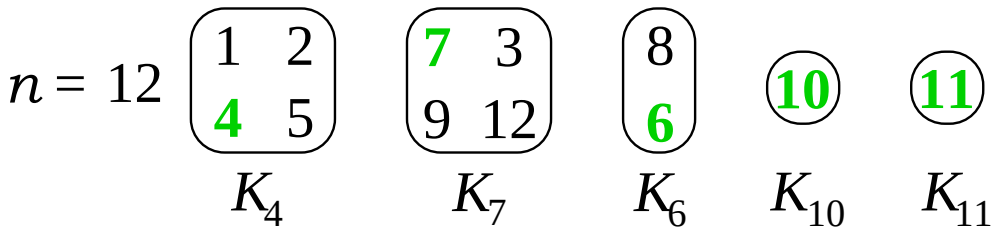
- *init* (Initialisierung)
- *union* (Vereinigung von Klassen der Partition)
- *find* („In welcher Klasse liegt i?“).

* In der Mathematik heißt die gesamte Aufteilung Partition, die Teile Klassen.
Bei der Speicheraufteilung in Computern bedeutet „Partition“ einen Teil. Nicht verwechseln!

Beispiel: $n = 12$.

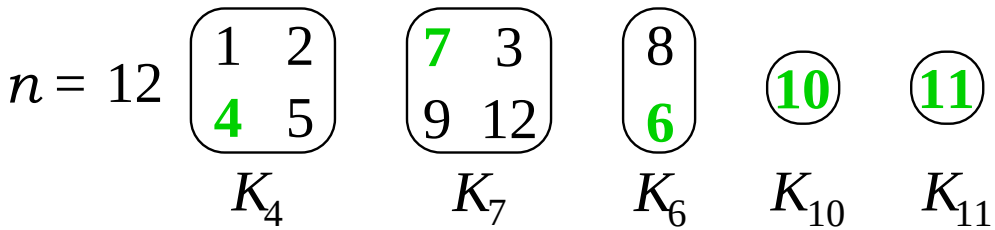


Beispiel: $n = 12$.



In jeder Klasse K der Partition ist ein **Repräsentant** $r \in K$ ausgezeichnet. Dieser fungiert als **Name** von K. Wir schreiben $\boldsymbol{K_r}$ für die Klasse mit Repräsentanten r.

Beispiel: $n = 12$.



In jeder Klasse K der Partition ist ein **Repräsentant** $r \in K$ ausgezeichnet. Dieser fungiert als **Name** von K. Wir schreiben $\boldsymbol{K_r}$ für die Klasse mit Repräsentanten r.

Kompakte Darstellung:
Funktion/Array r mit $r(i) =$ Repräsentant der Klasse von i:

i	1	2	3	4	5	6	7	8	9	10	11	12
$r(i)$	4	4	7	4	4	6	7	6	7	10	11	7

Operationen:

init(n): Erzeugt zu $n \geq 1$ die „diskrete Partition“ mit den n einelementigen Klassen $\{1\}, \{2\}, \ldots, \{n\}$, also $K_i = \{i\}$.

Operationen:

init(n): Erzeugt zu $n \geq 1$ die „diskrete Partition“ mit den n einelementigen Klassen $\{1\}, \{2\}, \ldots, \{n\}$, also $K_i = \{i\}$.

find(i): Gibt zu $i \in \{1, \ldots, n\}$ den Namen $r(i)$ der Klasse $K_{r(i)}$ aus, in der sich i (gegenwärtig) befindet.

Operationen:

init(n): Erzeugt zu $n \geq 1$ die „diskrete Partition“ mit den n einelementigen Klassen $\{1\}, \{2\}, \ldots, \{n\}$, also $K_i = \{i\}$.

find(i): Gibt zu $i \in \{1, \ldots, n\}$ den Namen $r(i)$ der Klasse $K_{r(i)}$ aus, in der sich i (gegenwärtig) befindet.

union(s, t): Die Argumente s und t müssen Repräsentanten **verschiedener Klassen** K_s bzw. K_t sein. Die Operation ersetzt in der Partition K_s und K_t durch die Vereinigung $K_s \cup K_t$.
Als Repräsentant von $K_s \cup K_t$ kann ein beliebiges Element verwendet werden. (Meistens: s oder t.)

Im Beispiel entfernt **union**$(4, 10)$ die Klassen $K_4 = \{1, 2, \mathbf{4}, 5\}$ und $K_{10} = \{\mathbf{10}\}$ und fügt $K'_{10} = \{1, 2, 4, 5, \mathbf{10}\}$ hinzu.

Im Beispiel entfernt **union**$(4, 10)$ die Klassen $K_4 = \{1, 2, \mathbf{4}, 5\}$ und $K_{10} = \{\mathbf{10}\}$ und fügt $K'_{10} = \{1, 2, 4, 5, \mathbf{10}\}$ hinzu.

Aus Sicht der r-Funktion entspricht diese Operation der Änderung der Funktionswerte auf der Klasse K_s:

$$r'(i) := \begin{cases} t & , \text{ falls } r(i) = s, \\ r(i) & , \text{ sonst.} \end{cases}$$

Im Beispiel entfernt **union**$(4, 10)$ die Klassen $K_4 = \{1, 2, \mathbf{4}, 5\}$ und $K_{10} = \{\mathbf{10}\}$ und fügt $K'_{10} = \{1, 2, 4, 5, \mathbf{10}\}$ hinzu.

Aus Sicht der r-Funktion entspricht diese Operation der Änderung der Funktionswerte auf der Klasse K_s:

$$r'(i) := \begin{cases} t & , \text{ falls } r(i) = s, \\ r(i) & , \text{ sonst.} \end{cases}$$

Im Beispiel: Die r-Werte in K_4 werden auf 10 geändert, Rest bleibt gleich.

i	1	2	3	4	5	6	7	8	9	10	11	12
$r'(i)$	**10**	**10**	7	**10**	**10**	6	7	6	7	10	11	7

Im Beispiel entfernt **union**$(4, 10)$ die Klassen $K_4 = \{1, 2, \mathbf{4}, 5\}$ und $K_{10} = \{\mathbf{10}\}$ und fügt $K'_{10} = \{1, 2, 4, 5, \mathbf{10}\}$ hinzu.

Aus Sicht der r-Funktion entspricht diese Operation der Änderung der Funktionswerte auf der Klasse K_s:

$$r'(i) := \begin{cases} t & , \text{ falls } r(i) = s, \\ r(i) & , \text{ sonst.} \end{cases}$$

Im Beispiel: Die r-Werte in K_4 werden auf 10 geändert, Rest bleibt gleich.

i	1	2	3	4	5	6	7	8	9	10	11	12
$r'(i)$	**10**	**10**	7	**10**	**10**	6	7	6	7	10	11	7

Zwei Implementierungsmöglichkeiten:

Im Beispiel entfernt **union**$(4, 10)$ die Klassen $K_4 = \{1, 2, \mathbf{4}, 5\}$ und $K_{10} = \{\mathbf{10}\}$ und fügt $K'_{10} = \{1, 2, 4, 5, \mathbf{10}\}$ hinzu.

Aus Sicht der r-Funktion entspricht diese Operation der Änderung der Funktionswerte auf der Klasse K_s:

$$r'(i) := \begin{cases} t & , \text{ falls } r(i) = s, \\ r(i) & , \text{ sonst.} \end{cases}$$

Im Beispiel: Die r-Werte in K_4 werden auf 10 geändert, Rest bleibt gleich.

i	1	2	3	4	5	6	7	8	9	10	11	12
$r'(i)$	**10**	**10**	7	**10**	**10**	6	7	6	7	10	11	7

Zwei Implementierungsmöglichkeiten:

1) **Arrays (mit Listen)** (erlaubt schnelle **find**s)

Im Beispiel entfernt **union**$(4, 10)$ die Klassen $K_4 = \{1, 2, \mathbf{4}, 5\}$ und $K_{10} = \{\mathbf{10}\}$ und fügt $K'_{10} = \{1, 2, 4, 5, \mathbf{10}\}$ hinzu.

Aus Sicht der r-Funktion entspricht diese Operation der Änderung der Funktionswerte auf der Klasse K_s:

$$r'(i) := \begin{cases} t & , \text{ falls } r(i) = s, \\ r(i) & , \text{ sonst.} \end{cases}$$

Im Beispiel: Die r-Werte in K_4 werden auf 10 geändert, Rest bleibt gleich.

i	1	2	3	4	5	6	7	8	9	10	11	12
$r'(i)$	**10**	**10**	7	**10**	**10**	6	7	6	7	10	11	7

Zwei Implementierungsmöglichkeiten:

1) **Arrays (mit Listen)** (erlaubt schnelle **find**s)
2) **Bäume** (erlaubt schnelle **union**s)

Algorithmus von Kruskal mit Union-Find

Input: Gewichteter zusammenhängender Graph $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$.

1. Schritt:

Sortiere Kanten $e_1, \ldots, e_m$ nach Gewichten $c_1 = c(e_1), \ldots, c_m = c(e_m)$ aufsteigend.

Resultat: Kantenliste $(v_1, w_1, c_1), \ldots, (v_m, w_m, c_m)$, $c_1 \leq \cdots \leq c_m$.

Algorithmus von Kruskal mit Union-Find

Input: Gewichteter zusammenhängender Graph $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$.

1. Schritt:
Sortiere Kanten $e_1, \ldots, e_m$ nach Gewichten $c_1 = c(e_1), \ldots, c_m = c(e_m)$ aufsteigend.

Resultat: Kantenliste $(v_1, w_1, c_1), \ldots, (v_m, w_m, c_m)$, $c_1 \leq \cdots \leq c_m$.

2. Schritt: Initialisiere Union-Find-Struktur für $\{1, \ldots, n\}$.

Algorithmus von Kruskal mit Union-Find

Input: Gewichteter zusammenhängender Graph $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$.

1. Schritt:
Sortiere Kanten $e_1, \ldots, e_m$ nach Gewichten $c_1 = c(e_1), \ldots, c_m = c(e_m)$ aufsteigend.

Resultat: Kantenliste $(v_1, w_1, c_1), \ldots, (v_m, w_m, c_m)$, $c_1 \leq \cdots \leq c_m$.

2. Schritt: Initialisiere Union-Find-Struktur für $\{1, \ldots, n\}$.

3. Schritt: Für $i = 1, 2, \ldots, m$ tue folgendes:

$\mathtt{s} \leftarrow$ **find**(v_i); $\mathtt{t} \leftarrow$ **find**(w_i);
if $\mathtt{s} \neq \mathtt{t}$ **then begin** $\mathtt{R} \leftarrow \mathtt{R} \cup \{e_i\}$; **union**$(\mathtt{s}, \mathtt{t})$ **end**;
// Optional: Beende Schleife, wenn $|\mathtt{R}| = n - 1$.

Algorithmus von Kruskal mit Union-Find

Input: Gewichteter zusammenhängender Graph $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$.

1. Schritt:
Sortiere Kanten $e_1, \ldots, e_m$ nach Gewichten $c_1 = c(e_1), \ldots, c_m = c(e_m)$ aufsteigend.

Resultat: Kantenliste $(v_1, w_1, c_1), \ldots, (v_m, w_m, c_m)$, $c_1 \leq \cdots \leq c_m$.

2. Schritt: Initialisiere Union-Find-Struktur für $\{1, \ldots, n\}$.

3. Schritt: Für $i = 1, 2, \ldots, m$ tue folgendes:

$\mathtt{s} \leftarrow$ **find**(v_i); $\mathtt{t} \leftarrow$ **find**(w_i);
if $\mathtt{s} \neq \mathtt{t}$ **then begin** $\mathtt{R} \leftarrow \mathtt{R} \cup \{e_i\}$; **union**$(\mathtt{s}, \mathtt{t})$ **end**;
// Optional: Beende Schleife, wenn $|\mathtt{R}| = n - 1$.

4. Schritt: Die Ausgabe ist R.

Algorithmus von Kruskal mit Union-Find

Input: Gewichteter zusammenhängender Graph $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$.

1. Schritt:
Sortiere Kanten $e_1, \ldots, e_m$ nach Gewichten $c_1 = c(e_1), \ldots, c_m = c(e_m)$ aufsteigend.

Resultat: Kantenliste $(v_1, w_1, c_1), \ldots, (v_m, w_m, c_m)$, $c_1 \leq \cdots \leq c_m$.

2. Schritt: Initialisiere Union-Find-Struktur für $\{1, \ldots, n\}$.

3. Schritt: Für $i = 1, 2, \ldots, m$ tue folgendes:

$\mathtt{s} \leftarrow$ **find**(v_i); $\mathtt{t} \leftarrow$ **find**(w_i);
if $\mathtt{s} \neq \mathtt{t}$ **then begin** $\mathtt{R} \leftarrow \mathtt{R} \cup \{e_i\}$; **union**$(\mathtt{s}, \mathtt{t})$ **end**;
// Optional: Beende Schleife, wenn $|\mathtt{R}| = n - 1$.

4. Schritt: Die Ausgabe ist R.

Bemerkung: Im Algorithmus kommt das Wort „Kreis“ nicht mehr vor.
Es wird vielmehr getestet, ob (v_i, w_i) zwei Zusammenhangskomponenten verbindet.

Satz 11.4.1

(a) Der Algorithmus von Kruskal in der Implementierung mit Union-Find ist korrekt.

Satz 11.4.1

(a) Der Algorithmus von Kruskal in der Implementierung mit Union-Find ist korrekt.

(b) Die Rechenzeit des Algorithmus ist $O(m \log n)$, wenn man die Union-Find-Datenstruktur mit **Arrays** implementiert.

Satz 11.4.1

(a) Der Algorithmus von Kruskal in der Implementierung mit Union-Find ist korrekt.

(b) Die Rechenzeit des Algorithmus ist $O(m \log n)$, wenn man die Union-Find-Datenstruktur mit **Arrays** implementiert.

(c) Die Rechenzeit des Algorithmus ist $O(m \log n)$, wenn man die Union-Find-Datenstruktur mit mit **wurzelgerichteten Bäumen** (mit **Pfadkompression**) implementiert.

Satz 11.4.1

(a) Der Algorithmus von Kruskal in der Implementierung mit Union-Find ist korrekt.

(b) Die Rechenzeit des Algorithmus ist $O(m \log n)$, wenn man die Union-Find-Datenstruktur mit **Arrays** implementiert.

(c) Die Rechenzeit des Algorithmus ist $O(m \log n)$, wenn man die Union-Find-Datenstruktur mit mit **wurzelgerichteten Bäumen** (mit **Pfadkompression**) implementiert.

Bem.: Teil (c) wird noch präzisiert.

Satz 11.4.1

(a) Der Algorithmus von Kruskal in der Implementierung mit Union-Find ist korrekt.

(b) Die Rechenzeit des Algorithmus ist $O(m \log n)$, wenn man die Union-Find-Datenstruktur mit **Arrays** implementiert.

(c) Die Rechenzeit des Algorithmus ist $O(m \log n)$, wenn man die Union-Find-Datenstruktur mit mit **wurzelgerichteten Bäumen** (mit **Pfadkompression**) implementiert.

Bem.: Teil (c) wird noch präzisiert.

Beweis: (a) Man zeigt durch Induktion über die Runden, dass nach Runde i die Klassen der Union-Find-Struktur genau die Zusammenhangskomponenten des Graphen (Waldes) (V, R_i) sind, für $R_i =$ Inhalt von `R` nach Runde i.

Daher testet „`s` $\leftarrow$ **find**(v_i); `t` $\leftarrow$ **find**(w_i); **if** `s` $\neq$ `t` . . . “ im 3. Schritt korrekt die Kreiseigenschaft.

(b), (c): Siehe unten.

11.4.2 Arrayimplementierung von Union-Find

11.4.2 Arrayimplementierung von Union-Find

Array `r`$[1..n]$ enthält in `r`$[i]$ den Repräsentanten $r(i)$ der Klasse $K_{r(i)}$, in der i **gegenwärtig** liegt. In unserem Beispiel sähe `r` folgendermaßen aus:

11.4.2 Arrayimplementierung von Union-Find

Array `r[1..n]` enthält in `r[i]` den Repräsentanten $r(i)$ der Klasse $K_{r(i)}$, in der i **gegenwärtig** liegt. In unserem Beispiel sähe `r` folgendermaßen aus:

	1	2	3	4	5	6	7	8	9	10	11	12
`r`:	4	4	7	4	4	6	7	6	7	10	11	7

11.4.2 Arrayimplementierung von Union-Find

Array `r[1..n]` enthält in `r[i]` den Repräsentanten $r(i)$ der Klasse $K_{r(i)}$, in der i **gegenwärtig** liegt. In unserem Beispiel sähe `r` folgendermaßen aus:

	1	2	3	4	5	6	7	8	9	10	11	12
`r`:	4	4	7	4	4	6	7	6	7	10	11	7

Offensichtlich ist **find**(i) in Zeit $O(1)$ ausführbar.

11.4.2 Arrayimplementierung von Union-Find

Array `r[1..n]` enthält in `r[i]` den Repräsentanten $r(i)$ der Klasse $K_{r(i)}$, in der i **gegenwärtig** liegt. In unserem Beispiel sähe `r` folgendermaßen aus:

	1	2	3	4	5	6	7	8	9	10	11	12
`r`:	4	4	7	4	4	6	7	6	7	10	11	7

Offensichtlich ist **find**(i) in Zeit $O(1)$ ausführbar.

Naiver Ansatz für **union**(s, t):

Durchlaufe das Array `r`; ändere dabei alle „s“ in „t“ (oder umgekehrt).

11.4.2 Arrayimplementierung von Union-Find

Array `r[1..n]` enthält in `r[i]` den Repräsentanten $r(i)$ der Klasse $K_{r(i)}$, in der i **gegenwärtig** liegt. In unserem Beispiel sähe `r` folgendermaßen aus:

	1	2	3	4	5	6	7	8	9	10	11	12
`r`:	4	4	7	4	4	6	7	6	7	10	11	7

Offensichtlich ist **find**(i) in Zeit $O(1)$ ausführbar.

Naiver Ansatz für **union**(s, t):

Durchlaufe das Array `r`; ändere dabei alle „s“ in „t“ (oder umgekehrt).

Kosten:

11.4.2 Arrayimplementierung von Union-Find

Array `r[1..n]` enthält in `r[i]` den Repräsentanten $r(i)$ der Klasse $K_{r(i)}$, in der i **gegenwärtig** liegt. In unserem Beispiel sähe `r` folgendermaßen aus:

	1	2	3	4	5	6	7	8	9	10	11	12
`r`:	4	4	7	4	4	6	7	6	7	10	11	7

Offensichtlich ist **find**(i) in Zeit $O(1)$ ausführbar.

Naiver Ansatz für **union**(s, t):

Durchlaufe das Array `r`; ändere dabei alle „s“ in „t“ (oder umgekehrt).

Kosten: $\Theta(n)$.

11.4.2 Arrayimplementierung von Union-Find

Array `r[1..n]` enthält in `r[i]` den Repräsentanten $r(i)$ der Klasse $K_{r(i)}$, in der i **gegenwärtig** liegt. In unserem Beispiel sähe `r` folgendermaßen aus:

	1	2	3	4	5	6	7	8	9	10	11	12
`r`:	4	4	7	4	4	6	7	6	7	10	11	7

Offensichtlich ist **find**(i) in Zeit $O(1)$ ausführbar.

Naiver Ansatz für **union**(s, t):

Durchlaufe das Array `r`; ändere dabei alle „s“ in „t“ (oder umgekehrt).

Kosten: $\Theta(n)$.

Dies kann man verbessern.

