16. Minimale Spannbäume

Definition 16.1:

1. Ein gewichteter ungerichteter Graph (G,w) ist ein ungerichteter Graph $G=(V,E)$ zusammen mit einer Gewichtsfunktion $w:E\rightarrow\mathbb{R}$

2. Ist $H=(U,F)$, $U\subseteq V$, $F\subseteq E$, ein Teilgraph von G, so ist das Gewicht $w(H)$ von H definiert als

$$w(H) = \sum_{e\in F} w(e)$$

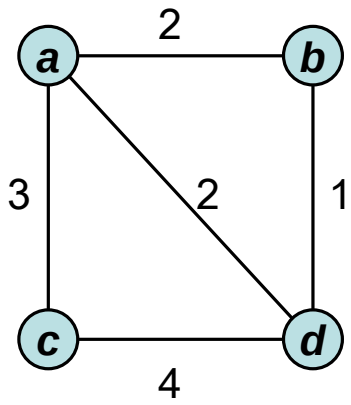
Minimale Spannbäume

Definition 16.2:

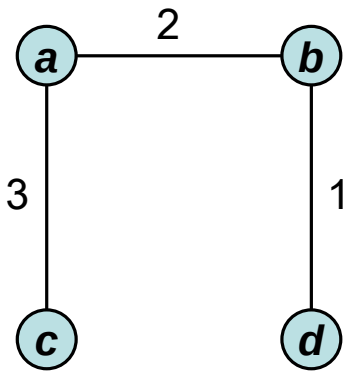
1. Ein Teilgraph eines ungerichteten Graphen G heißt Spannbaum von G=(V,E), wenn H ein Baum auf allen Knoten von G ist.

2. Ein Spannbaum S eines gewichteten, ungerichteten Graphen G heißt minimaler Spannbaum von G, wenn S minimales Gewicht unter allen Spannbäumen von G besitzt.

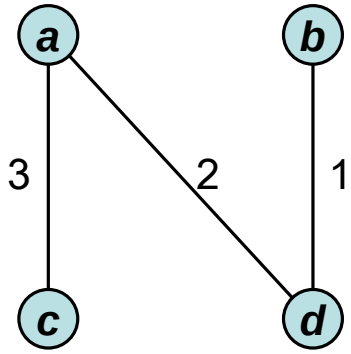
Illustration von minimalen Spannbäumen (1)



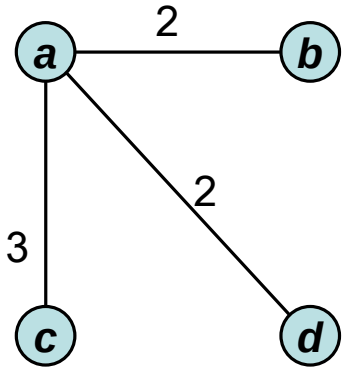
Graph G=(V,E)



Spannbaum für G (minimal)

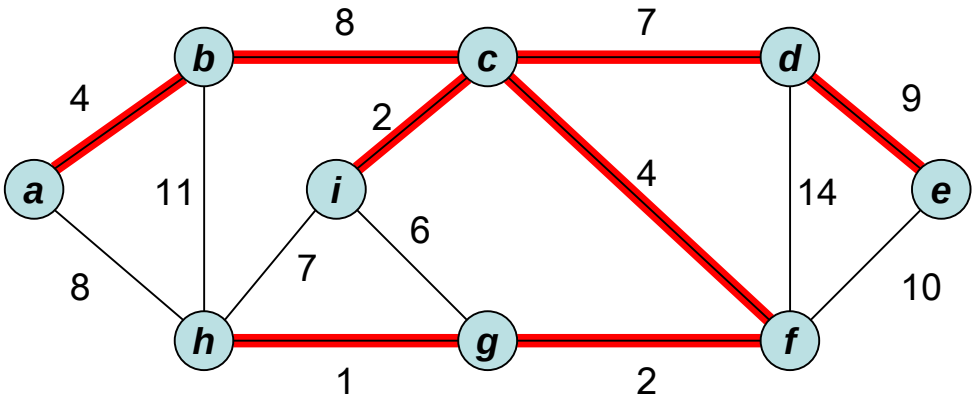


Spannbaum für G (minimal)



Spannbaum für G (nicht minimal)

Illustration von minimalen Spannbäumen (2)



Berechnung minimaler Spannbäume

Ziel: Gegeben ein gewichteter ungerichteter Graph (G,w) mit $G=(V,E)$, finde effizient einen minimalen Spannbaum von (G,w).

Vorgehensweise: Wir erweitern sukzessive eine Kantenmenge $A \subseteq E$ zu einem minimalen Spannbaum

1. Zu Beginn ist $A=\{\ \}$.
2. Wir ersetzen in jedem Schritt solange A durch $A \cup \{\ \{u,v\}\ \}$, wobei $\{u,v\}$ eine A-sichere Kante ist, bis $|A|=|V|-1$ ist.

Definition 16.3: $\{u,v\}$ heißt A-sicher, wenn mit A auch $A \cup \{\ \{u,v\}\ \}$ zu einem minimalen Spannbaum erweitert werden kann.

Generischer MST-Algorithmus

Generic-MST(G,w)

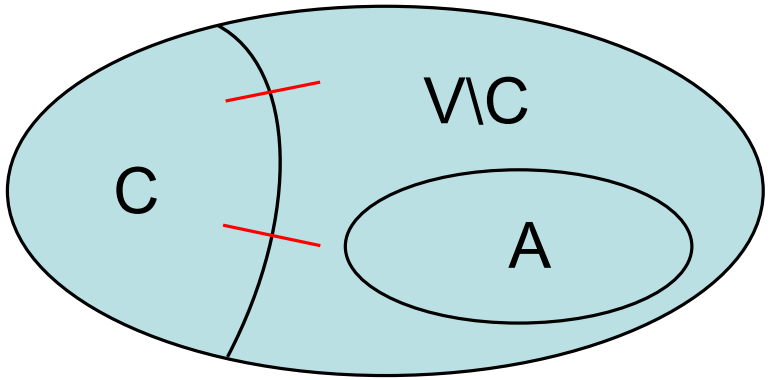
1. $A \leftarrow \{\ \}$
2. **while** A ist noch kein Spannbaum **do**
3. finde eine A-sichere Kante {u,v}
4. $A \leftarrow A \cup \{\ \{u,v\}\ \}$
5. **return** A

