Karlsruher Institut für Technologie

# Algorithmen I
# Tutorium 33

Woche 11 | 06. Juli 2018

**Daniel** Jungkind (daniel.jungkind@student.kit.edu)

INSTITUT FÜR THEORETISCHE INFORMATIK

# Inhalt

Spannbäume

Jarník-Prim

Kruskal

# Schwarzes Brett + Klausurinfos!

- **Achtung:** Blatt #11 hat **20 Punkte**,
  Abgabe bis **Freitag, 13.07.** um **15:00** Uhr!
- **Gesamtpunkte** der **ÜBs** stehen fest:
  Blatt #11 ist das letzte $\Rightarrow$ **130 P** insgesamt!
  $\Rightarrow$ für 25 % mind. notwendig: 32.5 P
  $\Rightarrow$ für 50 % mind. notwendig: 65 P
  $\Rightarrow$ für 75 % mind. notwendig: 97.5 P
- **Klausur** selbst am **04.09.2018** von **8–10 Uhr** (rechtzeitig anmelden!)
- Cheatsheet für Klausur **beidseitig** erlaubt! :D

# SPANNBÄUME

Spannung pur!

# Spannbäume

## Ein paar Definitionen

- Für heute: Alle Graphen $G = (V, E)$
  **ungerichtet**, **zusammenhängend**, mit **positiven** Kantengewichten
- **Spannbaum**: Baum $(V, T)$ von $G$ (also $T \subseteq E$),
  „spannt $G$ auf“ (= zusammenhängend)
- Bekannte Algorithmen, die (irgendwelche) Spannbäume bestimmen:
  - Tiefensuche
  - Breitensuche
  - Dijkstra
- Spannbaum hat **Gewicht** $\sum_{t \in T} c(t)$
- **Minimaler** Spannbaum (MST = Minimum Spanning Tree):
  Spannbaum mit **minimalem Gewicht**
- Minimales Gewicht ist **eindeutig**,
  minimaler Spannbaum jedoch i. A. **nicht**

# Spannbäume

## Ein paar Definitionen

- Für heute: Alle Graphen $G = (V, E)$
  **ungerichtet**, **zusammenhängend**, mit **positiven** Kantengewichten
- **Spannbaum**: Baum $(V, T)$ von $G$ (also $T \subseteq E$),
  „spannt $G$ auf“ (= zusammenhängend)
- Bekannte Algorithmen, die (irgendwelche) Spannbäume bestimmen:
  - Tiefensuche
  - Breitensuche
  - Dijkstra
- Spannbaum hat **Gewicht** $\sum_{t \in T} c(t)$
- **Minimaler** Spannbaum (MST = Minimum Spanning Tree):
  Spannbaum mit **minimalem Gewicht**
- Minimales Gewicht ist eindeutig,
  minimaler Spannbaum jedoch i. A. **nicht**

# Spannbäume

## Ein paar Definitionen

- Für heute: Alle Graphen $G = (V, E)$
  **ungerichtet**, **zusammenhängend**, mit **positiven** Kantengewichten
- **Spannbaum**: Baum $(V, T)$ von $G$ (also $T \subseteq E$),
  „spannt $G$ auf“ ($=$ zusammenhängend)
- Bekannte Algorithmen, die (irgendwelche) Spannbäume bestimmen:
  - Tiefensuche
  - Breitensuche
  - Dijkstra
- Spannbaum hat **Gewicht** $\sum_{t \in T} c(t)$
- **Minimaler** Spannbaum (MST = Minimum Spanning Tree):
  Spannbaum mit **minimalem Gewicht**
- Minimales Gewicht ist **eindeutig**,
  minimaler Spannbaum jedoch i. A. **nicht**

# Spannbäume

## Ein paar Definitionen

- Für heute: Alle Graphen $G = (V, E)$
  **ungerichtet**, **zusammenhängend**, mit **positiven** Kantengewichten
- **Spannbaum**: Baum $(V, T)$ von $G$ (also $T \subseteq E$),
  „spannt $G$ auf“ ($=$ zusammenhängend)
- Bekannte Algorithmen, die (irgendwelche) Spannbäume bestimmen:
  - Tiefensuche
  - Breitensuche
  - Dijkstra
- Spannbaum hat **Gewicht** $\sum_{t \in T} c(t)$
- **Minimaler** Spannbaum (MST = Minimum Spanning Tree):
  Spannbaum mit **minimalem Gewicht**
- Minimales Gewicht ist **eindeutig**,
  minimaler Spannbaum jedoch i. A. **nicht**

# Spannbäume

## Ein paar Definitionen

- Für heute: Alle Graphen $G = (V, E)$
  **ungerichtet**, **zusammenhängend**, mit **positiven** Kantengewichten
- **Spannbaum**: Baum $(V, T)$ von $G$ (also $T \subseteq E$),
  „spannt $G$ auf“ (= zusammenhängend)
- Bekannte Algorithmen, die (irgendwelche) Spannbäume bestimmen:
  - Tiefensuche
  - Breitensuche
  - Dijkstra
- Spannbaum hat **Gewicht** $\sum_{t \in T} c(t)$
- **Minimaler** Spannbaum (MST = Minimum Spanning Tree):
  Spannbaum mit **minimalem Gewicht**
- Minimales Gewicht ist eindeutig,
  minimaler Spannbaum jedoch i. A. nicht

# Spannbäume

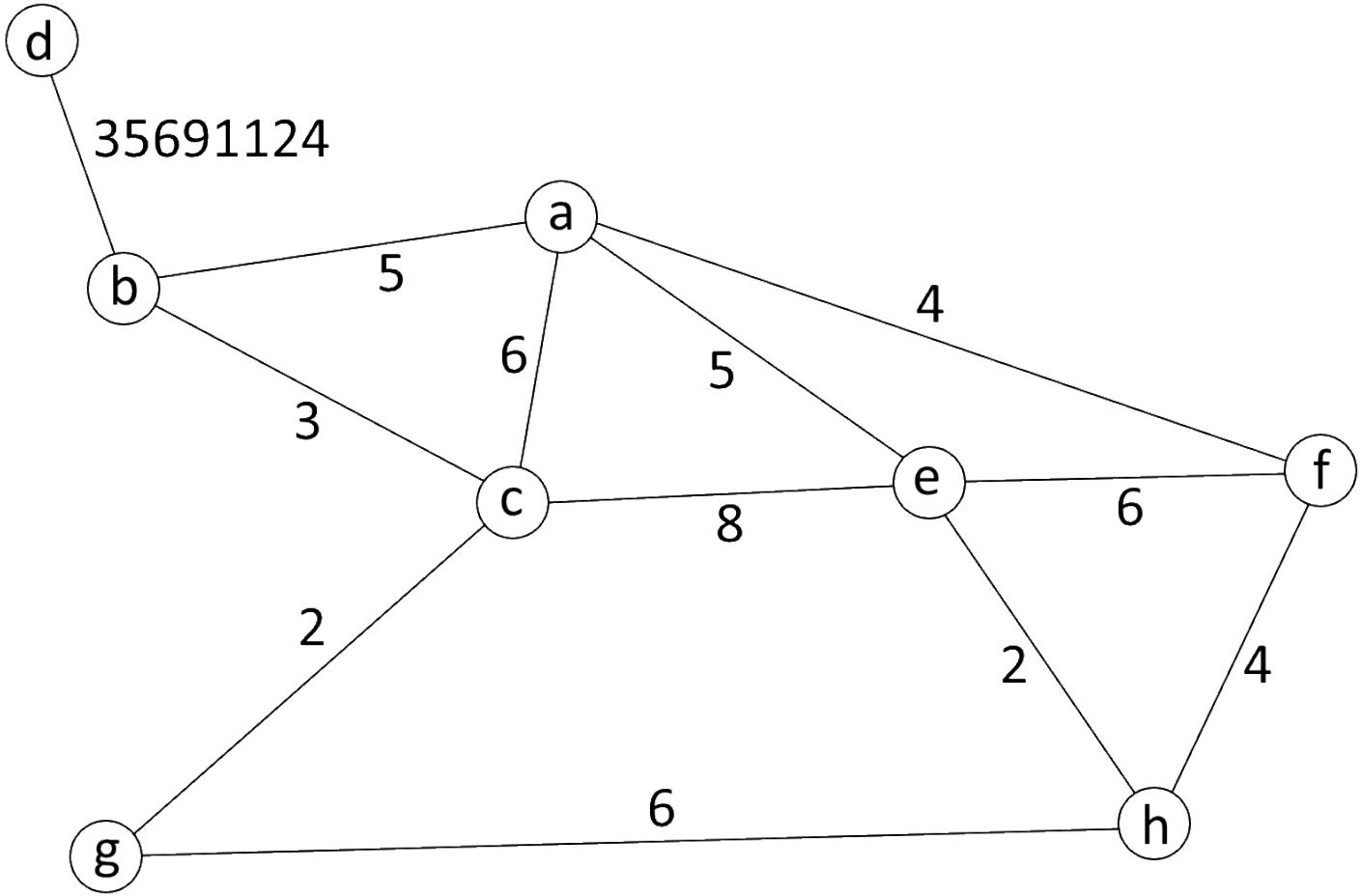
## Ein paar Definitionen

- Für heute: Alle Graphen $G = (V, E)$
  **ungerichtet**, **zusammenhängend**, mit **positiven** Kantengewichten
- **Spannbaum**: Baum $(V, T)$ von $G$ (also $T \subseteq E$),
  „spannt $G$ auf“ (= zusammenhängend)
- Bekannte Algorithmen, die (irgendwelche) Spannbäume bestimmen:
  - Tiefensuche
  - Breitensuche
  - Dijkstra
- Spannbaum hat **Gewicht** $\sum_{t \in T} c(t)$
- **Minimaler** Spannbaum (MST = Minimum Spanning Tree):
  Spannbaum mit **minimalem Gewicht**
- Minimales Gewicht ist **eindeutig**,
  minimaler Spannbaum jedoch i. A. **nicht**

# Spannbäume

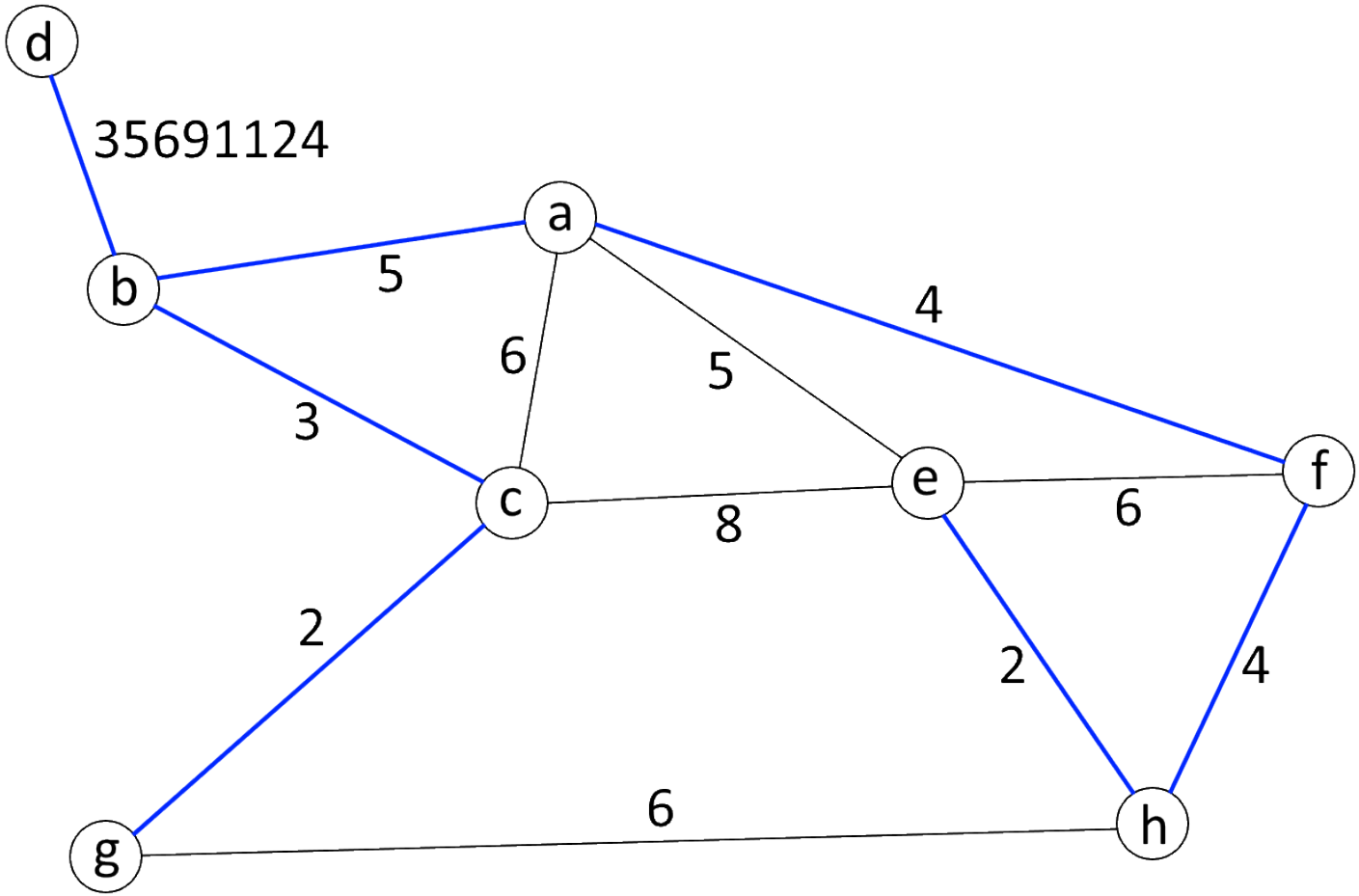
## Aufgabe 1: Scharf hinsehen

Gebt zu folgendem Graphen einen MST an:



# Spannbäume

## Lösung zu Aufgabe 1



# Spannbäume

## Die Schnitteigenschaft („Cut Property“)

- Sei $U \dot\cup W = V$ irgendeine „Aufteilung“ von $V$ und $C = \left\{ \{u, w\} \in E \;\middle|\; u \in U, w \in W \right\}$ alle „Brücken dazwischen“ (genannt **Schnitt**)

$\Rightarrow$ **Schnitteigenschaft:**
Die **leichteste** Kante $e \in C$ kann in einem MST verwendet werden.

- **Warum?**
  $\Rightarrow$ Betrachte MST $T$. $U$ und $W$ in $T$ durch $e$ verbunden?
  $\Rightarrow$ **Ja:** ✓
  $\Rightarrow$ **Nein:** Dann $U$ und $W$ durch andere Kante $e' \in C$ verbunden, und $e'$ darf **nicht schwerer** als $e$ sein (sonst $\lightning$ $T$ minimal) $\Longrightarrow$ $e$ und $e'$ **austauschbar**. □

# Spannbäume

## Die Schnitteigenschaft („Cut Property“)

- Sei $U \dot\cup W = V$ irgendeine „Aufteilung“ von $V$ und $C = \left\{\{u, w\} \in E \;\middle|\; u \in U, w \in W\right\}$ alle „Brücken dazwischen“ (genannt **Schnitt**)

$\Rightarrow$ **Schnitteigenschaft**:
Die **leichteste** Kante $e \in C$ kann in einem MST verwendet werden.

- **Warum?**
  $\Rightarrow$ Betrachte MST $T$. $U$ und $W$ in $T$ durch $e$ verbunden?
  $\Rightarrow$ Ja: ✓
  $\Rightarrow$ Nein: Dann $U$ und $W$ durch andere Kante $e' \in C$ verbunden, und $e'$ darf **nicht schwerer** als $e$ sein (sonst $\lightning$ $T$ minimal) $\Longrightarrow$ $e$ und $e'$ **austauschbar**. □

# Spannbäume

## Die Schnitteigenschaft („Cut Property“)

- Sei $U \dot\cup W = V$ irgendeine „Aufteilung“ von $V$ und $C = \left\{\{u, w\} \in E \;\middle|\; u \in U, w \in W\right\}$ alle „Brücken dazwischen“ (genannt **Schnitt**)

$\Rightarrow$ **Schnitteigenschaft**:
Die **leichteste** Kante $e \in C$ kann in einem MST verwendet werden.

- **Warum?**

$\Rightarrow$ Betrachte MST $T$. $U$ und $W$ in $T$ durch $e$ verbunden?

$\Rightarrow$ Ja: ✓

$\Rightarrow$ Nein: Dann $U$ und $W$ durch andere Kante $e' \in C$ verbunden, und $e'$ darf **nicht schwerer** als $e$ sein (sonst $\lightning$ $T$ minimal) $\Longrightarrow$ $e$ und $e'$ **austauschbar**. □

# Spannbäume

## Die Schnitteigenschaft („Cut Property“)

- Sei $U \dot\cup W = V$ irgendeine „Aufteilung“ von $V$ und $C = \left\{\{u, w\} \in E \;\middle|\; u \in U, w \in W\right\}$ alle „Brücken dazwischen“ (genannt **Schnitt**)

$\Rightarrow$ **Schnitteigenschaft**:
Die **leichteste** Kante $e \in C$ kann in einem MST verwendet werden.

- **Warum?**
  $\Rightarrow$ Betrachte MST $T$. $U$ und $W$ in $T$ durch $e$ verbunden?
  $\Rightarrow$ **Ja**: ✓
  $\Rightarrow$ **Nein**: Dann $U$ und $W$ durch andere Kante $e' \in C$ verbunden, und $e'$ darf **nicht schwerer** als $e$ sein (sonst ↯ $T$ minimal) $\Rightarrow$ $e$ und $e'$ **austauschbar**. □

# Spannbäume

## Die Kreiseigenschaft („Cycle Property“)

- Sei $C \subseteq E$ ein (beliebiger) Kreis in $G$

$\Rightarrow$ **Kreiseigenschaft:**
Für einen MST $T$ von $G$ wird die **schwerste** Kante $e \in C$ **nicht** benötigt.

- **Warum?**
  $\Rightarrow$ Angenommen, $e \in T$ und dafür **leichtere Kreiskante** $e' \notin T$:
  Durch **Austausch** von $e$ und $e'$ wird Gewicht von $T$ **kleiner**
  $\Rightarrow \lightning$ $T$ minimal $\Rightarrow$ Müssen $e$ rausschmeißen. $\square$

# Spannbäume

## Die Kreiseigenschaft („Cycle Property“)

- Sei $C \subseteq E$ ein (beliebiger) Kreis in $G$

$\Rightarrow$ **Kreiseigenschaft**:
Für einen MST $T$ von $G$ wird die **schwerste** Kante $e \in C$ **nicht** benötigt.

- **Warum?**
  $\Rightarrow$ Angenommen, $e \in T$ und dafür **leichtere Kreiskante** $e' \notin T$:
  Durch **Austausch** von $e$ und $e'$ wird Gewicht von $T$ **kleiner**
  $\Rightarrow \lightning$ $T$ minimal $\Rightarrow$ Müssen $e$ rausschmeißen. $\square$

# Spannbäume

## Die Kreiseigenschaft („Cycle Property“)

- Sei $C \subseteq E$ ein (beliebiger) Kreis in $G$

$\Rightarrow$ **Kreiseigenschaft**:
Für einen MST $T$ von $G$ wird die **schwerste** Kante $e \in C$ **nicht** benötigt.

- **Warum**?

$\Rightarrow$ Angenommen, $e \in T$ und dafür **leichtere** Kreiskante $e' \notin T$:
Durch **Austausch** von $e$ und $e'$ wird Gewicht von $T$ **kleiner**
$\Rightarrow \lightning$ $T$ minimal $\Rightarrow$ Müssen $e$ rausschmeißen. $\square$

# Spannbäume

## Die Kreiseigenschaft („Cycle Property“)

- Sei $C \subseteq E$ ein (beliebiger) Kreis in $G$

$\Rightarrow$ **Kreiseigenschaft**:
Für einen MST $T$ von $G$ wird die **schwerste** Kante $e \in C$ **nicht** benötigt.

- **Warum**?
  $\Rightarrow$ Angenommen, $e \in T$ und dafür **leichtere Kreiskante** $e' \notin T$:
  Durch **Austausch** von $e$ und $e'$ wird Gewicht von $T$ **kleiner**
  $\Rightarrow$ ↯ $T$ minimal $\Rightarrow$ Müssen $e$ rausschmeißen. □

# Spannbäume – Jarník-Prim

## Ein Algorithmus für MSTs: Jarník-Prim

$\Rightarrow$ **Idee**: Schnitteigenschaft irgendwie ausnutzen!

1. Starte ab beliebigem $s \in V$, setze $S := \{s\}$
2. **Erweitere** Knotenmenge $S$ und Baum $T$ **schrittweise** um die **minimale** Verbindungskante zu $V \setminus S$
   - Schnittkantenmenge $C$ zwischen $S$ und $V \setminus S$
   - $\Rightarrow$ Verwalte sie in einer **PriorityQueue** $PQ$

$\checkmark$ Funktioniert dank Schnitteigenschaft

# Spannbäume – Jarník-Prim

## Ein Algorithmus für MSTs: Jarník-Prim

$\Rightarrow$ **Idee**: Schnitteigenschaft irgendwie ausnutzen!

1. Starte ab beliebigem $s \in V$, setze $S := \{s\}$
2. **Erweitere** Knotenmenge $S$ und Baum $T$ **schrittweise** um die **minimale** Verbindungskante zu $V \setminus S$

- Schnittkantenmenge $C$ zwischen $S$ und $V \setminus S$
  $\Rightarrow$ Verwalte sie in einer **PriorityQueue** $PQ$

$\checkmark$ Funktioniert dank Schnitteigenschaft

# Spannbäume – Jarník-Prim

## Ein Algorithmus für MSTs: Jarník-Prim

$\Rightarrow$ **Idee**: Schnitteigenschaft irgendwie ausnutzen!