Trick 1:

Halte die Elemente jeder Klasse K_s (mit Repräsentant s) in einer linearen Liste L_s.

Trick 1:

Halte die Elemente jeder Klasse K_s (mit Repräsentant s) in einer linearen Liste L_s.

Dann muss man bei der Umbenennung von „s“ in „t“ nur L_s durchlaufen – die Kosten betragen $\Theta(|K_s|)$. Aus Listen L_s und L_t wird eine neue Liste L'_t.

Trick 1:

Halte die Elemente jeder Klasse K_s (mit Repräsentant s) in einer linearen Liste L_s.

Dann muss man bei der Umbenennung von „s“ in „t“ nur L_s durchlaufen – die Kosten betragen $\Theta(|K_s|)$. Aus Listen L_s und L_t wird eine neue Liste L'_t.

Trick 2:

Bei der Vereinigung von Klassen sollte man den Repräsentanten der **größeren** Klasse übernehmen und den der **kleineren** ändern, da damit die Zahl der Änderungen kleiner gehalten wird.

Trick 1:

Halte die Elemente jeder Klasse K_s (mit Repräsentant s) in einer linearen Liste L_s.

Dann muss man bei der Umbenennung von „s“ in „t“ nur L_s durchlaufen – die Kosten betragen $\Theta(|K_s|)$. Aus Listen L_s und L_t wird eine neue Liste L'_t.

Trick 2:

Bei der Vereinigung von Klassen sollte man den Repräsentanten der **größeren** Klasse übernehmen und den der **kleineren** ändern, da damit die Zahl der Änderungen kleiner gehalten wird. – Hierzu muss man die **Listenlängen/Klassengrößen** in einem zweiten Array `size`$[1..n]$ mitführen.

Im Beispiel:

	1	2	3	4	5	6	7	8	9	10	11	12
`r`:	4	4	7	4	4	6	7	6	7	10	11	7

Im Beispiel:

	1	2	3	4	5	6	7	8	9	10	11	12
`r`:	4	4	7	4	4	6	7	6	7	10	11	7

	1	2	3	4	5	6	7	8	9	10	11	12
`size`:	–	–	–	4	–	2	4	–	–	1	1	–

Im Beispiel:

	1	2	3	4	5	6	7	8	9	10	11	12
`r`:	4	4	7	4	4	6	7	6	7	10	11	7

	1	2	3	4	5	6	7	8	9	10	11	12
`size`:	–	–	–	4	–	2	4	–	–	1	1	–

Nur die Einträge `size`$[s]$ mit $r(s) = s$, also für Repräsentanten s, sind relevant. Die anderen Einträge (mit „$-$“ gekennzeichnet) sind unwesentlich.

Im Beispiel:

	1	2	3	4	5	6	7	8	9	10	11	12
`r`:	4	4	7	4	4	6	7	6	7	10	11	7

	1	2	3	4	5	6	7	8	9	10	11	12
`size`:	–	–	–	4	–	2	4	–	–	1	1	–

Nur die Einträge `size`$[s]$ mit $r(s) = s$, also für Repräsentanten s, sind relevant. Die anderen Einträge (mit „$-$“ gekennzeichnet) sind unwesentlich.

Listen: $L_4 = (4, 2, 1, 5)$, $L_7 = (7, 3, 12, 9)$, $L_6 = (6, 8)$, $L_{10} = (10)$, $L_{11} = (11)$.

Im Beispiel:

	1	2	3	4	5	6	7	8	9	10	11	12
`r`:	4	4	7	4	4	6	7	6	7	10	11	7

	1	2	3	4	5	6	7	8	9	10	11	12
`size`:	–	–	–	4	–	2	4	–	–	1	1	–

Nur die Einträge `size[`s`]` mit $r(s) = s$, also für Repräsentanten s, sind relevant. Die anderen Einträge (mit „$-$“ gekennzeichnet) sind unwesentlich.

Listen: $L_4 = (4, 2, 1, 5)$, $L_7 = (7, 3, 12, 9)$, $L_6 = (6, 8)$, $L_{10} = (10)$, $L_{11} = (11)$.

Für eine besonders sparsame Implementierung der Listen ist es nützlich, wenn L_s mit s beginnt.

Beispiel: **union(7, 6)**.

Weil `size[6]` $<$ `size[7]`, werden die Einträge `r[`i`]` für i in L_6 auf **7** geändert und $L_6 = (\mathbf{6}, \mathbf{8})$ wird unmittelbar nach dem ersten Element **7** in L_7 eingefügt:

	1	2	3	4	5	6	7	8	9	10	11	12
`r`:	4	4	7	4	4	7	7	7	7	10	11	7

	1	2	3	4	5	6	7	8	9	10	11	12
`size`:	–	–	–	4	–	–	6	–	–	1	1	–

Aus $L_7 = (7, 3, 12, 9)$ und $L_6 = (6, 8)$ wird die neue Liste $L_7 = (7, \mathbf{6}, \mathbf{8}, 3, 12, 9)$.

Zeitaufwand: $\Theta($Länge der kleineren Liste$)$ für die Umbenennungen im Array `r` und $O(1)$ für das Ändern des Eintrags `size`$[t]$ sowie das Kombinieren der Listen.

Zeitaufwand: $\Theta($Länge der kleineren Liste$)$ für die Umbenennungen im Array `r` und $O(1)$ für das Ändern des Eintrags `size`$[t]$ sowie das Kombinieren der Listen.

Zur Effizienzverbesserung (um konstanten Faktor) und zur Platzersparnis gibt es noch einen Trick.

Zeitaufwand: $\Theta($Länge der kleineren Liste$)$ für die Umbenennungen im Array `r` und $O(1)$ für das Ändern des Eintrags `size`$[t]$ sowie das Kombinieren der Listen.

Zur Effizienzverbesserung (um konstanten Faktor) und zur Platzersparnis gibt es noch einen Trick.
Da alle Listen zusammen stets die Einträge $1, 2, \ldots, n$ haben, brauchen wir keine dynamisch erzeugten Listenelemente, sondern können alle Listen kompakt in *einem* Array speichern.

Zeitaufwand: Θ(Länge der kleineren Liste) für die Umbenennungen im Array `r` und $O(1)$ für das Ändern des Eintrags `size`$[t]$ sowie das Kombinieren der Listen.

Zur Effizienzverbesserung (um konstanten Faktor) und zur Platzersparnis gibt es noch einen Trick.
Da alle Listen zusammen stets die Einträge $1, 2, \ldots, n$ haben, brauchen wir keine dynamisch erzeugten Listenelemente, sondern können alle Listen kompakt in *einem* Array speichern.

`next`$[1..n]$`: integer` mit

$$\texttt{next}[i] = \begin{cases} j, & \text{falls } j \text{ Nachfolger von } i \text{ in einer der Listen } L_s, \\ 0, & \text{falls } j \textbf{ letzter} \text{ Eintrag in seiner Liste } L_{r(j)}. \end{cases}$$

Zeitaufwand: Θ(Länge der kleineren Liste) für die Umbenennungen im Array `r` und $O(1)$ für das Ändern des Eintrags `size`$[t]$ sowie das Kombinieren der Listen.

Zur Effizienzverbesserung (um konstanten Faktor) und zur Platzersparnis gibt es noch einen Trick.
Da alle Listen zusammen stets die Einträge $1, 2, \ldots, n$ haben, brauchen wir keine dynamisch erzeugten Listenelemente, sondern können alle Listen kompakt in *einem* Array speichern.

`next`$[1..n]$`: integer` mit

$$\texttt{next}[i] = \begin{cases} j, & \text{falls } j \text{ Nachfolger von } i \text{ in einer der Listen } L_s, \\ 0, & \text{falls } j \textbf{ letzter} \text{ Eintrag in seiner Liste } L_{r(j)}. \end{cases}$$

Beispiel:

	1	2	3	4	5	6	7	8	9	10	11	12
`next:`	–	$\cdots$	12	$\cdots$	$\cdots$	8	6	3	0	$\cdots$	$\cdots$	9

Darstellung von $L_7 = (7, 6, 8, 3, 12, 9)$. (Hier sieht man, wieso L_s mit s beginnen muss.)

Implementierung der Operationen im Detail:

Implementierung der Operationen im Detail:

Prozedur init(n) // Initialisierung einer Union-Find-Struktur

(1) Erzeuge `r, size, next`: Arrays der Länge n für `int`-Einträge
(2) **for** `i` **from** 1 **to** n **do**
(3) `r[i]` $\leftarrow$ `i`;
(4) `size[i]` $\leftarrow 1$;
(5) `next[i]` $\leftarrow 0$.

Zeitaufwand: $\Theta(n)$.

Implementierung der Operationen im Detail:

Prozedur init(n) // Initialisierung einer Union-Find-Struktur

(1) Erzeuge `r, size, next`: Arrays der Länge n für `int`-Einträge
(2) **for** `i` **from** 1 **to** n **do**
(3) `r[i]` $\leftarrow$ `i`;
(4) `size[i]` $\leftarrow 1$;
(5) `next[i]` $\leftarrow 0$.

Zeitaufwand: $\Theta(n)$.

Prozedur find(i)

(1) **return** `r[`i`]`.

Zeitaufwand: $O(1)$.

Prozedur union(s, t)

```
    // Ausgangspunkt: s, t sind verschiedene Repräsentanten
(1) if size[s] > size[t] then vertausche s, t;
(2)    // nun: size[s] ≤ size[t]
(3) z ← s;
(4) r[z] ← t;
(5) while next[z] ≠ 0 do        // durchlaufe L_s, setzt r-Werte um
(6)    z ← next[z];
(7)    r[z] ← t;
(8) // nun: z enthält letztes Element von L_s; next[z] = 0
(9) // L_s nach dem ersten Eintrag in L_t einhängen:
(10) next[z] ← next[t]; next[t] ← s;
(11) size[t] ← size[t] + size[s].
```

Aufwand: $O(\text{Länge der kürzeren der Listen } L_s, L_t)$.

Behauptung:

$n-1$ **union**-Aufrufe (mehr kann es nicht geben!) benötigen Zeit $O(n \log n)$.

Wir behaupten **nicht**, dass jeder einzelne **union**-Aufruf Zeit $O(\log n)$ benötigt (hierfür kann man leicht Gegenbeispiele konstruieren)

Behauptung:

$n-1$ **union**-Aufrufe (mehr kann es nicht geben!) benötigen Zeit $O(n \log n)$.

Wir behaupten **nicht**, dass jeder einzelne **union**-Aufruf Zeit $O(\log n)$ benötigt (hierfür kann man leicht Gegenbeispiele konstruieren) – aber in der Summe entfällt auf jeden solchen Aufruf ein Anteil von $O(\log n)$:

„Amortisierte Analyse“

Behauptung:

$n-1$ **union**-Aufrufe (mehr kann es nicht geben!) benötigen Zeit $O(n \log n)$.

Wir behaupten **nicht**, dass jeder einzelne **union**-Aufruf Zeit $O(\log n)$ benötigt (hierfür kann man leicht Gegenbeispiele konstruieren) – aber in der Summe entfällt auf jeden solchen Aufruf ein Anteil von $O(\log n)$:

„Amortisierte Analyse“

Wir nummerieren die **union**-Aufrufe mit $p = 1, 2, \ldots, n-1$ durch. Die beiden in Aufruf Nummer p vereinigten Klassen seien $\boldsymbol{K_{p,1}}$ und $\boldsymbol{K_{p,2}}$, wobei $\boldsymbol{|K_{p,1}| \leq |K_{p,2}|}$ gelten soll.

Behauptung:

$n-1$ **union**-Aufrufe (mehr kann es nicht geben!) benötigen Zeit $O(n \log n)$.

Wir behaupten **nicht**, dass jeder einzelne **union**-Aufruf Zeit $O(\log n)$ benötigt (hierfür kann man leicht Gegenbeispiele konstruieren) – aber in der Summe entfällt auf jeden solchen Aufruf ein Anteil von $O(\log n)$:

„Amortisierte Analyse“

Wir nummerieren die **union**-Aufrufe mit $p = 1, 2, \ldots, n-1$ durch. Die beiden in Aufruf Nummer p vereinigten Klassen seien $\boldsymbol{K_{p,1}}$ und $\boldsymbol{K_{p,2}}$, wobei $\boldsymbol{|K_{p,1}| \leq |K_{p,2}|}$ gelten soll.

Für den p-ten **union**-Aufruf veranschlagen wir Kosten $\boldsymbol{|K_{p,1}|}$.

Behauptung:

$n-1$ **union**-Aufrufe (mehr kann es nicht geben!) benötigen Zeit $O(n \log n)$.

Wir behaupten **nicht**, dass jeder einzelne **union**-Aufruf Zeit $O(\log n)$ benötigt (hierfür kann man leicht Gegenbeispiele konstruieren) – aber in der Summe entfällt auf jeden solchen Aufruf ein Anteil von $O(\log n)$:

„Amortisierte Analyse“

Wir nummerieren die **union**-Aufrufe mit $p = 1, 2, \ldots, n-1$ durch. Die beiden in Aufruf Nummer p vereinigten Klassen seien $\boldsymbol{K_{p,1}}$ und $\boldsymbol{K_{p,2}}$, wobei $\boldsymbol{|K_{p,1}| \leq |K_{p,2}|}$ gelten soll.

Für den p-ten **union**-Aufruf veranschlagen wir Kosten $\boldsymbol{|K_{p,1}|}$.

Dann ist der Gesamt-Zeitaufwand für alle **union**s zusammen

$$O(n) + O\Big(\sum_{1 \leq p < n} |K_{p,1}| \Big).$$

Behauptung:

$n-1$ **union**-Aufrufe (mehr kann es nicht geben!) benötigen Zeit $O(n \log n)$.

Wir behaupten **nicht**, dass jeder einzelne **union**-Aufruf Zeit $O(\log n)$ benötigt (hierfür kann man leicht Gegenbeispiele konstruieren) – aber in der Summe entfällt auf jeden solchen Aufruf ein Anteil von $O(\log n)$:

„Amortisierte Analyse“

Wir nummerieren die **union**-Aufrufe mit $p = 1, 2, \ldots, n-1$ durch. Die beiden in Aufruf Nummer p vereinigten Klassen seien $\boldsymbol{K_{p,1}}$ und $\boldsymbol{K_{p,2}}$, wobei $\boldsymbol{|K_{p,1}| \leq |K_{p,2}|}$ gelten soll.

Für den p-ten **union**-Aufruf veranschlagen wir Kosten $\boldsymbol{|K_{p,1}|}$.

Dann ist der Gesamt-Zeitaufwand für alle **union**s zusammen

$$O(n) + O\Big(\sum_{1 \leq p < n} |K_{p,1}| \Big).$$

Wir wollen $\sum_{1 \leq p < n} |K_{p,1}|$ abschätzen.

Trick: Ermittle den Beitrag zu dieser Summe aus Sicht der einzelnen Elemente i.

Trick: Ermittle den Beitrag zu dieser Summe aus Sicht der einzelnen Elemente i.

Für $1 \leq i \leq n$, $1 \leq p < n$ definiere:

$$a_{i,p} := \begin{cases} 1 & \text{falls } i \in K_{p,1}, \\ 0 & \text{sonst.} \end{cases}$$

Trick: Ermittle den Beitrag zu dieser Summe aus Sicht der einzelnen Elemente i.

Für $1 \leq i \leq n$, $1 \leq p < n$ definiere:

$$a_{i,p} := \begin{cases} 1 & \text{falls } i \in K_{p,1}, \\ 0 & \text{sonst.} \end{cases}$$

Dann erhält man durch Umordnen der Summation:

$$\sum_{1 \leq p < n} |K_{p,1}|$$

Trick: Ermittle den Beitrag zu dieser Summe aus Sicht der einzelnen Elemente i.

Für $1 \leq i \leq n$, $1 \leq p < n$ definiere:

$$a_{i,p} := \begin{cases} 1 & \text{falls } i \in K_{p,1}, \\ 0 & \text{sonst.} \end{cases}$$

Dann erhält man durch Umordnen der Summation:

$$\sum_{1 \leq p < n} |K_{p,1}| = \sum_{1 \leq p < n} \sum_{1 \leq i \leq n} a_{i,p}$$

Trick: Ermittle den Beitrag zu dieser Summe aus Sicht der einzelnen Elemente i.

Für $1 \leq i \leq n$, $1 \leq p < n$ definiere:

$$a_{i,p} := \begin{cases} 1 & \text{falls } i \in K_{p,1}, \\ 0 & \text{sonst.} \end{cases}$$

Dann erhält man durch Umordnen der Summation:

$$\sum_{1 \leq p < n} |K_{p,1}| = \sum_{1 \leq p < n} \sum_{1 \leq i \leq n} a_{i,p} = \sum_{1 \leq i \leq n} \Big(\sum_{1 \leq p < n} a_{i,p} \Big).$$

In der letzten inneren Summe $c_i = \sum_{1 \leq p < n} a_{i,p}$ wird für jedes $i \in \{1, \ldots, n\}$ gezählt, wie oft es bei **union**-Operationen in der kleineren Menge $K_{p,1}$ enthalten ist

In der letzten inneren Summe $c_i = \sum_{1 \leq p < n} a_{i,p}$ wird für jedes $i \in \{1, \ldots, n\}$ gezählt, wie oft es bei **union**-Operationen in der kleineren Menge $K_{p,1}$ enthalten ist (d. h. wie oft insgesamt der Repräsentant seiner Klasse gewechselt hat).

In der letzten inneren Summe $c_i = \sum_{1 \leq p < n} a_{i,p}$ wird für jedes $i \in \{1, \ldots, n\}$ gezählt, wie oft es bei **union**-Operationen in der kleineren Menge $K_{p,1}$ enthalten ist (d. h. wie oft insgesamt der Repräsentant seiner Klasse gewechselt hat).

Bei jeder **union**-Operation, bei der i in der kleineren Klasse ist, wird die Größe der Klasse mindestens **verdoppelt**,

In der letzten inneren Summe $c_i = \sum_{1 \leq p < n} a_{i,p}$ wird für jedes $i \in \{1, \ldots, n\}$ gezählt, wie oft es bei **union**-Operationen in der kleineren Menge $K_{p,1}$ enthalten ist (d. h. wie oft insgesamt der Repräsentant seiner Klasse gewechselt hat).

Bei jeder **union**-Operation, bei der i in der kleineren Klasse ist, wird die Größe der Klasse mindestens **verdoppelt**, startend mit der Klasse $\{i\}$. Da Klassen nicht größer als n werden können, gilt $2^{c_i} \leq n$,

In der letzten inneren Summe $c_i = \sum_{1 \leq p < n} a_{i,p}$ wird für jedes $i \in \{1, \ldots, n\}$ gezählt, wie oft es bei **union**-Operationen in der kleineren Menge $K_{p,1}$ enthalten ist (d. h. wie oft insgesamt der Repräsentant seiner Klasse gewechselt hat).

Bei jeder **union**-Operation, bei der i in der kleineren Klasse ist, wird die Größe der Klasse mindestens **verdoppelt**, startend mit der Klasse $\{i\}$. Da Klassen nicht größer als n werden können, gilt $2^{c_i} \leq n$, also $c_i \leq \log n$,

In der letzten inneren Summe $c_i = \sum_{1 \leq p < n} a_{i,p}$ wird für jedes $i \in \{1, \ldots, n\}$ gezählt, wie oft es bei **union**-Operationen in der kleineren Menge $K_{p,1}$ enthalten ist (d. h. wie oft insgesamt der Repräsentant seiner Klasse gewechselt hat).

