Algorithmen und Datenstrukturen
Kapitel 9
Minimale Spannbäume
und
Kürzeste Pfade

Frank Heitmann
heitmann@informatik.uni-hamburg.de

9. Dezember 2015

Problemstellung

Definition (Bestimmung eines minimalen Spannbaums)

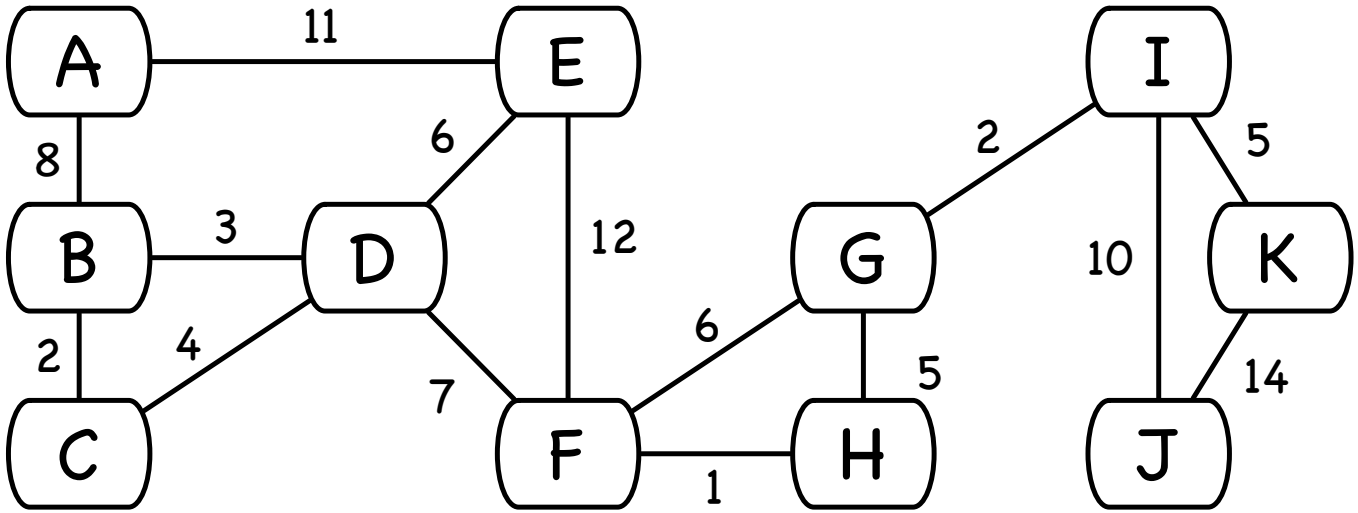
Eingabe: Gegeben ein **ungerichteter** Graph $G = (V, E)$ und eine Gewichtsfunktion $w : E \to \mathbb{R}^+$.
Gesucht: Ein Spannbaum T derart, dass

$$w(T) = \sum_{(u,v) \in E(T)} w(u, v)$$

minimal ist. Mit $E(T)$ werden dabei die Kanten von T bezeichnet. T wird minimaler Spannbaum oder MST genannt.

Ein Spannbaum von G ist ein zusammenhängender und azyklischer Teilgraph (also ein Baum) von G, der alle Knoten von G enthält.