1. Starte ab beliebigem $s \in V$, setze $S := \{s\}$
2. **Erweitere** Knotenmenge $S$ und Baum $T$ **schrittweise** um die **minimale** Verbindungskante zu $V \setminus S$

- Schnittkantenmenge $C$ zwischen $S$ und $V \setminus S$
  $\Rightarrow$ Verwalte sie in einer **PriorityQueue** $PQ$

✓ Funktioniert dank Schnitteigenschaft

# Spannbäume – Jarník-Prim

## Ein Algorithmus für MSTs: Jarník-Prim

⇒ **Idee**: Schnitteigenschaft irgendwie ausnutzen!

1. Starte ab beliebigem $s \in V$, setze $S := \{s\}$
2. **Erweitere** Knotenmenge $S$ und Baum $T$ **schrittweise** um die **minimale** Verbindungskante zu $V \setminus S$

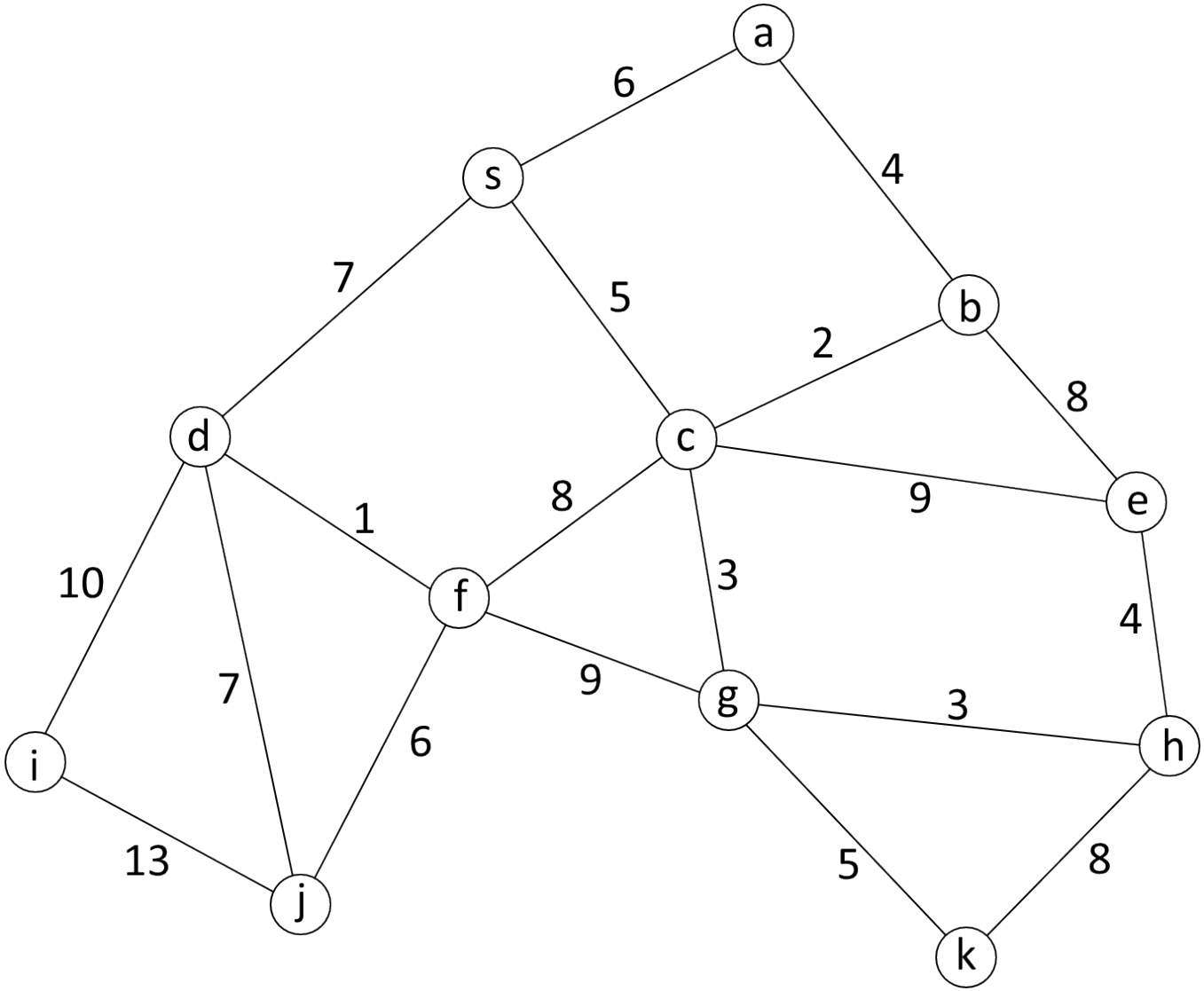
- Schnittkantenmenge $C$ zwischen $S$ und $V \setminus S$
  ⇒ Verwalte sie in einer **PriorityQueue** $PQ$

✓ Funktioniert dank Schnitteigenschaft

# Spannbäume – Jarník-Prim

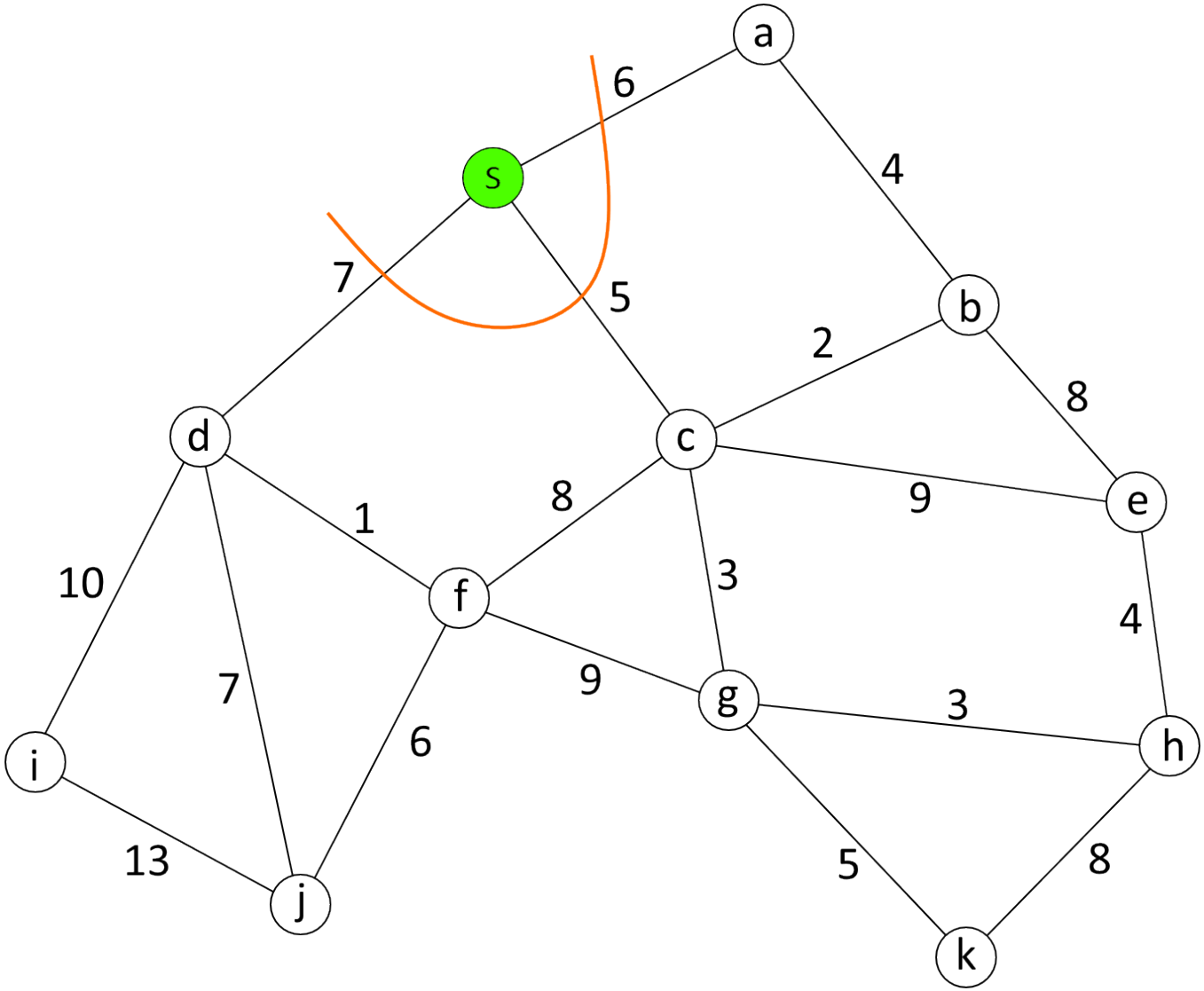
```
function Jarník-Prim(G = (V, E))    // sieht aus wie Dijkstra
    pick any s ∈ V
    d := (∞, ..., ∞) : array[1...n] of ℝ    // Distanz zu Knotenmenge S, nicht zu
      s ∈ V!
    parent := (⊥, ..., ⊥) : array[1...n] of V
    PQ = ⟨s⟩ : PriorityQueue
    parent[s] := s
    while PQ ≠ ∅ do
        u := PQ.deleteMin(),    d[u] := 0
        foreach e = {u, v} ∈ E do
            if c(e) < d[v] then
                d[v] := c(e)
                parent[v] := u
                if v ∈ PQ then  PQ.decreaseKey(v)
                else            PQ.insert(v)
    return { (parent[v], v) | s ≠ v ∈ V }
```

# Spannbäume – Beispiel Jarník-Prim

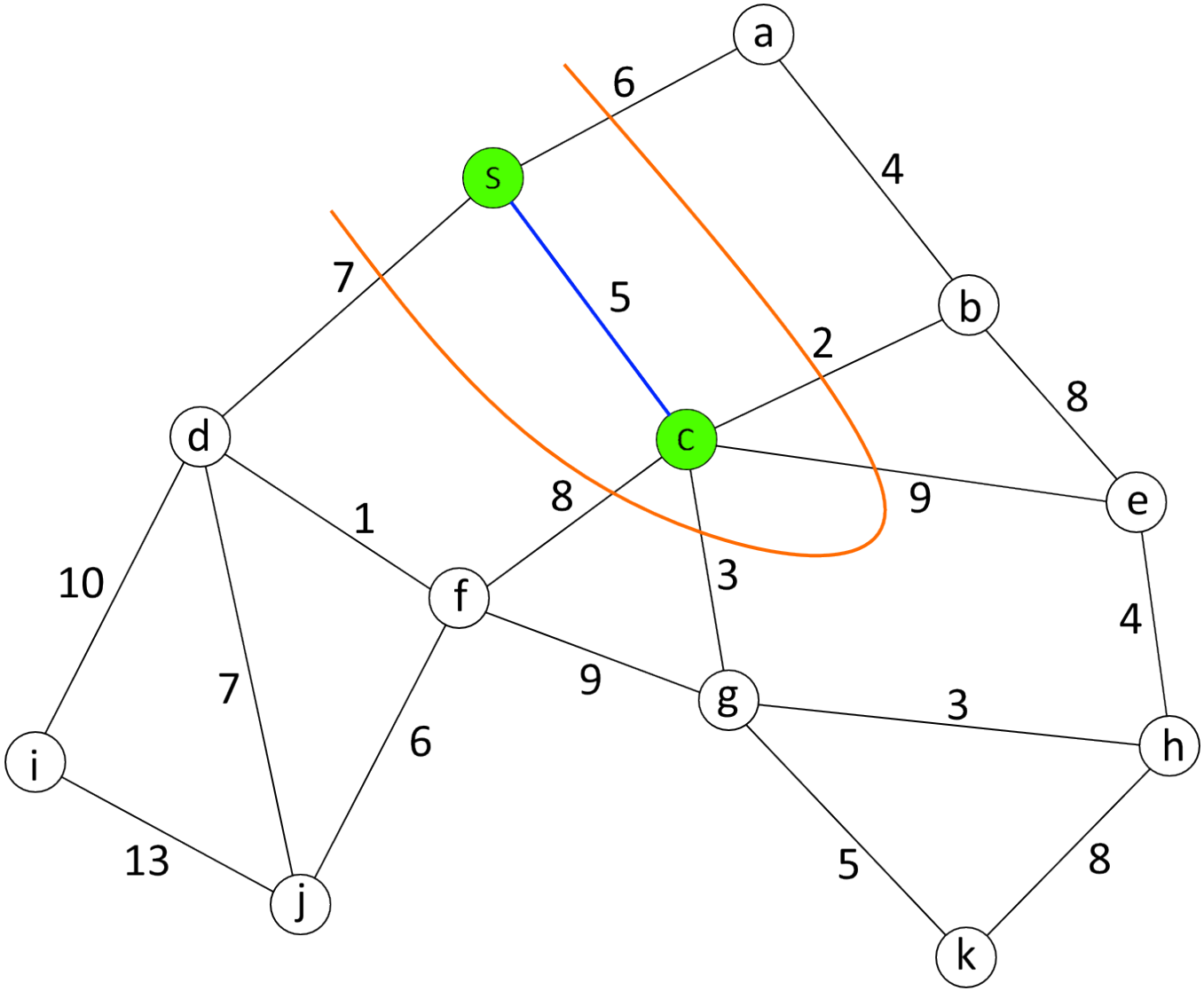



Hier klicken, um das Beispiel zu überspringen.