Bei jeder **union**-Operation, bei der i in der kleineren Klasse ist, wird die Größe der Klasse mindestens **verdoppelt**, startend mit der Klasse $\{i\}$. Da Klassen nicht größer als n werden können, gilt $2^{c_i} \leq n$, also $c_i \leq \log n$, also

$$\sum_{1 \leq p < n} |K_{p,1}|$$

In der letzten inneren Summe $c_i = \sum_{1 \leq p < n} a_{i,p}$ wird für jedes $i \in \{1, \ldots, n\}$ gezählt, wie oft es bei **union**-Operationen in der kleineren Menge $K_{p,1}$ enthalten ist (d. h. wie oft insgesamt der Repräsentant seiner Klasse gewechselt hat).

Bei jeder **union**-Operation, bei der i in der kleineren Klasse ist, wird die Größe der Klasse mindestens **verdoppelt**, startend mit der Klasse $\{i\}$. Da Klassen nicht größer als n werden können, gilt $2^{c_i} \leq n$, also $c_i \leq \log n$, also

$$\sum_{1 \leq p < n} |K_{p,1}| = \sum_{1 \leq i \leq n} c_i$$

In der letzten inneren Summe $c_i = \sum_{1 \leq p < n} a_{i,p}$ wird für jedes $i \in \{1, \ldots, n\}$ gezählt, wie oft es bei **union**-Operationen in der kleineren Menge $K_{p,1}$ enthalten ist (d. h. wie oft insgesamt der Repräsentant seiner Klasse gewechselt hat).

Bei jeder **union**-Operation, bei der i in der kleineren Klasse ist, wird die Größe der Klasse mindestens **verdoppelt**, startend mit der Klasse $\{i\}$. Da Klassen nicht größer als n werden können, gilt $2^{c_i} \leq n$, also $c_i \leq \log n$, also

$$\sum_{1 \leq p < n} |K_{p,1}| = \sum_{1 \leq i \leq n} c_i \leq n \log n.$$

In der letzten inneren Summe $c_i = \sum_{1 \leq p < n} a_{i,p}$ wird für jedes $i \in \{1, \ldots, n\}$ gezählt, wie oft es bei **union**-Operationen in der kleineren Menge $K_{p,1}$ enthalten ist (d. h. wie oft insgesamt der Repräsentant seiner Klasse gewechselt hat).

Bei jeder **union**-Operation, bei der i in der kleineren Klasse ist, wird die Größe der Klasse mindestens **verdoppelt**, startend mit der Klasse $\{i\}$. Da Klassen nicht größer als n werden können, gilt $2^{c_i} \leq n$, also $c_i \leq \log n$, also

$$\sum_{1 \leq p < n} |K_{p,1}| = \sum_{1 \leq i \leq n} c_i \leq n \log n.$$

Damit ist die Behauptung bewiesen. □

Satz 11.4.2

In der Implementierung der Union-Find-Struktur mit Arrays hat jede **find**-Operation Rechenzeit $O(1)$, $n-1$ **union**-Operationen haben Rechenzeit $O(n \log n)$.

Satz 11.4.2

In der Implementierung der Union-Find-Struktur mit Arrays hat jede **find**-Operation Rechenzeit $O(1)$, $n-1$ **union**-Operationen haben Rechenzeit $O(n \log n)$.

Satz 11.4.1 (Vollversion)

(a) . . .

(b) Die Rechenzeit des Algorithmus von Kruskal ist $O(m \log n) + O(m) + O(n \log n)$, also $O(m \log n)$, wenn man die Union-Find-Struktur mit **Arrays** implementiert.

Satz 11.4.2

In der Implementierung der Union-Find-Struktur mit Arrays hat jede **find**-Operation Rechenzeit $O(1)$, $n-1$ **union**-Operationen haben Rechenzeit $O(n \log n)$.

Satz 11.4.1 (Vollversion)

(a) . . .

(b) Die Rechenzeit des Algorithmus von Kruskal ist $O(m \log n) + O(m) + O(n \log n)$, also $O(m \log n)$, wenn man die Union-Find-Struktur mit **Arrays** implementiert.

Beweis:
Sortieren der Kanten hat Kosten

Satz 11.4.2

In der Implementierung der Union-Find-Struktur mit Arrays hat jede **find**-Operation Rechenzeit $O(1)$, $n-1$ **union**-Operationen haben Rechenzeit $O(n \log n)$.

Satz 11.4.1 (Vollversion)

(a) . . .

(b) Die Rechenzeit des Algorithmus von Kruskal ist $O(m \log n) + O(m) + O(n \log n)$, also $O(m \log n)$, wenn man die Union-Find-Struktur mit **Arrays** implementiert.

Beweis:
Sortieren der Kanten hat Kosten $O(m \log m) = O(m \log n)$.

Satz 11.4.2

In der Implementierung der Union-Find-Struktur mit Arrays hat jede **find**-Operation Rechenzeit $O(1)$, $n-1$ **union**-Operationen haben Rechenzeit $O(n \log n)$.

Satz 11.4.1 (Vollversion)

(a) . . .

(b) Die Rechenzeit des Algorithmus von Kruskal ist $O(m \log n) + O(m) + O(n \log n)$, also $O(m \log n)$, wenn man die Union-Find-Struktur mit **Arrays** implementiert.

Beweis:
Sortieren der Kanten hat Kosten $O(m \log m) = O(m \log n)$.
$2m$ **find**-Operationen haben Kosten

Satz 11.4.2

In der Implementierung der Union-Find-Struktur mit Arrays hat jede **find**-Operation Rechenzeit $O(1)$, $n-1$ **union**-Operationen haben Rechenzeit $O(n \log n)$.

Satz 11.4.1 (Vollversion)

(a) . . .

(b) Die Rechenzeit des Algorithmus von Kruskal ist $O(m \log n) + O(m) + O(n \log n)$, also $O(m \log n)$, wenn man die Union-Find-Struktur mit **Arrays** implementiert.

Beweis:
Sortieren der Kanten hat Kosten $O(m \log m) = O(m \log n)$.
$2m$ **find**-Operationen haben Kosten $O(m)$.

Satz 11.4.2

In der Implementierung der Union-Find-Struktur mit Arrays hat jede **find**-Operation Rechenzeit $O(1)$, $n-1$ **union**-Operationen haben Rechenzeit $O(n \log n)$.

Satz 11.4.1 (Vollversion)

(a) . . .

(b) Die Rechenzeit des Algorithmus von Kruskal ist $O(m \log n) + O(m) + O(n \log n)$, also $O(m \log n)$, wenn man die Union-Find-Struktur mit **Arrays** implementiert.

Beweis:
Sortieren der Kanten hat Kosten $O(m \log m) = O(m \log n)$.
$2m$ **find**-Operationen haben Kosten $O(m)$.
$n-1$ **union**-Operationen haben Kosten

Satz 11.4.2

In der Implementierung der Union-Find-Struktur mit Arrays hat jede **find**-Operation Rechenzeit $O(1)$, $n-1$ **union**-Operationen haben Rechenzeit $O(n \log n)$.

Satz 11.4.1 (Vollversion)

(a) . . .

(b) Die Rechenzeit des Algorithmus von Kruskal ist $O(m \log n) + O(m) + O(n \log n)$, also $O(m \log n)$, wenn man die Union-Find-Struktur mit **Arrays** implementiert.

Beweis:
Sortieren der Kanten hat Kosten $O(m \log m) = O(m \log n)$.
$2m$ **find**-Operationen haben Kosten $O(m)$.
$n-1$ **union**-Operationen haben Kosten $O(n \log n)$. □

PAUSE

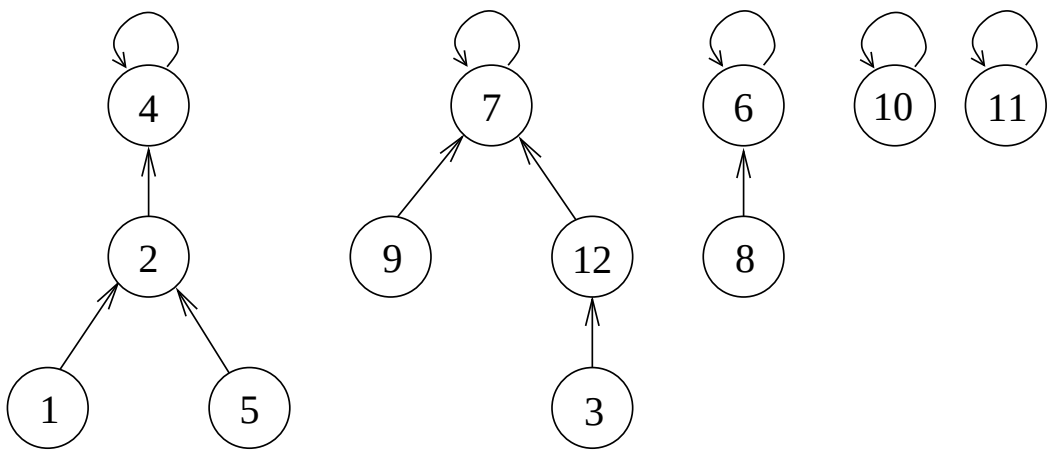
Es folgt: Baumimplementierung von Union-Find

11.4.3 Baumimplementierung von Union-Find

11.4.3 Baumimplementierung von Union-Find

Eine attraktive Implementierung der Union-Find-Datenstruktur verwendet einen **„wurzelgerichteten Wald“**.

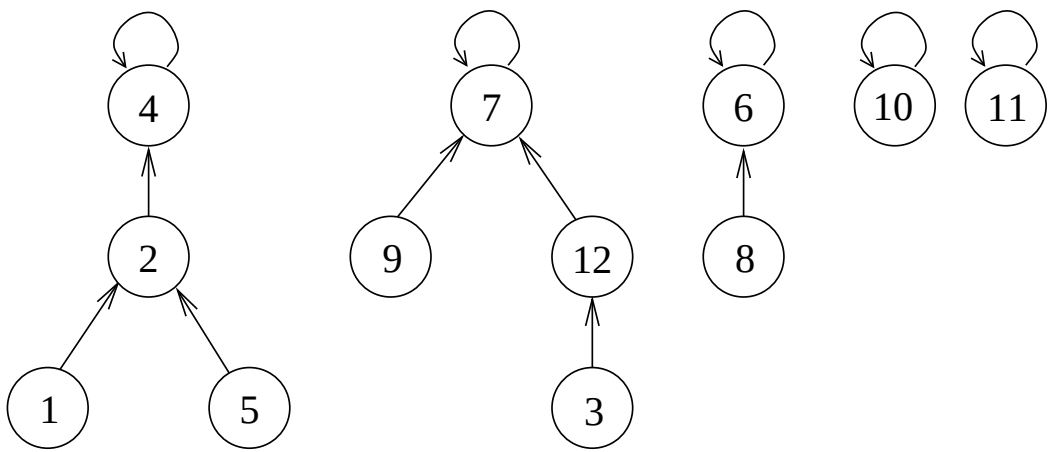
Beispiel: Partition $\{1, 2, \mathbf{4}, 5\}, \{3, \mathbf{7}, 9, 12\}, \{\mathbf{6}, 8\}, \{\mathbf{10}\}, \{\mathbf{11}\}$ wird dargestellt durch:



11.4.3 Baumimplementierung von Union-Find

Eine attraktive Implementierung der Union-Find-Datenstruktur verwendet einen **„wurzelgerichteten Wald“**.

Beispiel: Partition $\{1, 2, \mathbf{4}, 5\}, \{3, \mathbf{7}, 9, 12\}, \{\mathbf{6}, 8\}, \{\mathbf{10}\}, \{\mathbf{11}\}$ wird dargestellt durch:

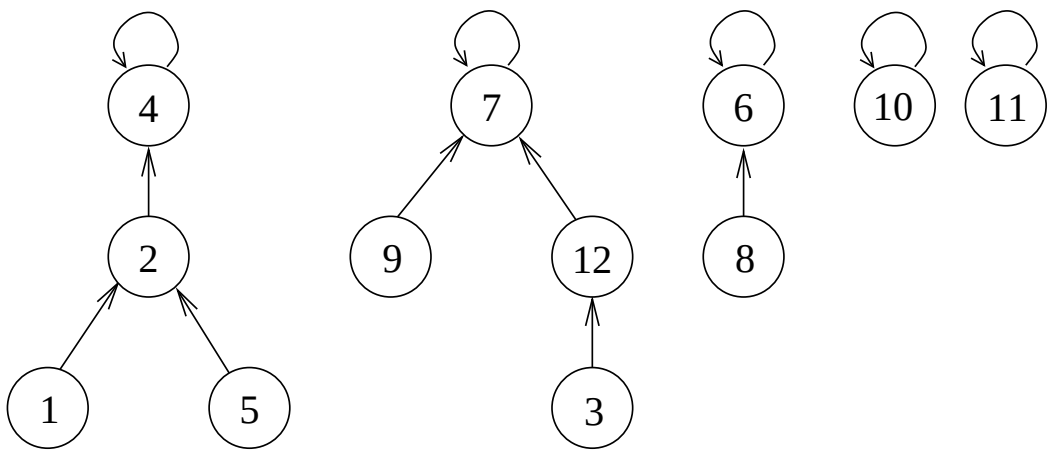


Für jede Klasse K_s gibt es genau einen Baum B_s.

11.4.3 Baumimplementierung von Union-Find

Eine attraktive Implementierung der Union-Find-Datenstruktur verwendet einen **„wurzelgerichteten Wald“**.

Beispiel: Partition $\{1, 2, \mathbf{4}, 5\}, \{3, \mathbf{7}, 9, 12\}, \{\mathbf{6}, 8\}, \{\mathbf{10}\}, \{\mathbf{11}\}$ wird dargestellt durch:

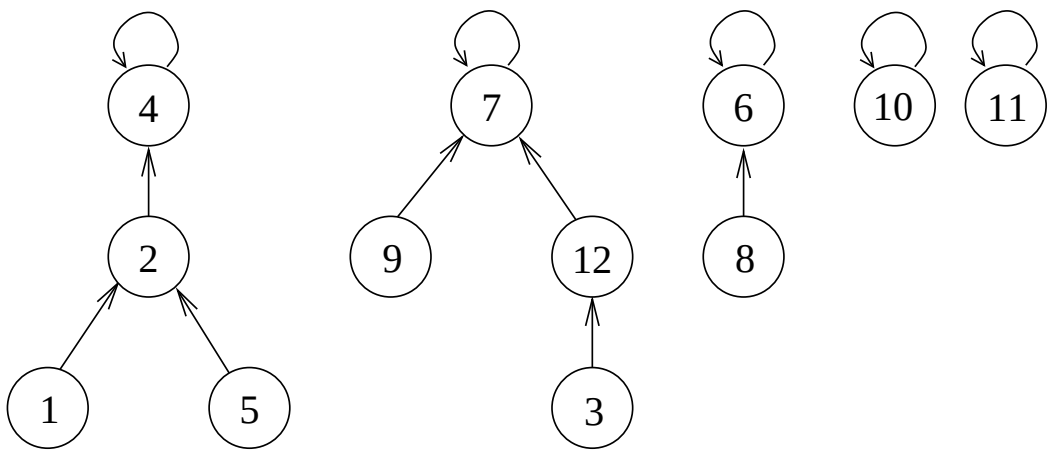


Für jede Klasse K_s gibt es genau einen Baum B_s. Jedes $i \in K_s$ ist ein Knoten im Baum B_s.

11.4.3 Baumimplementierung von Union-Find

Eine attraktive Implementierung der Union-Find-Datenstruktur verwendet einen **„wurzelgerichteten Wald“**.

Beispiel: Partition $\{1, 2, \mathbf{4}, 5\}, \{3, \mathbf{7}, 9, 12\}, \{\mathbf{6}, 8\}, \{\mathbf{10}\}, \{\mathbf{11}\}$ wird dargestellt durch:

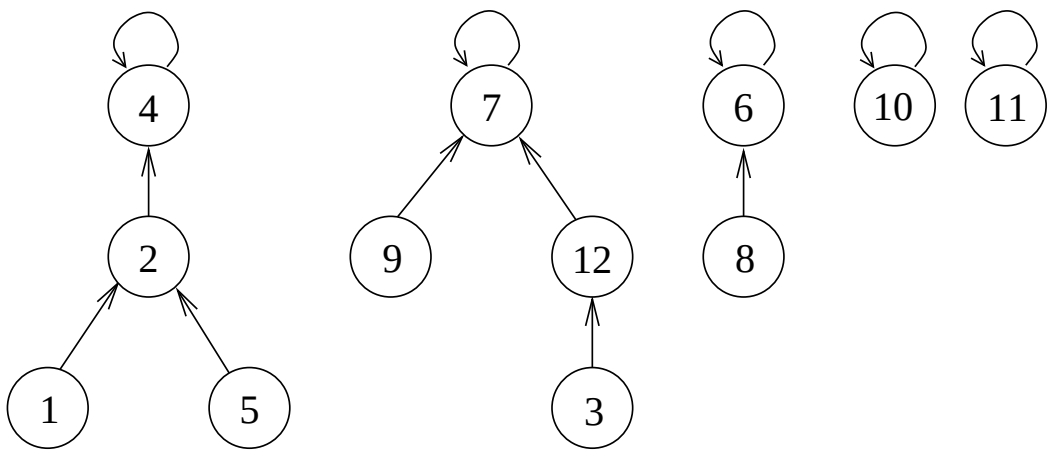


Für jede Klasse K_s gibt es genau einen Baum B_s. Jedes $i \in K_s$ ist ein Knoten im Baum B_s.
Es gibt nur Zeiger in Richtung auf die Wurzel zu.

11.4.3 Baumimplementierung von Union-Find

Eine attraktive Implementierung der Union-Find-Datenstruktur verwendet einen **„wurzelgerichteten Wald“**.

Beispiel: Partition $\{1, 2, \mathbf{4}, 5\}, \{3, \mathbf{7}, 9, 12\}, \{\mathbf{6}, 8\}, \{\mathbf{10}\}, \{\mathbf{11}\}$ wird dargestellt durch:



Für jede Klasse K_s gibt es genau einen Baum B_s. Jedes $i \in K_s$ ist ein Knoten im Baum B_s.
Es gibt nur Zeiger in Richtung auf die Wurzel zu. $p(i)$ ist der Vorgänger von i.
Die Wurzel ist der Repräsentant s.
Sie zeigt auf sich selbst als „Vorgänger“: $p(i) = i$ genau dann wenn i Repräsentant ist.

Eine kostengünstige Darstellung eines solchen Waldes benutzt nur ein Integerarray $\texttt{p}[1..n]$. Für Knoten i gibt der Eintrag $\texttt{p}[i]$ den Vorgängerknoten $p(i)$ an.

Eine kostengünstige Darstellung eines solchen Waldes benutzt nur ein Integerarray $\texttt{p}[1..n]$. Für Knoten i gibt der Eintrag $\texttt{p}[i]$ den Vorgängerknoten $p(i)$ an.

Das dem Wald im Beispiel entsprechende Array sieht so aus:

	1	2	3	4	5	6	7	8	9	10	11	12
p:	2	4	12	4	2	6	7	6	7	10	11	7

Eine kostengünstige Darstellung eines solchen Waldes benutzt nur ein Integerarray $\texttt{p}[1..n]$. Für Knoten i gibt der Eintrag $\texttt{p}[i]$ den Vorgängerknoten $p(i)$ an.

Das dem Wald im Beispiel entsprechende Array sieht so aus:

	1	2	3	4	5	6	7	8	9	10	11	12
p:	2	4	12	4	2	6	7	6	7	10	11	7

Prozedur find(i)

Eine kostengünstige Darstellung eines solchen Waldes benutzt nur ein Integerarray `p[1..n]`. Für Knoten i gibt der Eintrag `p[`i`]` den Vorgängerknoten $p(i)$ an.