Korrektheit: ergibt sich aus der Definition A-sicherer Kanten

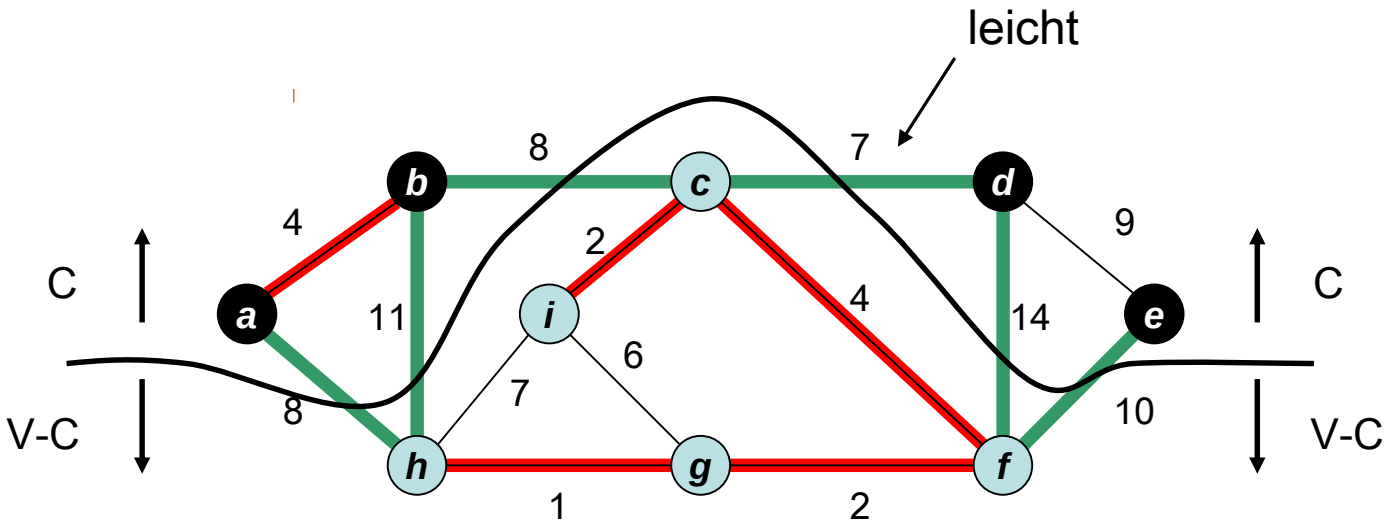
Schnitte in Graphen

Definition 16.4:

1. Ein Schnitt (C,V\C) in einem Graphen G=(V,E) ist eine Partition der Knotenmenge V des Graphen.
2. Eine Kante von G kreuzt einen Schnitt (V,V\C), wenn ein Knoten der Kante in C und der andere Knoten in V\C liegt.
3. Ein Schnitt (C,V\C) ist mit einer Teilmenge $A \subseteq E$ verträglich, wenn kein Element von A den Schnitt kreuzt.
4. Eine (C,V\C) kreuzende Kante heißt leicht, wenn sie eine Kante minimalen Gewichts unter den (C,V\C) kreuzenden Kanten ist.



Schnitt in einem Graphen (1)



Schnittkanten

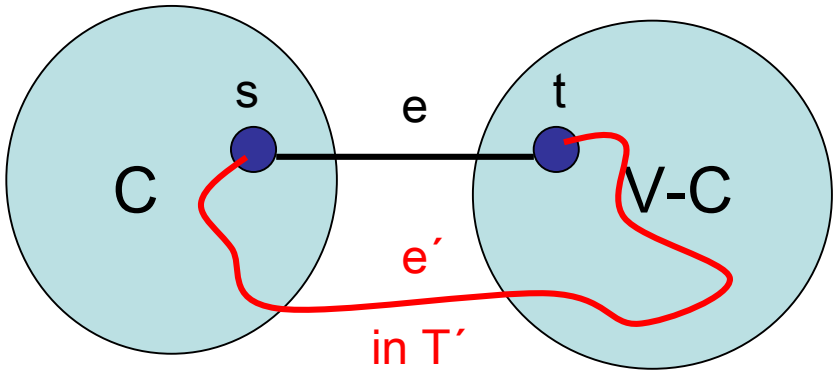
Charakterisierung sicherer Kanten

Satz 16.5: Sei (G,w) mit $G=(V,E)$ ein gewichteter ungerichteter Graph. Die Kantenmenge $A \subseteq E$ sei in einem minimalen Spannbaum von (G,w) enthalten. Weiter sei $(C,V\backslash C)$ ein mit A verträglicher Schnitt und $\{u,v\}$ sei eine leichte $(C,V\backslash C)$ kreuzende Kante. Dann ist $\{u,v\}$ eine A-sichere Kante.

Korollar 16.6: Sei (G,w) mit $G=(V,E)$ ein gewichteter ungerichteter Graph. Die Kantenmenge $A \subseteq E$ sei in einem minimalen Spannbaum von (G,w) enthalten. Ist $\{u,v\}$ ein Kante minimalen Gewichts, die eine Zusammenhangskomponente C von $G_A=(V,A)$ mit dem Rest des Graphen G_A verbindet, dann ist $\{u,v\}$ eine A-sichere Kante.

Beweis des Satzes - Illustration

- Betrachte beliebigen MSB T´, der A enthält
- (C,V\C): mit A verträglicher Schnitt
- e={s,t}: (C,V\C)-Kante minimalen Gewichts (e∉A)



- Ersetzung von e´ durch e führt zu Baum T, der e und A enthält und höchstens Kosten von MSB T´ hat, also MSB ist

Algorithmus von Prim - Idee

- Zu jedem Zeitpunkt des Algorithmus besteht der Graph $G_A=(V,A)$ aus einem Baum T_A und einer Menge von isolierten Knoten I_A.
- Eine Kante minimalen Gewichts, die einen Knoten in I_A mit T_A verbindet, wird zu A hinzugefügt.

Algorithmus von Prim - Idee

- Zu jedem Zeitpunkt des Algorithmus besteht der Graph $G_A=(V,A)$ aus einem Baum T_A und einer Menge von isolierten Knoten I_A.
- Eine Kante minimalen Gewichts, die einen Knoten in I_A mit T_A verbindet, wird zu A hinzugefügt.

- Die Knoten in I_A sind in einem Min-Heap organisiert. Dabei ist der Schlüssel $d[v]$ eines Knotens $v \in I_A$ gegeben durch das minimale Gewicht einer Kante, die v mit T_A verbindet.

Algorithmus von Prim - Pseudocode

Prim(G,w,s)

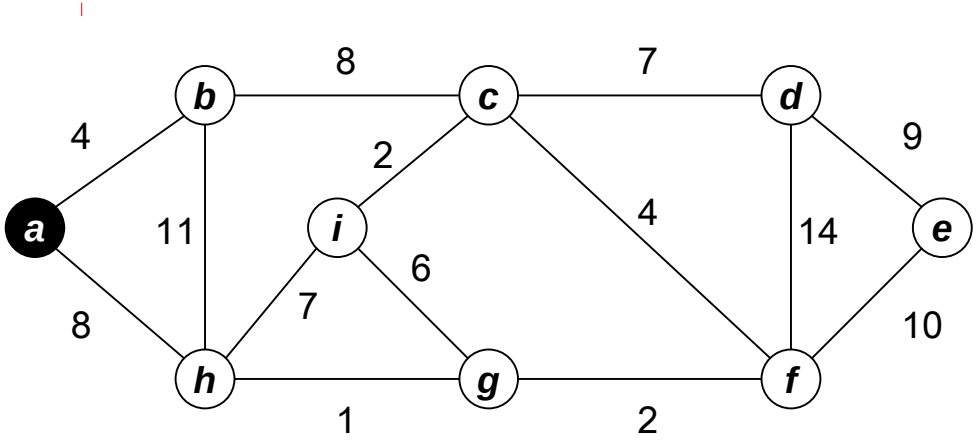
1. **for each** vertex $v \in V$ **do** $d[v] \leftarrow \infty$; $\pi[v] \leftarrow$ **nil**
2. $d[s] \leftarrow 0$; $Q \leftarrow$ BuildHeap(V)
3. **while** $Q \neq \emptyset$ **do**
4. u←deleteMin(Q)
5. **for each** vertex $v \in$ Adj[u] **do**
6. **if** $v \in Q$ and $d[v] > w(u,v)$ **then**
7. Decrease-Key(Q,v,w(u,v)); $\pi[v] \leftarrow u$