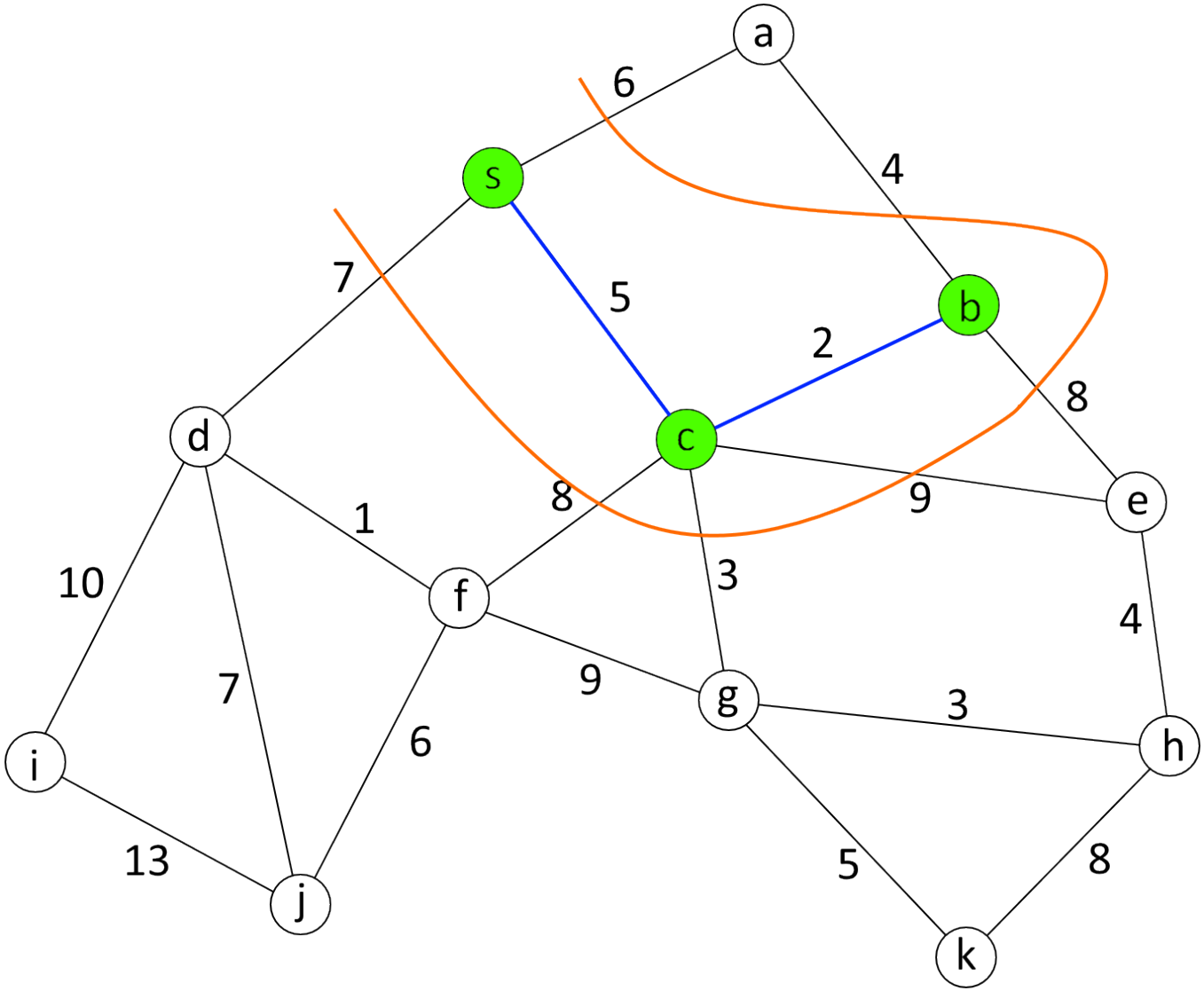
# Spannbäume – Beispiel Jarník-Prim



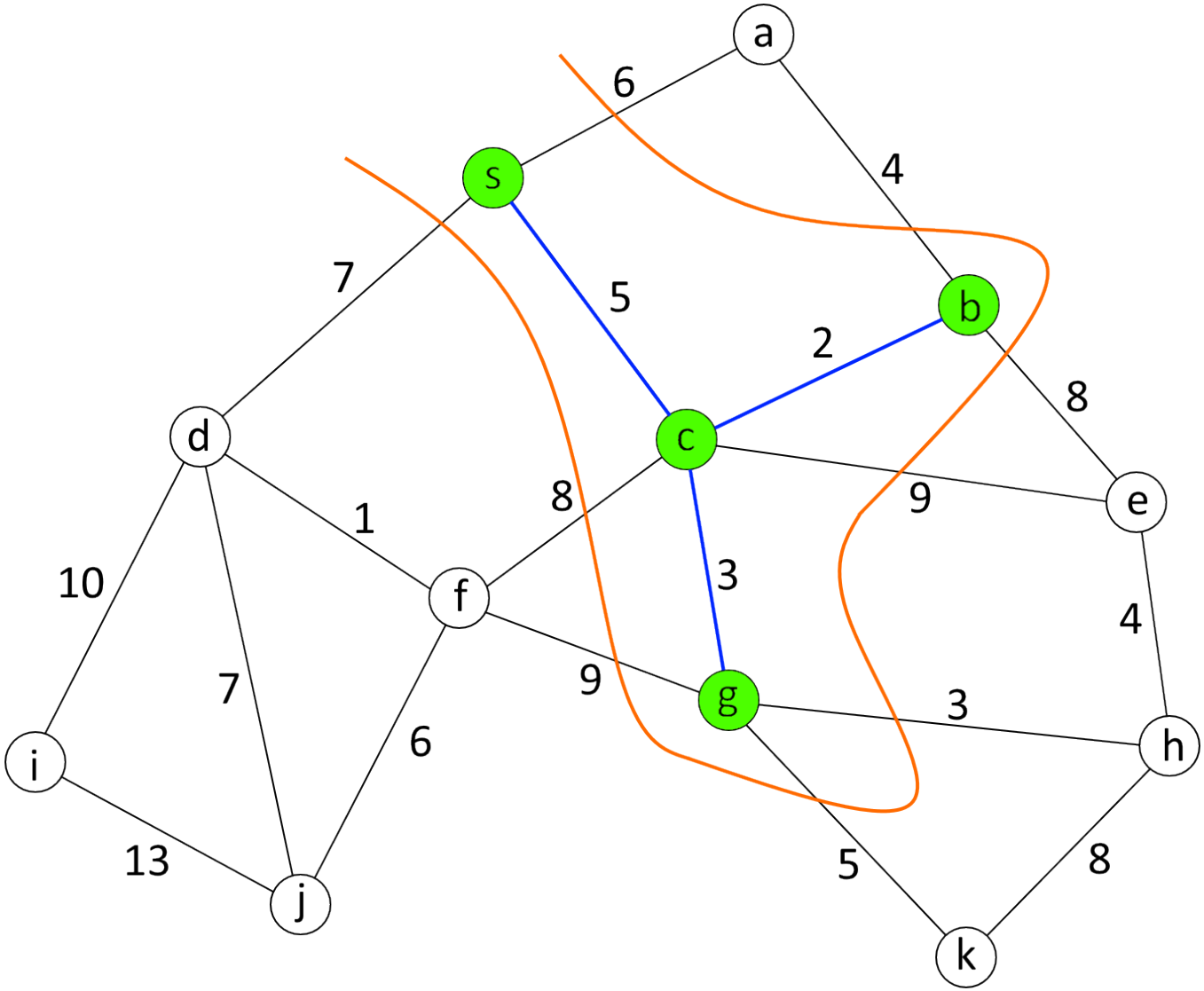
# Spannbäume – Beispiel Jarník-Prim



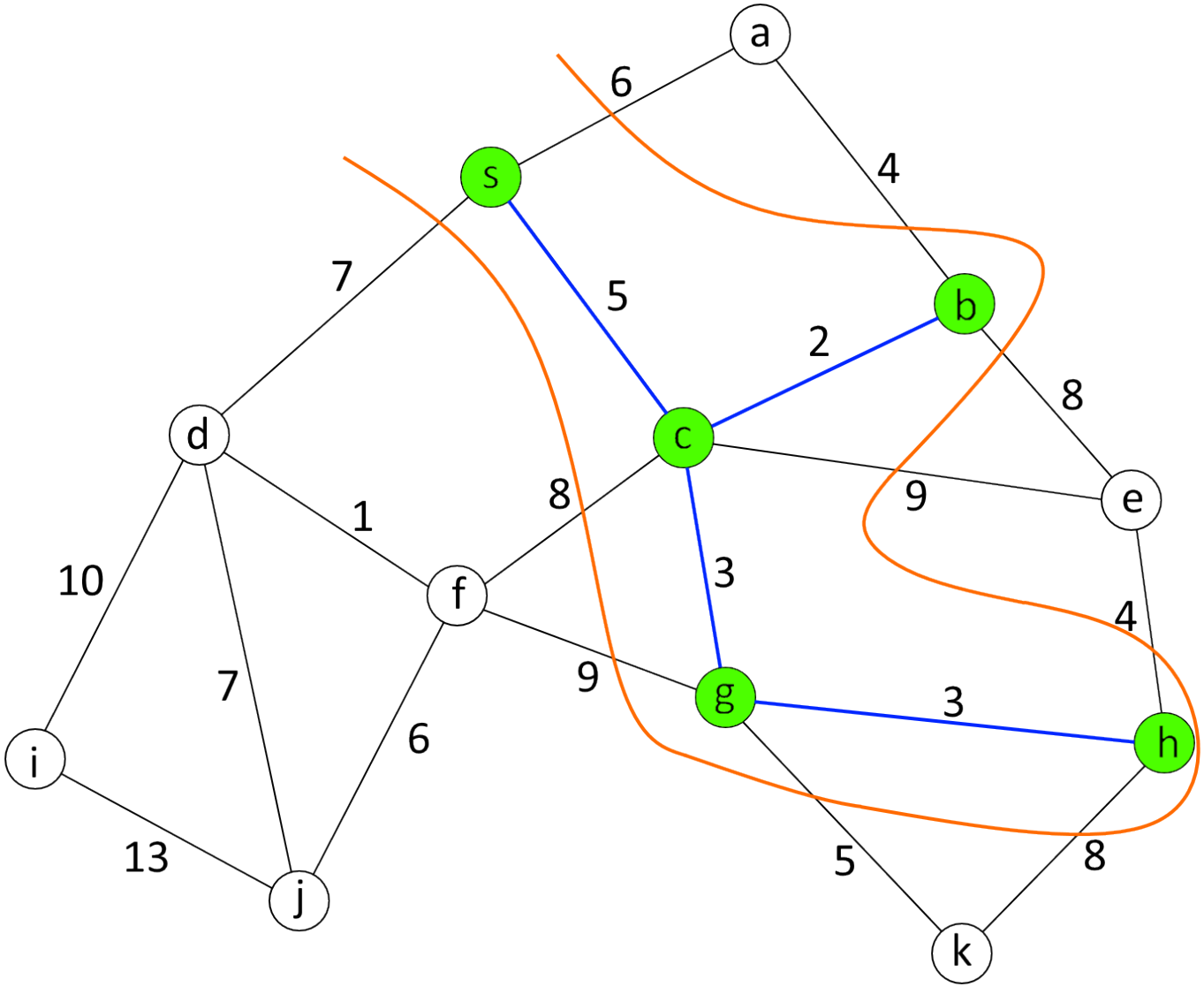
# Spannbäume – Beispiel Jarník-Prim



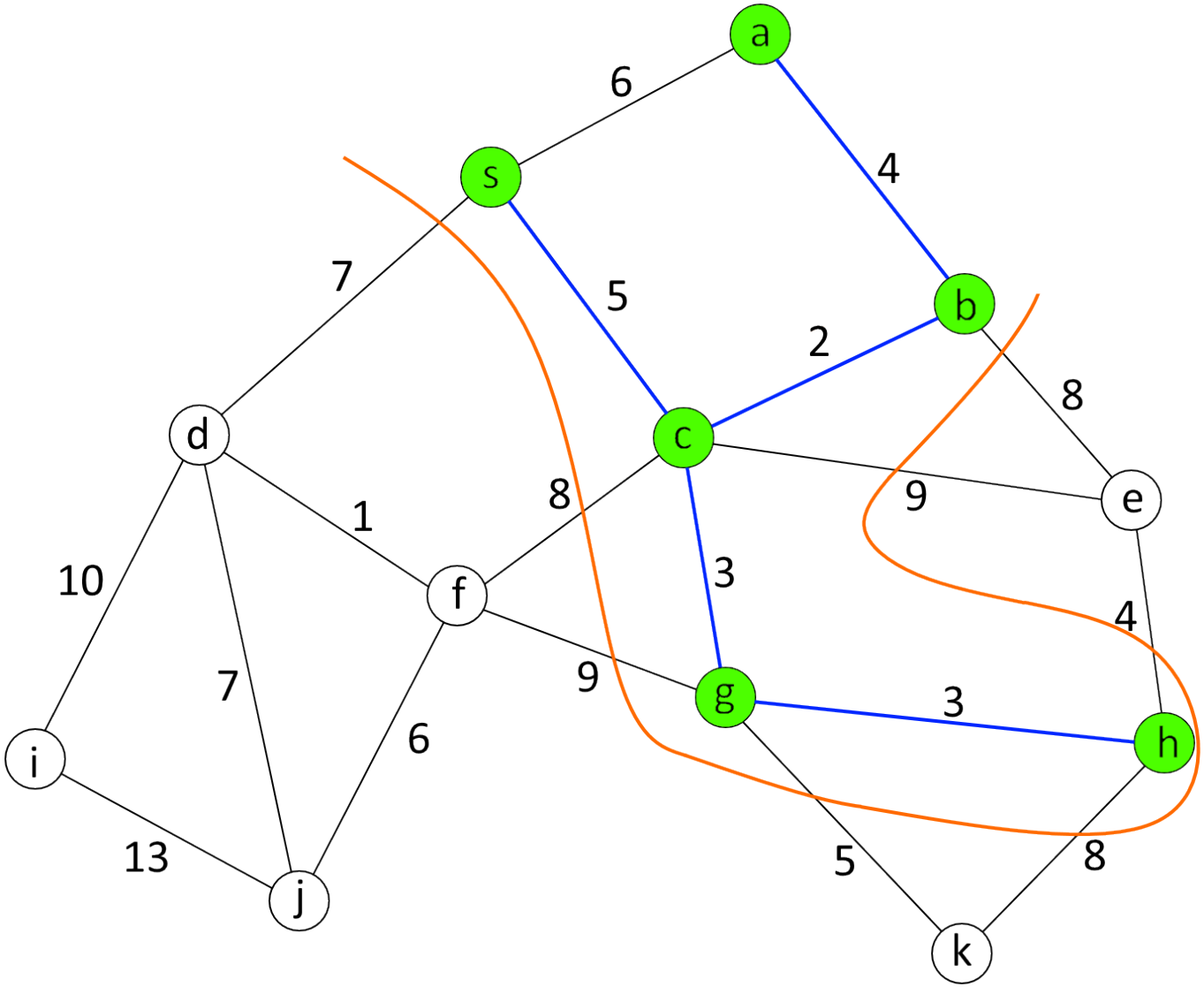
# Spannbäume – Beispiel Jarník-Prim



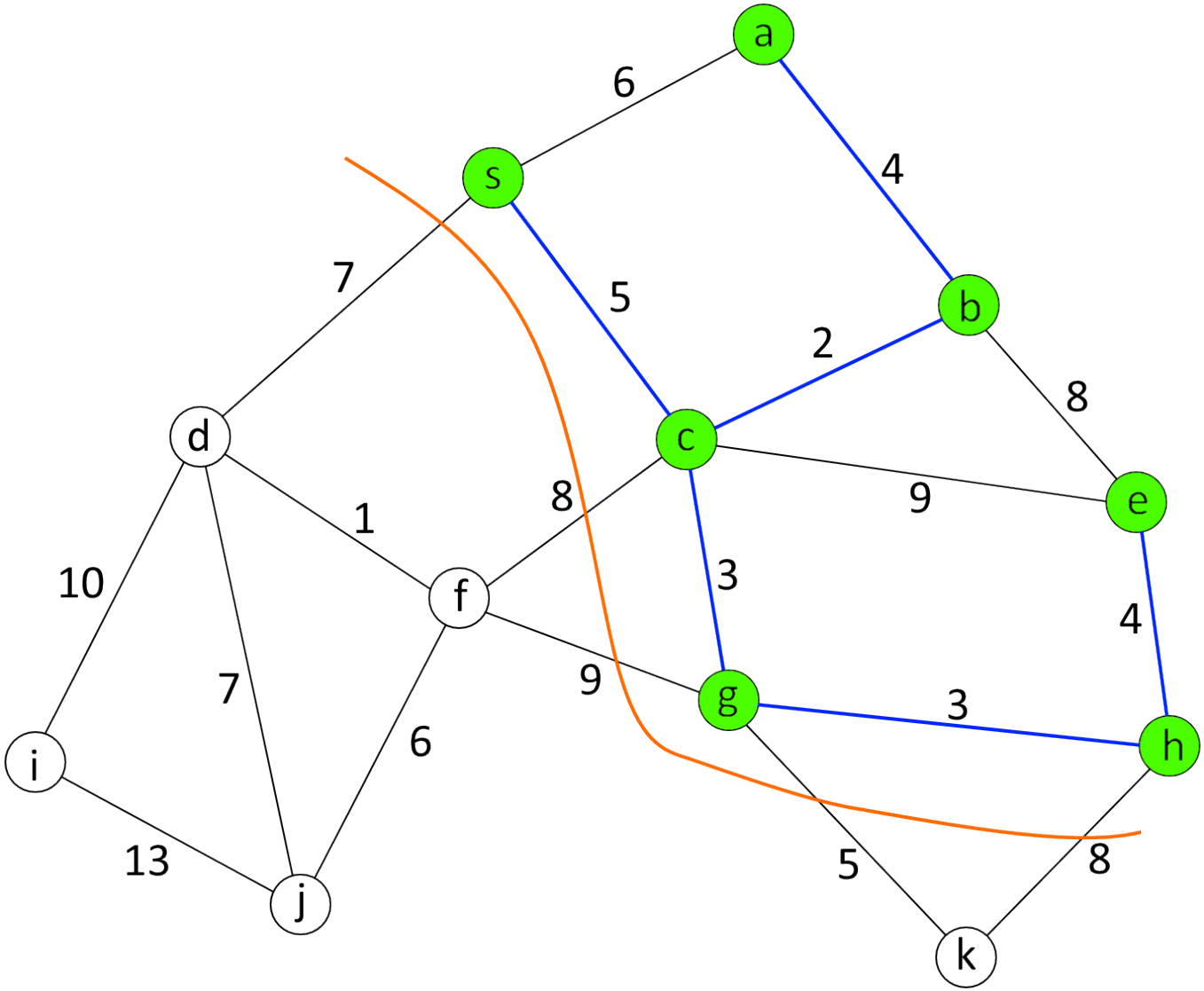
# Spannbäume – Beispiel Jarník-Prim



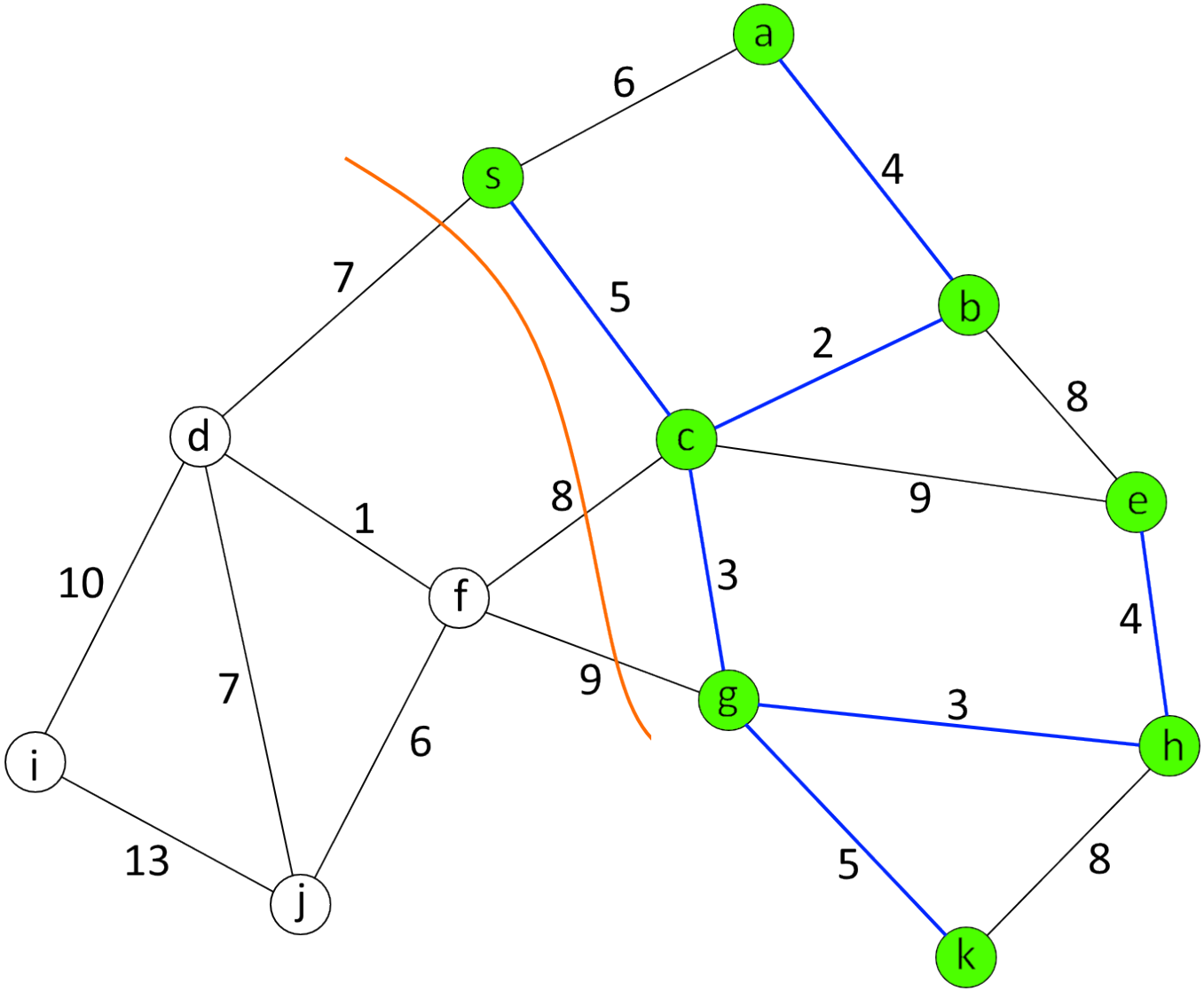
# Spannbäume – Beispiel Jarník-Prim



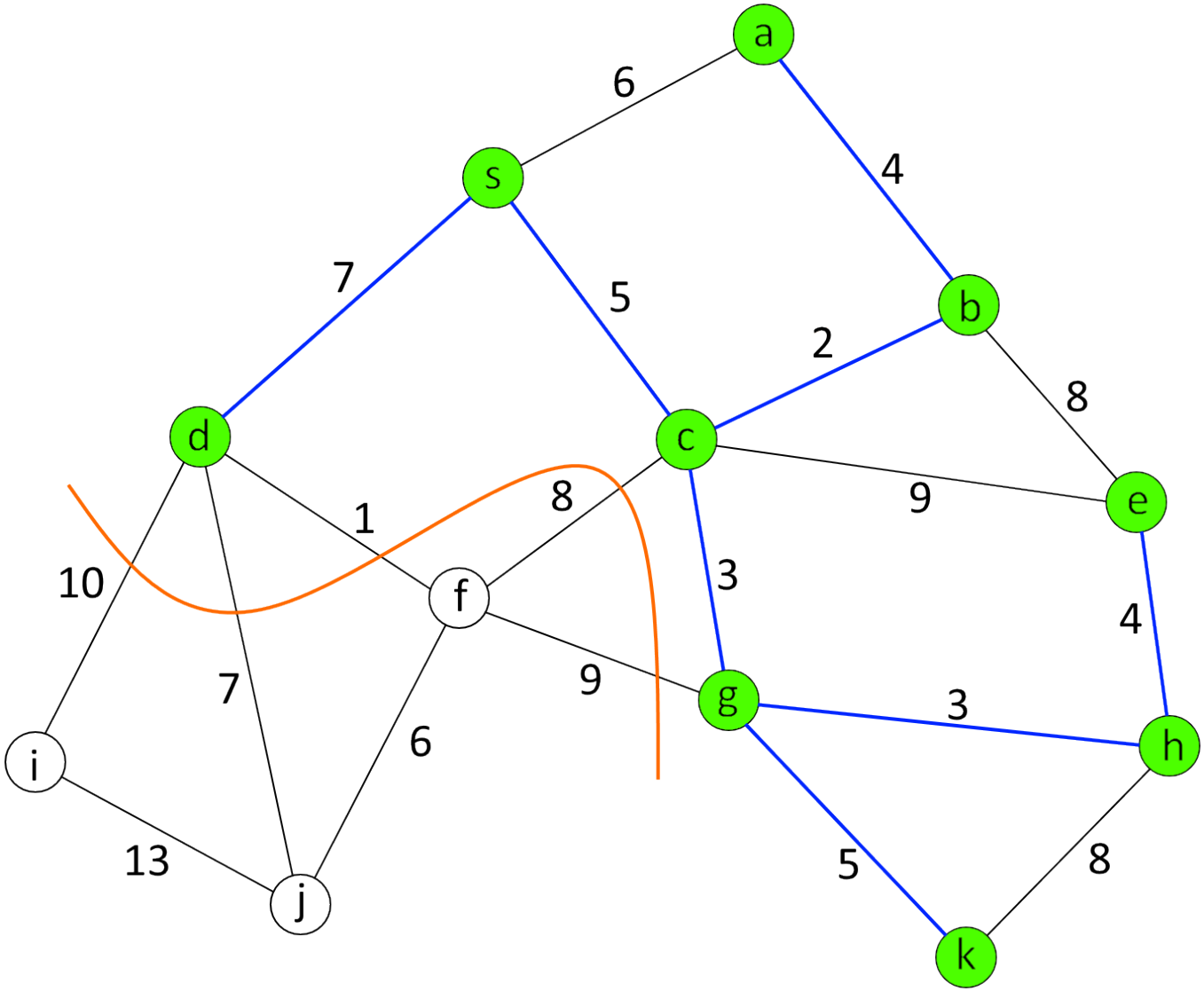
# Spannbäume – Beispiel Jarník-Prim



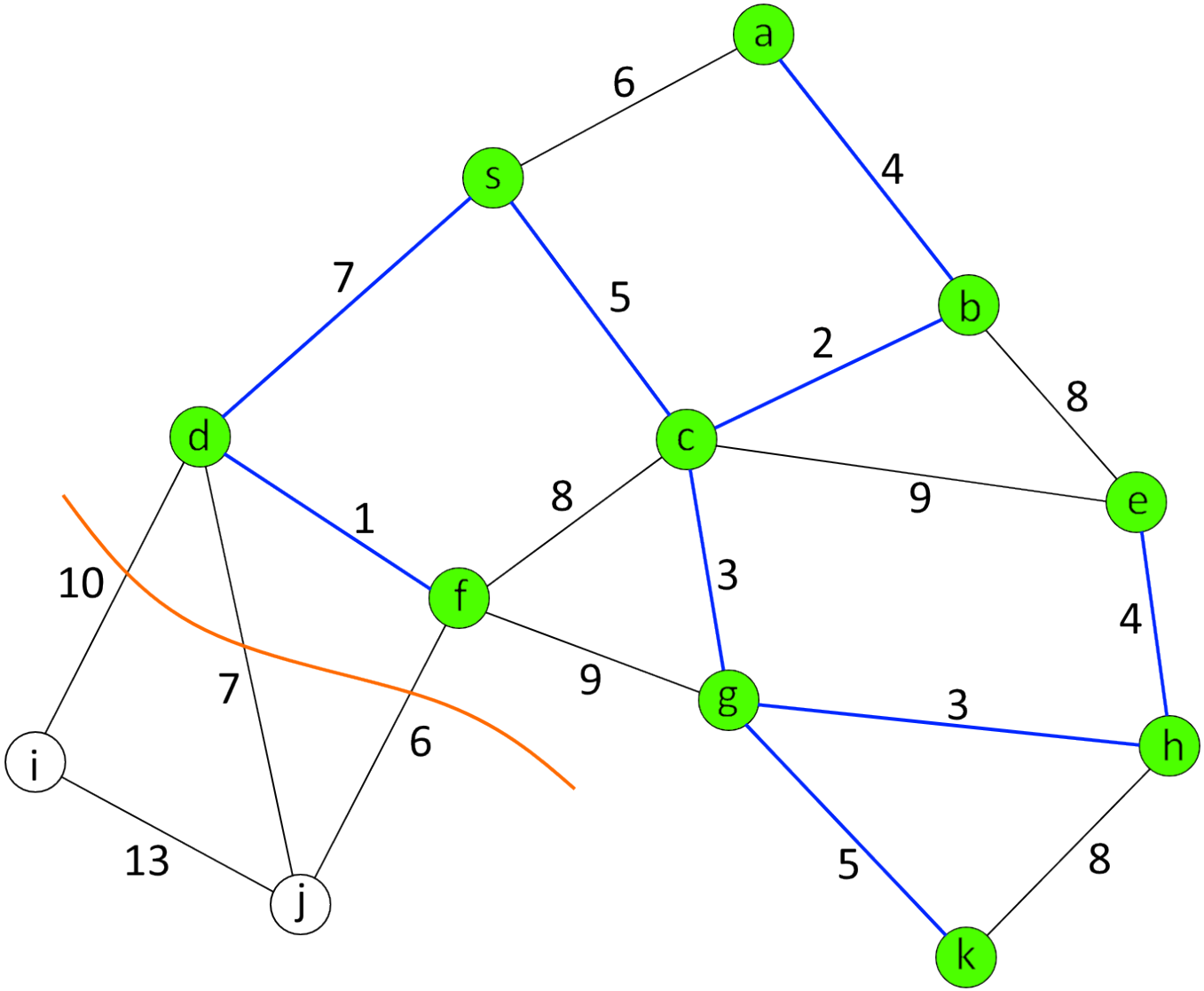
# Spannbäume – Beispiel Jarník-Prim



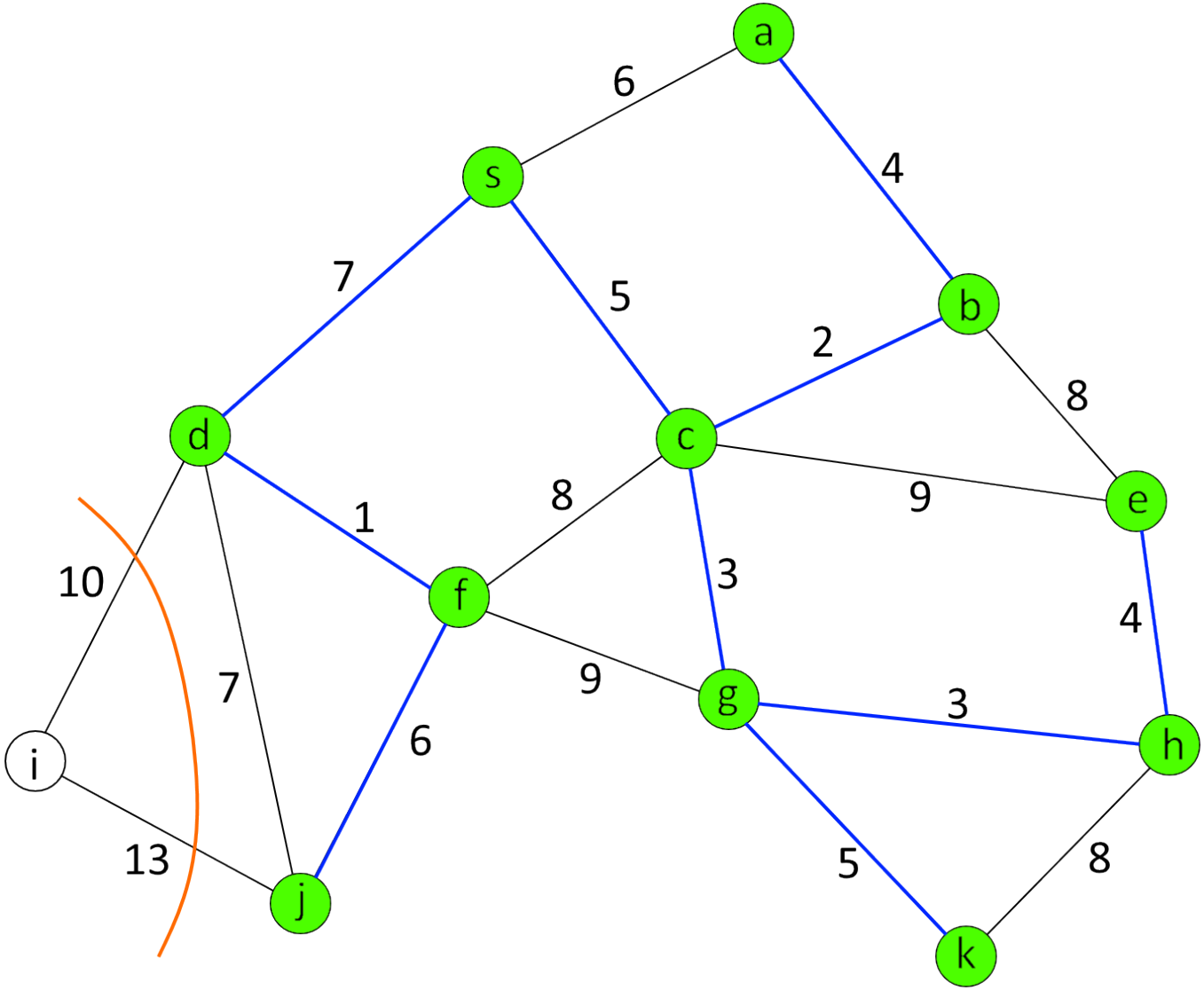
# Spannbäume – Beispiel Jarník-Prim



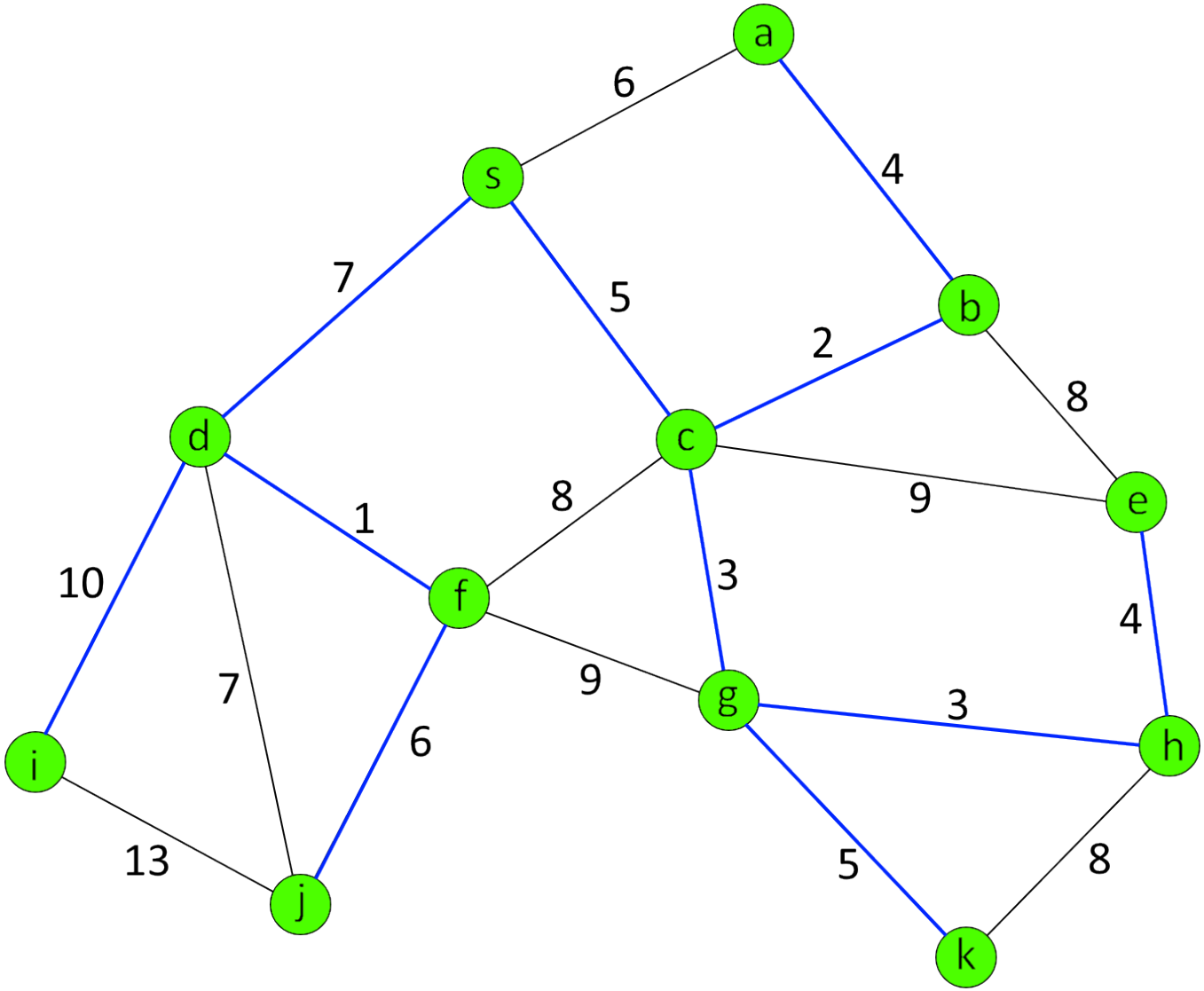
# Spannbäume – Beispiel Jarník-Prim



# Spannbäume – Beispiel Jarník-Prim



# Spannbäume – Beispiel Jarník-Prim



# Spannbäume – Jarník-Prim

**Laufzeit von Jarník-Prim: same as Dijkstra**

Im Worst-Case $m$-mal *decreaseKey*

$+$ Genau $n$-mal *deleteMin* und *insert*

$=$ Mit binärem Heap: $O((m+n)\log n)$

$=$ Mit Fibonacci-Heap: $O(m + n\log n)$ (amortisiert und mit höheren konstanten Faktoren)

# Spannbäume – Jarník-Prim

**Laufzeit von Jarník-Prim: same as Dijkstra**

Im Worst-Case $m$-mal *decreaseKey*

$+$ Genau $n$-mal *deleteMin* und *insert*

$=$ Mit binärem Heap: $O((m+n)\log n)$

$=$ Mit Fibonacci-Heap: $O(m + n\log n)$ (amortisiert und mit höheren konstanten Faktoren)

# Spannbäume – Jarník-Prim

**Laufzeit von Jarník-Prim: same as Dijkstra**

Im Worst-Case $m$-mal *decreaseKey*

$+$ Genau $n$-mal *deleteMin* und *insert*

$=$ Mit binärem Heap: $O((m+n)\log n)$

$=$ Mit Fibonacci-Heap: $O(m + n\log n)$ (amortisiert und mit höheren konstanten Faktoren)

# Spannbäume – Kruskal

## Rosinen rauspicken mit Kruskal

- **Idee**: Baum (bzw. Wald) **schrittweise wachsen** lassen