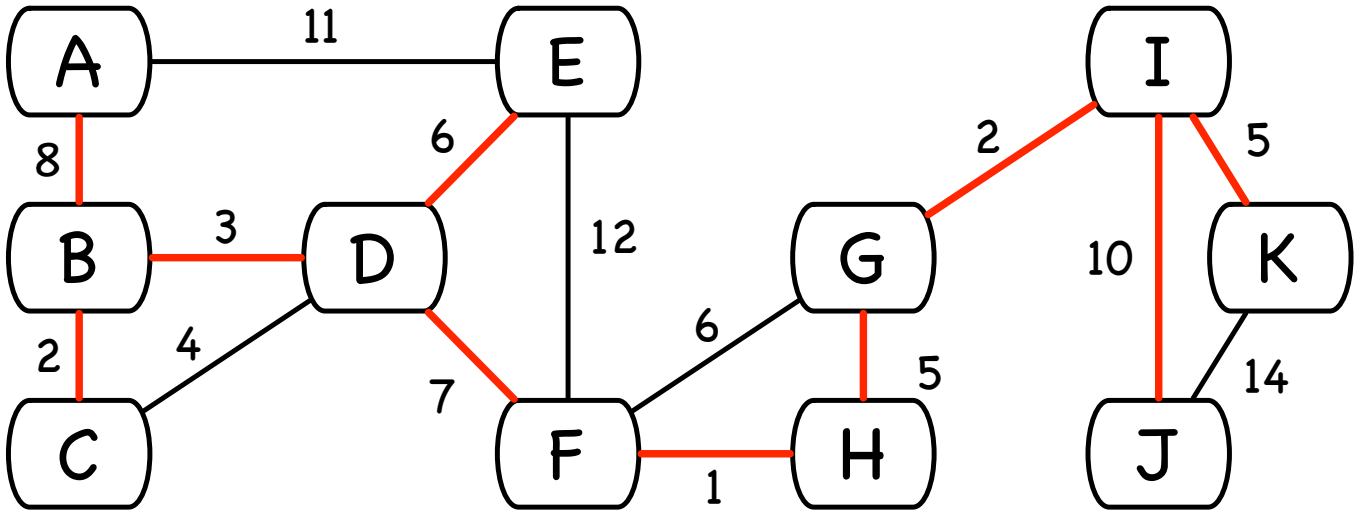
Ein Beispiel



Anmerkung

Alle Spannbäume haben $|V| - 1$ Kanten. Es ist oben also wichtig, dass das summierte Gewicht dieser Kanten minimiert werden soll. Bei ungewichteten (!) Graphen liefern Breiten- und Tiefensuche also bereits minimale Spannbäume.

Ein Beispiel



Anmerkung

Alle Spannbäume haben $|V| - 1$ Kanten. Es ist oben also wichtig, dass das summierte Gewicht dieser Kanten minimiert werden soll. Bei ungewichteten (!) Graphen liefern Breiten- und Tiefensuche also bereits minimale Spannbäume.

Die generelle Idee

Die Idee ist, dass der Algorithmus sukzessive eine Kantenmenge A aufbauen soll, wobei folgende Schleifeninvariante erhalten bleiben soll:

> *Unmittelbar vor jeder Iteration ist A eine Teilmenge eines minimalen Spannbaums.*

In jedem Schritt wird eine Kante bestimmt, die hinzugefügt werden kann, ohne dass die Invariante verletzt wird. Solche Kanten werden als **sicher** bezeichnet. Eine Kante e ist also *sicher* für eine Teilmenge A eines minimalen Spannbaums, wenn $A \cup \{e\}$ weiterhin Teilmenge eines (möglicherweise anderen) minimalen Spannbaums ist.

Die generelle Idee

Algorithmus 1 GenericMST(G, w)

1: $A = \emptyset$
2: **while** A bildet keinen Spannbaum **do**
3: bestimme eine für A sichere Kante (u, v)
4: $A = A \cup \{(u, v)\}$
5: **end while**
6: **return** A

Anmerkung

A ist die bisher ermittelte Kantenmenge des Spannbaums. In jedem Schritt wird zu A eine *sichere* Kante hinzugefügt. **Der schwierige Teil ist die Bestimmung einer sicheren Kante.**

Die Schleifeninvariante

Algorithmus 2 GenericMST(G, w)

1: $A = \emptyset$
2: **while** A bildet keinen Spannbaum **do**
3: bestimme eine für A sichere Kante (u, v)
4: $A = A \cup \{(u, v)\}$
5: **end while**
6: **return** A

Pause to Ponder

Unmittelbar vor jeder Iteration ist A eine Teilmenge eines minimalen Spannbaums.

Initialisierung? Fortsetzung? Terminierung? Und gibt es immer eine sichere Kante?