Das dem Wald im Beispiel entsprechende Array sieht so aus:

	1	2	3	4	5	6	7	8	9	10	11	12
p:	2	4	12	4	2	6	7	6	7	10	11	7

Prozedur find(i)

```
(1) j ← i;
(2) jj ← p[j];          // (2)–(6): verfolge Vorgängerzeiger bis zur Wurzel
(3) while jj ≠ j do
(4)    j ← jj;
(5)    jj ← p[j] ;
(6) return j.
```

Eine kostengünstige Darstellung eines solchen Waldes benutzt nur ein Integerarray `p[1..n]`. Für Knoten i gibt der Eintrag `p[`i`]` den Vorgängerknoten $p(i)$ an.

Das dem Wald im Beispiel entsprechende Array sieht so aus:

	1	2	3	4	5	6	7	8	9	10	11	12
p:	2	4	12	4	2	6	7	6	7	10	11	7

Prozedur find(i)

```
(1) j ← i;
(2) jj ← p[j];          // (2)–(6): verfolge Vorgängerzeiger bis zur Wurzel
(3) while jj ≠ j do
(4)    j ← jj;
(5)    jj ← p[j] ;
(6) return j.
```

Zeitaufwand:

Eine kostengünstige Darstellung eines solchen Waldes benutzt nur ein Integerarray `p[1..n]`. Für Knoten i gibt der Eintrag `p[`i`]` den Vorgängerknoten $p(i)$ an.

Das dem Wald im Beispiel entsprechende Array sieht so aus:

	1	2	3	4	5	6	7	8	9	10	11	12
p:	2	4	12	4	2	6	7	6	7	10	11	7

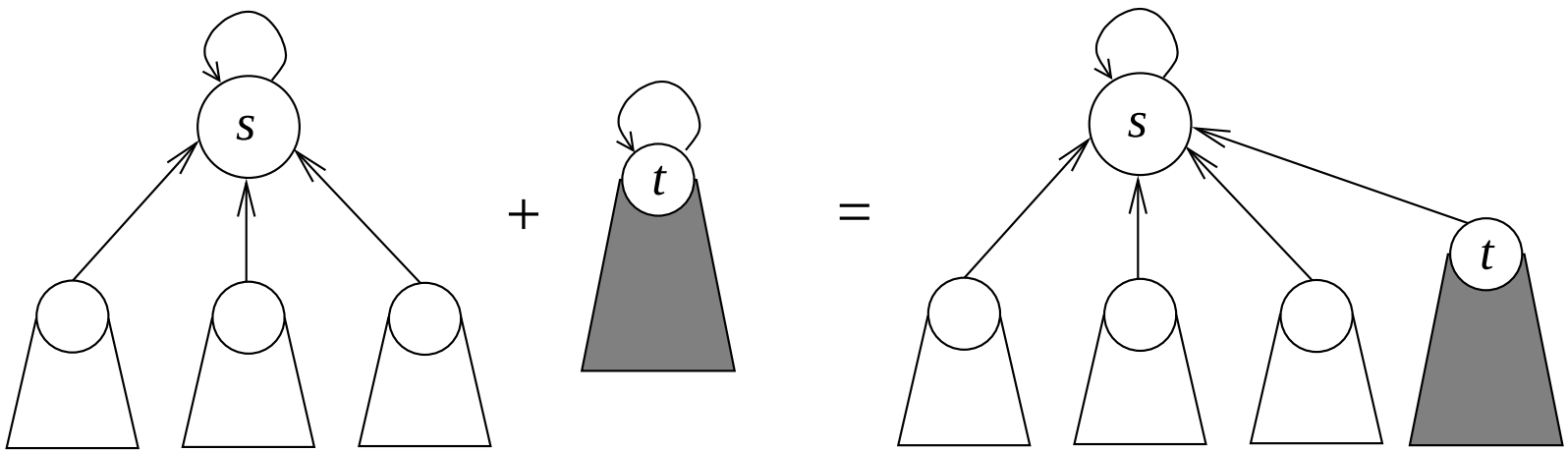
Prozedur find(i)

```
(1) j ← i;
(2) jj ← p[j];            // (2)–(6): verfolge Vorgängerzeiger bis zur Wurzel
(3) while jj ≠ j do
(4)    j ← jj;
(5)    jj ← p[j] ;
(6) return j.
```

Zeitaufwand: $\Theta(\mathit{depth}(i)) = \Theta(\text{Tiefe von } i \text{ in seinem Baum})$.

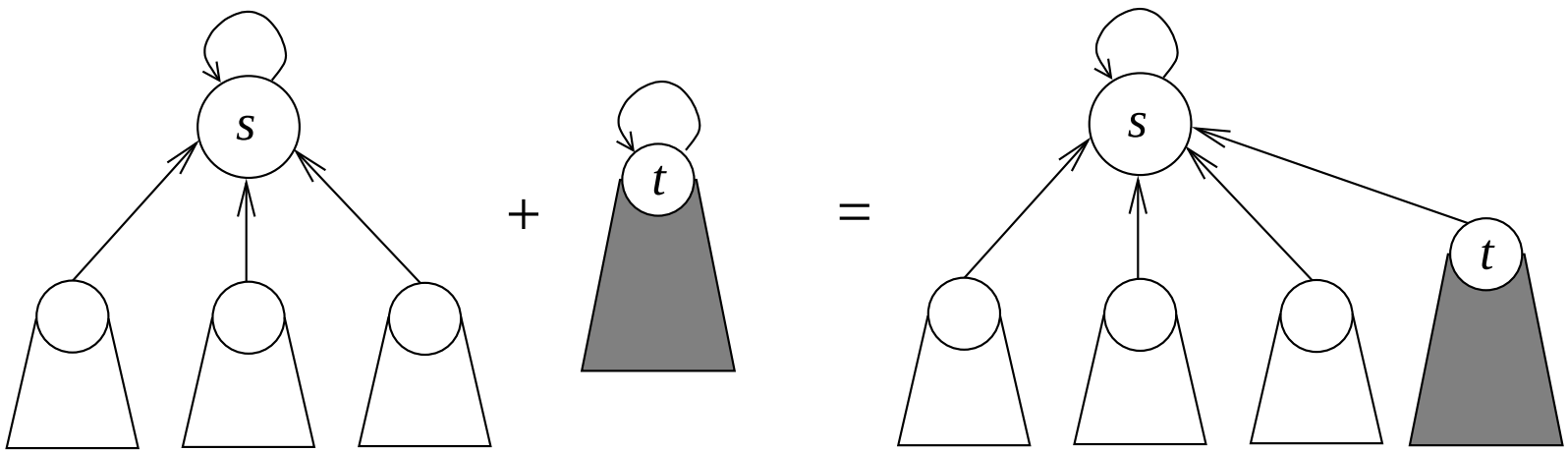
union(s,t): Gegeben: Verschiedene Repräsentanten s und t.

Man macht einen der Repräsentanten zum Kind des anderen, setzt also $\mathrm{p}(s) := t$ **oder** $\mathrm{p}(t) := s$.



union(s, t): Gegeben: Verschiedene Repräsentanten s und t.

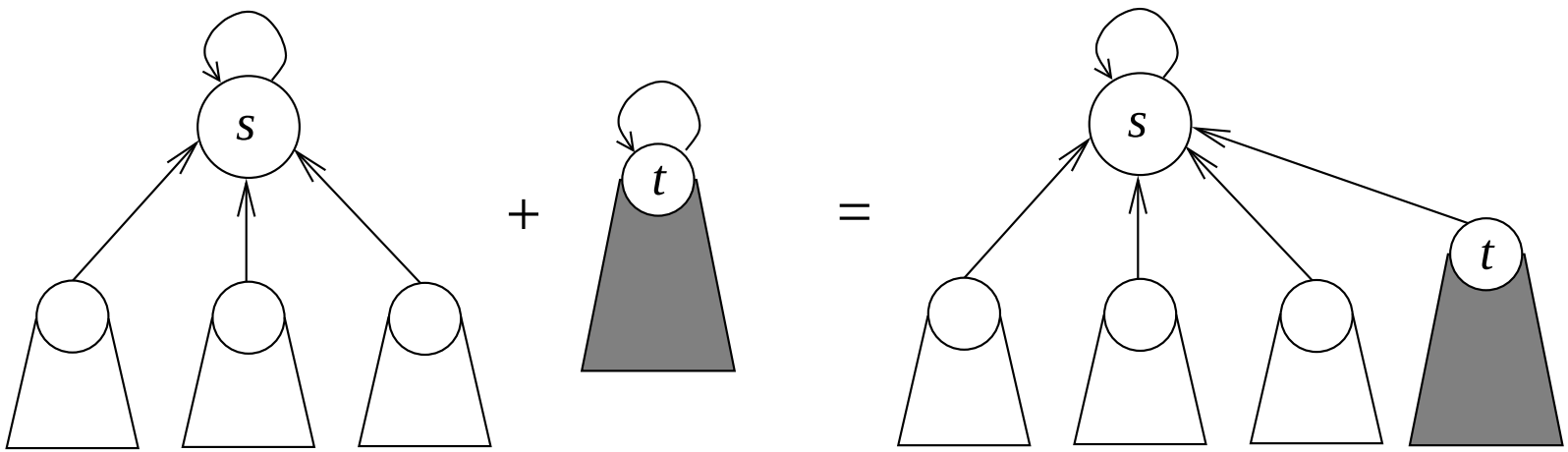
Man macht einen der Repräsentanten zum Kind des anderen, setzt also $\mathrm{p}(s) := t$ **oder** $\mathrm{p}(t) := s$.



? Welche der beiden Optionen sollte man nehmen?

union(s, t): Gegeben: Verschiedene Repräsentanten s und t.

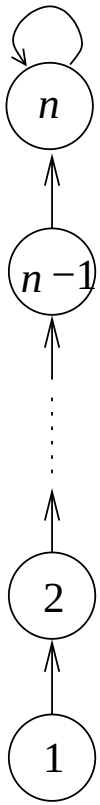
Man macht einen der Repräsentanten zum Kind des anderen, setzt also $\mathrm{p}(s) := t$ **oder** $\mathrm{p}(t) := s$.



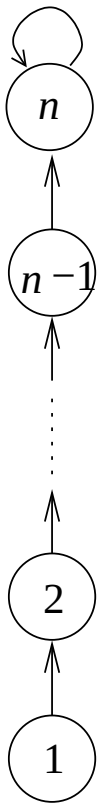
? Welche der beiden Optionen sollte man nehmen?

Ungeschickt: union$(i, i+1)$, $i = 1, \ldots, n-1$, dadurch ausführen, dass man nacheinander $\mathrm{p}(i) := i + 1$ ausführt.

Dies führt zu dem Baum



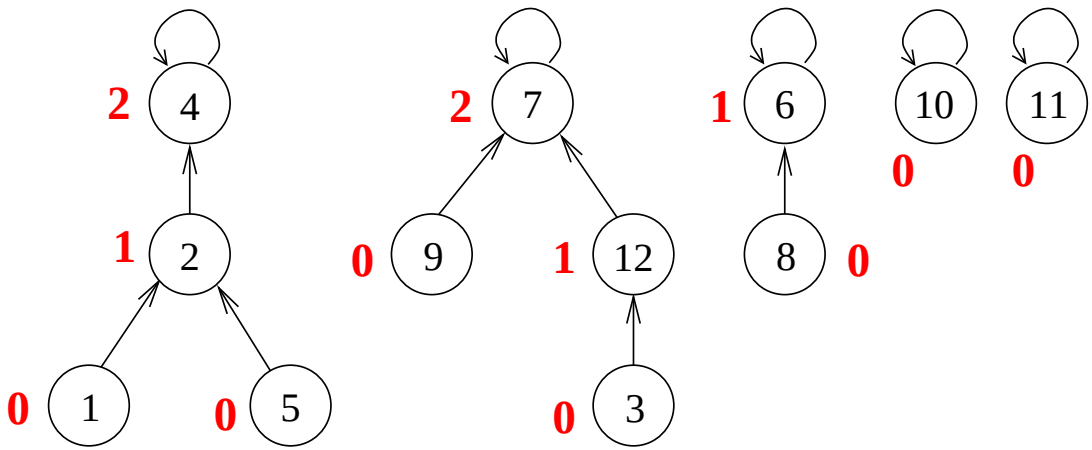
Dies führt zu dem Baum



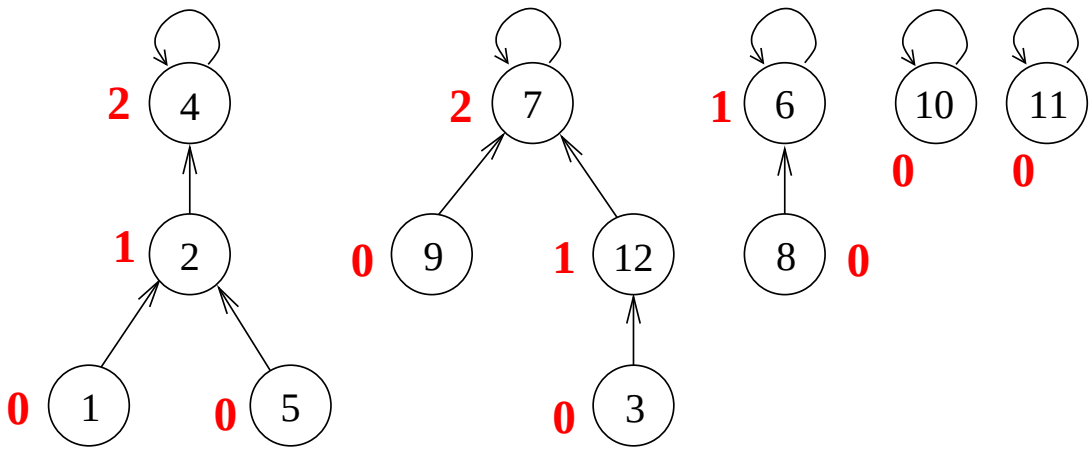
find-Operationen sind nun sehr teuer!
Ein **find**(i) für jedes $i \in \{1, \ldots, n\}$ führt zu Gesamtkosten $\Theta(n^2)$!

Trick: Führe für jeden Knoten i eine Zahl *rank*(i) **(Rang)** mit, in einem Array `rank[1..`n`]: array of int`, mit `rank[`i`]` $=$ Tiefe des Teilbaums mit Wurzel i.

Trick: Führe für jeden Knoten i eine Zahl *rank*(i) **(Rang)** mit, in einem Array `rank[1..n]: array of int`, mit `rank[`i`]` $=$ Tiefe des Teilbaums mit Wurzel i.



Trick: Führe für jeden Knoten i eine Zahl *rank*(i) **(Rang)** mit, in einem Array `rank[1..n]: array of int`, mit `rank[`i`]` $=$ Tiefe des Teilbaums mit Wurzel i.



Für **union**(s, t) benutzen wir die Regel „*union by rank*“:
Die Wurzel mit dem größeren Rang wird Wurzel des neuen Baums, keine Änderung der Ränge.

Bei gleichen Rängen ist die Wahl der Wurzel gleichgültig.
In diesem Fall erhöht sich der Rang des Knotens, der Wurzel bleibt, um 1.

Operationen:

Prozedur init(n) // Initialisierung

Operationen:

Prozedur init(n) // Initialisierung

(1) Erzeuge `p`, `rank`: Arrays der Länge n für `int`-Einträge

(2) **for** `i` **from** 1 **to** n **do**

(3) `p[i]` $\leftarrow$ `i`; `rank[i]` $\leftarrow 0$; // n Bäume mit je einem (Wurzel-)Knoten vom Rang 0

Operationen:

Prozedur init(n) // Initialisierung
(1) Erzeuge `p`, `rank`: Arrays der Länge n für `int`-Einträge
(2) **for** `i` **from** 1 **to** n **do**
(3) `p[i]` $\leftarrow$ `i`; `rank[i]` $\leftarrow 0$; // n Bäume mit je einem (Wurzel-)Knoten vom Rang 0

Zeitaufwand: $\Theta(n)$.

Prozedur union(s, t) // „union by rank“

Operationen:

Prozedur init(n) // Initialisierung

(1) Erzeuge p, rank: Arrays der Länge n für int-Einträge
(2) **for** i **from** 1 **to** n **do**
(3) p[i] $\leftarrow$ i; rank[i] $\leftarrow 0$; // n Bäume mit je einem (Wurzel-)Knoten vom Rang 0

Zeitaufwand: $\Theta(n)$.

Prozedur union(s, t) // „union by rank“

// Ausgangspunkt: s, t sind verschiedene Repräsentanten von Klassen
// D.h.: p[s] $= s$ und p[t] $= t$

(1) **if** rank[s] $>$ rank[t]
(2) **then** p[t] $\leftarrow s$
(3) **elseif** rank[t] $>$ rank[s]
(4) **then** p[s] $\leftarrow t$
(5) **else** p[t] $\leftarrow s$; rank[s] $\leftarrow$ rank[s] $+ 1$.

Operationen:

Prozedur init(n) // Initialisierung
(1) Erzeuge p, rank: Arrays der Länge n für int-Einträge
(2) **for** i **from** 1 **to** n **do**
(3) p[i] $\leftarrow$ i; rank[i] $\leftarrow 0$; // n Bäume mit je einem (Wurzel-)Knoten vom Rang 0

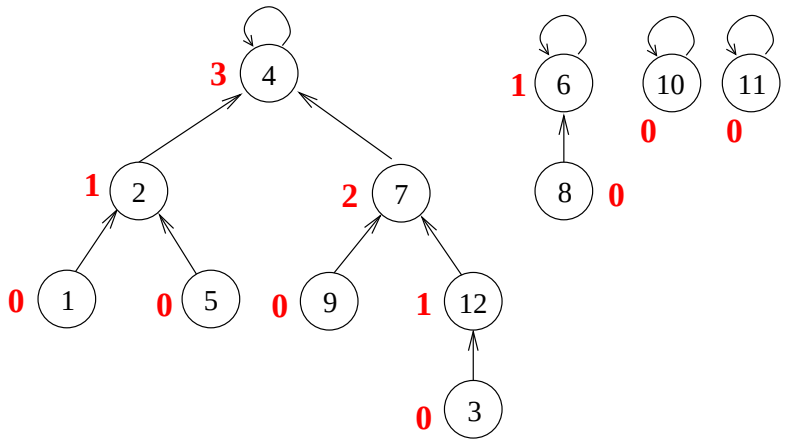
Zeitaufwand: $\Theta(n)$.

Prozedur union(s, t) // „union by rank“
// Ausgangspunkt: s, t sind verschiedene Repräsentanten von Klassen
// D.h.: p[s] $= s$ und p[t] $= t$
(1) **if** rank[s] $>$ rank[t]
(2) **then** p[t] $\leftarrow s$
(3) **elseif** rank[t] $>$ rank[s]
(4) **then** p[s] $\leftarrow t$
(5) **else** p[t] $\leftarrow s$; rank[s] $\leftarrow$ rank[s] $+ 1$.

Zeitaufwand: $O(1)$.