⇒ Durchlaufe **Kanten** nach **aufsteigendem** Gewicht:

- Nehme Kante zum Wald **dazu**, falls dadurch **kein Kreis** entsteht (also wenn die Kante **zwei separate** Bäume vereinigt)

- Am **Ende**: Alle ausgewählten Kanten ergeben einen MST

✓ Funktioniert dank Schnitt- und Kreiseigenschaft

# Spannbäume – Kruskal

## Rosinen rauspicken mit Kruskal

- **Idee**: Baum (bzw. Wald) **schrittweise wachsen** lassen

$\Rightarrow$ Durchlaufe **Kanten** nach **aufsteigendem** Gewicht:
Nehme Kante zum Wald **dazu**, falls dadurch **kein Kreis** entsteht
(also wenn die Kante **zwei separate** Bäume vereinigt)

- Am Ende: Alle ausgewählten Kanten ergeben einen MST

✓ Funktioniert dank Schnitt- und Kreiseigenschaft

# Spannbäume – Kruskal

## Rosinen rauspicken mit Kruskal

- **Idee**: Baum (bzw. Wald) **schrittweise wachsen** lassen

$\Rightarrow$ Durchlaufe **Kanten** nach **aufsteigendem** Gewicht:
Nehme Kante zum Wald **dazu**, falls dadurch **kein Kreis** entsteht
(also wenn die Kante **zwei separate** Bäume vereinigt)

- Am **Ende**: Alle ausgewählten Kanten ergeben einen MST

✓ Funktioniert dank Schnitt- und Kreiseigenschaft

# Spannbäume – Kruskal

Karlsruher Institut für Technologie

## Rosinen rauspicken mit Kruskal

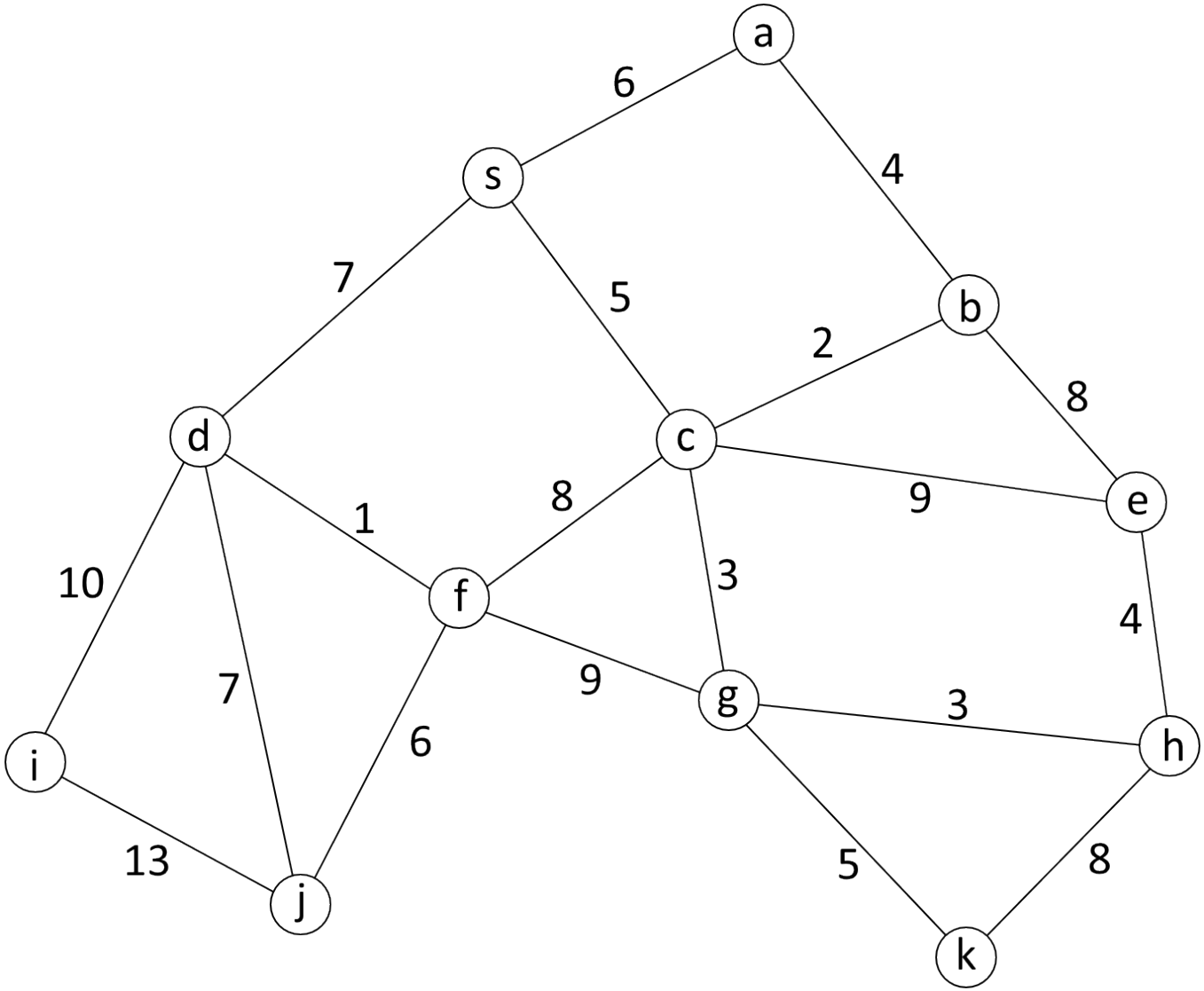
- **Idee**: Baum (bzw. Wald) **schrittweise wachsen** lassen

$\Rightarrow$ Durchlaufe **Kanten** nach **aufsteigendem** Gewicht:
Nehme Kante zum Wald **dazu**, falls dadurch **kein Kreis** entsteht
(also wenn die Kante **zwei separate** Bäume vereinigt)

- Am **Ende**: Alle ausgewählten Kanten ergeben einen MST

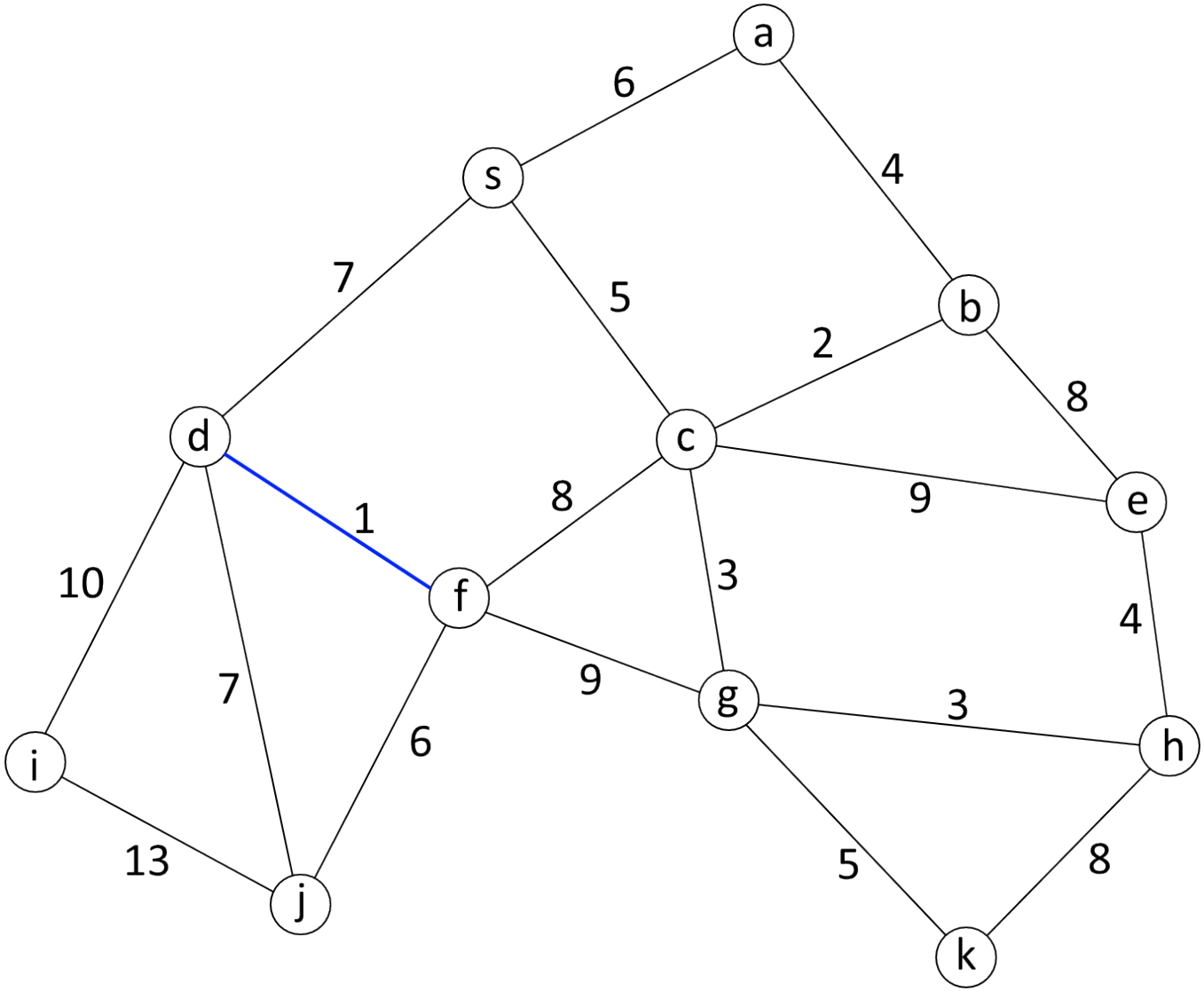
✓ Funktioniert dank Schnitt- und Kreiseigenschaft

# Spannbäume – Beispiel Kruskal

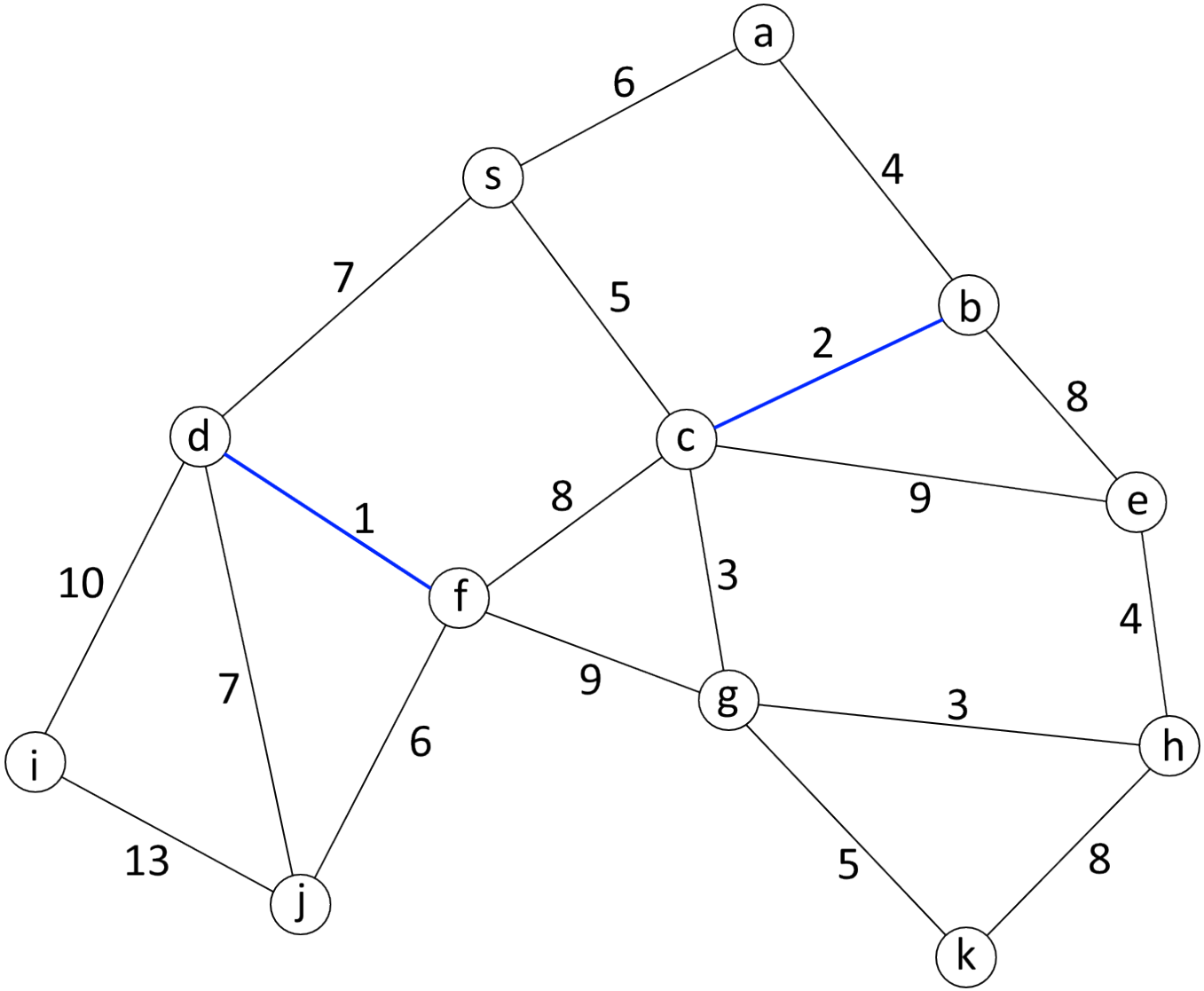



Hier klicken, um das Beispiel zu überspringen.