Einige Definitionen

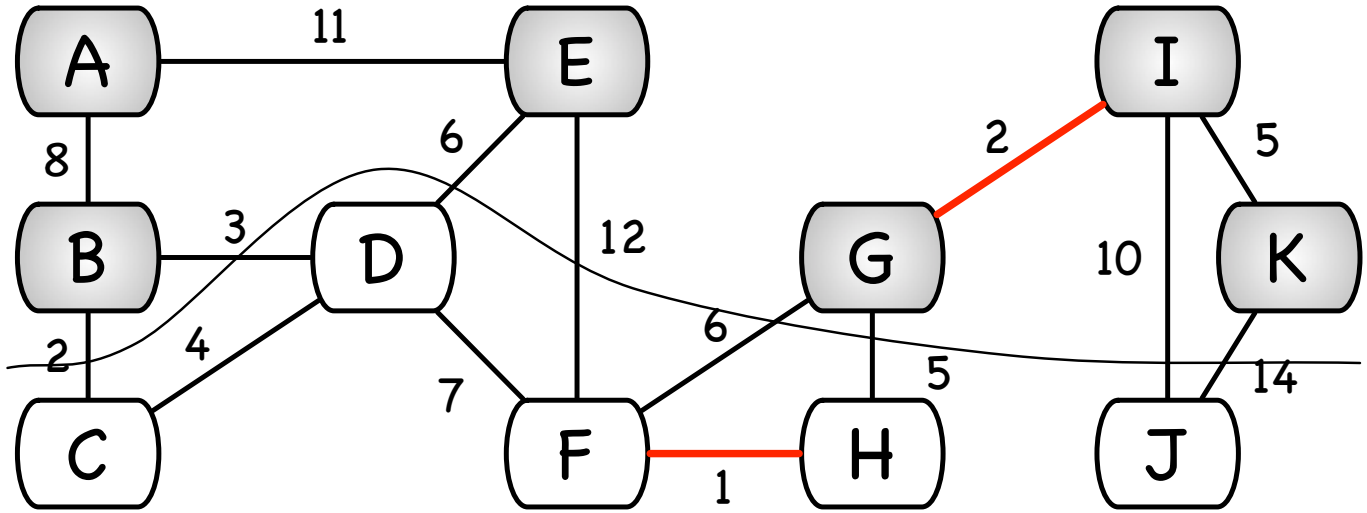
Wir ermitteln gleich eine Regel, mit der sichere Kanten erkannt werden können. Diese wird dann in zwei Algorithmen genutzt, um effizient sichere Kanten zu bestimmen. Vorher einige Definitionen...

Definition

Sei $G = (V, E)$ ein ungerichteter Graph.

- Ein *Schnitt* $(S, V - S)$ ist eine Partitionierung von V.
- Eine Kante $(u, v) \in E$ *kreuzt* den Schnitt, wenn ihre Endpunkte in verschiedenen Partitionen liegen.
- Eine Kante ist eine *leichte, den Schnitt kreuzende Kante*, falls sie das kleinste Gewicht aller Kanten hat, die den Schnitt kreuzen.
- Ein Schnitt *respektiert* eine Kantenmenge A, falls keine Kante von A den Schnitt kreuzt.

Zur Illustration...



Eine wichtige Regel

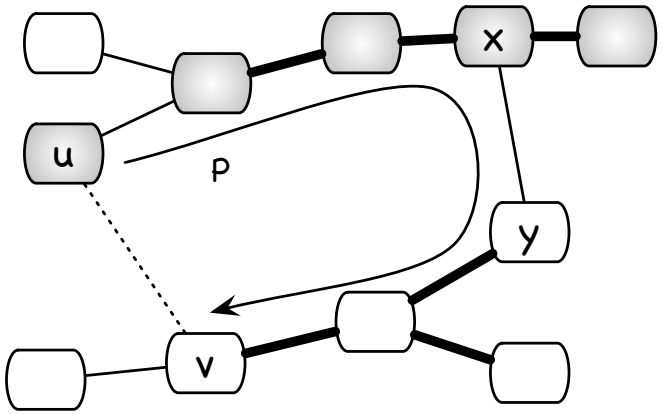
Satz

Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph und $w : E \to \mathbb{R}^+$ eine Gewichtsfunktion. Sei $A \subseteq E$ Teilmenge eines minimalen Spannbaums von G, $(S, V - S)$ ein Schnitt von G, der A respektiert und (u, v) eine leichte Kante, die $(S, V - S)$ kreuzt. Dann ist die Kante (u, v) für A sicher.

Beweis

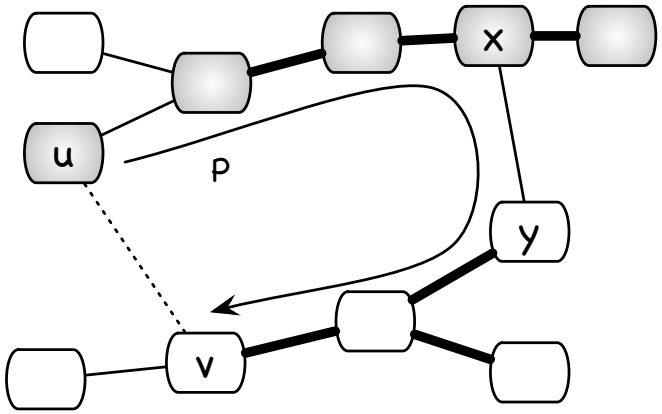
Sei T ein minimaler Spannbaum, der A enthält und gelte $(u, v) \notin E(T)$, sonst wären wir sofort fertig. Wir konstruieren nun durch *'Ausschneiden und Einfügen'* einen anderen minimalen Spannbaum T' mit $A \cup \{(u, v)\} \subseteq E(T')$, woraus dann folgt, dass (u, v) sicher für A ist. ...