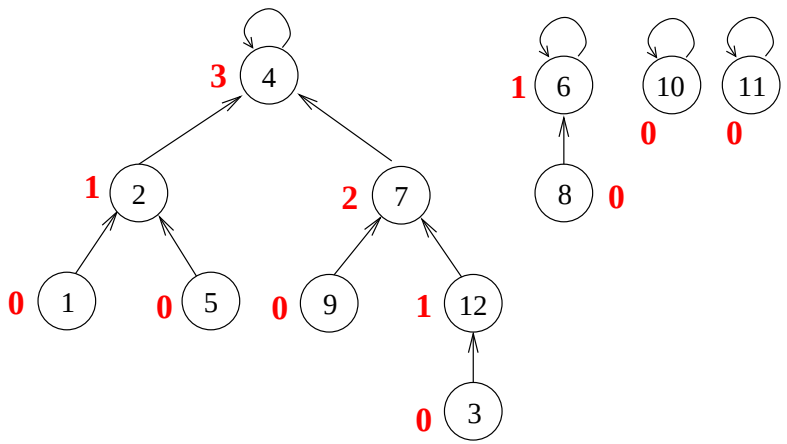
Beispiele:

union$(4, 7)$ würde liefern:

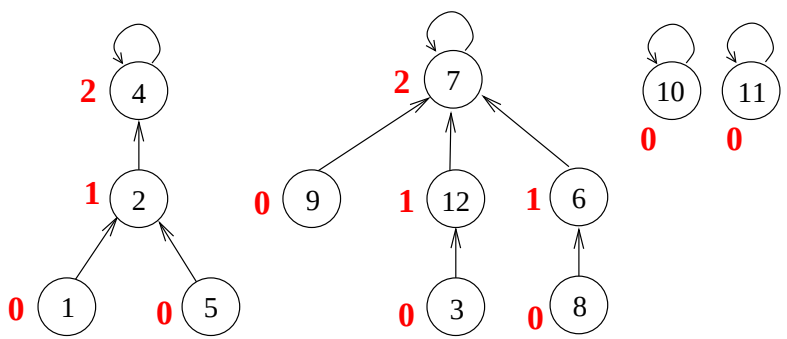


Beispiele:

union$(4, 7)$ würde liefern:



union$(6, 7)$ würde liefern:



Satz 11.4.3

Die eben beschriebene Implementierung einer Union-Find-Struktur mit einem wurzelgerichteten Wald hat folgende Eigenschaften:

a) Sie ist korrekt (d. h. hat das vorgesehene Ein-/Ausgabeverhalten).

Satz 11.4.3

Die eben beschriebene Implementierung einer Union-Find-Struktur mit einem wurzelgerichteten Wald hat folgende Eigenschaften:

a) Sie ist korrekt (d. h. hat das vorgesehene Ein-/Ausgabeverhalten).

b) **init**(n) benötigt Zeit $\Theta(n)$;

Satz 11.4.3

Die eben beschriebene Implementierung einer Union-Find-Struktur mit einem wurzelgerichteten Wald hat folgende Eigenschaften:

a) Sie ist korrekt (d. h. hat das vorgesehene Ein-/Ausgabeverhalten).

b) **init**(n) benötigt Zeit $\Theta(n)$; **find**(i) benötigt Zeit $O(\log n)$;

Satz 11.4.3

Die eben beschriebene Implementierung einer Union-Find-Struktur mit einem wurzelgerichteten Wald hat folgende Eigenschaften:

a) Sie ist korrekt (d. h. hat das vorgesehene Ein-/Ausgabeverhalten).

b) **init**(n) benötigt Zeit $\Theta(n)$; **find**(i) benötigt Zeit $O(\log n)$; **union**(s,t) benötigt Zeit $O(1)$.

Satz 11.4.3

Die eben beschriebene Implementierung einer Union-Find-Struktur mit einem wurzelgerichteten Wald hat folgende Eigenschaften:

a) Sie ist korrekt (d. h. hat das vorgesehene Ein-/Ausgabeverhalten).

b) **init**(n) benötigt Zeit $\Theta(n)$; **find**(i) benötigt Zeit $O(\log n)$; **union**(s,t) benötigt Zeit $O(1)$.

Beweis: (a) ist unmittelbar klar.

(b) Wir beobachten:

Fakt 1: $i \neq p(i) \Rightarrow \mathit{rank}(i) < \mathit{rank}(p(i))$.

Satz 11.4.3

Die eben beschriebene Implementierung einer Union-Find-Struktur mit einem wurzelgerichteten Wald hat folgende Eigenschaften:

a) Sie ist korrekt (d. h. hat das vorgesehene Ein-/Ausgabeverhalten).

b) **init**(n) benötigt Zeit $\Theta(n)$; **find**(i) benötigt Zeit $O(\log n)$; **union**(s,t) benötigt Zeit $O(1)$.

Beweis: (a) ist unmittelbar klar.

(b) Wir beobachten:

Fakt 1: $i \neq p(i) \Rightarrow \mathit{rank}(i) < \mathit{rank}(p(i))$.

Beweis: Wenn s bei einer **union**-Operation Kind von t wird, dann ist entweder $\mathit{rank}(s) < \mathit{rank}(t)$ (und das bleibt so)

Satz 11.4.3

Die eben beschriebene Implementierung einer Union-Find-Struktur mit einem wurzelgerichteten Wald hat folgende Eigenschaften:

a) Sie ist korrekt (d. h. hat das vorgesehene Ein-/Ausgabeverhalten).

b) **init**(n) benötigt Zeit $\Theta(n)$; **find**(i) benötigt Zeit $O(\log n)$; **union**(s,t) benötigt Zeit $O(1)$.

Beweis: (a) ist unmittelbar klar.

(b) Wir beobachten:

Fakt 1: $i \neq p(i) \Rightarrow \mathit{rank}(i) < \mathit{rank}(p(i))$.

Beweis: Wenn s bei einer **union**-Operation Kind von t wird, dann ist entweder $\mathit{rank}(s) < \mathit{rank}(t)$ (und das bleibt so) oder es ist $\mathit{rank}(s) = \mathit{rank}(t)$, und der Rang von t erhöht sich nun um 1.

Satz 11.4.3

Die eben beschriebene Implementierung einer Union-Find-Struktur mit einem wurzelgerichteten Wald hat folgende Eigenschaften:

a) Sie ist korrekt (d. h. hat das vorgesehene Ein-/Ausgabeverhalten).

b) **init**(n) benötigt Zeit $\Theta(n)$; **find**(i) benötigt Zeit $O(\log n)$; **union**(s,t) benötigt Zeit $O(1)$.

Beweis: (a) ist unmittelbar klar.

(b) Wir beobachten:

Fakt 1: $i \neq p(i) \Rightarrow \mathit{rank}(i) < \mathit{rank}(p(i))$.

Beweis: Wenn s bei einer **union**-Operation Kind von t wird, dann ist entweder $\mathit{rank}(s) < \mathit{rank}(t)$ (und das bleibt so) oder es ist $\mathit{rank}(s) = \mathit{rank}(t)$, und der Rang von t erhöht sich nun um 1.
Nachher ist s keine Wurzel mehr. Daher ändert sich $\mathit{rank}(s)$ nicht mehr, und auch $p(s)$ bleibt gleich. Der Rang von $p(s)$ kann im weiteren Verlauf nur noch wachsen. Daher bleibt die behauptete Ungleichung erhalten.

Fakt 2: Wenn s Wurzel des Baums B_s ist und $h = \mathit{rank}(s)$ gilt, dann enthält B_s mindestens 2^h Knoten.

Fakt 2: Wenn s Wurzel des Baums B_s ist und $h = \mathit{rank}(s)$ gilt, dann enthält B_s mindestens 2^h Knoten.

Beweis: Für $h = 0$ ist die Aussage richtig. Eine Wurzel vom Rang $h \geq 1$ entsteht, wenn zwei Bäume vereinigt werden, deren Wurzeln beide Rang $h - 1$ haben. Daraus folgt Fakt 2 leicht durch Induktion über $h = 0, 1, \ldots$.

Fakt 2: Wenn s Wurzel des Baums B_s ist und $h = \mathit{rank}(s)$ gilt, dann enthält B_s mindestens 2^h Knoten.

Beweis: Für $h = 0$ ist die Aussage richtig. Eine Wurzel vom Rang $h \geq 1$ entsteht, wenn zwei Bäume vereinigt werden, deren Wurzeln beide Rang $h - 1$ haben. Daraus folgt Fakt 2 leicht durch Induktion über $h = 0, 1, \ldots$.

Fakt 3: Bei n Knoten insgesamt gibt es höchstens $n/2^h$ Knoten von Rang h.

Fakt 2: Wenn s Wurzel des Baums B_s ist und $h = \mathit{rank}(s)$ gilt, dann enthält B_s mindestens 2^h Knoten.

Beweis: Für $h = 0$ ist die Aussage richtig. Eine Wurzel vom Rang $h \geq 1$ entsteht, wenn zwei Bäume vereinigt werden, deren Wurzeln beide Rang $h - 1$ haben. Daraus folgt Fakt 2 leicht durch Induktion über $h = 0, 1, \ldots$.

Fakt 3: Bei n Knoten insgesamt gibt es höchstens $n/2^h$ Knoten von Rang h.

Beweis: Wegen Fakt 1 können Knoten mit Rang h nicht Nachfahren voneinander sein. Also sind alle Unterbäume, deren Wurzeln Rang h haben, disjunkt. Aus Fakt 2 folgt leicht, dass auch für Knoten i im Inneren von Bäumen mit $\mathit{rank}(i) = h$ gilt, dass ihr Unterbaum mindestens 2^h Knoten hat. Also kann es nicht mehr als $n/2^h$ solche Knoten geben.

Fakt 2: Wenn s Wurzel des Baums B_s ist und $h = \mathit{rank}(s)$ gilt, dann enthält B_s mindestens 2^h Knoten.

Beweis: Für $h = 0$ ist die Aussage richtig. Eine Wurzel vom Rang $h \geq 1$ entsteht, wenn zwei Bäume vereinigt werden, deren Wurzeln beide Rang $h - 1$ haben. Daraus folgt Fakt 2 leicht durch Induktion über $h = 0, 1, \ldots$.

Fakt 3: Bei n Knoten insgesamt gibt es höchstens $n/2^h$ Knoten von Rang h.

Beweis: Wegen Fakt 1 können Knoten mit Rang h nicht Nachfahren voneinander sein. Also sind alle Unterbäume, deren Wurzeln Rang h haben, disjunkt. Aus Fakt 2 folgt leicht, dass auch für Knoten i im Inneren von Bäumen mit $\mathit{rank}(i) = h$ gilt, dass ihr Unterbaum mindestens 2^h Knoten hat. Also kann es nicht mehr als $n/2^h$ solche Knoten geben.

Aus Fakt 3 folgt, dass es keine Knoten mit Rang größer als $\log n$ geben kann.

Fakt 2: Wenn s Wurzel des Baums B_s ist und $h = \mathit{rank}(s)$ gilt, dann enthält B_s mindestens 2^h Knoten.

Beweis: Für $h = 0$ ist die Aussage richtig. Eine Wurzel vom Rang $h \geq 1$ entsteht, wenn zwei Bäume vereinigt werden, deren Wurzeln beide Rang $h - 1$ haben. Daraus folgt Fakt 2 leicht durch Induktion über $h = 0, 1, \ldots$.

Fakt 3: Bei n Knoten insgesamt gibt es höchstens $n/2^h$ Knoten von Rang h.

Beweis: Wegen Fakt 1 können Knoten mit Rang h nicht Nachfahren voneinander sein. Also sind alle Unterbäume, deren Wurzeln Rang h haben, disjunkt. Aus Fakt 2 folgt leicht, dass auch für Knoten i im Inneren von Bäumen mit $\mathit{rank}(i) = h$ gilt, dass ihr Unterbaum mindestens 2^h Knoten hat. Also kann es nicht mehr als $n/2^h$ solche Knoten geben.

Aus Fakt 3 folgt, dass es keine Knoten mit Rang größer als $\log n$ geben kann.
Also hat kein Baum Tiefe größer als $\log n$,

Fakt 2: Wenn s Wurzel des Baums B_s ist und $h = \mathit{rank}(s)$ gilt, dann enthält B_s mindestens 2^h Knoten.

Beweis: Für $h = 0$ ist die Aussage richtig. Eine Wurzel vom Rang $h \geq 1$ entsteht, wenn zwei Bäume vereinigt werden, deren Wurzeln beide Rang $h - 1$ haben. Daraus folgt Fakt 2 leicht durch Induktion über $h = 0, 1, \ldots$.

Fakt 3: Bei n Knoten insgesamt gibt es höchstens $n/2^h$ Knoten von Rang h.

Beweis: Wegen Fakt 1 können Knoten mit Rang h nicht Nachfahren voneinander sein. Also sind alle Unterbäume, deren Wurzeln Rang h haben, disjunkt. Aus Fakt 2 folgt leicht, dass auch für Knoten i im Inneren von Bäumen mit $\mathit{rank}(i) = h$ gilt, dass ihr Unterbaum mindestens 2^h Knoten hat. Also kann es nicht mehr als $n/2^h$ solche Knoten geben.

Aus Fakt 3 folgt, dass es keine Knoten mit Rang größer als $\log n$ geben kann.
Also hat kein Baum Tiefe größer als $\log n$, also kosten **find**-Operationen maximal Zeit $O(\log n)$. □

Pfadkompression

Pfadkompression

Eine interessante Variante der Union-Find-Datenstruktur, die mit einem wurzelgerichteten Wald implementiert ist, ist der Ansatz der „**Pfadkompression**“ (oder „**Pfadverkürzung**“).

Pfadkompression

Eine interessante Variante der Union-Find-Datenstruktur, die mit einem wurzelgerichteten Wald implementiert ist, ist der Ansatz der „**Pfadkompression**“ (oder „**Pfadverkürzung**“).

Bei einem **find**(i) muss man den ganzen Weg von Knoten i zu seiner Wurzel $r(i)$ ablaufen; Aufwand $O(\mathit{depth}(i))$.

Pfadkompression

Eine interessante Variante der Union-Find-Datenstruktur, die mit einem wurzelgerichteten Wald implementiert ist, ist der Ansatz der „**Pfadkompression**“ (oder „**Pfadverkürzung**“).

Bei einem **find**(i) muss man den ganzen Weg von Knoten i zu seiner Wurzel $r(i)$ ablaufen; Aufwand $O(\mathit{depth}(i))$.

Ein gutes strategisches Ziel ist also, diese Wege möglichst kurz zu halten.

Pfadkompression

Eine interessante Variante der Union-Find-Datenstruktur, die mit einem wurzelgerichteten Wald implementiert ist, ist der Ansatz der „**Pfadkompression**“ (oder „**Pfadverkürzung**“).

Bei einem **find**(i) muss man den ganzen Weg von Knoten i zu seiner Wurzel $r(i)$ ablaufen; Aufwand $O(\mathit{depth}(i))$.

Ein gutes strategisches Ziel ist also, diese Wege möglichst kurz zu halten.

Idee: Man investiert bei **find**(i) etwas mehr Arbeit, jedoch immer noch im Rahmen $O(\mathit{depth}(i))$, um dabei einige **Wege zu verkürzen** und damit spätere **find**s billiger zu machen.

Pfadkompression

Eine interessante Variante der Union-Find-Datenstruktur, die mit einem wurzelgerichteten Wald implementiert ist, ist der Ansatz der „**Pfadkompression**“ (oder „**Pfadverkürzung**“).

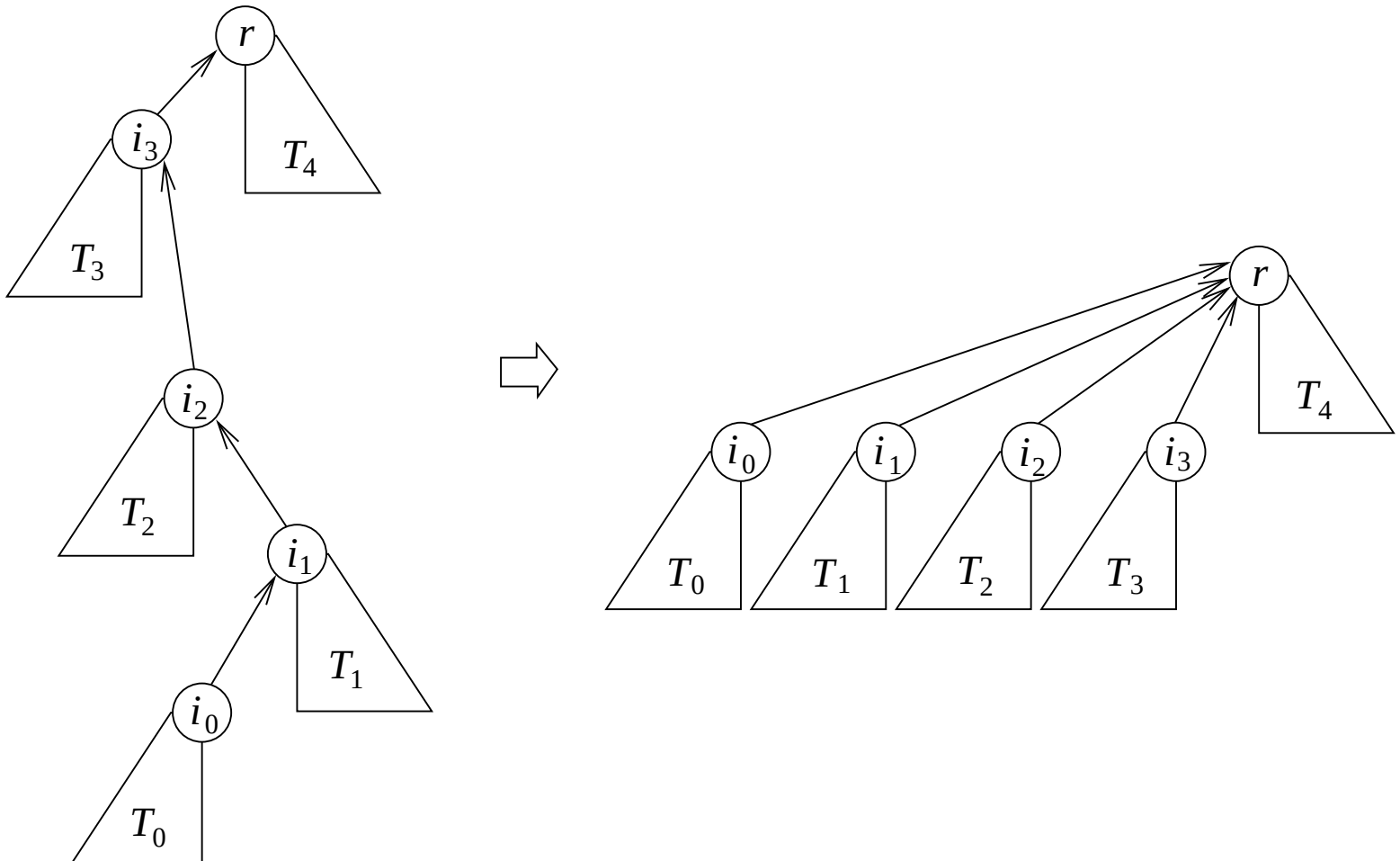
Bei einem **find**(i) muss man den ganzen Weg von Knoten i zu seiner Wurzel $r(i)$ ablaufen; Aufwand $O(\mathit{depth}(i))$.

Ein gutes strategisches Ziel ist also, diese Wege möglichst kurz zu halten.

Idee: Man investiert bei **find**(i) etwas mehr Arbeit, jedoch immer noch im Rahmen $O(\mathit{depth}(i))$, um dabei einige **Wege zu verkürzen** und damit spätere **find**s billiger zu machen.

Jeder Knoten $i = i_0, i_1, \ldots, i_{d-1}$ auf dem Weg von i zur Wurzel $i_d = r(i)$ wird direkt als Kind an die Wurzel gehängt.
Kosten pro Knoten/Ebene: $O(1)$, insgesamt also $O(\mathit{depth}(i))$.