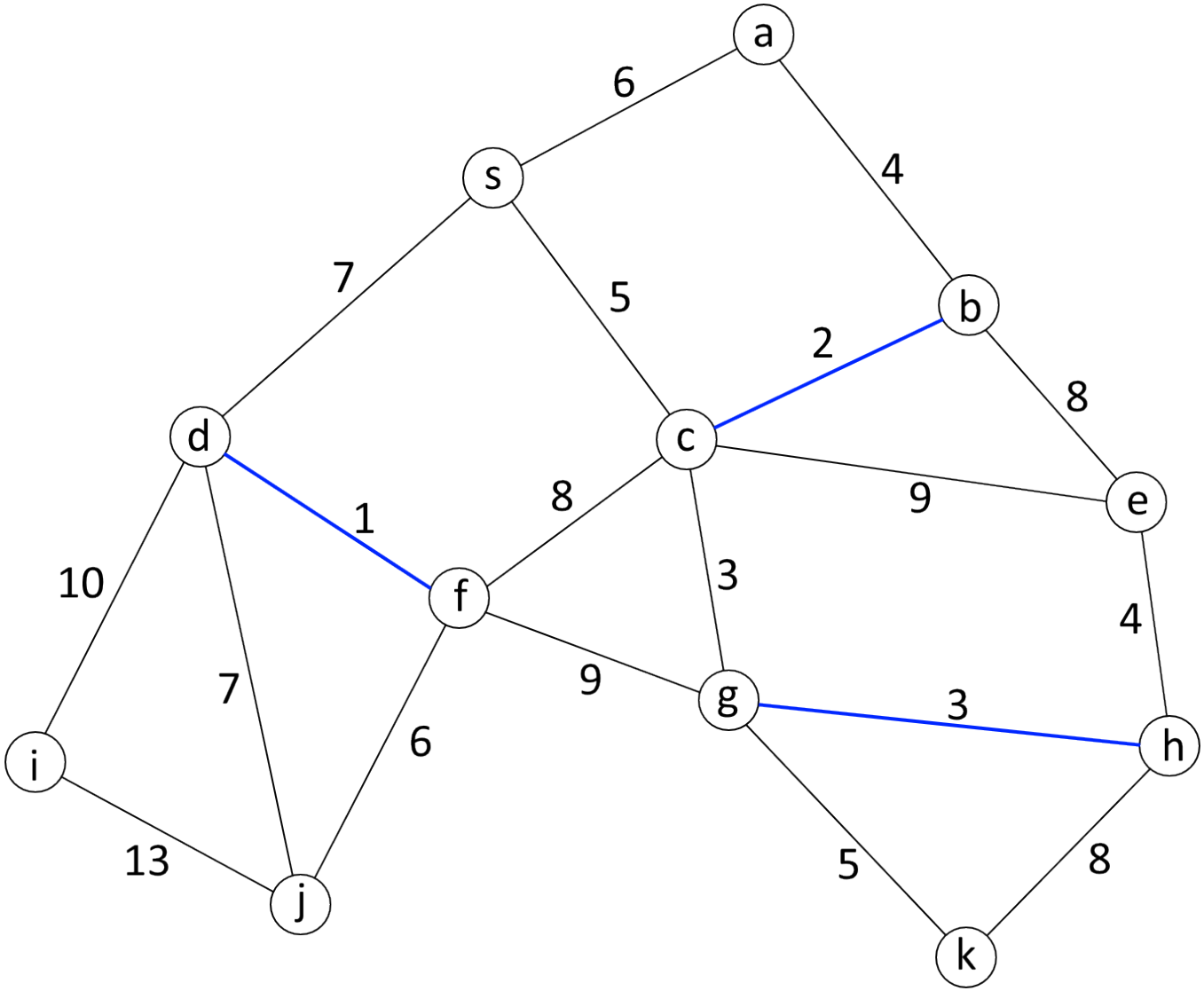
# Spannbäume – Beispiel Kruskal



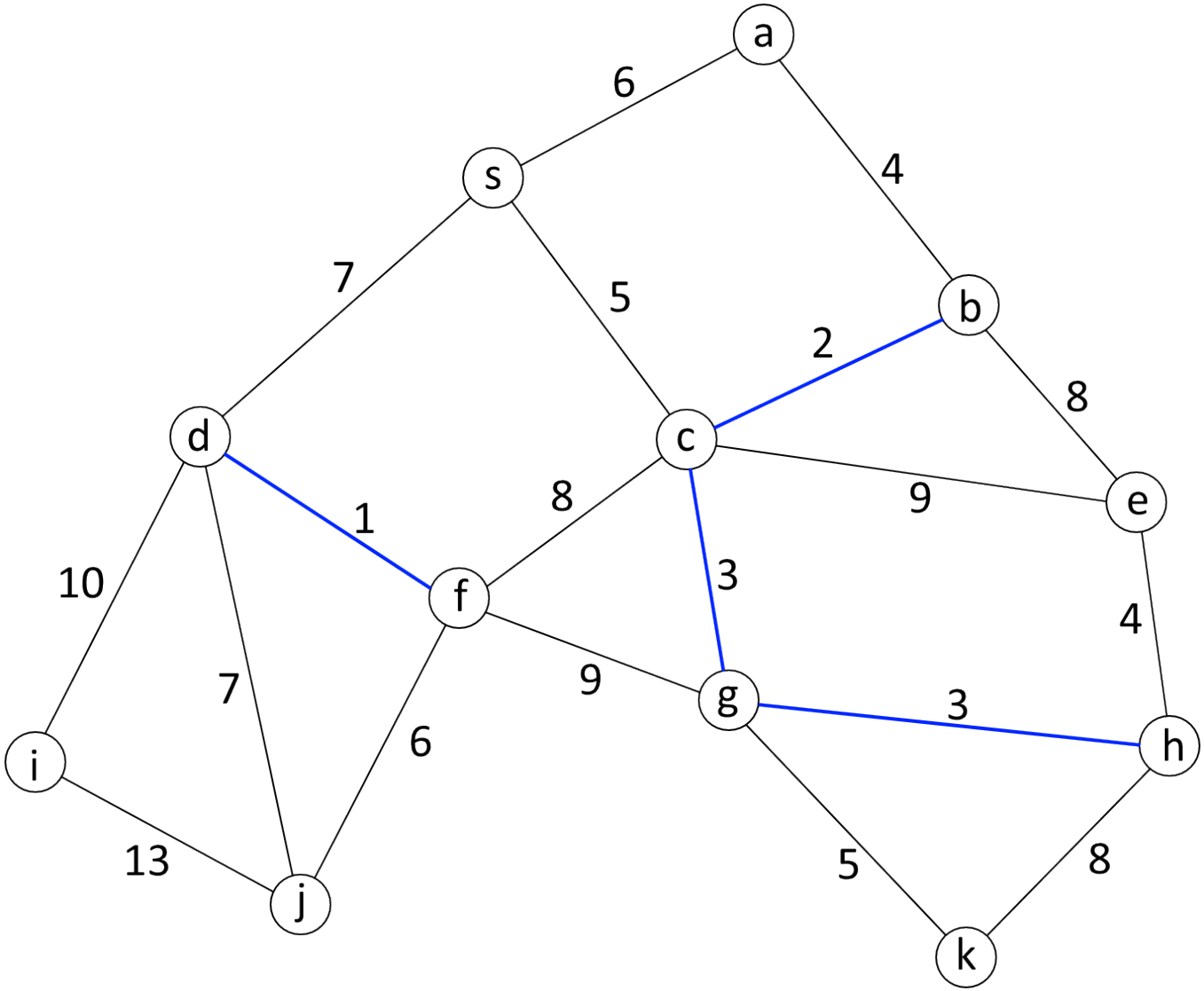
# Spannbäume – Beispiel Kruskal



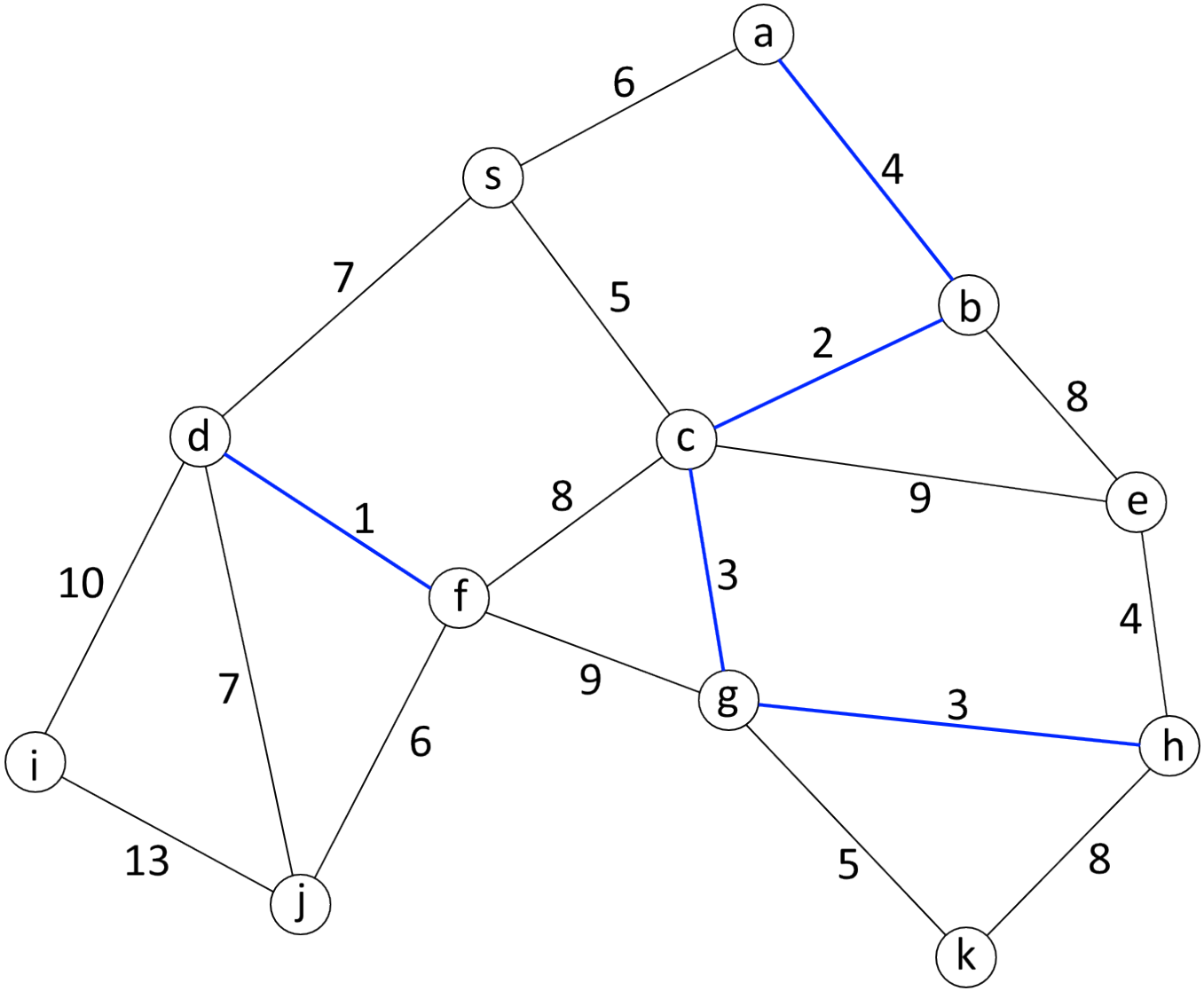
# Spannbäume – Beispiel Kruskal



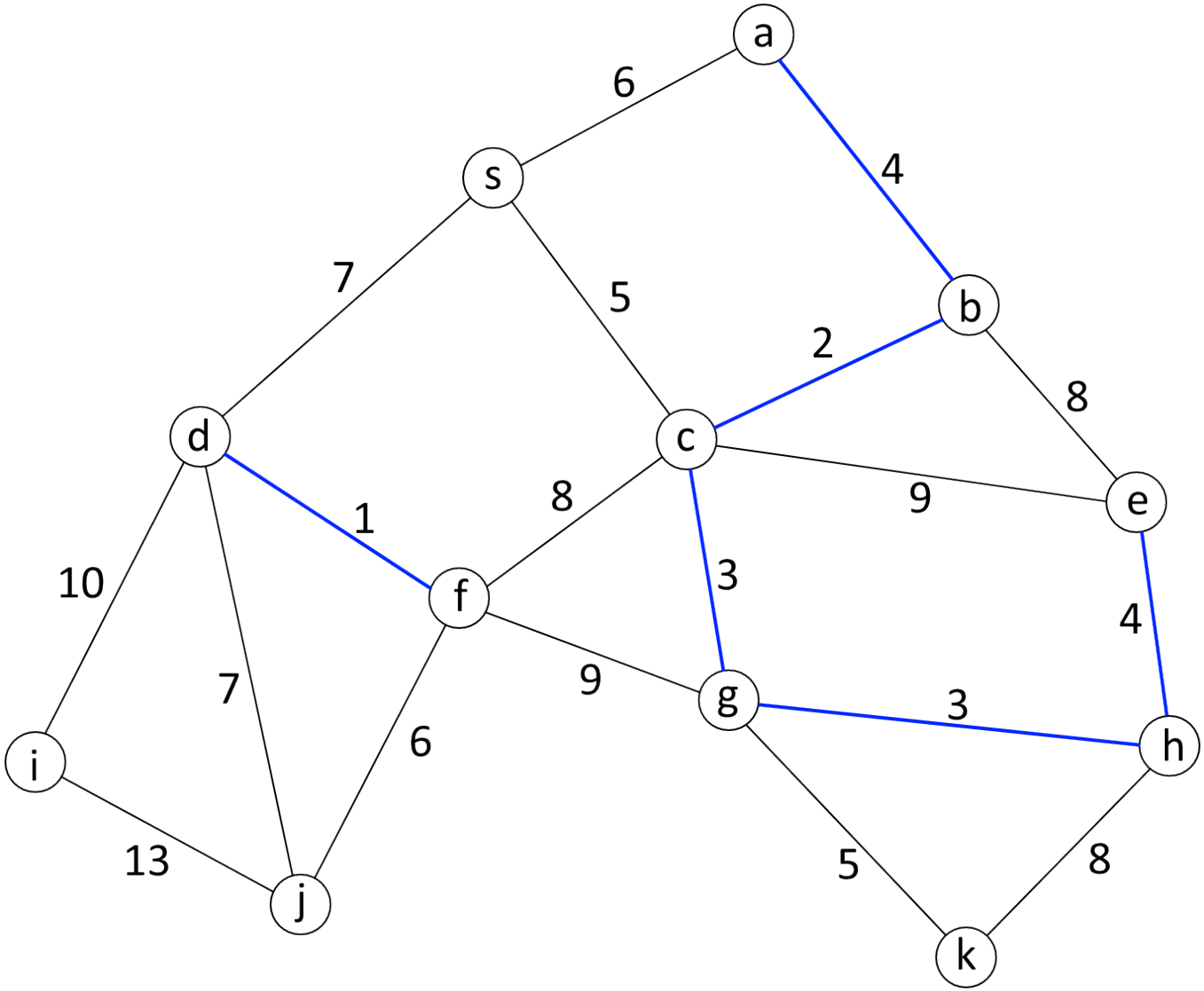
# Spannbäume – Beispiel Kruskal



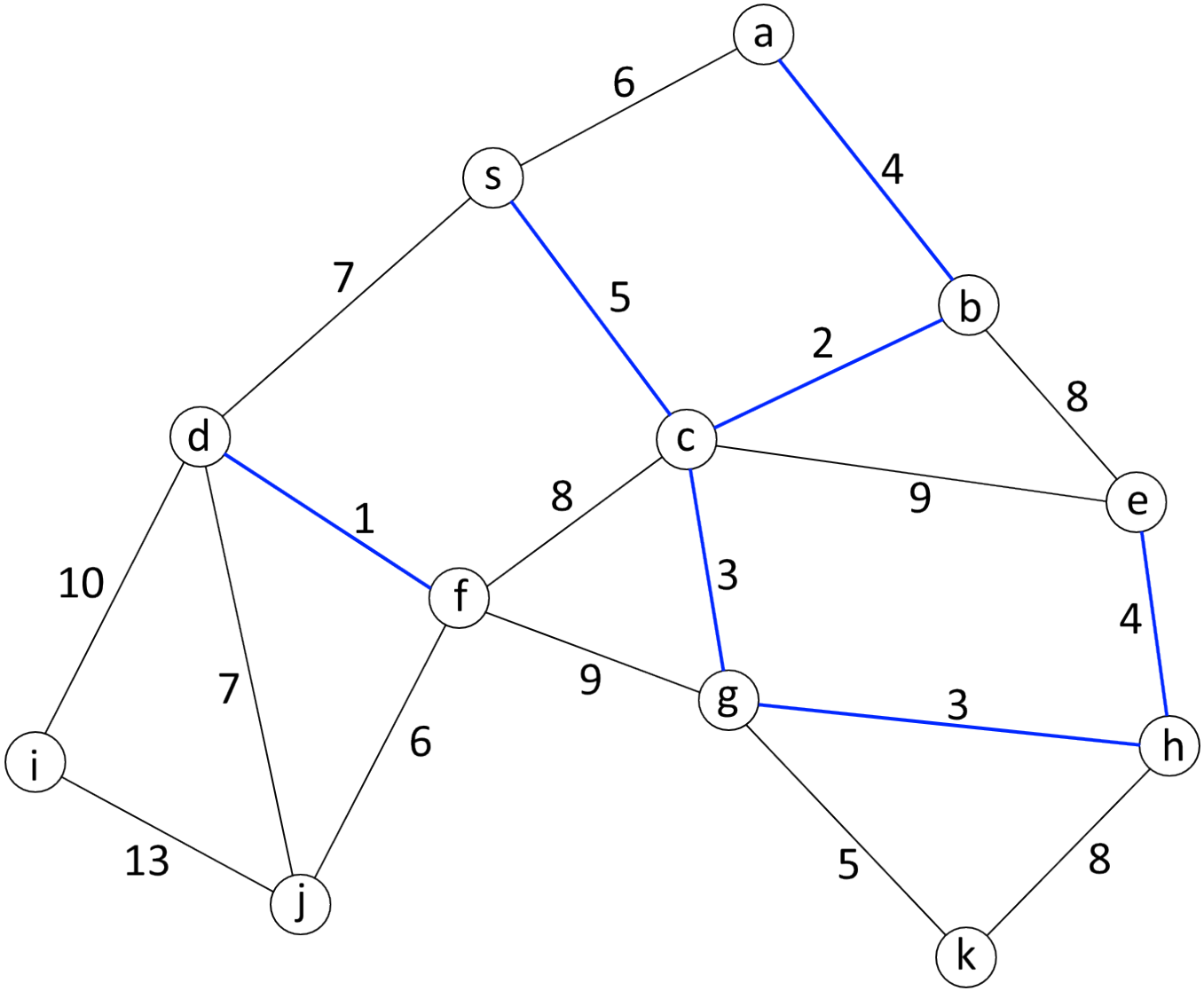
# Spannbäume – Beispiel Kruskal



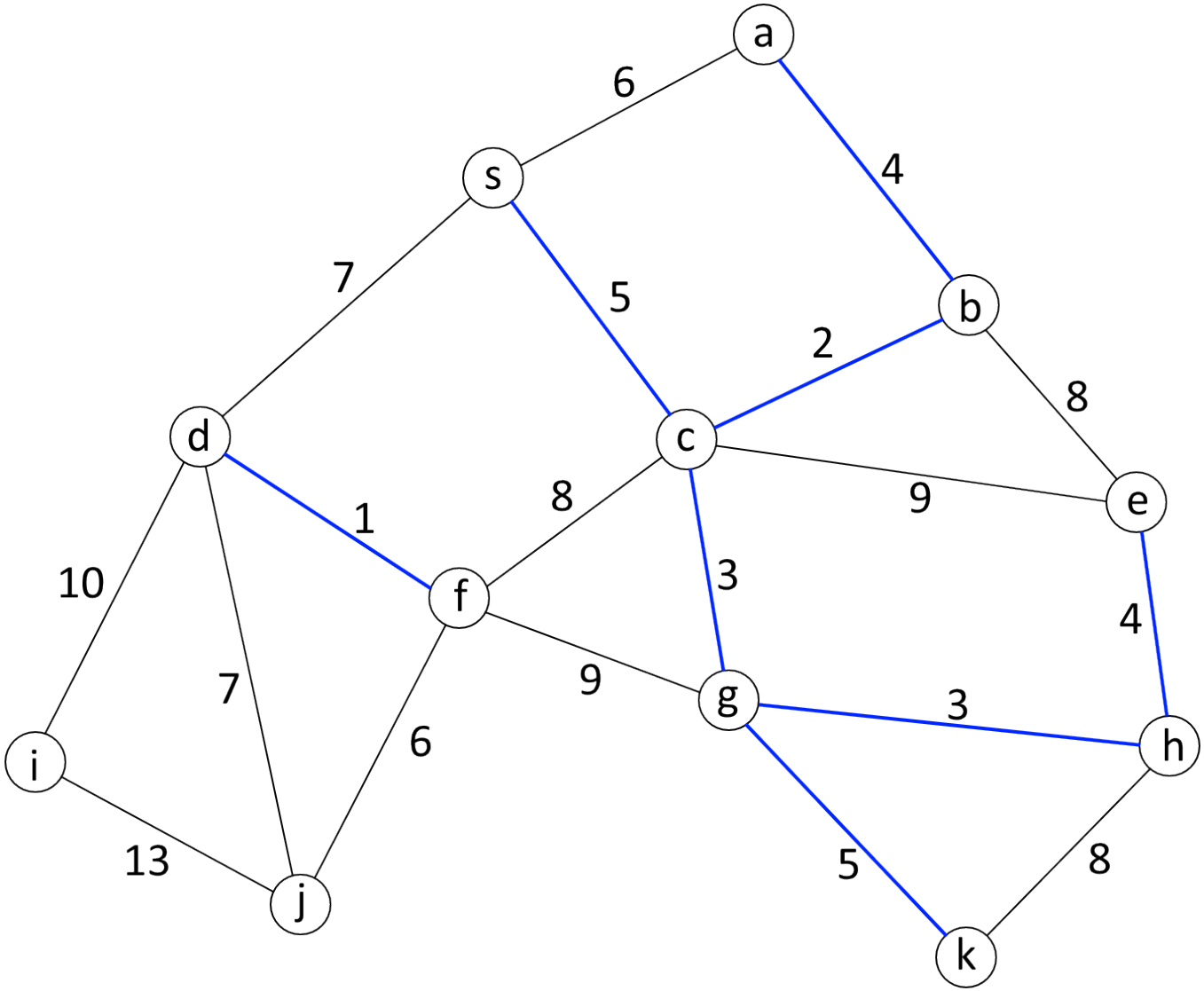
# Spannbäume – Beispiel Kruskal



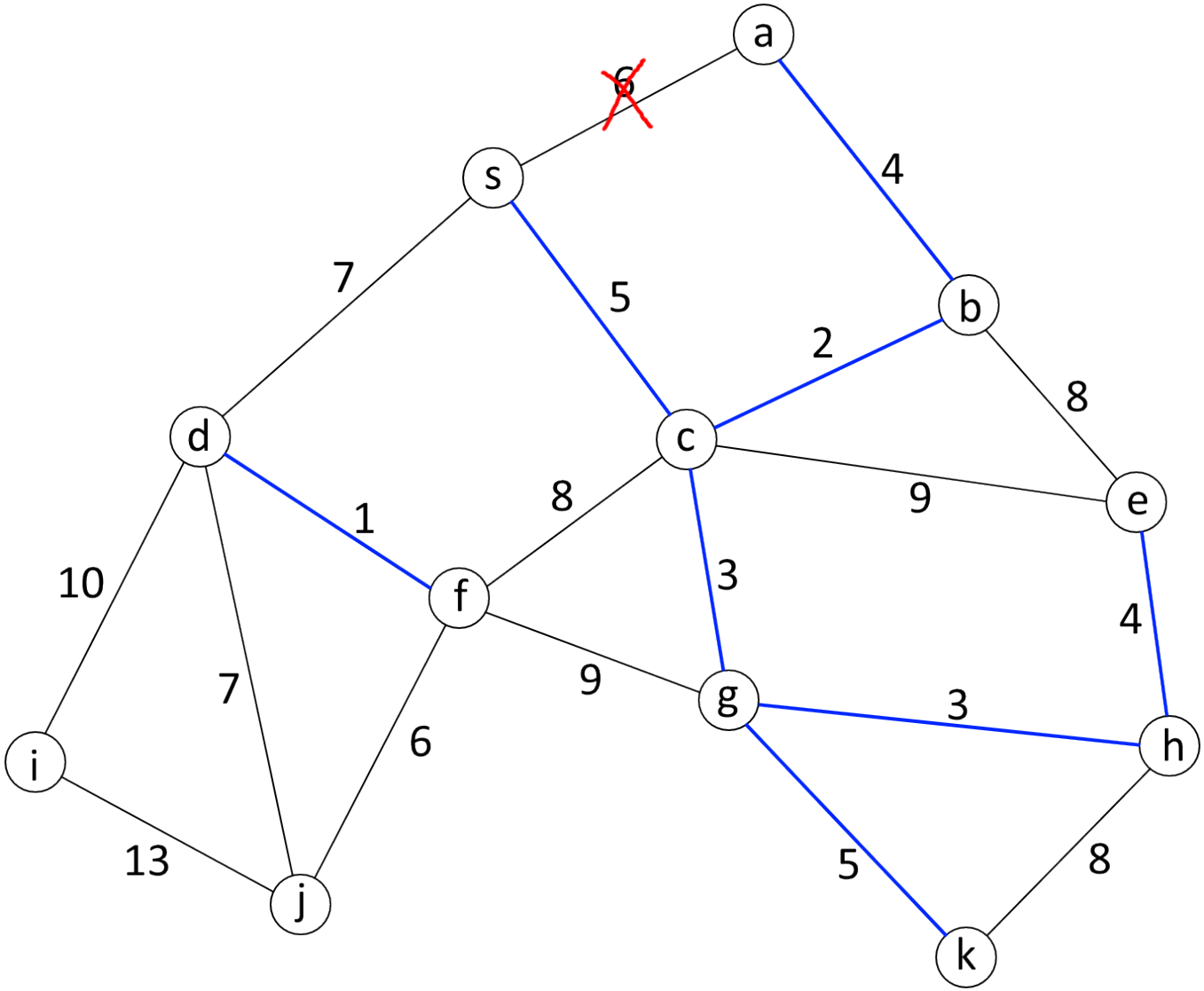