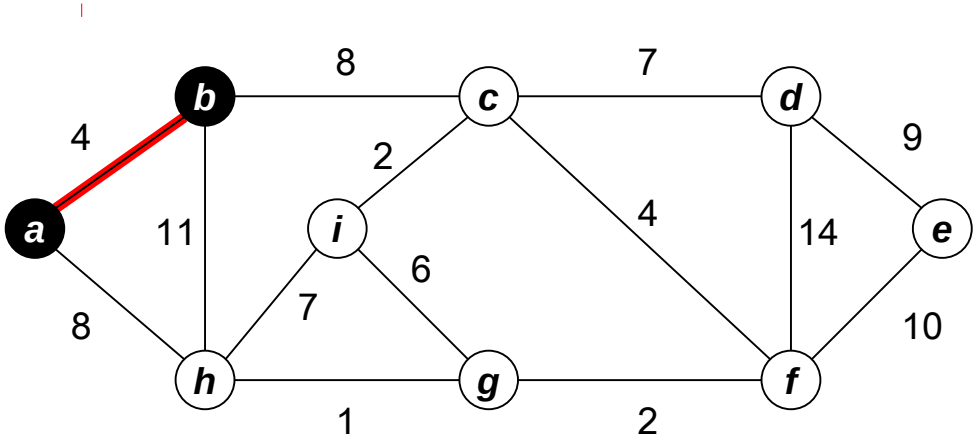
Vergleich Dijkstra zu Prim

Dijkstra(G,w,s)

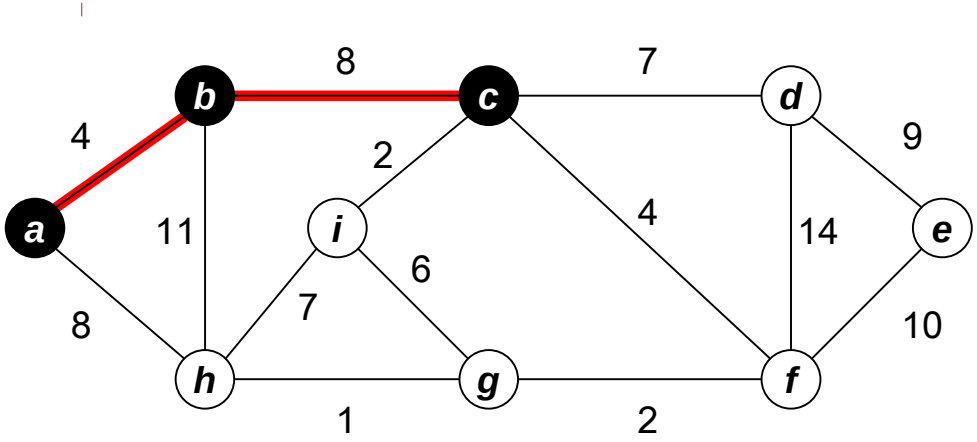
1. **for each** vertex $v \in V$ **do** $d[v] \leftarrow \infty$; $\pi[v] \leftarrow$ **nil**
2. $d[s] \leftarrow 0$; $Q \leftarrow$ BuildHeap(V)
3. **while** $Q \neq \emptyset$ **do**
4. u←deleteMin(Q)
5. **for each** vertex $v \in$ Adj[u] **do**
6. **if** d[v]>d[u]+w(u,v) **then**
7. Decrease-Key(Q,v,d[u]+w(u,v)); π[v]←u

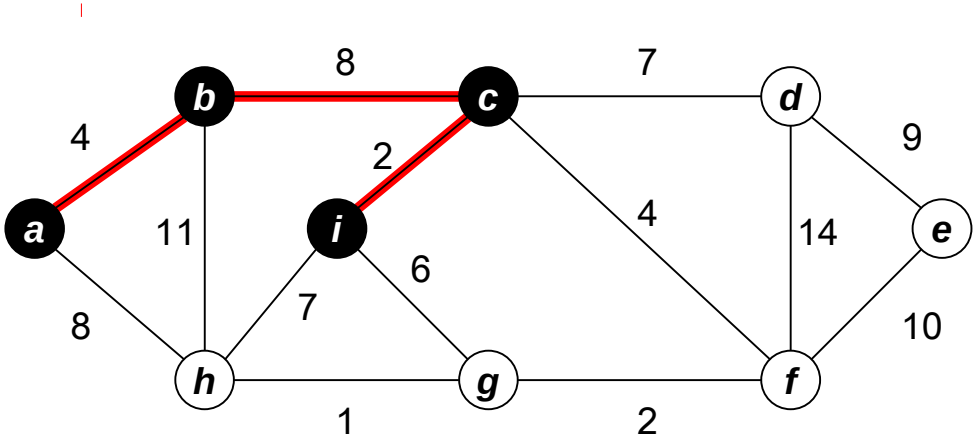
Algorithmus von Prim – Illustration (1)



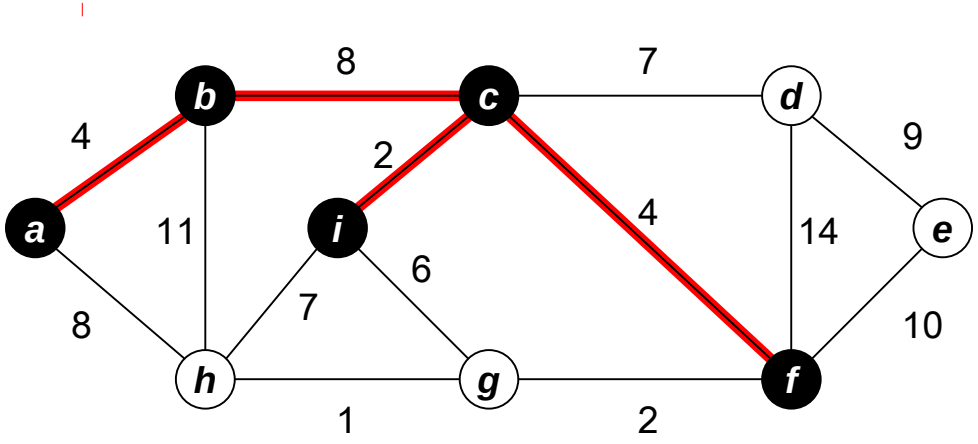


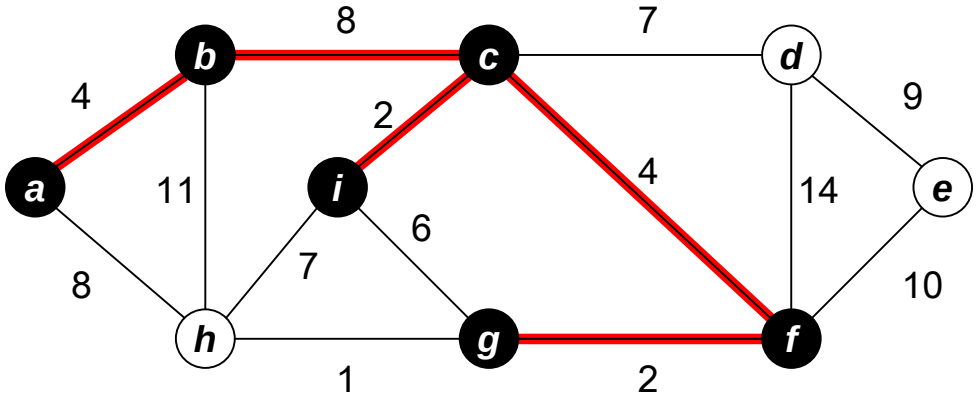
Algorithmus von Prim – Illustration (2)