Beispiel: $i_0 = i$ mit $d = \mathit{depth}(i) = 4$.



Die neue Version der **find**-Operation:

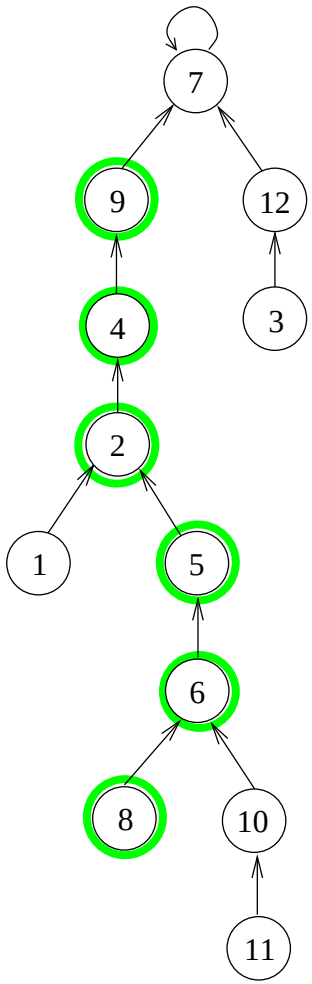
Prozedur find(i) // Pfadkompressions-Version

Die neue Version der **find**-Operation:

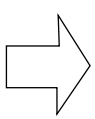
Prozedur find(i) // Pfadkompressions-Version

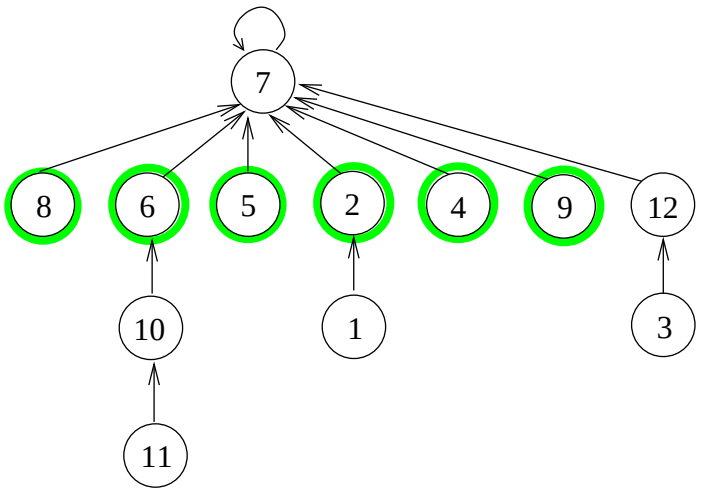
```
(1)   j ← i;
      // verfolge Vorgängerzeiger bis zur Wurzel r(i):
(2)   jj ← p[j];
(3)   while jj ≠ j do
(4)     j ← jj;
(5)     jj ← p[j] ;
(6)   r ← j;   // r enthält nun Wurzel r(i)
      // Erneuter Lauf zur Wurzel, Vorgängerzeiger umhängen:
(7)   j ← i;
(8)   jj ← p[j];
(9)   while jj ≠ j do
(10)    p[j] ← r;
(11)    j ← jj;
(12)    jj ← p[j];
(13)  return r.
```

Beispiel:



find (8)





PAUSE

Es folgt: Analyse der Pfadkompression, Details nicht prüfungsrelevant.

Analyse: Interessant, **aber nicht prüfungsrelevant (2020)**.

Analyse: Interessant, **aber nicht prüfungsrelevant (2020)**.

Prüfungsrelevant ist nur das Ergebnis, also Satz 11.4.5.

Analyse: Interessant, **aber nicht prüfungsrelevant (2020)**.

Prüfungsrelevant ist nur das Ergebnis, also Satz 11.4.5.

Die **union**-Operationen werden exakt wie bisher durchgeführt.

Analyse: Interessant, **aber nicht prüfungsrelevant (2020)**.

Prüfungsrelevant ist nur das Ergebnis, also Satz 11.4.5.

Die **union**-Operationen werden exakt wie bisher durchgeführt.

Beobachtungen: (1) Die Werte im `rank`-Array werden aktualisiert wie vorher.

Analyse: Interessant, **aber nicht prüfungsrelevant (2020)**.

Prüfungsrelevant ist nur das Ergebnis, also Satz 11.4.5.

Die **union**-Operationen werden exakt wie bisher durchgeführt.

Beobachtungen: (1) Die Werte im `rank`-Array werden aktualisiert wie vorher.
Man kann sie aber nicht mehr als Baumtiefen interpretieren.

Analyse: Interessant, **aber nicht prüfungsrelevant (2020)**.

Prüfungsrelevant ist nur das Ergebnis, also Satz 11.4.5.

Die **union**-Operationen werden exakt wie bisher durchgeführt.

Beobachtungen: (1) Die Werte im `rank`-Array werden aktualisiert wie vorher.
Man kann sie aber nicht mehr als Baumtiefen interpretieren.

(2) Nach (1) fällt bei einer **union**-Operationen die Entscheidung, welcher Knoten Kind eines anderen wird, exakt wie im Verfahren ohne Pfadkompression. Daher verändert die Pfadkompression die Knotenmengen der Bäume (also die Klassen) und die Repräsentanten nicht.

Analyse: Interessant, **aber nicht prüfungsrelevant (2020)**.

Prüfungsrelevant ist nur das Ergebnis, also Satz 11.4.5.

Die **union**-Operationen werden exakt wie bisher durchgeführt.

Beobachtungen: (1) Die Werte im `rank`-Array werden aktualisiert wie vorher.
Man kann sie aber nicht mehr als Baumtiefen interpretieren.

(2) Nach (1) fällt bei einer **union**-Operationen die Entscheidung, welcher Knoten Kind eines anderen wird, exakt wie im Verfahren ohne Pfadkompression. Daher verändert die Pfadkompression die Knotenmengen der Bäume (also die Klassen) und die Repräsentanten nicht.

Aus (1) und (2) folgt, dass **Fakt 2** weiterhin gilt.

Analyse: Interessant, **aber nicht prüfungsrelevant (2020)**.

Prüfungsrelevant ist nur das Ergebnis, also Satz 11.4.5.

Die **union**-Operationen werden exakt wie bisher durchgeführt.

Beobachtungen: (1) Die Werte im `rank`-Array werden aktualisiert wie vorher.
Man kann sie aber nicht mehr als Baumtiefen interpretieren.

(2) Nach (1) fällt bei einer **union**-Operationen die Entscheidung, welcher Knoten Kind eines anderen wird, exakt wie im Verfahren ohne Pfadkompression. Daher verändert die Pfadkompression die Knotenmengen der Bäume (also die Klassen) und die Repräsentanten nicht.

Aus (1) und (2) folgt, dass **Fakt 2** weiterhin gilt.

Bem.: Nicht-Wurzeln vom Rang h können – weil sie bei der Pfadkompression Kindknoten verlieren können – weniger als 2^h Nachfahren haben.

(3) Wenn ein Knoten i Kind eines anderen wird (also aufhört, Repräsentant zu sein), dann **ändert** sich sein **Rang**, wie in `rank`$[i]$ gespeichert, **nie mehr**. Dies gilt in der Version ohne und in der Version mit Pfadkompression.

Diesen Wert werden wir als den „endgültigen“ Rang von i ansehen.

(3) Wenn ein Knoten i Kind eines anderen wird (also aufhört, Repräsentant zu sein), dann **ändert** sich sein **Rang**, wie in `rank`$[i]$ gespeichert, **nie mehr**. Dies gilt in der Version ohne und in der Version mit Pfadkompression.

Diesen Wert werden wir als den „endgültigen“ Rang von i ansehen.

Die Rang-Werte der Nicht-Wurzeln sind dann in der Version mit und in der Version ohne Pfadkompression **identisch**.

Solange ein Knoten Wurzel ist, ist sein Rang nicht endgültig, aber in beiden Versionen gleich.

Insbesondere: **Fakt 3** gilt weiterhin.

(3) Wenn ein Knoten i Kind eines anderen wird (also aufhört, Repräsentant zu sein), dann **ändert** sich sein **Rang**, wie in `rank`$[i]$ gespeichert, **nie mehr**. Dies gilt in der Version ohne und in der Version mit Pfadkompression.

Diesen Wert werden wir als den „endgültigen“ Rang von i ansehen.

Die Rang-Werte der Nicht-Wurzeln sind dann in der Version mit und in der Version ohne Pfadkompression **identisch**.

Solange ein Knoten Wurzel ist, ist sein Rang nicht endgültig, aber in beiden Versionen gleich.

Insbesondere: **Fakt 3** gilt weiterhin.

(4) Weder **union**- noch **find**-Operationen (mit Pfadkompression) ändern etwas an **Fakt 1**: Ränge wachsen strikt von unten nach oben entlang der Wege in den Bäumen.
(Beweis durch Induktion über ausgeführte Operationen, Übung.)

Definition: $F(0) := 1, \qquad F(i) := 2^{F(i-1)}$, für $i \geq 1$.

$F(i)$ ist die Zahl, die von einem „Zweierpotenzturm“ der Höhe i berechnet wird:

$$F(i) = \; 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \; i \text{ Zweien}$$

Definition: $F(0) := 1, \qquad F(i) := 2^{F(i-1)}$, für $i \geq 1$.

$F(i)$ ist die Zahl, die von einem „Zweierpotenzturm“ der Höhe i berechnet wird:

$$F(i) = \; 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \; i \text{ Zweien}$$

Beispielwerte:

i	0	1	2	3	4	5	6
$F(i)$	1	2	4	16	65536	2^{65536}	$2^{2^{65536}}$

Definition: $F(0) := 1, \qquad F(i) := 2^{F(i-1)}$, für $i \geq 1$.

$F(i)$ ist die Zahl, die von einem „Zweierpotenzturm“ der Höhe i berechnet wird:

$$F(i) = \; 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \quad i \text{ Zweien}$$

Beispielwerte:

i	0	1	2	3	4	5	6
$F(i)$	1	2	4	16	65536	2^{65536}	$2^{2^{65536}}$

Definition: $\log^* n := \min\{k \mid F(k) \geq n\}$.

Definition: $F(0) := 1, \qquad F(i) := 2^{F(i-1)}$, für $i \geq 1$.

$F(i)$ ist die Zahl, die von einem „Zweierpotenzturm“ der Höhe i berechnet wird:

$$F(i) = \; 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \; i \text{ Zweien}$$

Beispielwerte:

i	0	1	2	3	4	5	6
$F(i)$	1	2	4	16	65536	2^{65536}	$2^{2^{65536}}$

Definition: $\log^* n := \min\{k \mid F(k) \geq n\}$.

Leicht zu sehen: $\log^* n = \min\{k \mid \underbrace{\log \log \ldots \log}_{k} n \leq 1\}$.

Nun teilen wir die Knoten j mit (endgültigem) Rang > 0 (keine Blätter, keine Wurzeln) in **„Ranggruppen“** $G_0, G_1, G_2, \ldots$ ein:

Nun teilen wir die Knoten j mit (endgültigem) Rang > 0 (keine Blätter, keine Wurzeln) in **„Ranggruppen“** $G_0, G_1, G_2, \ldots$ ein:

Knoten j gehört zu Ranggruppe G_k, wenn $F(k-1) < \mathit{rank}(j) \leq F(k)$ gilt.

Nun teilen wir die Knoten j mit (endgültigem) Rang > 0 (keine Blätter, keine Wurzeln) in **„Ranggruppen“** $G_0, G_1, G_2, \ldots$ ein:

Knoten j gehört zu Ranggruppe G_k, wenn $F(k-1) < \mathit{rank}(j) \leq F(k)$ gilt.

G_0: Rang 1; G_1: Rang 2; G_2: Ränge $3, 4$; G_3: Ränge $5, \ldots, 16$; G_4: Ränge $17, \ldots, 65536$; etc.

Nun teilen wir die Knoten j mit (endgültigem) Rang > 0 (keine Blätter, keine Wurzeln) in **„Ranggruppen“** $G_0, G_1, G_2, \ldots$ ein:

Knoten j gehört zu Ranggruppe G_k, wenn $F(k-1) < \mathit{rank}(j) \leq F(k)$ gilt.

G_0: Rang 1; G_1: Rang 2; G_2: Ränge $3, 4$; G_3: Ränge $5, \ldots, 16$; G_4: Ränge $17, \ldots, 65536$; etc.

Wenn j in Ranggruppe G_k ist, folgt (mit Fakt 3):

Nun teilen wir die Knoten j mit (endgültigem) Rang > 0 (keine Blätter, keine Wurzeln) in **„Ranggruppen“** $G_0, G_1, G_2, \ldots$ ein:

Knoten j gehört zu Ranggruppe G_k, wenn $F(k-1) < \mathit{rank}(j) \leq F(k)$ gilt.

G_0: Rang 1; G_1: Rang 2; G_2: Ränge $3, 4$; G_3: Ränge $5, \ldots, 16$; G_4: Ränge $17, \ldots, 65536$; etc.

Wenn j in Ranggruppe G_k ist, folgt (mit Fakt 3):

$F(k) = 2^{F(k-1)} < 2^{\mathit{rank}(j)} \leq 2^{\log n} = n.$

Nun teilen wir die Knoten j mit (endgültigem) Rang > 0 (keine Blätter, keine Wurzeln) in **„Ranggruppen“** $G_0, G_1, G_2, \ldots$ ein:

Knoten j gehört zu Ranggruppe G_k, wenn $F(k-1) < \mathit{rank}(j) \leq F(k)$ gilt.

G_0: Rang 1; G_1: Rang 2; G_2: Ränge $3, 4$; G_3: Ränge $5, \ldots, 16$; G_4: Ränge $17, \ldots, 65536$; etc.

Wenn j in Ranggruppe G_k ist, folgt (mit Fakt 3):

$F(k) = 2^{F(k-1)} < 2^{\mathit{rank}(j)} \leq 2^{\log n} = n.$

Also: $k < \log^* n$, d. h. es gibt maximal $\log^* n$ nichtleere Ranggruppen.

Nun teilen wir die Knoten j mit (endgültigem) Rang > 0 (keine Blätter, keine Wurzeln) in **„Ranggruppen“** $G_0, G_1, G_2, \ldots$ ein:

Knoten j gehört zu Ranggruppe G_k, wenn $F(k-1) < \mathit{rank}(j) \leq F(k)$ gilt.

G_0: Rang 1; G_1: Rang 2; G_2: Ränge $3, 4$; G_3: Ränge $5, \ldots, 16$; G_4: Ränge $17, \ldots, 65536$; etc.

Wenn j in Ranggruppe G_k ist, folgt (mit Fakt 3):

$F(k) = 2^{F(k-1)} < 2^{\mathit{rank}(j)} \leq 2^{\log n} = n.$

Also: $k < \log^* n$, d. h. es gibt maximal $\log^* n$ nichtleere Ranggruppen.

Beachte: $2^{65536} > 1000^{6553} > 10^{19500}$; Werte n mit $\log^* n > 5$, in denen also Ranggruppe G_5 eventuell nicht leer ist, kommen in wirklichen algorithmischen Anwendungen nicht vor $\Rightarrow$ Man wird nie mehr als fünf nichtleere Ranggruppen sehen.

Nun teilen wir die Knoten j mit (endgültigem) Rang > 0 (keine Blätter, keine Wurzeln) in **„Ranggruppen“** $G_0, G_1, G_2, \ldots$ ein:

Knoten j gehört zu Ranggruppe G_k, wenn $F(k-1) < \mathit{rank}(j) \leq F(k)$ gilt.

G_0: Rang 1; G_1: Rang 2; G_2: Ränge $3, 4$; G_3: Ränge $5, \ldots, 16$; G_4: Ränge $17, \ldots, 65536$; etc.

Wenn j in Ranggruppe G_k ist, folgt (mit Fakt 3):

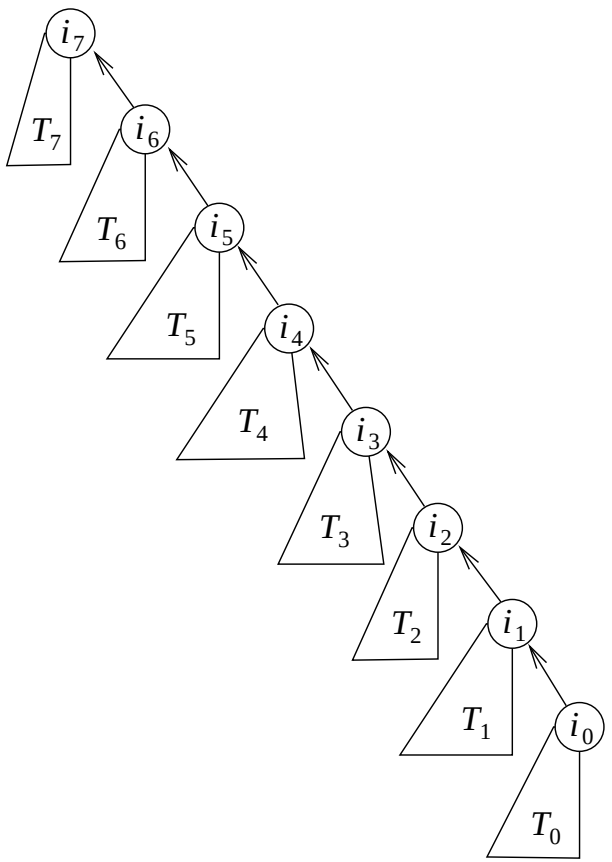
$F(k) = 2^{F(k-1)} < 2^{\mathit{rank}(j)} \leq 2^{\log n} = n.$

Also: $k < \log^* n$, d. h. es gibt maximal $\log^* n$ nichtleere Ranggruppen.

Beachte: $2^{65536} > 1000^{6553} > 10^{19500}$; Werte n mit $\log^* n > 5$, in denen also Ranggruppe G_5 eventuell nicht leer ist, kommen in wirklichen algorithmischen Anwendungen nicht vor $\Rightarrow$ Man wird nie mehr als fünf nichtleere Ranggruppen sehen.

(Es gilt aber: $\lim_{n\to\infty} \log^* n = \infty$.)

Wir wollen nun die Kosten von m **find**-Operationen berechnen. Bei **find**(i) läuft man einen Weg $i = i_0, i_1 = p(i_0), i_2 = p(i_1), \ldots, i_d = p(i_{d-1})$ entlang, von i bis zur Wurzel $r(i) = i_d$.
Beispiel: $d = 7$.



Wir veranschlagen Kosten $d + 1$ für diese Operation, also 1 für jeden Knoten j auf dem Weg.
Die **Rechenzeit** für alle m **find**s zusammen ist dann $O($Summe aller „Kosten“$)$.

Wir veranschlagen Kosten $d + 1$ für diese Operation, also 1 für jeden Knoten j auf dem Weg.
Die **Rechenzeit** für alle m **find**s zusammen ist dann $O(\text{Summe aller „Kosten“})$.
Für die Kostenanalyse benutzen wir die sogenannte **Bankkontomethode**, in einer anschaulichen Form.

Wir veranschlagen Kosten $d + 1$ für diese Operation, also 1 für jeden Knoten j auf dem Weg.
Die **Rechenzeit** für alle m **find**s zusammen ist dann $O(\text{Summe aller „Kosten“})$.
Für die Kostenanalyse benutzen wir die sogenannte **Bankkontomethode**, in einer anschaulichen Form.

Ein Knoten in Ranggruppe G_k bekommt $F(k)$ € Taschengeld, in dem Moment, in dem er aufhört, Repräsentant/Baumwurzel zu sein, sein Rang also endgültig feststeht.