# Spannbäume – Beispiel Kruskal



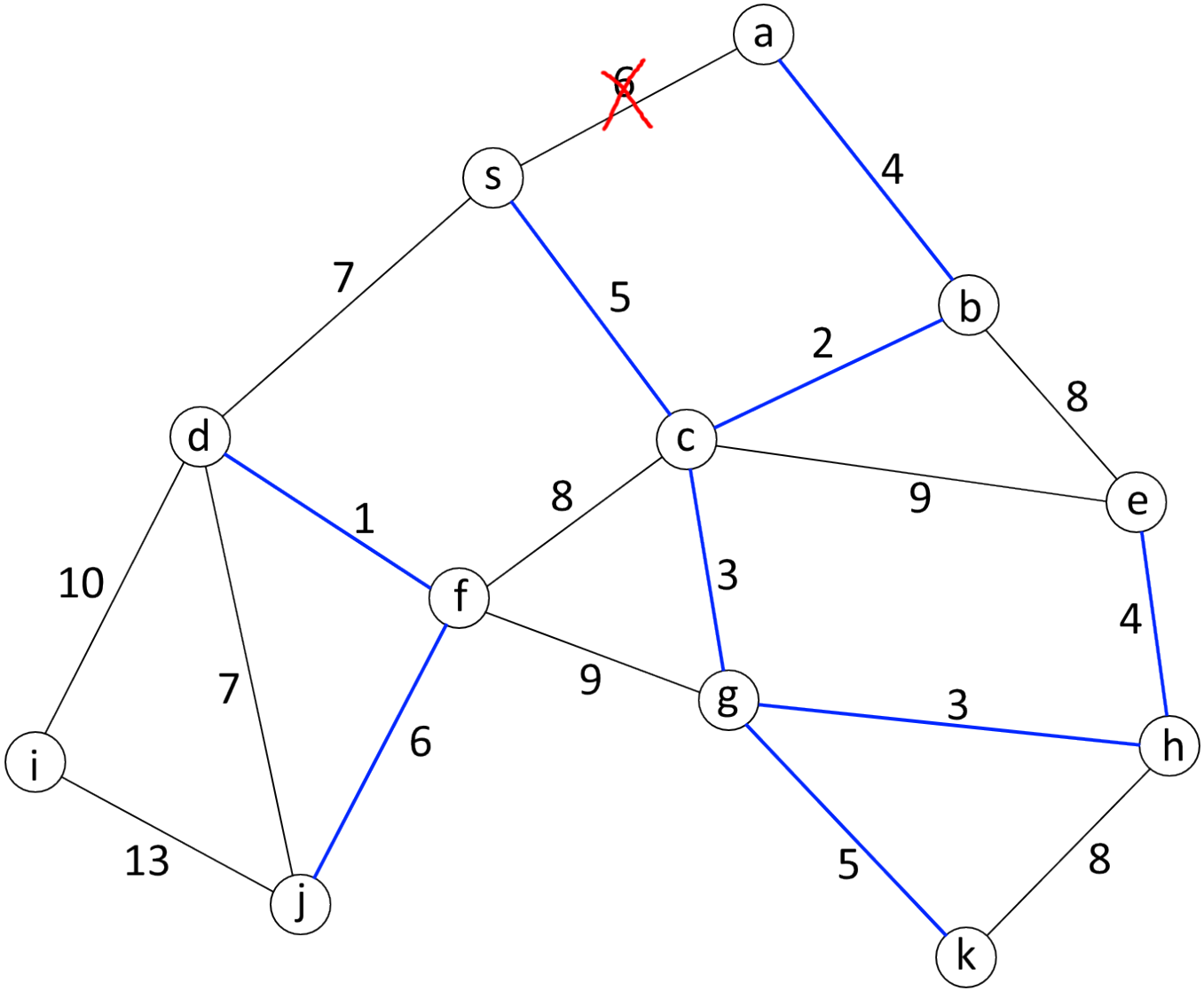
# Spannbäume – Beispiel Kruskal



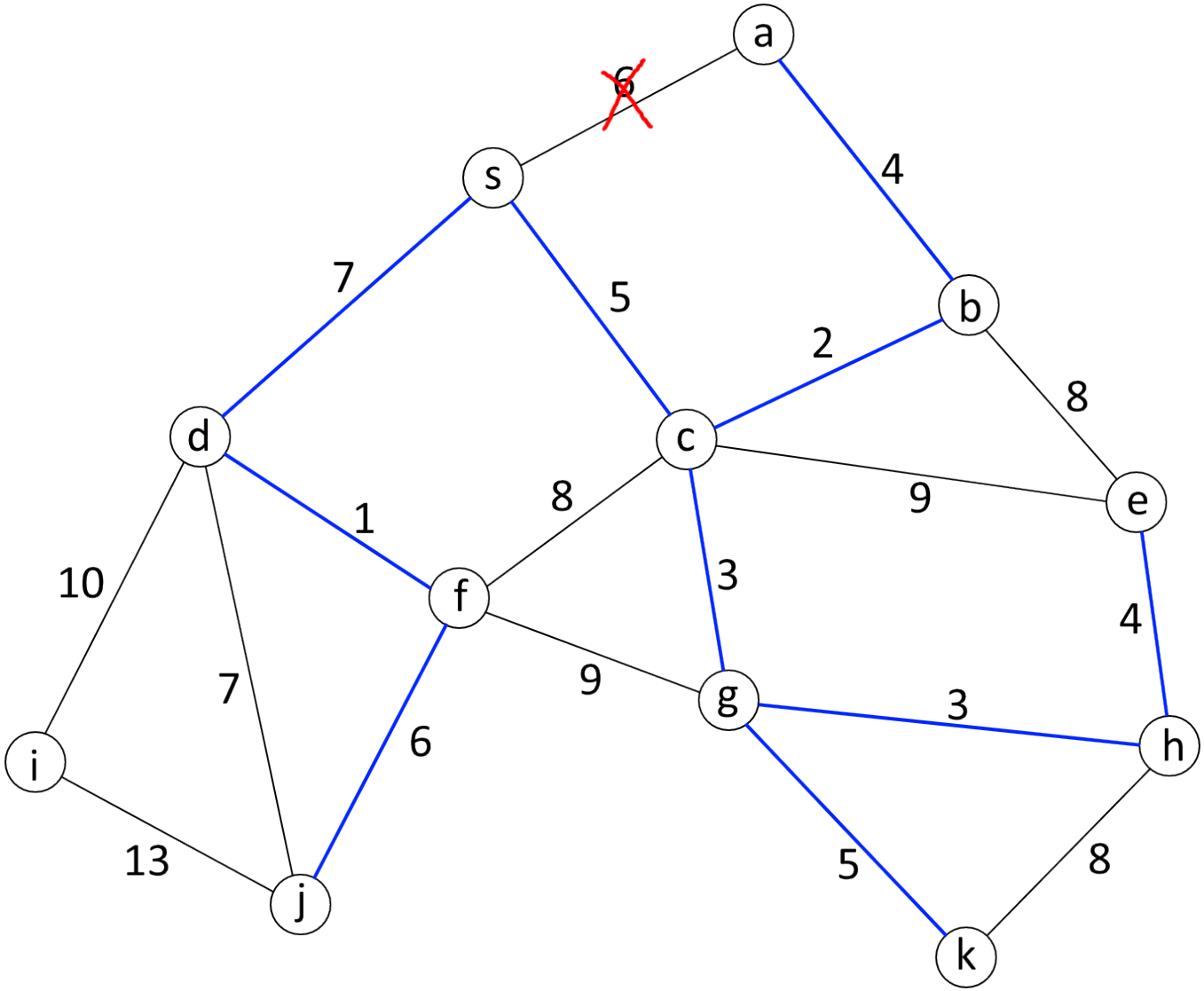
# Spannbäume – Beispiel Kruskal



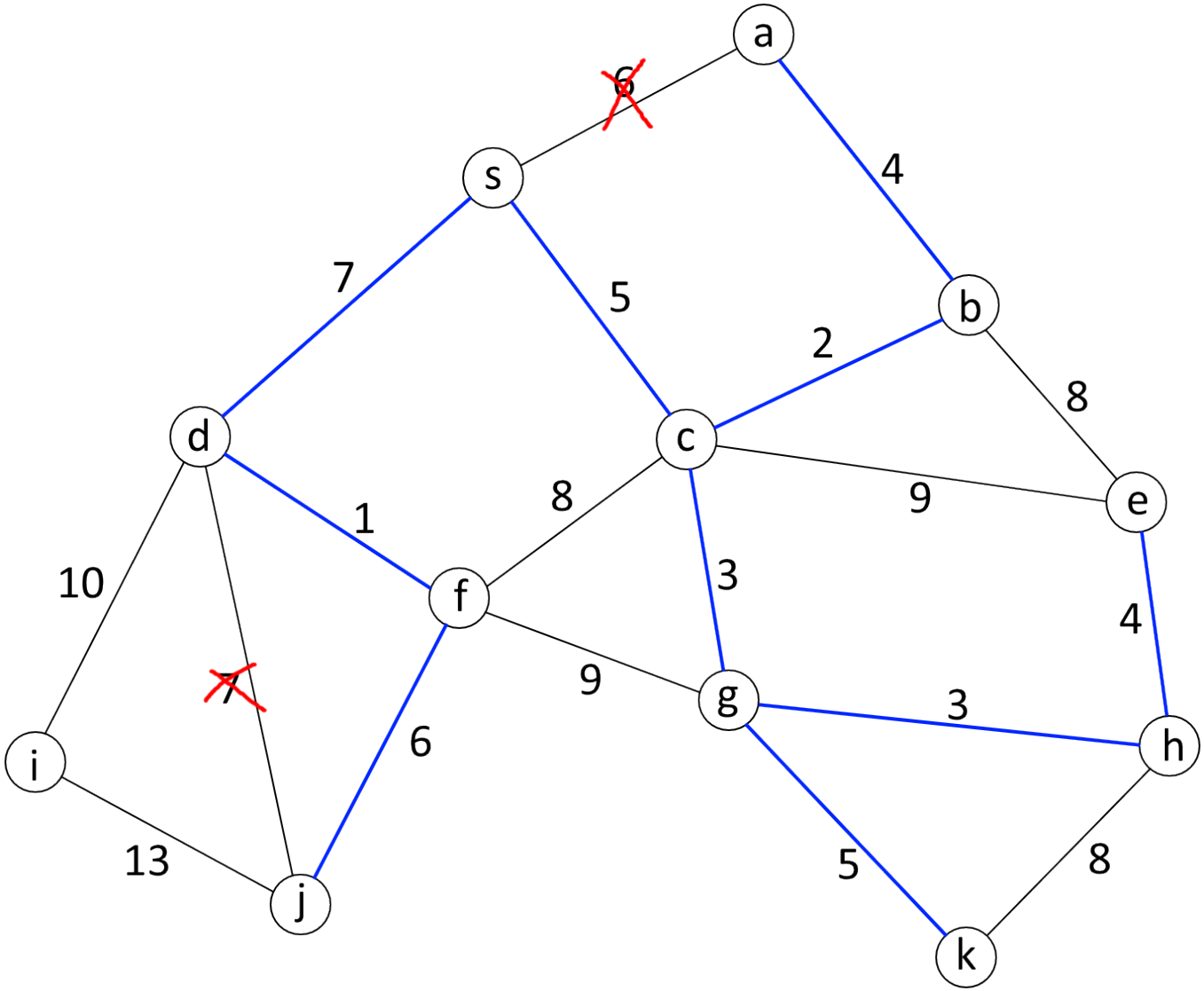
# Spannbäume – Beispiel Kruskal



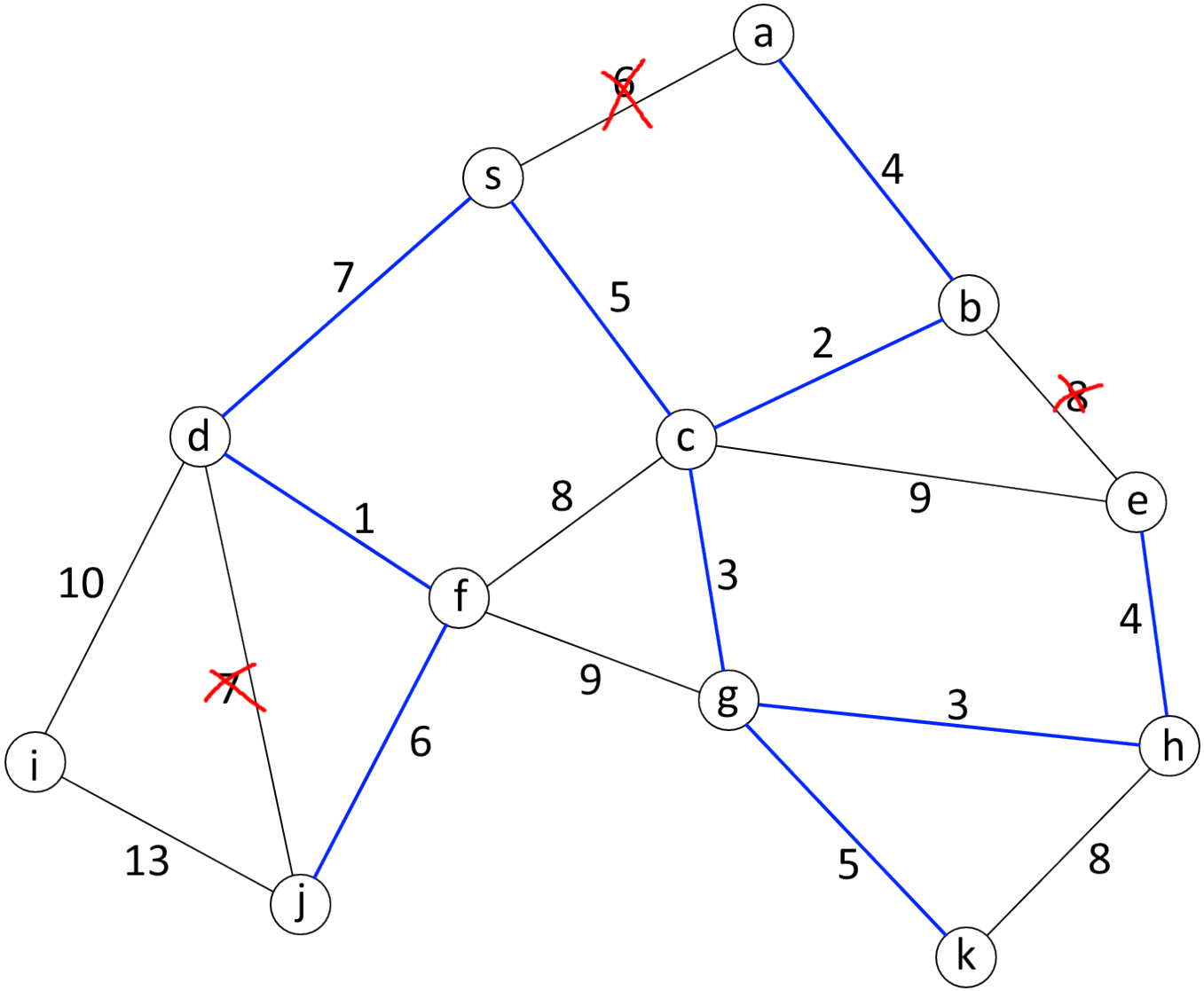
# Spannbäume – Beispiel Kruskal



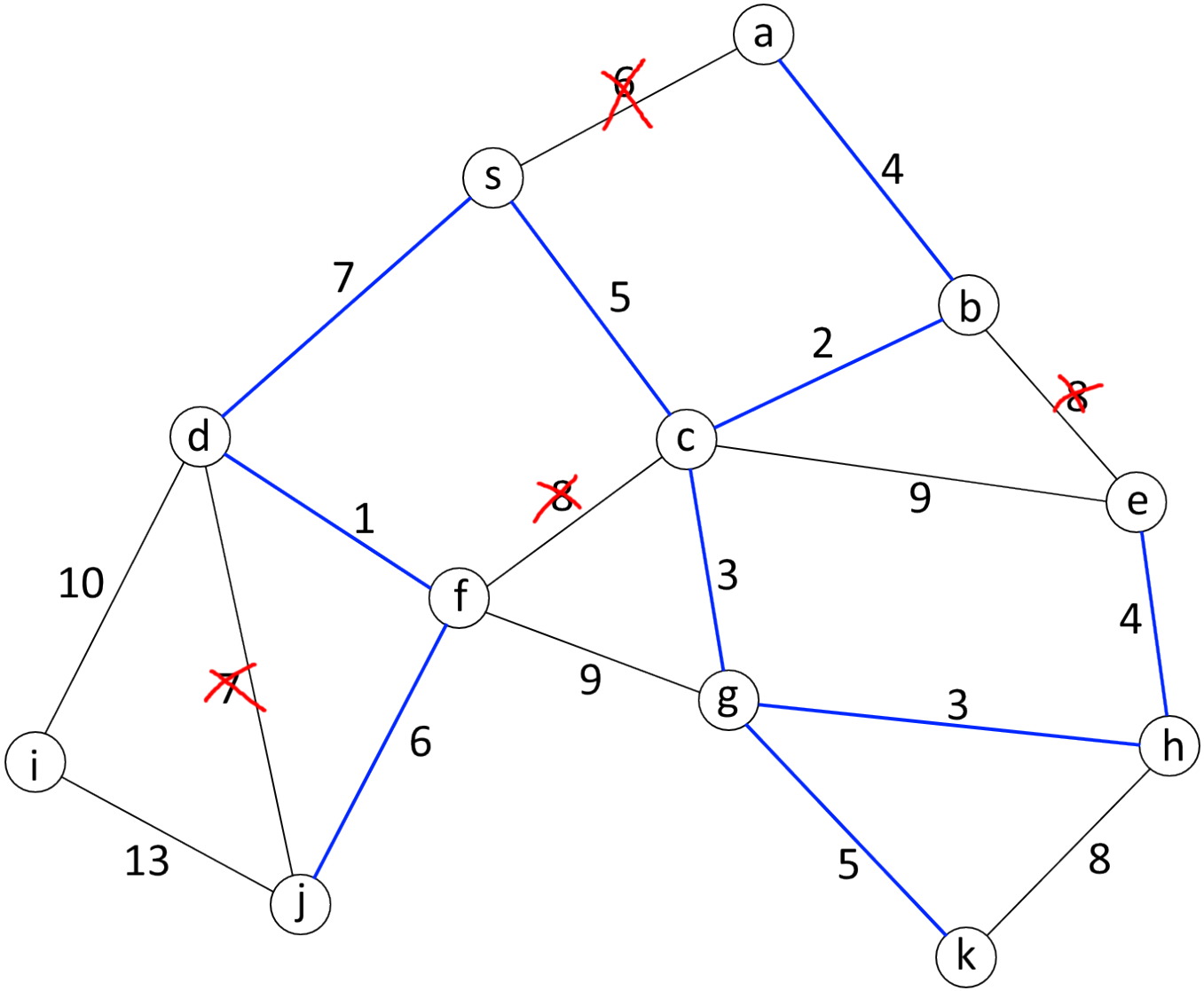
# Spannbäume – Beispiel Kruskal



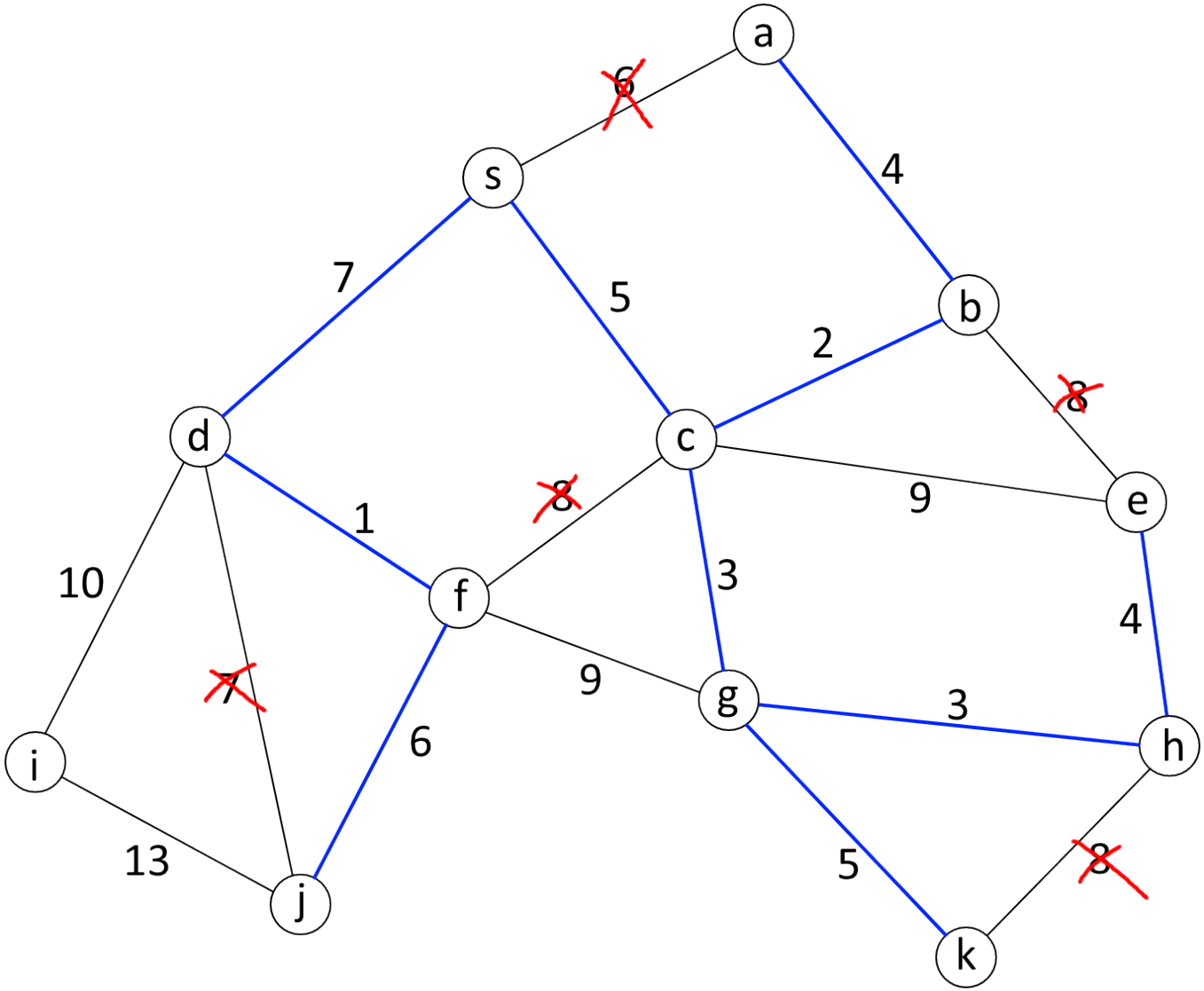