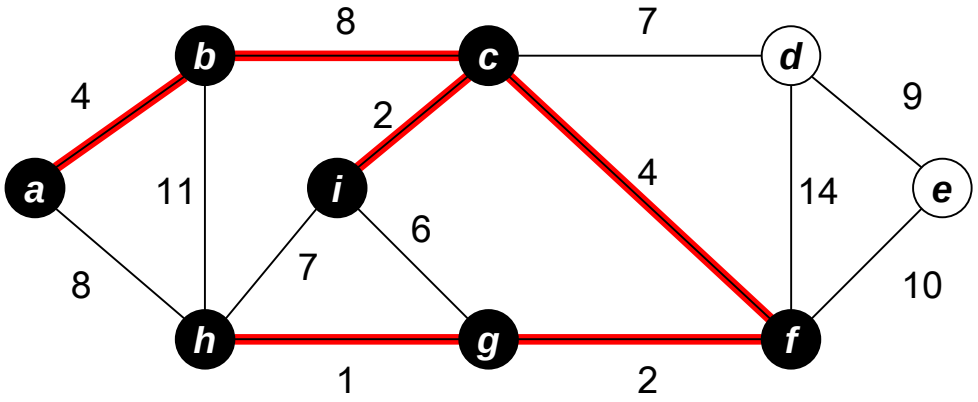


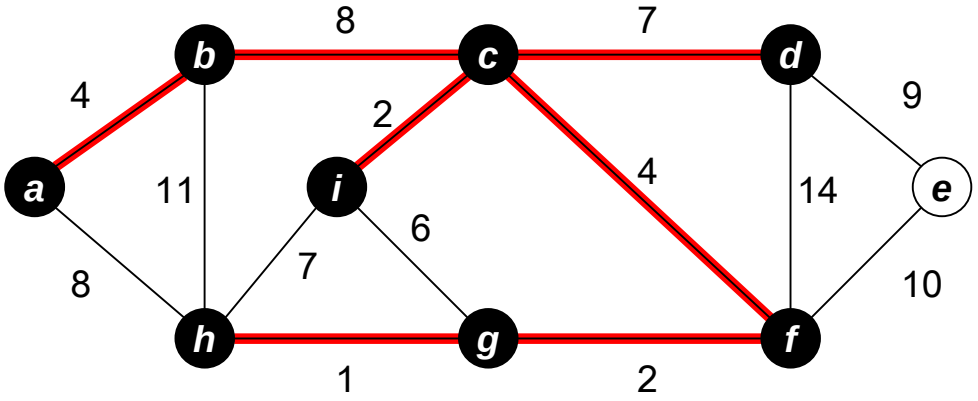
Algorithmus von Prim – Illustration (3)



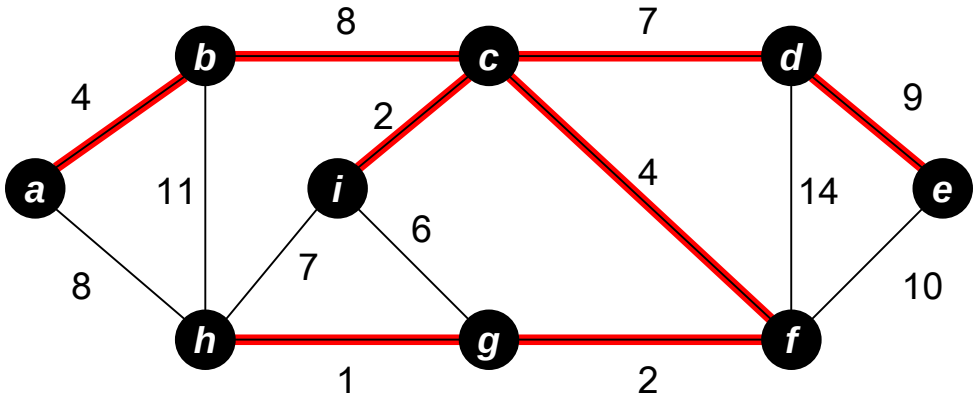


Algorithmus von Prim – Illustration (4)





Algorithmus von Prim – Illustration (5)



Korrektheit von Prims Algorithmus für allgemeine Instanzen (G,w): ergibt sich aus der Korrektheit von Generic-MST und Korollar 16.6.

Algorithmus von Prim – Laufzeit

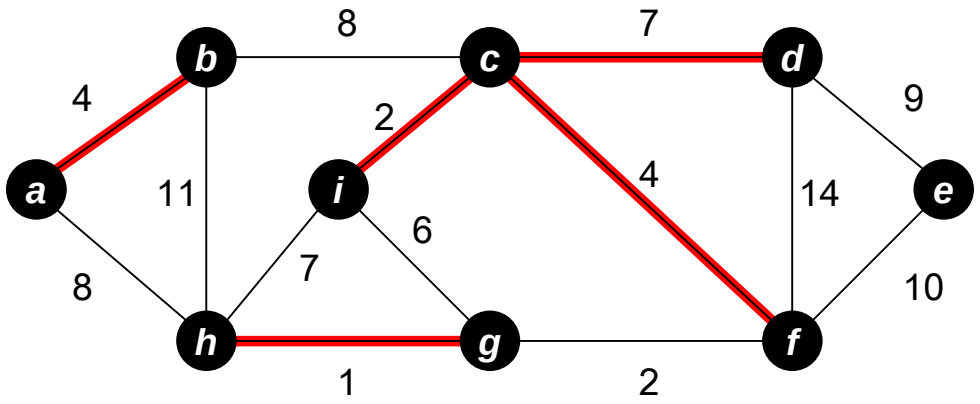
- Zeile 7 pro Durchlauf $O(\log(|V|))$, wird $|V|$-mal durchlaufen.
- Zeilen 8-12 pro Durchlauf $O(\log(|V|))$ (für Decrease-Key).
- Zeilen 9-12 $2|E|$-mal durchlaufen, da Größe aller Adjazenzlisten zusammen genau $2|E|$.

Satz 16.7: Der Algorithmus von Prim berechnet in Zeit $O(|V|\log(|V|)+|E|\log(|V|))$ einen minimalen Spannbaum eines gewichteten ungerichteten Graphen (G,w), G=(V,E).

Mit Fibonacci-Heaps Verbesserung auf $O(|V|\log(|V|)+|E|)$ möglich.

Algorithmus von Kruskal – Idee (1)

- Kruskals Algorithmus berechnet minimale Spannbäume mit anderer Strategie als Prims Algoirthmus.
- Kruskals Algorithmus erweitert auch sukzessive Kantenmenge A zu einem Spannbaum. Aber $G_A=(V,A)$ kann aus einer Menge von Bäumen bestehen.
- Strategie: Eine Kante minimalen Gewichts, die in G_A keinen Kreis erzeugt, wird zu A in jedem Schritt hinzu gefügt.