Wir veranschlagen Kosten $d + 1$ für diese Operation, also 1 für jeden Knoten j auf dem Weg.
Die **Rechenzeit** für alle m **find**s zusammen ist dann $O(\text{Summe aller „Kosten“})$.
Für die Kostenanalyse benutzen wir die sogenannte **Bankkontomethode**, in einer anschaulichen Form.

Ein Knoten in Ranggruppe G_k bekommt $F(k)$ € Taschengeld, in dem Moment, in dem er aufhört, Repräsentant/Baumwurzel zu sein, sein Rang also endgültig feststeht.

In G_k sind Knoten mit Rängen $F(k-1)+1, \ldots, F(k)$.

Wir veranschlagen Kosten $d+1$ für diese Operation, also 1 für jeden Knoten j auf dem Weg.
Die **Rechenzeit** für alle m **find**s zusammen ist dann $O(\text{Summe aller „Kosten“})$.
Für die Kostenanalyse benutzen wir die sogenannte **Bankkontomethode**, in einer anschaulichen Form.

Ein Knoten in Ranggruppe G_k bekommt $F(k)$ € Taschengeld, in dem Moment, in dem er aufhört, Repräsentant/Baumwurzel zu sein, sein Rang also endgültig feststeht.

In G_k sind Knoten mit Rängen $F(k-1)+1, \ldots, F(k)$. Nach Fakt 3 gilt:

$$|G_k| \leq \frac{n}{2^{F(k-1)+1}} + \frac{n}{2^{F(k-1)+2}} + \cdots + \frac{n}{2^{F(k)}} < \frac{n}{2^{F(k-1)}} = \frac{n}{F(k)}.$$

Wir veranschlagen Kosten $d+1$ für diese Operation, also 1 für jeden Knoten j auf dem Weg.
Die **Rechenzeit** für alle m **find**s zusammen ist dann $O(\text{Summe aller „Kosten“})$.
Für die Kostenanalyse benutzen wir die sogenannte **Bankkontomethode**, in einer anschaulichen Form.

Ein Knoten in Ranggruppe G_k bekommt $F(k)$ € Taschengeld, in dem Moment, in dem er aufhört, Repräsentant/Baumwurzel zu sein, sein Rang also endgültig feststeht.

In G_k sind Knoten mit Rängen $F(k-1)+1, \ldots, F(k)$. Nach Fakt 3 gilt:

$$|G_k| \leq \frac{n}{2^{F(k-1)+1}} + \frac{n}{2^{F(k-1)+2}} + \cdots + \frac{n}{2^{F(k)}} < \frac{n}{2^{F(k-1)}} = \frac{n}{F(k)}.$$

Also bekommen alle Knoten in G_k zusammen höchstens n €, und in allen höchstens $\log^* n$ Ranggruppen zusammen werden nicht mehr als $n \log^* n$ € als Taschengeld ausgezahlt.

Wir veranschlagen Kosten $d + 1$ für diese Operation, also 1 für jeden Knoten j auf dem Weg.
Die **Rechenzeit** für alle m **find**s zusammen ist dann $O(\text{Summe aller „Kosten“})$.
Für die Kostenanalyse benutzen wir die sogenannte **Bankkontomethode**, in einer anschaulichen Form.

Ein Knoten in Ranggruppe G_k bekommt $F(k)$ € Taschengeld, in dem Moment, in dem er aufhört, Repräsentant/Baumwurzel zu sein, sein Rang also endgültig feststeht.

In G_k sind Knoten mit Rängen $F(k-1)+1, \ldots, F(k)$. Nach Fakt 3 gilt:

$$|G_k| \leq \frac{n}{2^{F(k-1)+1}} + \frac{n}{2^{F(k-1)+2}} + \cdots + \frac{n}{2^{F(k)}} < \frac{n}{2^{F(k-1)}} = \frac{n}{F(k)}.$$

Also bekommen alle Knoten in G_k zusammen höchstens n €, und in allen höchstens $\log^* n$ Ranggruppen zusammen werden nicht mehr als $n \log^* n$ € als Taschengeld ausgezahlt.

Dies ist ein entscheidender Punkt in der Analyse: Die Ranggruppen sind sehr klein, daher kann man sich ein großzügiges Taschengeld leisten.

Weiter legen wir $m(2 + \log^* n)$ € in einer „Gemeinschaftskasse“ bereit.

Ziel: Wir wollen zeigen, dass das Taschengeld und das Geld in der Gemeinschaftskasse zusammen ausreichen, um die Kosten aller m **find**-Operationen zu bestreiten.

Weiter legen wir $m(2 + \log^* n)$ € in einer „Gemeinschaftskasse“ bereit.

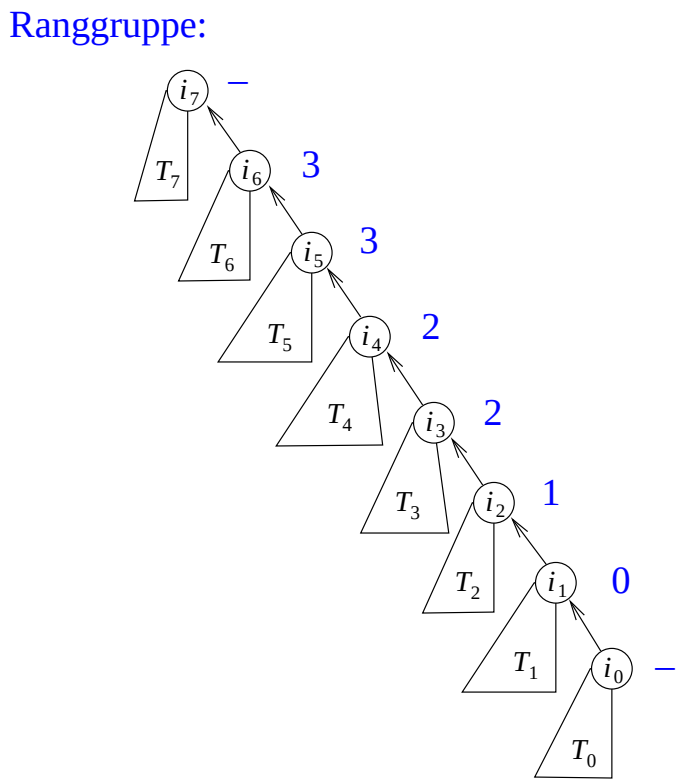
Ziel: Wir wollen zeigen, dass das Taschengeld und das Geld in der Gemeinschaftskasse zusammen ausreichen, um die Kosten aller m **find**-Operationen zu bestreiten.

Wir betrachten eine **find**(i)-Operation. Jeder Knoten j auf dem Weg verursacht Kosten 1.

Weiter legen wir $m(2 + \log^* n)$ € in einer „Gemeinschaftskasse“ bereit.

Ziel: Wir wollen zeigen, dass das Taschengeld und das Geld in der Gemeinschaftskasse zusammen ausreichen, um die Kosten aller m **find**-Operationen zu bestreiten.

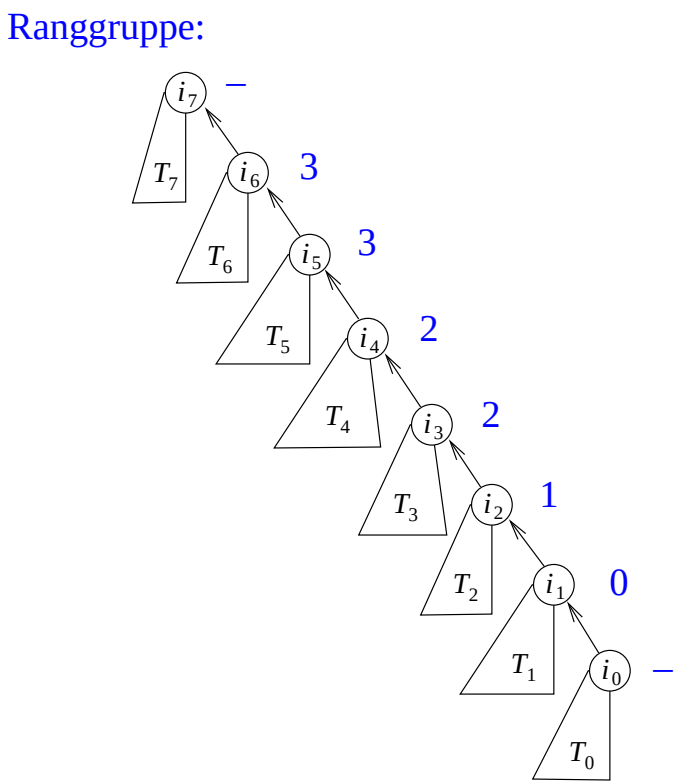
Wir betrachten eine **find**(i)-Operation. Jeder Knoten j auf dem Weg verursacht Kosten 1.



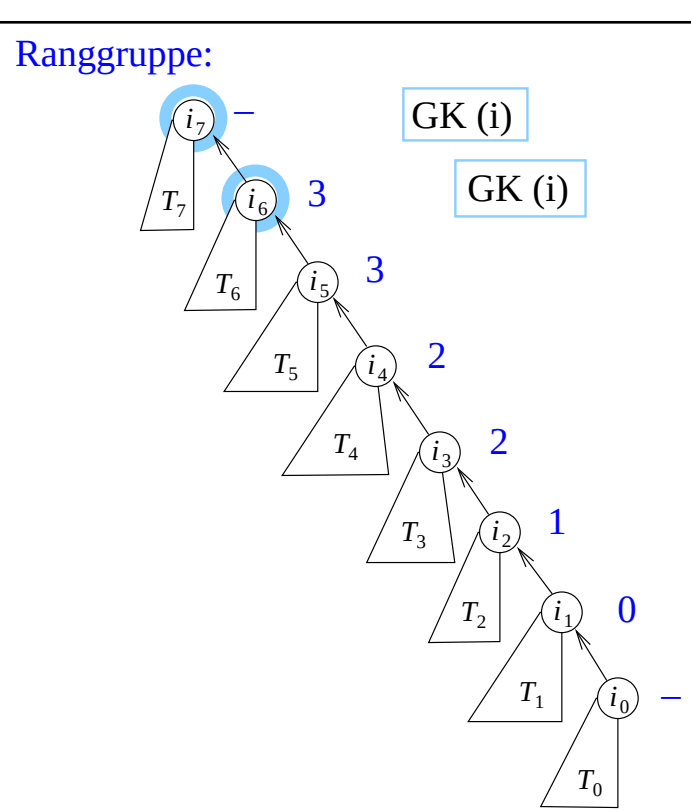
Weiter legen wir $m(2 + \log^* n)$ € in einer „Gemeinschaftskasse“ bereit.

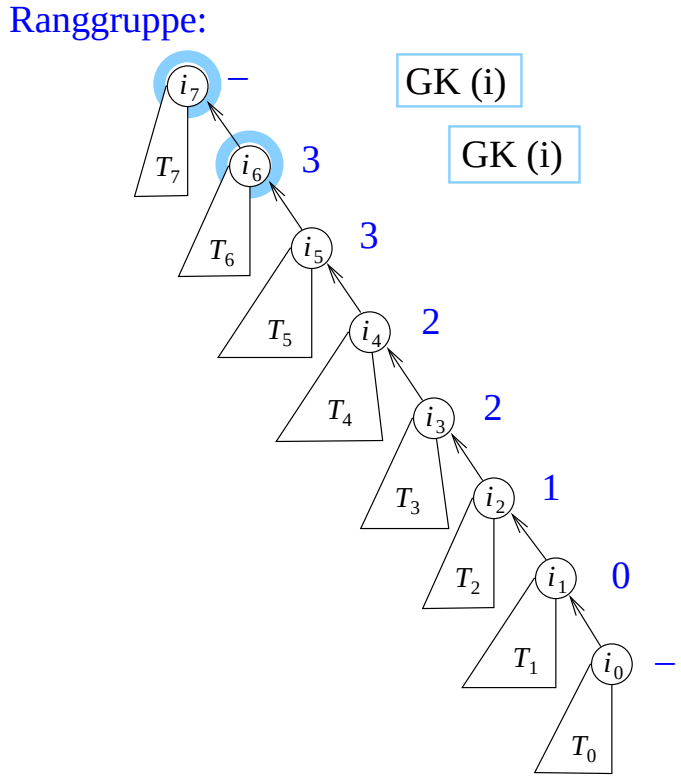
Ziel: Wir wollen zeigen, dass das Taschengeld und das Geld in der Gemeinschaftskasse zusammen ausreichen, um die Kosten aller m **find**-Operationen zu bestreiten.

Wir betrachten eine **find**(i)-Operation. Jeder Knoten j auf dem Weg verursacht Kosten 1.

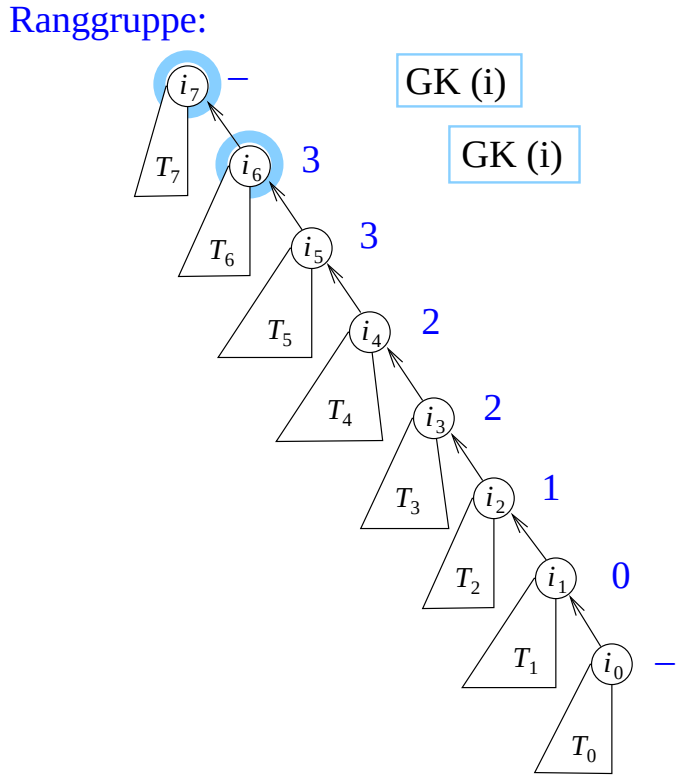


Es gibt drei Fälle.

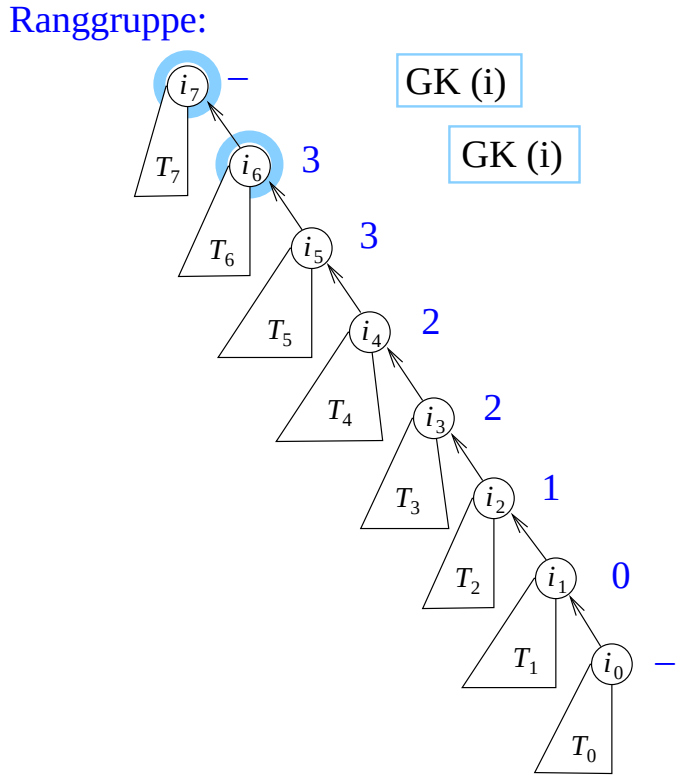

Ranggruppe:
GK (i)
GK (i)
i_7
i_6
i_5
i_4
i_3
i_2
i_1
i_0
T_7
T_6
T_5
T_4
T_3
T_2
T_1
T_0
–
3
3
2
2
1
0
–



(i) j ist **Wurzel** oder ist der **vorletzte** Knoten auf dem Weg von i nach $r(i)$.
Die Kosten von j bestreiten wir aus der Gemeinschaftskasse.

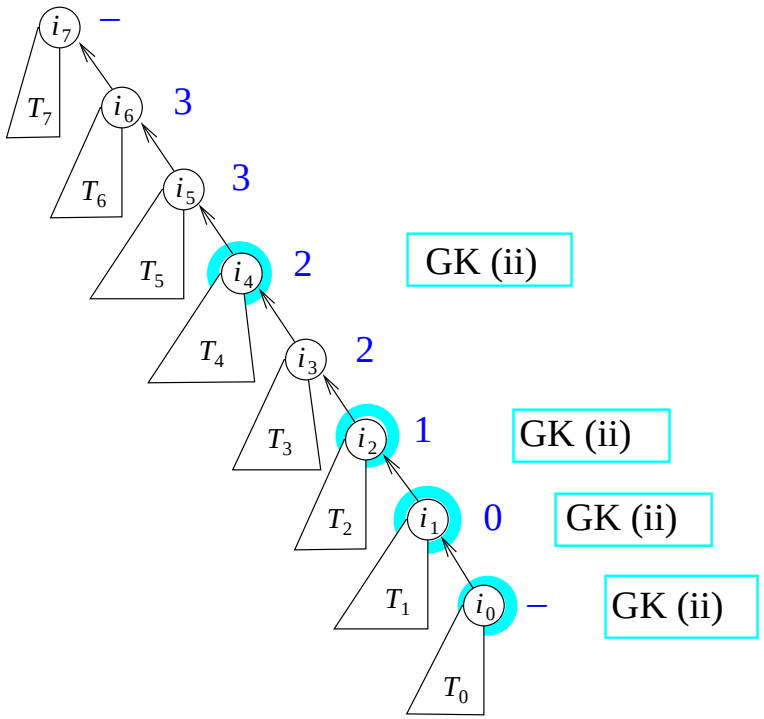


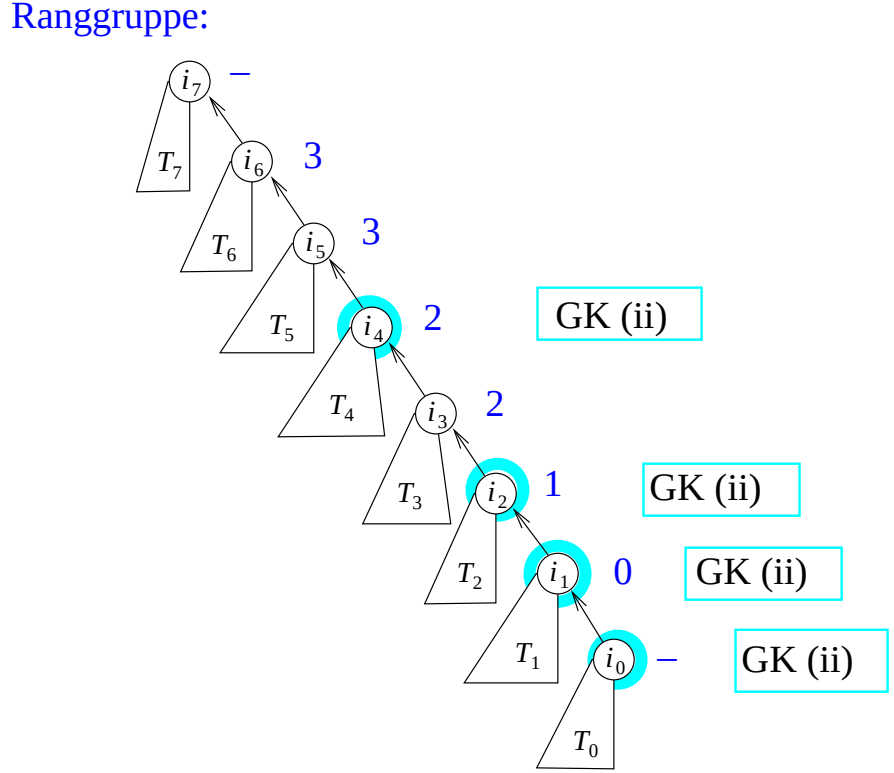
(i) j ist **Wurzel** oder ist der **vorletzte** Knoten auf dem Weg von i nach $r(i)$.
Die Kosten von j bestreiten wir aus der Gemeinschaftskasse.
Dieser Fall kostet maximal 2€ für die **find**(i)-Operation.