# Spannbäume – Beispiel Kruskal



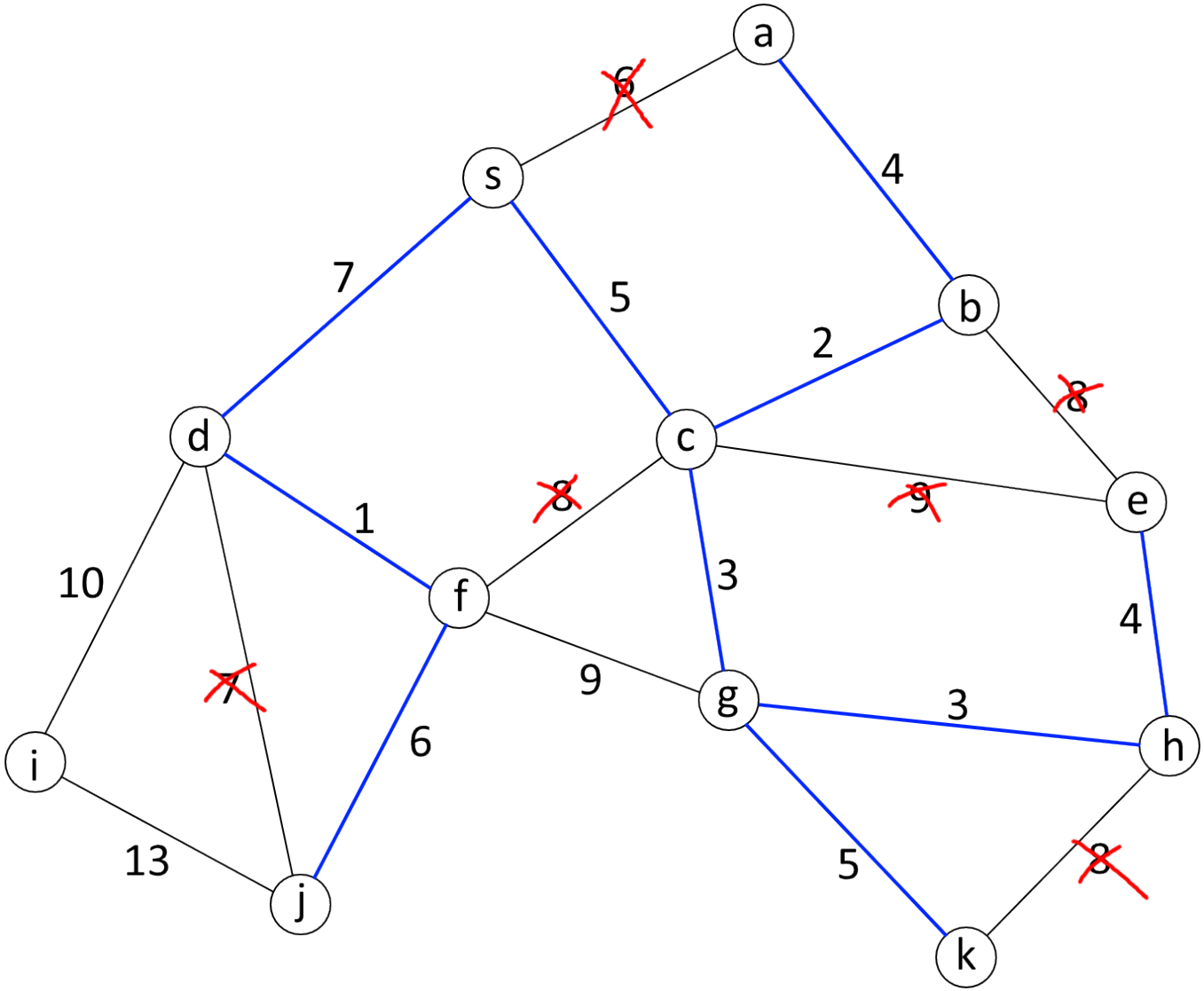
# Spannbäume – Beispiel Kruskal



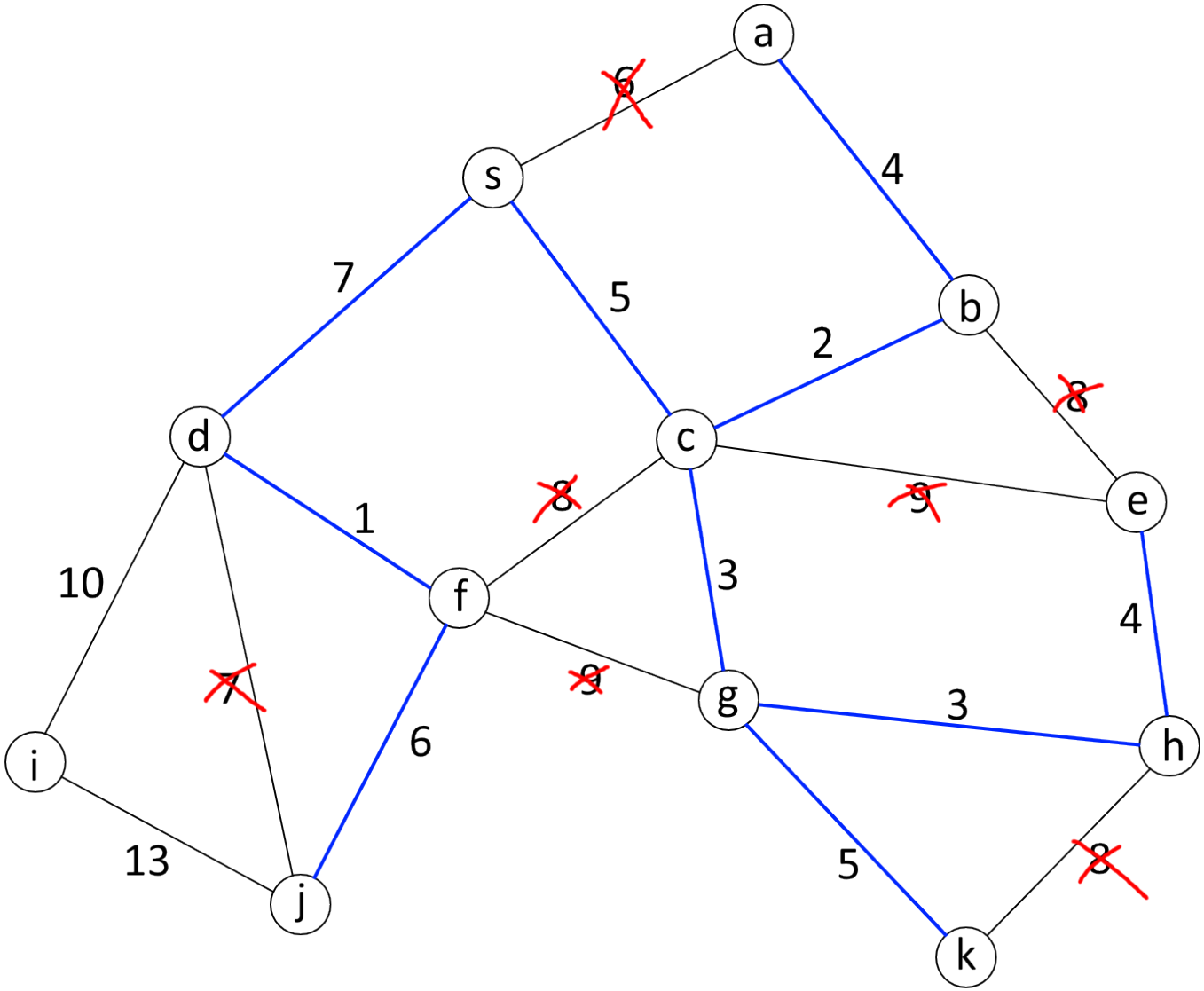
# Spannbäume – Beispiel Kruskal



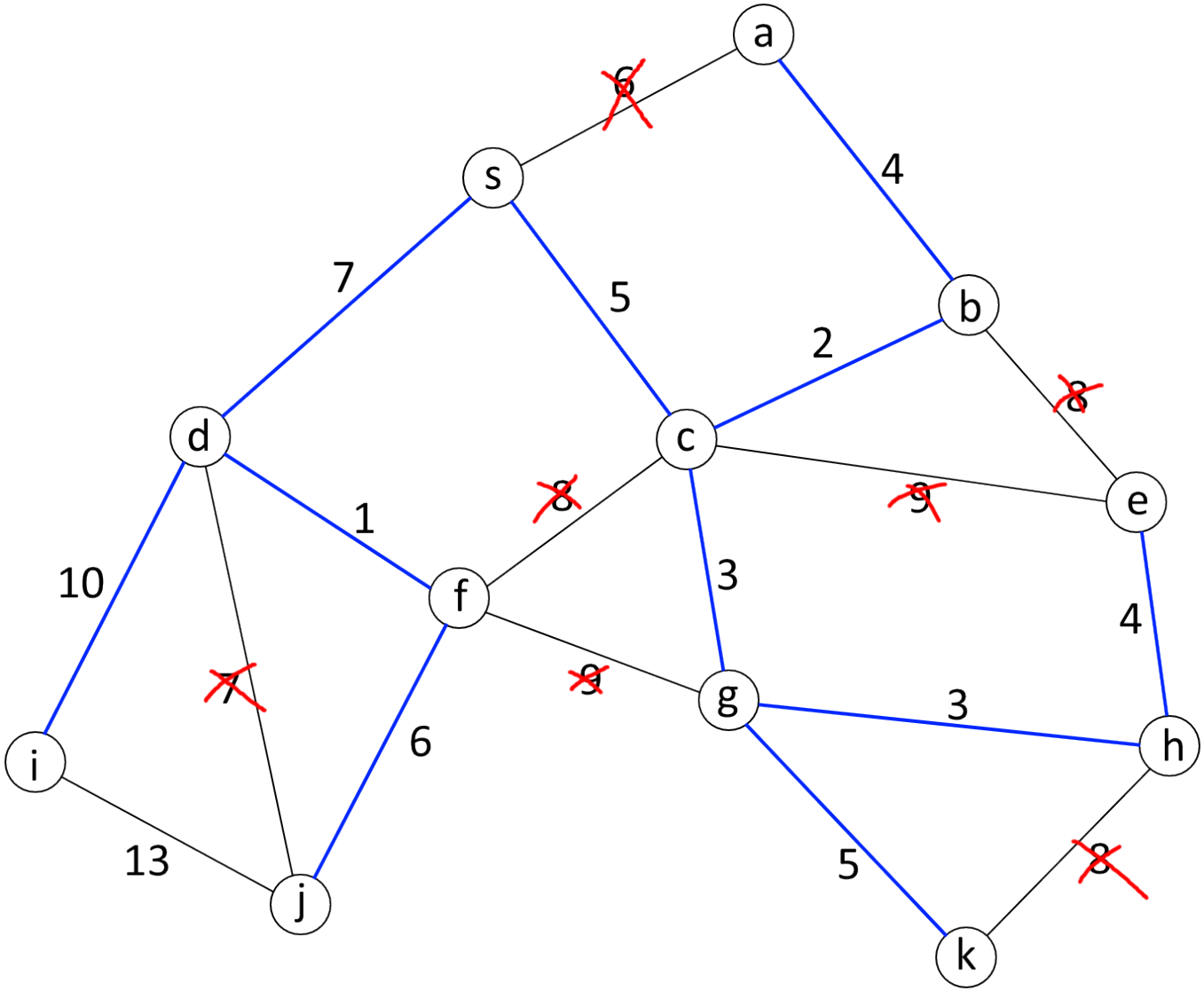
# Spannbäume – Beispiel Kruskal



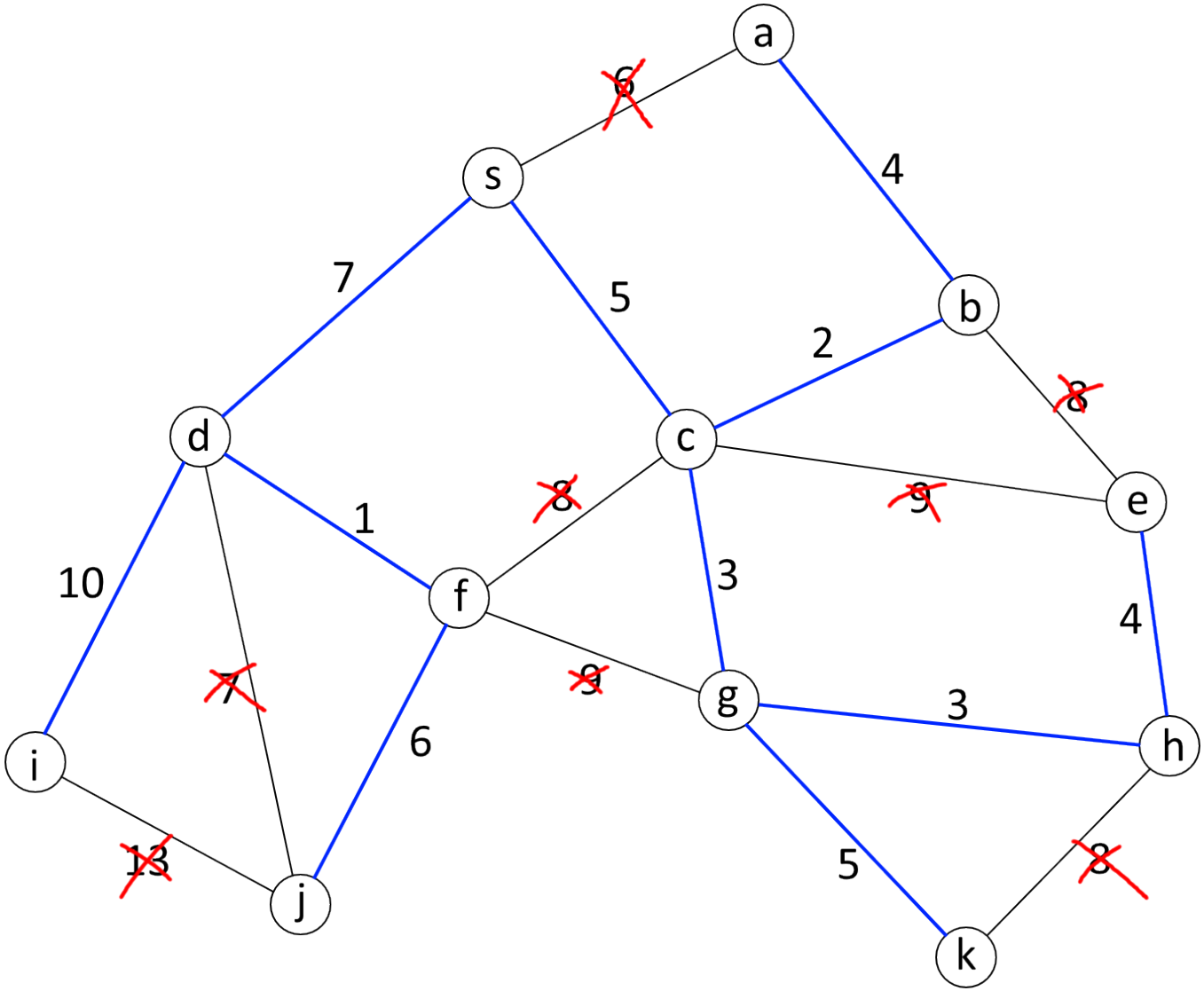
# Spannbäume – Beispiel Kruskal



# Spannbäume – Beispiel Kruskal



# Spannbäume – Beispiel Kruskal



# Union-Find

**Implementierung**

Wollen dafür effizient...