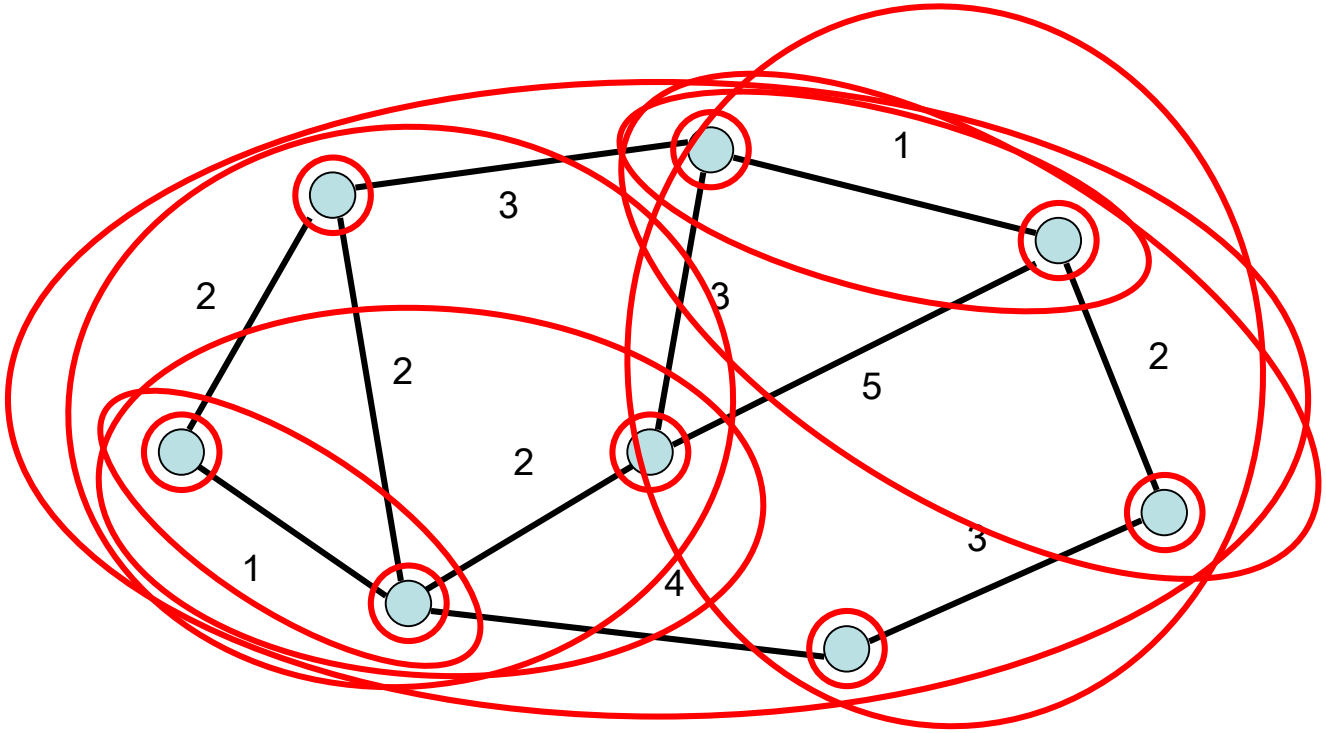
- Korrektheit folgt dann aus Korollar 16.6.

Zur Erinnerung

Satz 16.5: Sei (G,w) mit $G=(V,E)$ ein gewichteter ungerichteter Graph. Die Kantenmenge $A \subseteq E$ sei in einem minimalen Spannbaum von (G,w) enthalten. Weiter sei $(C,V\backslash C)$ ein mit A verträglicher Schnitt und $\{u,v\}$ sei eine leichte $(C,V\backslash C)$ kreuzende Kante. Dann ist $\{u,v\}$ eine A-sichere Kante.

Korollar 16.6: Sei (G,w) mit $G=(V,E)$ ein gewichteter ungerichteter Graph. Die Kantenmenge $A \subseteq E$ sei in einem minimalen Spannbaum von (G,w) enthalten. Ist $\{u,v\}$ ein Kante minimalen Gewichts, die eine Zusammenhangskomponente C von $G_A=(V,A)$ mit dem Rest des Graphen G_A verbindet, dann ist $\{u,v\}$ eine A-sichere Kante.

Algorithmus von Kruskal – Illustration



Algorithmus von Kruskal - Datenstruktur

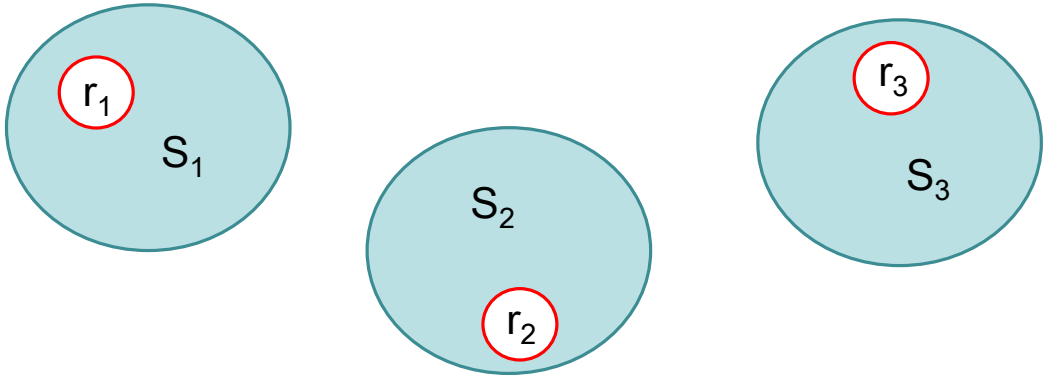
Kruskals Algorithmus benötigt eine Datenstruktur, mit deren Hilfe

1. für jede Kante $\{u,v\} \in E$ effizient entschieden werden kann, ob u und v in derselben Zusammenhangs-komponente von G_A liegen und
2. Zusammenhangskomponenten effizient verschmolzen werden können.

Solch eine Datenstruktur ist die Union-Find Datenstruktur für disjunkte dynamische Mengen.

Disjunkte dynamische Mengen (1)

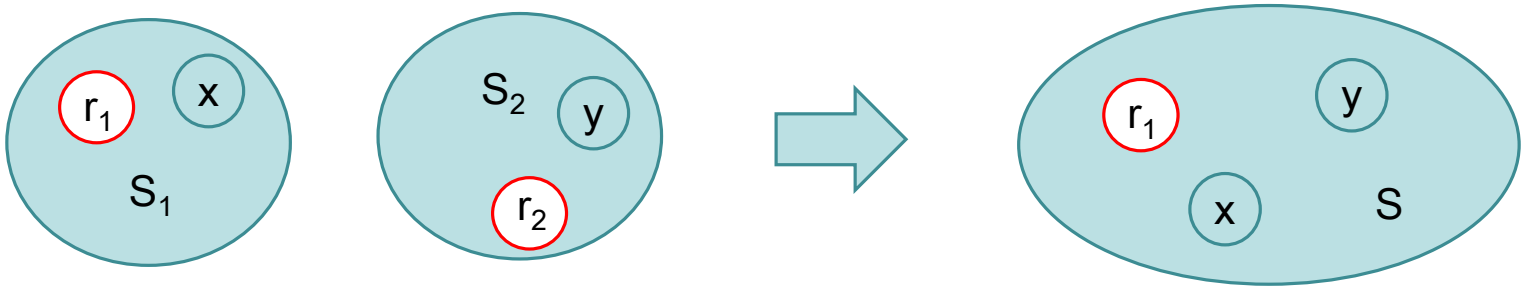
- Gegeben ist eine Menge U von Objekten.
- Verwaltet wird immer eine Familie $\{S_1, S_2, \ldots, S_k\}$ von disjunkten Teilmengen von U.
- Teilmengen S_i werden identifiziert mithilfe eines Elements aus S_i, einem sogenannten Repräsentanten.