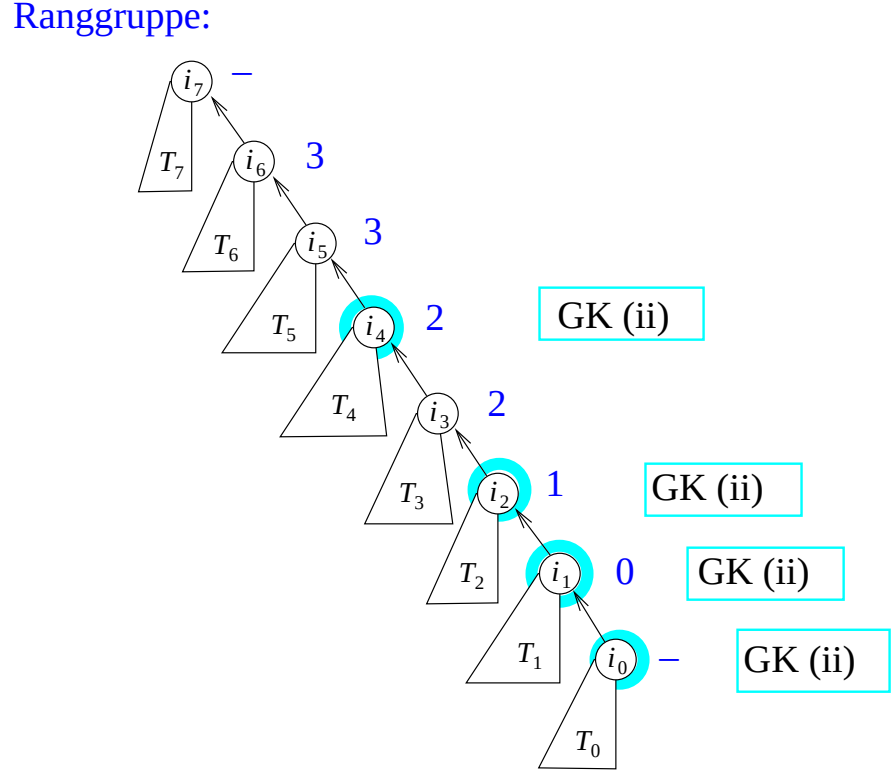
(i) j ist **Wurzel** oder ist der **vorletzte** Knoten auf dem Weg von i nach $r(i)$.
Die Kosten von j bestreiten wir aus der Gemeinschaftskasse.
Dieser Fall kostet maximal 2€ für die **find**(i)-Operation.
Ab hier: j ist nicht Wurzel oder vorletzter Knoten.

Ranggruppe:


i_7
–
T_7
i_6
3
T_6
i_5
3
T_5
i_4
2
GK (ii)
T_4
i_3
2
T_3
i_2
1
GK (ii)
T_2
i_1
0
GK (ii)
T_1
i_0
–
GK (ii)
T_0

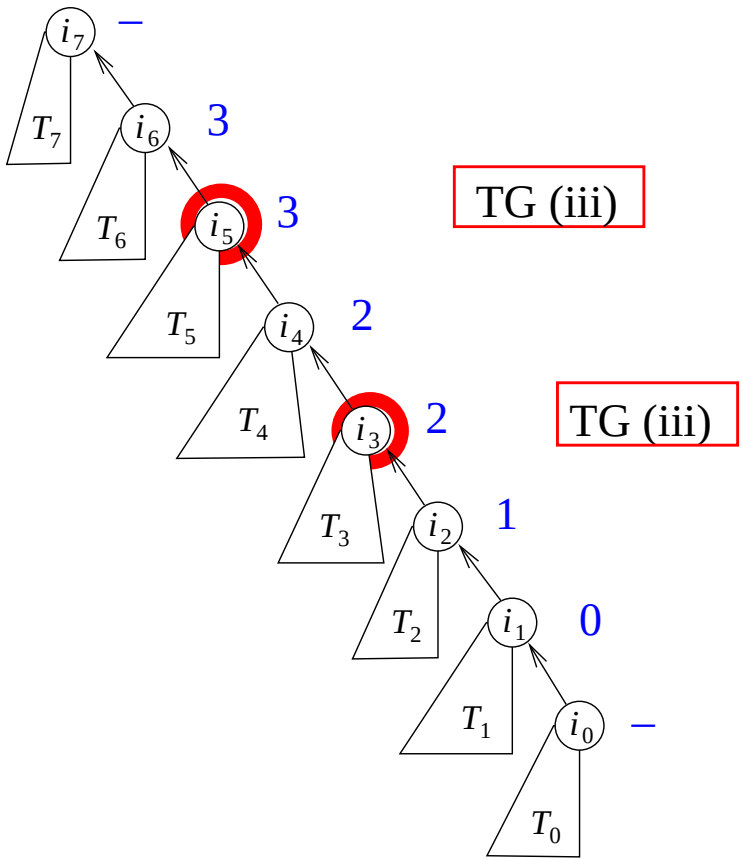


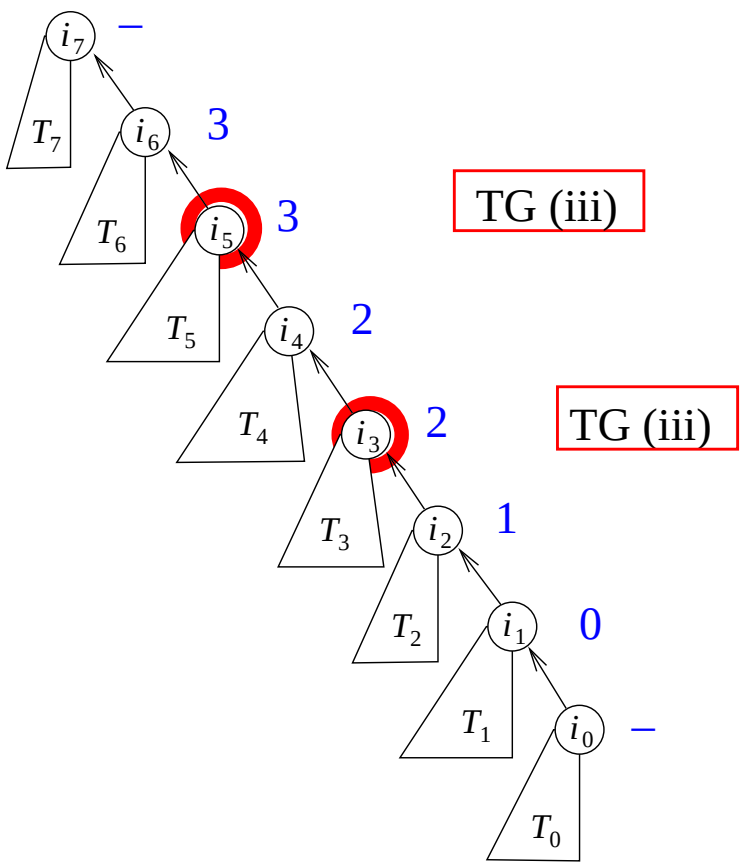
(ii) $\mathit{rank}(j) = 0$ oder $p(j)$ ist in einer höheren Ranggruppe als j.
Es gibt höchstens $\log^* n$ viele solche j's, weil Ränge entlang des Weges strikt wachsen (Fakt 1).



(ii) *rank*$(j) = 0$ oder $p(j)$ ist in einer höheren Ranggruppe als j.
Es gibt höchstens $\log^* n$ viele solche j's, weil Ränge entlang des Weges strikt wachsen (Fakt 1).
Kosten dieser j sind $\leq \log^* n$ €, werden aus Gemeinschaftskasse bezahlt.

Ranggruppe:





(iii) $p(j)$ ist in derselben Ranggruppe wie j.
In diesem Fall muss j mit $1\,€$ aus seinem Taschengeld bezahlen.

Nun wird abgerechnet:

(A) Die Gesamtkosten aus Fällen (i) und (ii), summiert über alle m **find**s, betragen höchstens $m(2 + \log^* n)$; der Inhalt der Gemeinschaftskasse reicht.

Nun wird abgerechnet:

(A) Die Gesamtkosten aus Fällen (i) und (ii), summiert über alle m **find**s, betragen höchstens $m(2 + \log^* n)$; der Inhalt der Gemeinschaftskasse reicht.

(B) Wie steht es mit Fall (iii) und dem Taschengeld?

Nun wird abgerechnet:

(A) Die Gesamtkosten aus Fällen (i) und (ii), summiert über alle m **find**s, betragen höchstens $m(2 + \log^* n)$; der Inhalt der Gemeinschaftskasse reicht.

(B) Wie steht es mit Fall (iii) und dem Taschengeld?

Wir betrachten nun **(Trick!)** einen festen Knoten j.

Nun wird abgerechnet:

(A) Die Gesamtkosten aus Fällen (i) und (ii), summiert über alle m **find**s, betragen höchstens $m(2 + \log^* n)$; der Inhalt der Gemeinschaftskasse reicht.

(B) Wie steht es mit Fall (iii) und dem Taschengeld?

Wir betrachten nun **(Trick!)** einen festen Knoten j.

Immer wenn j bezahlen muss, nimmt er an einer Pfadverkürzung teil. Sein neuer Vorgängerknoten $p'(j)$ wird seine aktuelle Wurzel (verschieden vom bisherigen Vorgänger $p(j)$).

Nun wird abgerechnet:

(A) Die Gesamtkosten aus Fällen (i) und (ii), summiert über alle m **find**s, betragen höchstens $m(2 + \log^* n)$; der Inhalt der Gemeinschaftskasse reicht.

(B) Wie steht es mit Fall (iii) und dem Taschengeld?

Wir betrachten nun **(Trick!)** einen festen Knoten j.

Immer wenn j bezahlen muss, nimmt er an einer Pfadverkürzung teil. Sein neuer Vorgängerknoten $p'(j)$ wird seine aktuelle Wurzel (verschieden vom bisherigen Vorgänger $p(j)$).

Wegen Fakt 1 hat der **neue Vorgänger** $p'(j)$ einen **höheren Rang** als der **alte**.

Nun wird abgerechnet:

(A) Die Gesamtkosten aus Fällen (i) und (ii), summiert über alle m **find**s, betragen höchstens $m(2 + \log^* n)$; der Inhalt der Gemeinschaftskasse reicht.

(B) Wie steht es mit Fall (iii) und dem Taschengeld?

Wir betrachten nun **(Trick!)** einen festen Knoten j.

Immer wenn j bezahlen muss, nimmt er an einer Pfadverkürzung teil. Sein neuer Vorgängerknoten $p'(j)$ wird seine aktuelle Wurzel (verschieden vom bisherigen Vorgänger $p(j)$).

Wegen Fakt 1 hat der **neue Vorgänger** $p'(j)$ einen **höheren Rang** als der **alte**.

Sei G_k die Ranggruppe von j. In dieser Ranggruppe gibt es nicht mehr als $F(k)$ verfügbare Rang-Werte. Daher ändert sich der Vorgänger von j weniger als $F(k)$-mal, bevor ein Knoten aus einer höheren Ranggruppe Vorgänger von j wird (und alle späteren Vorgänger ebenfalls aus höheren Ranggruppen kommen).

Nun wird abgerechnet:

(A) Die Gesamtkosten aus Fällen (i) und (ii), summiert über alle m **find**s, betragen höchstens $m(2 + \log^* n)$; der Inhalt der Gemeinschaftskasse reicht.

(B) Wie steht es mit Fall (iii) und dem Taschengeld?

Wir betrachten nun **(Trick!)** einen festen Knoten j.

Immer wenn j bezahlen muss, nimmt er an einer Pfadverkürzung teil. Sein neuer Vorgängerknoten $p'(j)$ wird seine aktuelle Wurzel (verschieden vom bisherigen Vorgänger $p(j)$).

Wegen Fakt 1 hat der **neue Vorgänger** $p'(j)$ einen **höheren Rang** als der **alte**.

Sei G_k die Ranggruppe von j. In dieser Ranggruppe gibt es nicht mehr als $F(k)$ verfügbare Rang-Werte. Daher ändert sich der Vorgänger von j weniger als $F(k)$-mal, bevor ein Knoten aus einer höheren Ranggruppe Vorgänger von j wird (und alle späteren Vorgänger ebenfalls aus höheren Ranggruppen kommen).

Also tritt Situation (iii) für Knoten j weniger als $F(k)$-mal ein. Daher reicht das Taschengeld von Knoten j aus, um für diese Situationen zu bezahlen.

Nun wird abgerechnet:

(A) Die Gesamtkosten aus Fällen (i) und (ii), summiert über alle m **find**s, betragen höchstens $m(2 + \log^* n)$; der Inhalt der Gemeinschaftskasse reicht.

(B) Wie steht es mit Fall (iii) und dem Taschengeld?

Wir betrachten nun **(Trick!)** einen festen Knoten j.

Immer wenn j bezahlen muss, nimmt er an einer Pfadverkürzung teil. Sein neuer Vorgängerknoten $p'(j)$ wird seine aktuelle Wurzel (verschieden vom bisherigen Vorgänger $p(j)$).

Wegen Fakt 1 hat der **neue Vorgänger** $p'(j)$ einen **höheren Rang** als der **alte**.

Sei G_k die Ranggruppe von j. In dieser Ranggruppe gibt es nicht mehr als $F(k)$ verfügbare Rang-Werte. Daher ändert sich der Vorgänger von j weniger als $F(k)$-mal, bevor ein Knoten aus einer höheren Ranggruppe Vorgänger von j wird (und alle späteren Vorgänger ebenfalls aus höheren Ranggruppen kommen).

Also tritt Situation (iii) für Knoten j weniger als $F(k)$-mal ein. Daher reicht das Taschengeld von Knoten j aus, um für diese Situationen zu bezahlen.

Die Gesamtkosten aus Fall (iii), summiert über alle j's, betragen also nicht mehr als das gesamte Taschengeld, also höchstens $n \log^* n$.

Satz 11.4.4

In der Implementierung der Union-Find-Struktur mit wurzelgerichteten Wäldern und Pfadkompression haben m **find**-Operationen insgesamt Kosten von maximal $(2 + m + n) \log^* n$, benötigen also Rechenzeit $O((m + n) \log^* n)$.
Jede einzelne **union**-Operation benötigt Zeit $O(1)$.

Satz 11.4.4

In der Implementierung der Union-Find-Struktur mit wurzelgerichteten Wäldern und Pfadkompression haben m **find**-Operationen insgesamt Kosten von maximal $(2 + m + n) \log^* n$, benötigen also Rechenzeit $O((m + n) \log^* n)$.
Jede einzelne **union**-Operation benötigt Zeit $O(1)$.

Bemerkung:
(a) Da für real vorkommende n die Zahl $\log^* n$ nicht größer als 5 ist, wird man in der Praxis eine Rechenzeit beobachten, die linear in $n + m$ ist.

Satz 11.4.4

In der Implementierung der Union-Find-Struktur mit wurzelgerichteten Wäldern und Pfadkompression haben m **find**-Operationen insgesamt Kosten von maximal $(2 + m + n) \log^* n$, benötigen also Rechenzeit $O((m + n) \log^* n)$.
Jede einzelne **union**-Operation benötigt Zeit $O(1)$.

Bemerkung:
(a) Da für real vorkommende n die Zahl $\log^* n$ nicht größer als 5 ist, wird man in der Praxis eine Rechenzeit beobachten, die linear in $n + m$ ist.
(b) Implementierungen von Union-Find-Strukturen als wurzelgerichteter Wald mit Pfadkompression verhalten sich in der Praxis sehr effizient.

Algorithmus von Kruskal mit Union-Find

Input: Gewichteter zusammenhängender Graph $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$.

1. Schritt: Sortiere die Kanten $e_1, \ldots, e_m$ nach den Gewichten $c_1 = c(e_1), \ldots, c_m = c(e_m)$ aufsteigend.

Resultat: Sortierte Kantenliste $(v_1, w_1, c_1), \ldots, (v_m, w_m, c_m)$.

2. Schritt: Initialisiere Union-Find-Struktur für $\{1, \ldots, n\}$.

3. Schritt: Für $i = 1, 2, \ldots, m$ tue folgendes:

$\mathtt{s} \leftarrow$ **find**(v_i); $\mathtt{t} \leftarrow$ **find**(w_i);

if $\mathtt{s} \neq \mathtt{t}$ **then begin** $\mathtt{R} \leftarrow \mathtt{R} \cup \{e_i\}$; **union**$(\mathtt{s}, \mathtt{t})$ **end**;

// Optional: Beende Schleife, wenn $|\mathtt{R}| = n - 1$.

4. Schritt: Die Ausgabe ist R.

Satz 11.4.1 (Vollversion)

(a) Der Algorithmus von Kruskal in der Implementierung mit Union-Find ist korrekt.

(b) Die Rechenzeit des Algorithmus ist $O(m \log m) + O(m) + O(n \log n)$, also $O(m \log n)$, wenn man die Union-Find-Struktur mit **Arrays** implementiert.

(c) Die Rechenzeit des Algorithmus ist $O(m \log m) + O(m \log^* n)$, also $O(m \log n)$, wenn die Union-Find-Struktur mit **wurzelgerichteten Bäumen** mit **Pfadkompression** implementiert wird.

Beweis: (a) haben wir schon gesehen.

Beweis: (a) haben wir schon gesehen.

(b), (c): Der erste Term $O(m \log m)$ benennt die Kosten für das Sortieren. Danach sind $2m$ **find**-Operationen und $n-1$ **union**-Operationen durchzuführen. Die Zeiten hierfür in beiden Implementierungen wurden in den Sätzen 11.4.2, 11.4.3, 11.4.4 begründet. □

Bemerkung

Wenn die Kanten schon sortiert sind oder billig sortiert werden können ($O(m)$ Zeit, z. B. mittels Radixsort, siehe Abschn. 6.7), und $G = (V, E)$ sehr wenige Kanten hat, ergeben sich noch günstigere Rechenzeiten:

- Union-Find mit Arrays und Listen liefert Rechenzeit $O(m + n \log n)$: das ist linear, sobald $m = \Omega(n \log n)$ ist, also für nicht ganz dünn besetzte Graphen;
- Union-Find mit Pfadkompression liefert Rechenzeit $O(m \log^* n)$: fast linear.

Beachte aber: Für $m = \Omega(n \log n)$ bietet der Algorithmus von Jarník/Prim mit Fibonacci-Heaps lineare Rechenzeit ohne Annahmen über die Sortierkosten.

PAUSE

Ende von Kapitel 11.