- ...heraus**finden**, zu welcher Menge ein Element gehört (*find*)
- ...die Mengen zweier Elemente **vereinigen** (*union*)

# Union-Find

$\Rightarrow$ **Eine neue Datenstruktur!**

Fürs Grobe...

- Repräsentiere Menge $M \subseteq \{1...n\}$ durch **beliebiges** Element $w \in M$
- Intern: $M$ wird als **Baum**, $w$ als **Wurzel** von $M$ behandelt

$\Rightarrow$ Verwalte $parent : \text{array}[1...n]$ of $1...n$, wobei

$parent[v] = v \Leftrightarrow v$ ist Wurzel

Am **Anfang**: $parent[v] := v$ // Alle Elemente sind eigene Wurzel

...und für die Effizienz

$\Rightarrow$ Verwalte $rank := (0, ..., 0) : \text{array}[1..n]$ of $0..\log n$, wobei

$$rank[v] := \begin{cases} \text{Höhe von Baum von } v \\ \text{(ohne Pfadkompression)} \end{cases} \quad v \text{ ist Wurzel}$$

$$garbage \quad v \text{ ist keine Wurzel}$$

# Union-Find

$\Rightarrow$ **Eine neue Datenstruktur!**

Fürs Grobe...

- Repräsentiere Menge $M \subseteq \{1...n\}$ durch **beliebiges** Element $w \in M$
- Intern: $M$ wird als **Baum**, $w$ als **Wurzel** von $M$ behandelt

$\Rightarrow$ Verwalte $parent : \text{array}[1...n]$ of $1...n$, wobei

$parent[v] = v \Leftrightarrow v$ ist Wurzel

Am **Anfang**: $parent[v] := v$ // Alle Elemente sind eigene Wurzel

...und für die Effizienz

$\Rightarrow$ Verwalte $rank := (0, ..., 0) : \text{array}[1..n]$ of $0...\log n$, wobei

$$rank[v] := \begin{cases} \text{Höhe von Baum von } v \\ \text{(ohne Pfadkompression)} & v \text{ ist Wurzel} \\ garbage & v \text{ ist keine Wurzel} \end{cases}$$

# Union-Find

$\Rightarrow$ **Eine neue Datenstruktur!**

Fürs Grobe...

- Repräsentiere Menge $M \subseteq \{1...n\}$ durch **beliebiges** Element $w \in M$
- Intern: $M$ wird als **Baum**, $w$ als **Wurzel** von $M$ behandelt

$\Rightarrow$ Verwalte *parent* : **array**$[1...n]$ **of** $1...n$, wobei
$parent[v] = v \Leftrightarrow v$ ist Wurzel
Am **Anfang**: $parent[v] := v$ *// Alle Elemente sind eigene Wurzel*


...und für die Effizienz

$\Rightarrow$ Verwalte $rank = (0, ..., 0) : \mathbf{array}[1..n]$ **of** $0...\log n$, wobei

$$rank[v] = \begin{cases} \text{Höhe von Baum von } v \\ \text{(ohne Pfadkompression)} \\ garbage, \end{cases} \quad \begin{array}{l} v \text{ ist Wurzel} \\ \\ v \text{ ist keine Wurzel} \end{array}$$

# Union-Find

$\Rightarrow$ **Eine neue Datenstruktur!**

Fürs Grobe...

- Repräsentiere Menge $M \subseteq \{1...n\}$ durch **beliebiges** Element $w \in M$
- Intern: $M$ wird als **Baum**, $w$ als **Wurzel** von $M$ behandelt

$\Rightarrow$ Verwalte *parent* : **array**$[1...n]$ **of** $1...n$, wobei
$parent[v] = v \Leftrightarrow v$ ist Wurzel
Am **Anfang**: $parent[v] := v$ *// Alle Elemente sind eigene Wurzel*


...und für die Effizienz

$\Rightarrow$ Verwalte $rank = (0, ..., 0)$ : **array**$[1...n]$ **of** $0... \log n$, wobei

$$rank[v] = \begin{cases} \text{Höhe von Baum von } v \text{ (ohne Pfadkompression)}, & v \text{ ist Wurzel} \\ garbage, & v \text{ ist } \textbf{keine} \text{ Wurzel} \end{cases}$$

# Union-Find – Operationen

## Finden

```
function find(e : 1..n) : 1..n
    if parent[e] = e then
        return e
    else
        w := find(parent[e])
        // Pfadkompression
        parent[e] := w
        return w
```

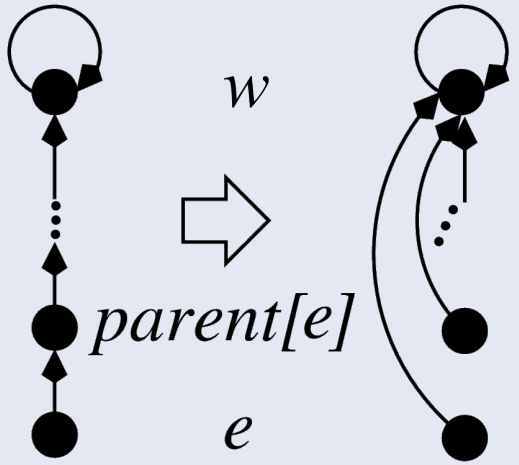



Abbildung: Pfadkompression

# Union-Find – Operationen

## Vereinigen

**procedure** union($a', b' : 1...n$)

$$\begin{pmatrix} a \\ b \end{pmatrix} := \begin{pmatrix} \text{find}(a') \\ \text{find}(b') \end{pmatrix}$$

**if** $a \neq b$ **then**

*// union by rank*

**if** $\text{rank}[a] < \text{rank}[b]$ **then**

$\text{parent}[a] := b$

**else**

$\text{parent}[b] := a$

**if** $\text{rank}[a] = \text{rank}[b]$ **then**

$\text{rank}[a]{+}{+}$

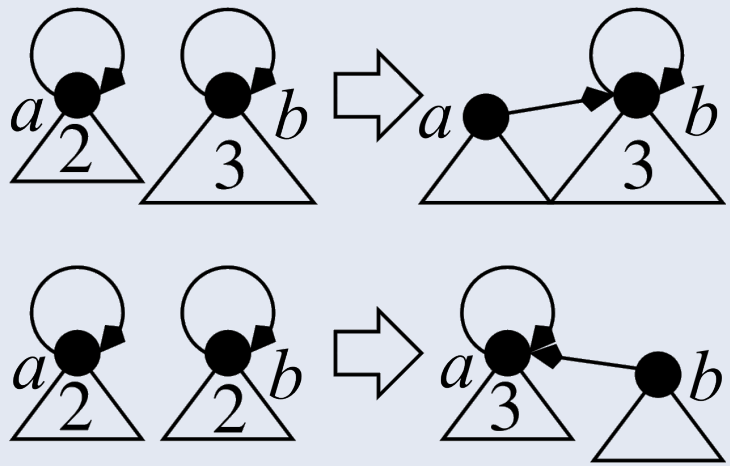



Abbildung: Union-by-rank

# Spannbäume – Kruskal

## Kruskal – Pseudocode

```
function Kruskal(G = (V, E))
    forest := new UnionFind(n)
    T := ∅
    Sort(E) by c(·)
    foreach e = {u, v} ∈ E do
        if forest.find(u) ≠ forest.find(v) then
            T := T ∪ {e}
            forest.union(u, v)
    return T
```

# Spannbäume – Kruskal

## Laufzeit von Kruskal

$O(m \log m)$ fürs Sortieren von $E$

$+ \; O\left(m \cdot \alpha_T(m,n)\right)$ für $m \times \text{find}() + n \times \text{union}()$ laut VL

$\approx O(m \cdot 5)$ (**Inv. Ackermannfunktion** $\alpha_T(\cdot,\cdot) \leqslant 5$ for all sane inputs)

$= O(m \log m)$.

Bei Kantengewichten in $\mathbb{Z}_{\leqslant L}$ sogar schneller möglich.

# Spannbäume – Kruskal

## Laufzeit von Kruskal

$$
\begin{aligned}
& O(m \log m) && \text{fürs Sortieren von } E \\
+\ & O\left(m \cdot \alpha_T(m, n)\right) && \text{für } m \times \text{find}() + n \times \text{union}() \text{ laut VL} \\
& \approx O(m \cdot 5) && (\textbf{Inv. Ackermannfunktion } \alpha_T(\cdot, \cdot) \leqslant 5 \text{ for all sane inputs}) \\
= & O(m \log m).
\end{aligned}
$$

Bei Kantengewichten in $\mathbb{Z}_{\leq}$ sogar schneller möglich.

# Spannbäume – Kruskal

## Laufzeit von Kruskal

$$
\begin{aligned}
& O(m \log m) && \text{fürs Sortieren von } E \\
+\ & O\left(m \cdot \alpha_T(m, n)\right) && \text{für } m \times \text{find}() + n \times \text{union}() \text{ laut VL} \\
& \approx O(m \cdot 5) && (\textbf{Inv. Ackermannfunktion } \alpha_T(\cdot, \cdot) \leqslant 5 \text{ for all sane inputs}) \\
=\ & O(m \log m). && \\
& \text{Bei Kantengewichten in } \mathbb{Z}_+ \text{ sogar schneller möglich.} &&
\end{aligned}
$$

# Spannbäume

Die Union-Find-Datenstruktur bei Kruskals Algorithmus repräsentiert den bisher gefundenen MST. **?**

# Spannbäume

Die Union-Find-Datenstruktur bei Kruskals Algorithmus repräsentiert den bisher gefundenen MST. **Falsch.**

Bloß interne Hierarchie für die Knoten. MST-Kanten tauchen da drin gar nicht auf.

# Spannbäume

Die Union-Find-Datenstruktur bei Kruskals Algorithmus repräsentiert den bisher gefundenen MST. **Falsch.**

Bloß interne Hierarchie für die Knoten. MST-Kanten tauchen da drin gar nicht auf.

Dijkstra ist zur Bestimmung eines MST bei gerichteten Graphen (mit nichtnegativen Kantengewichten) geeignet. **?**

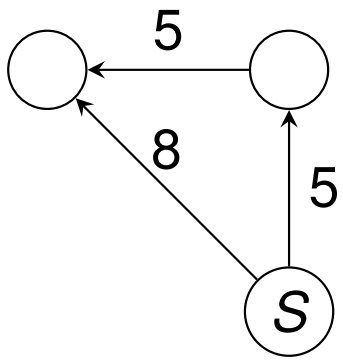
# Spannbäume

Die Union-Find-Datenstruktur bei Kruskals Algorithmus repräsentiert den bisher gefundenen MST. **Falsch.**

Bloß interne Hierarchie für die Knoten. MST-Kanten tauchen da drin gar nicht auf.

Dijkstra ist zur Bestimmung eines MST bei gerichteten Graphen (mit nichtnegativen Kantengewichten) geeignet. **Falsch.**

Dijkstra bestimmt im Allgemeinen **keinen MST** (und MST auf gerichteten Graphen nicht in dieser VL).



# Spannbäume

Sowohl der Algorithmus von Jarník-Prim als auch Kruskals Algorithmus funktionieren auch bei negativen Kantengewichten. **?**

# Spannbäume

Sowohl der Algorithmus von Jarník-Prim als auch Kruskals Algorithmus funktionieren auch bei negativen Kantengewichten. **Wahr.**

(JP braucht kleine Anpassung: $d[u]$ muss beim Rausholen aus der PQ auf $-\infty$ gesetzt werden.)

# Spannbäume

Sowohl der Algorithmus von Jarník-Prim als auch Kruskals Algorithmus funktionieren auch bei negativen Kantengewichten. **Wahr.**

(JP braucht kleine Anpassung: $d[u]$ muss beim Rausholen aus der PQ auf $-\infty$ gesetzt werden.)

Dijkstra funktioniert nicht, wenn negative Kantengewichte vorhanden sind. **?**

# Spannbäume

Sowohl der Algorithmus von Jarník-Prim als auch Kruskals Algorithmus funktionieren auch bei negativen Kantengewichten. **Wahr.**

(JP braucht kleine Anpassung: $d[u]$ muss beim Rausholen aus der PQ auf $-\infty$ gesetzt werden.)

Dijkstra funktioniert nicht, wenn negative Kantengewichte vorhanden sind. **Falsch.**

Dijkstra funktioniert nicht, wenn es negative **Zyklen** gibt ($\Rightarrow$ Endlosschleife).

Ansonsten bestimmt Dijkstra auch **mit** negativen Kantengewichten **korrekte kürzeste Pfade**, aber in **deutlich schlechterer Laufzeit**.

# Spannbäume

Sowohl der Algorithmus von Jarník-Prim als auch Kruskals Algorithmus funktionieren auch bei negativen Kantengewichten. **Wahr.**

(JP braucht kleine Anpassung: $d[u]$ muss beim Rausholen aus der PQ auf $-\infty$ gesetzt werden.)

Dijkstra funktioniert nicht, wenn negative Kantengewichte vorhanden sind. **Falsch.**

Dijkstra funktioniert nicht, wenn es negative **Zyklen** gibt ($\Rightarrow$ Endlosschleife).
Ansonsten bestimmt Dijkstra auch **mit** negativen Kantengewichten **korrekte kürzeste Pfade**, aber in **deutlich schlechterer Laufzeit**.

Bellman-Ford bestimmt stets einen **beliebigen** Spannbaum. **?**

# Spannbäume

Sowohl der Algorithmus von Jarník-Prim als auch Kruskals Algorithmus funktionieren auch bei negativen Kantengewichten. **Wahr.**

(JP braucht kleine Anpassung: $d[u]$ muss beim Rausholen aus der PQ auf $-\infty$ gesetzt werden.)

Dijkstra funktioniert nicht, wenn negative Kantengewichte vorhanden sind. **Falsch.**

Dijkstra funktioniert nicht, wenn es negative **Zyklen** gibt ($\Rightarrow$ Endlosschleife).
Ansonsten bestimmt Dijkstra auch **mit** negativen Kantengewichten **korrekte kürzeste Pfade**, aber in **deutlich schlechterer Laufzeit**.

Bellman-Ford bestimmt stets einen **beliebigen** Spannbaum. **Falsch.**

Nur wenn keine negativen Zyklen vorhanden sind!

# Spannbäume

**Aufgabe 2: Streaming MST**

Gegeben sei ein ungerichteter zusammenhängender Graph $G = (V, E)$ mit $n$ Knoten, $m$ Kanten und positiven Kantengewichten. Die Knoten sind lokal gespeichert, die Kanten sind hingegen zunächst **unbekannt** und können nur **stückweise** (und in zufälliger Reihenfolge) aus dem Netz angefordert und im Speicher gehalten werden, da **nur** $O(n)$ **Platz** zur Verfügung steht. Gebt einen Algorithmus an, der unter diesen Einschränkungen einen MST von $G$ bestimmt.

## Lösung zu Aufgabe 2

- Verwende eine **Union-Find-Datenstruktur** (wie bei Kruskal).
- Falls die neu angeforderte Kante **zwei Teilbäume verbindet**, füge sie hinzu.
- Falls sie zwei Knoten **im selben** Teilbaum verbindet, füge sie provisorisch hinzu, schmeiße auf dem Pfad zwischen den Knoten die **schwerste Kante** raus.
- **Laufzeit** in $O(m \cdot n)$, da bei der Bestimmung des Pfades maximal $n - 1$ Kanten abgelaufen werden (es sind zu jedem Zeitpunkt maximal $n$ Kanten im Graphen enthalten).

# Spannbäume

**Aufgabe 3: Streaming MST mit Dünger**

Wie bei Aufgabe 2:

> Gegeben sei ein ungerichteter zusammenhängender Graph $G = (V, E)$ mit $n$ Knoten, $m$ Kanten und positiven Kantengewichten. Die Knoten sind lokal gespeichert, die Kanten sind hingegen zunächst **unbekannt** und können nur **stückweise** (und in zufälliger Reihenfolge) aus dem Netz angefordert und im Speicher gehalten werden, da **nur** $O(n)$ **Platz** zur Verfügung steht. Gebt einen Algorithmus an, der unter diesen Einschränkungen einen MST von $G$ bestimmt.

Er darf **nur** $O(m \log n)$ **Rechenzeit benötigen**.

# Spannbäume

**Lösung zu Aufgabe 3**

Wir besorgen uns Kanten-Pakete mit jeweils $n$ Kanten.
Mit dem **ersten** Paket bestimmen wir (mit Kruskal) einen
Minimum Spanning Forest (MSF) und entfernen alle anderen Kanten.
**Restliche** Pakete behandeln wir so:

- Füge die $n$ neuen Kanten zum Graphen hinzu
- Bestimme auf diesem neuen Graphen mit (maximal) $2n - 1$ Kanten einen MSF und entferne alle anderen Kanten

$\Rightarrow$ **Laufzeit** in $O(m \log n)$, denn:
Jeder dieser ca. $\frac{m}{n}$ Schritte benötigt Zeit in $O(n \log n)$.

# Spannbäume

**Aufgabe 4: Ein Algorithmus mit Ecken und Kanten**

Erneut betrachten wir einen ungerichteten zusammenhängenden Graphen $G = (V, E)$ mit $V = \{1...n\}$ und Kantengewichten in $\{1, 3\}$. $G$ sei in Form eines Adjazenzfeldes gegeben. Gebt einen Algorithmus an, der in $O(m)$ einen MST von $G$ berechnet (und begründet das Laufzeitverhalten).

# Spannbäume

## Lösung zu Aufgabe 4

Getweaktes **Jarník-Prim**:

- Komplette PriorityQueue wäre **overkill** $\Rightarrow$ **stattdessen** zwei einfache Queues $Q_1$ und $Q_3$ (eine für jedes Kantengewicht)
- Merke zu jedem Knoten Pointer in die Queue (oder $\perp$)

$\Rightarrow$ *insert*, *deleteMin* und *decreaseKey* in $O(1)$ durch einfaches Pointer-Umhängen

$\Rightarrow$ **Laufzeit**: $O(n+m) = O(m)$
(da für zusammenhängende Graphen $n \in O(m)$).

(Eine clevere Alternative wäre, die Kanten mit **Bucketsort** zu sortieren und dann **Kruskal** draufloszulassen $\Rightarrow$ Laufzeit: $O\left(m \cdot \alpha_T(m,n)\right) \overset{\text{realistisch}}{\approx} O(5 \cdot m) = O(m)$. Aber Achtung, $\alpha_T$ ist **nicht** konstant, deshalb „$\approx$"!)

# Spannbäume

## Lösung zu Aufgabe 4

Getweaktes **Jarník-Prim**:

- Komplette PriorityQueue wäre **overkill** $\Rightarrow$ **stattdessen** zwei einfache Queues $Q_1$ und $Q_3$ (eine für jedes Kantengewicht)
- Merke zu jedem Knoten Pointer in die Queue (oder $\perp$)

$\Rightarrow$ *insert*, *deleteMin* und *decreaseKey* in $O(1)$ durch einfaches Pointer-Umhängen

$\Rightarrow$ **Laufzeit**: $O(n+m) = O(m)$
(da für zusammenhängende Graphen $n \in O(m)$).

(Eine clevere Alternative wäre, die Kanten mit **Bucketsort** zu sortieren und dann **Kruskal** draufloszulassen $\Rightarrow$ Laufzeit:
$O\left(m \cdot \alpha_T(m,n)\right) \overset{\text{realistisch}}{\approx} O(5 \cdot m) = O(m)$. Aber Achtung, $\alpha_T$ ist **nicht** konstant, deshalb „$\approx$“!)

# Danke für eure Aufmerksamkeit! ☺