Disjunkte dynamische Mengen (2)

Die folgenden Operationen sollen unterstützt werden:

1. Make-Set(x): erzeugt Teilmenge {x} mit Repräsentant x.
2. Union(x,y): vereinigt die beiden Teilmengen, die x bzw. y enthalten.

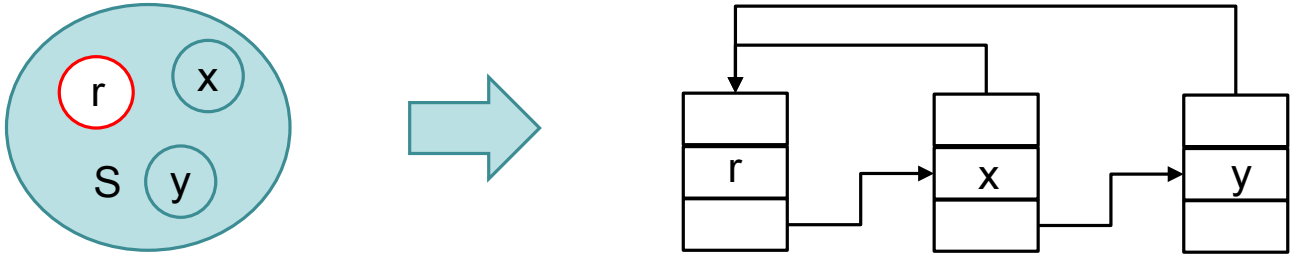


3. Find-Set(x): liefert den Repräsentanten derjenigen Menge, die x enthält.

Union-Find mit verketteten Listen

Realisierung einer Datenstruktur für disjunkte dynamische Mengen kann mit verketteten Listen erfolgen:

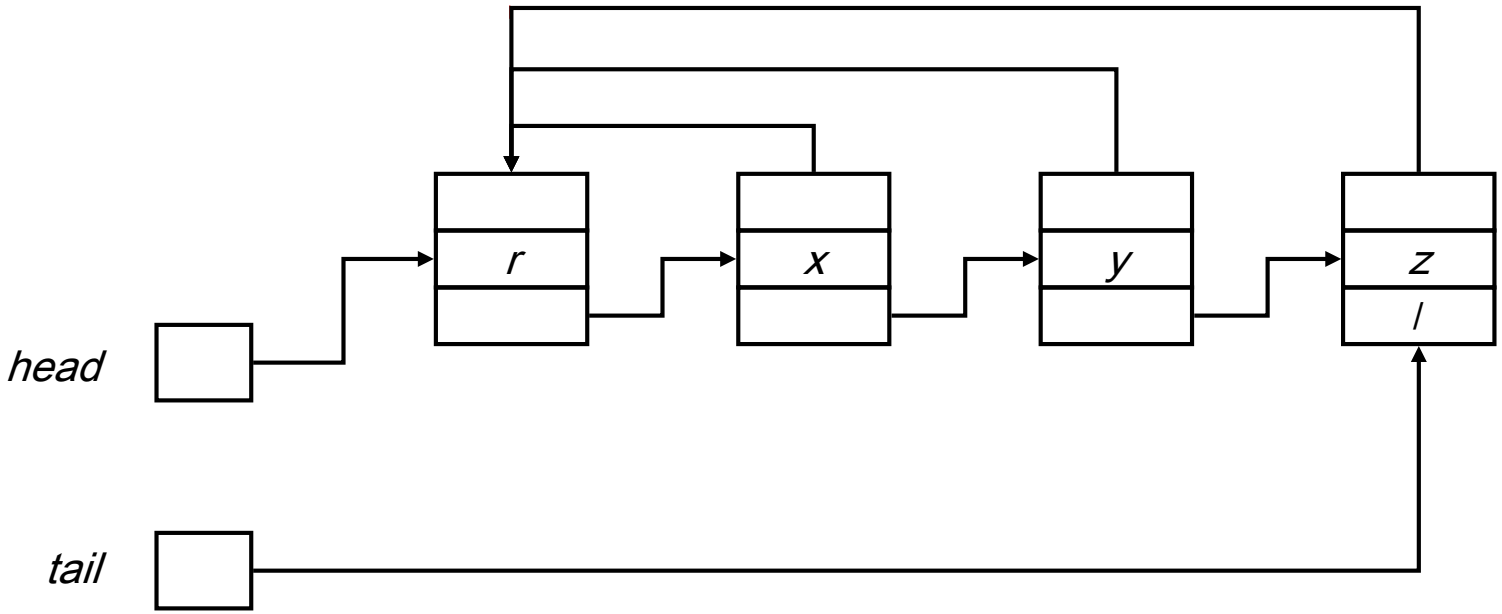
1. Für jede Teilmenge S gibt es eine verkettete Liste der Objekte in S.



2. Repräsentant ist dabei das erste Objekt in der Liste.
3. Für jedes Element x der Liste Verweis rep[x] auf Repräsentanten der Liste.

Union-Find mit verketteten Listen - Illustration

4. Zusätzlich enthalten head[S] und tail[S] Verweise auf das erste bzw. letzte Element der Liste zu S. Feld size[S] speichert die Größe von der Liste.



Realisierung der Operationen

- Make-Set(x): erzeuge 1-elementige Liste mit x.
- Find-Set(x): gib Repräsentant rep[x] in x aus.
- Union(x,y):
  S_x enthalte x und S_y enthalte y.

  1. Bestimme, welche der Listen kürzer ist.
  2. Hänge kürzere Liste an längere Liste.
  3. Setze Repräsentanten der Elemente in kürzerer Liste auf den der längeren Liste.

Lemma 16.8: Die Operation Union kann in Zeit proportional zur Länge der kürzeren Liste durchgeführt werden.

Analyse vieler Operationen

Satz 16.9: Werden verkettete Listen als Union-Find Datenstruktur benutzt und werden insgesamt n Make-Set und Union Operationen und m Find-Set Operationen ausgeführt, so können alle Operationen zusammen in Zeit $O(m+n \log(n))$ ausgeführt werden.

Beweis:

- Make-Set: Erzeugen einer 1-elementigen Liste möglich in Zeit $O(1)$.
- Find-Set(x): Ausgabe des Repräsentanten rep[x] möglich in Zeit $O(1)$.
- Union(x,y): Worst-case Laufzeit $\Theta(n)$. Aufaddierung über n Union Operationen würde dann aber Laufzeitschranke ergeben, die größer als die in Satz 16.9 ist!
- Strategie: betrachte alle Union Operationen auf einmal.

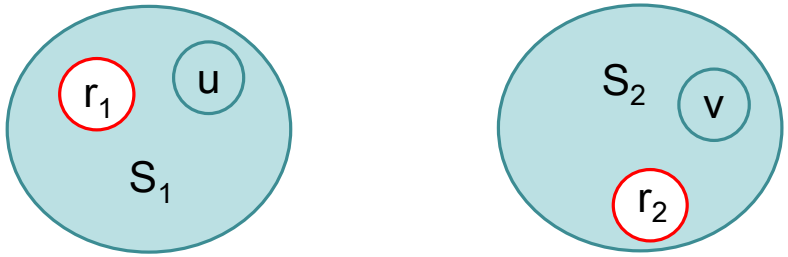
Analyse vieler Operationen

Beweis von Satz 16.9:

- Strategie: betrachte alle Union Operationen auf einmal.
- Frage: Wie oft muss Repräsentant eines Knotens geändert werden?
- Beim ersten Mal: Liste hat Länge mindestens 2.
- Beim zweiten Mal: Liste hat Länge mindestens 4.
- D.h. nach k Malen muss Liste mindestens Länge 2^k haben. Weiterhin ist $2^k \leq n$.
- D.h. für jeden Knoten wechselt der Repräsentant höchstens $\log n$ mal.
- Da die Gesamtkosten der Union-Operationen $= O($Anzahl der Male, die sich Repräsentant eines Knotens ändert$)$ ist, ist die Gesamtlaufzeit $O(n \log n)$.

Kruskals Algorithmus und Union-Find

- Wir benutzen Union-Find-Datenstruktur, um Zusammenhangskomponenten von $G_A=(V,A)$ zu verwalten.
- Test, ob {u,v} zwei verschiedene Zusammenhangskomponenten verbindet, durch Test, ob Find-Set(u) ≠ Find-Set(v).



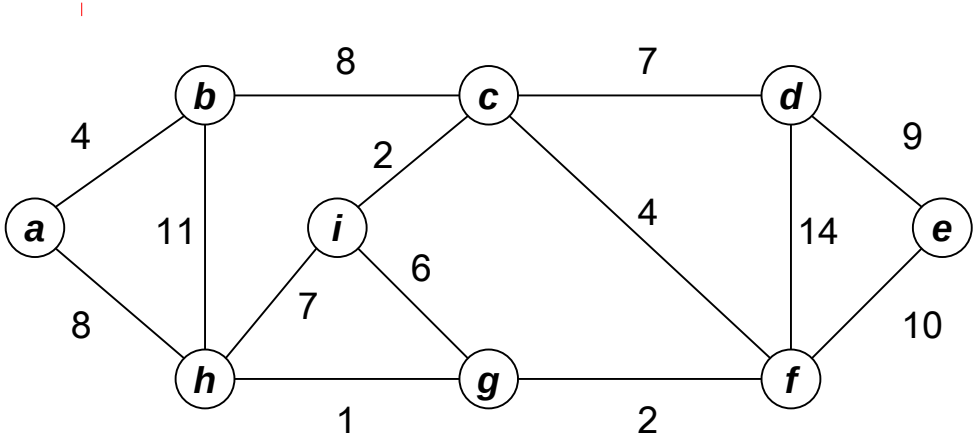
- Verschmelzen von Zusammenhangskomponenten durch Union

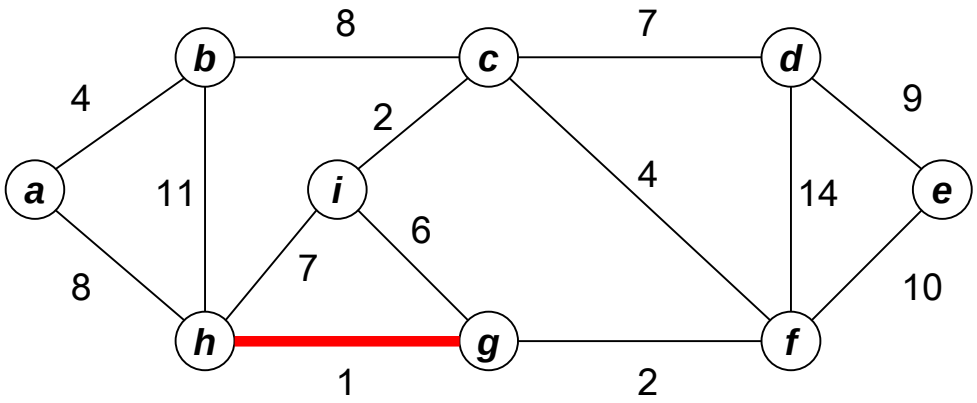
Algorithmus von Kruskal - Pseudocode

Kruskal-MST(G,w)

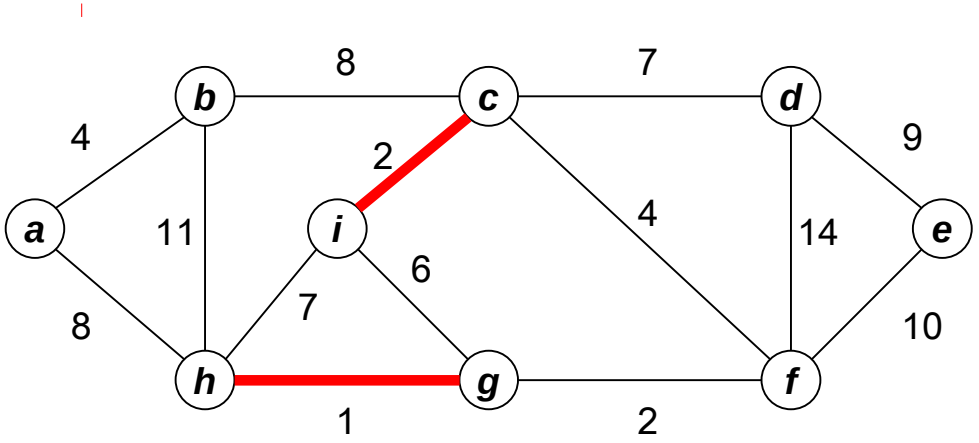
1. $A \leftarrow \{\ \}$
2. **for each** $v \in V$ **do** Make-Set(v)
3. Sortiere Kanten von G nach aufsteigendem Gewicht
4. **for each** $\{u,v\} \in E$ in aufsteigender Reihenfolge **do**
5. **if** Find-Set(u)$\neq$Find-Set(v) **then**
6. $A \leftarrow A \cup \{\ \{u,v\}\ \}$
7. Union(u,v)

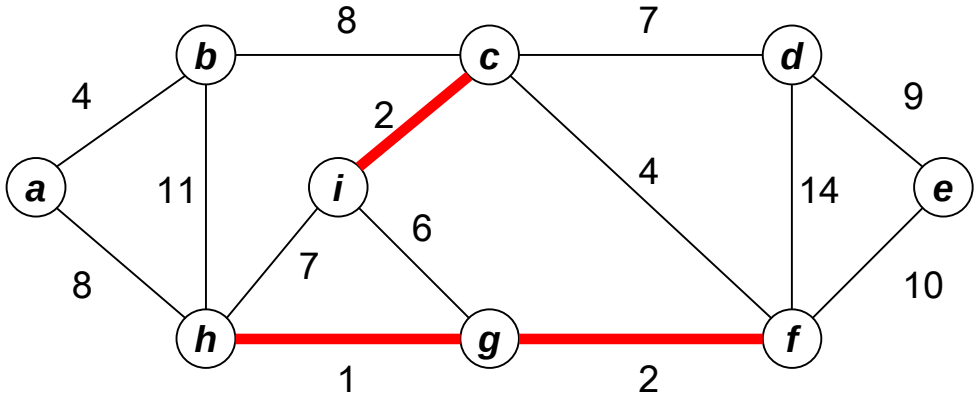
Algorithmus von Kruskal – Illustration (1)



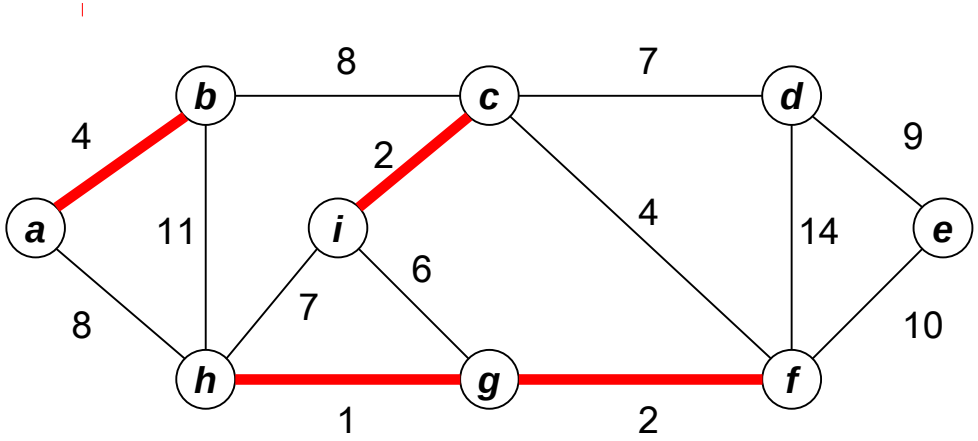


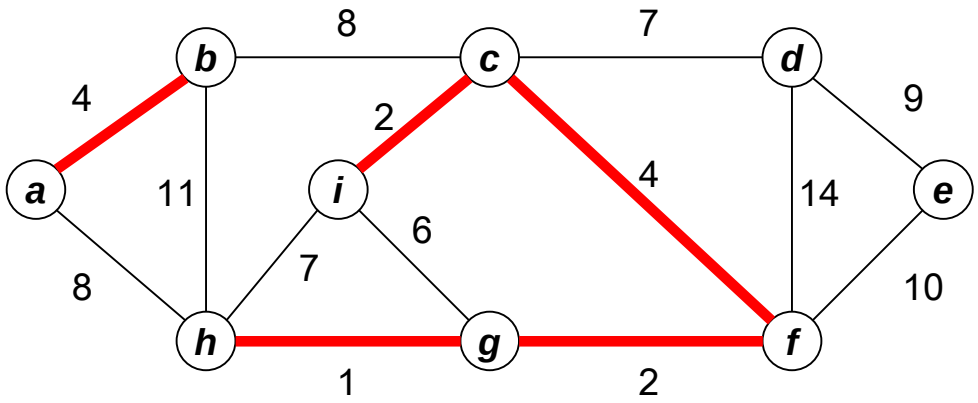
Algorithmus von Kruskal – Illustration (2)



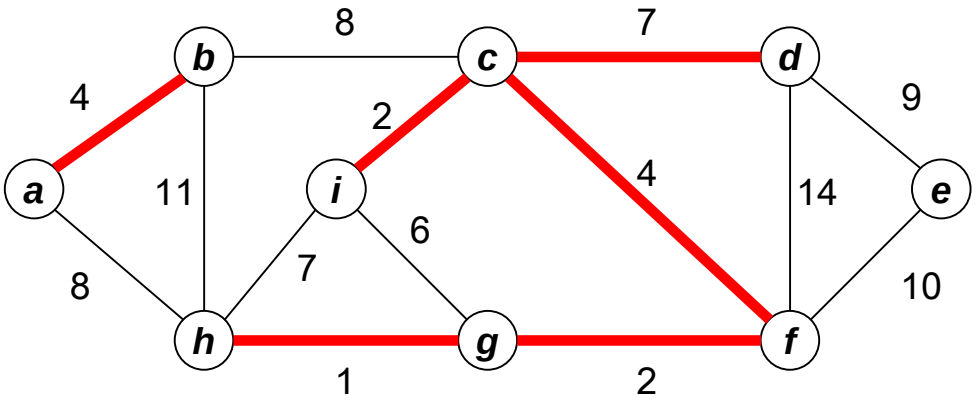


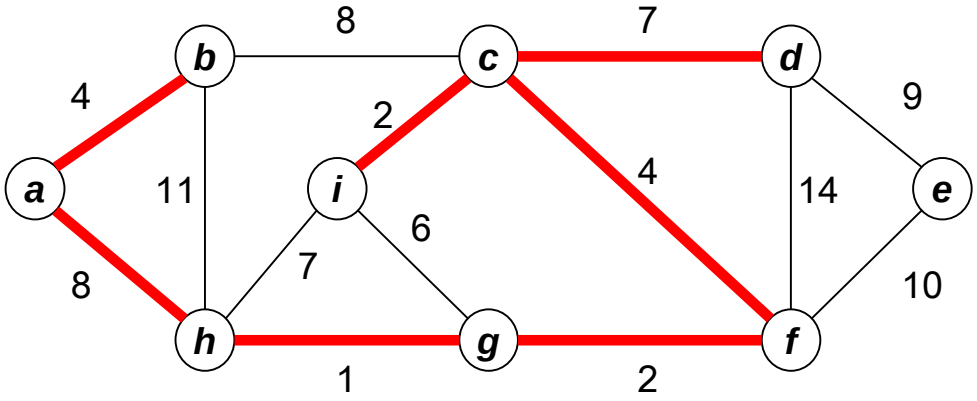
Algorithmus von Kruskal – Illustration (3)



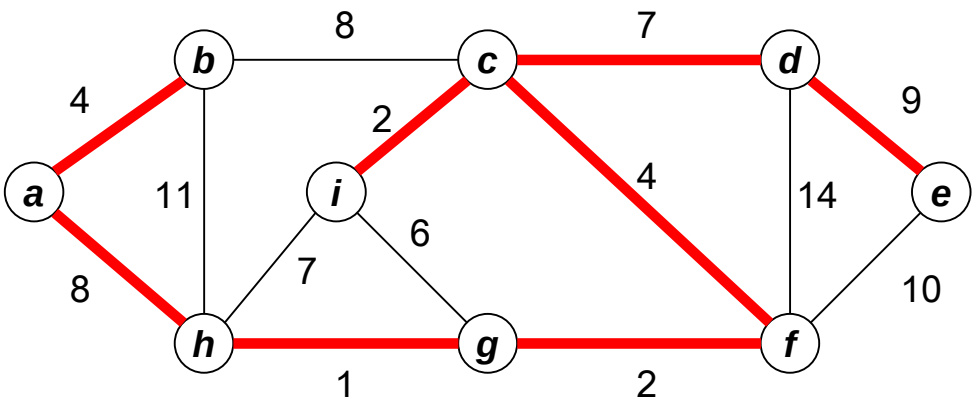


Algorithmus von Kruskal – Illustration (4)





Algorithmus von Kruskal – Illustration (5)



Laufzeitanalyse von Kruskals Algorithmus

Satz 16.10: Werden verkettete Listen als Union-Find-Datenstruktur benutzt, so ist die Laufzeit von Kruskals Algorithmus $O\left(|V|\log(|V|)+|E|\log(|E|)\right)$.