Zur Illustration des Beweises



Die Knoten von S sind grau, die von $S - V$ weiß. Die Kanten von T sind eingezeichnet, nicht die von G. Kanten von A sind dick. Die Kante (u, v) gehört zu G, nicht zu T. Sie ist eine leichte, den Schnitt $(S, V - S)$ kreuzende Kante. Die Kante (x, y) liegt auf dem Pfad P in T von u nach v. Man erhält T', indem man (x, y) aus T entfernt und die Kante (u, v) hinzufügt.

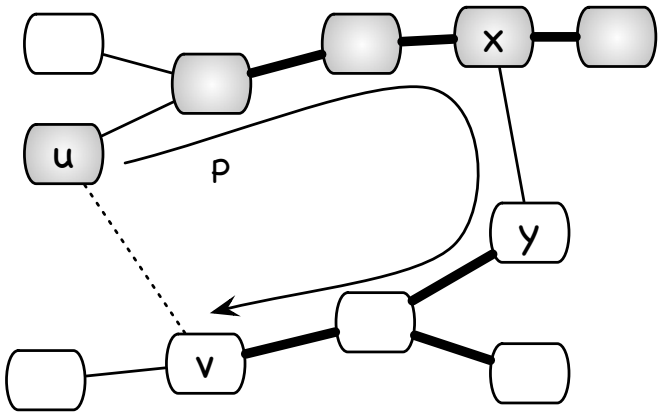
Eine wichtige Regel - Der Beweis



Beweis

- Ziel: $A \cup \{(u, v)\} \subseteq E(T')$ und T' ist MST.
- (u, v) bildet mit dem Pfad P, der in T von u nach v führt einen Zyklus.
- (u, v) kreuzt den Schnitt $(S, V - S)$, daher gibt es einen Kante (x, y) auf P, die den Schnitt (ebenfalls) kreuzt.

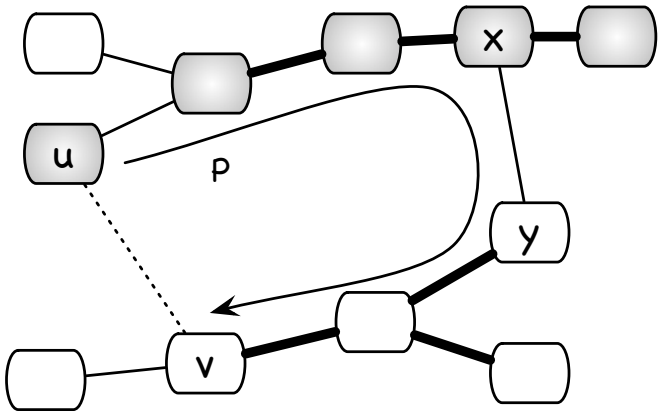
Eine wichtige Regel - Der Beweis



Beweis

- $(x, y) \notin A$, da der Schnitt A respektiert.
- Da T ein Baum ist, zerfällt er durch entfernen von (x, y) in zwei Komponenten und wird durch hinzufügen von (u, v) wieder zu einem Spannbaum: $T' = (T - \{(x, y)\}) \cup \{(u, v)\}$

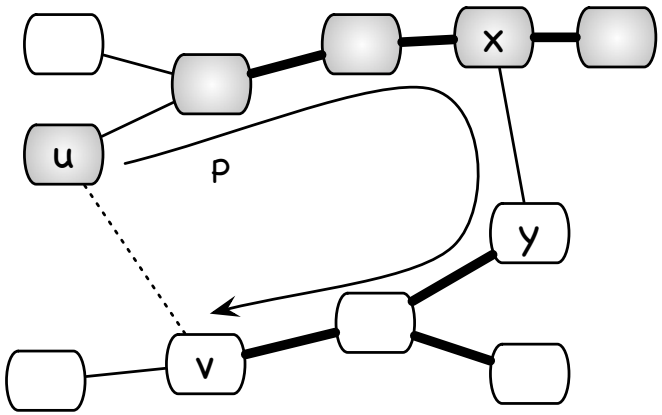
Eine wichtige Regel - Der Beweis



Beweis

- Ziel: $A \cup \{(u, v)\} \subseteq E(T')$ und T' ist MST.
- Z.Z.: T' ist (auch) ein MST. Da (x, y) und (u, v) beide den Schnitt kreuzen, (u, v) aber eine leichte Kante ist, gilt zunächst $w(u, v) \leq w(x, y)$. Damit folgt $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T) (\leq w(T'))$, also $w(T') = w(T)$ und beides sind MSTs.

Eine wichtige Regel - Der Beweis



Beweis.

- Ziel: $A \cup \{(u,v)\} \subseteq E(T')$ und T' ist MST.
- Z.Z.: $A \cup \{(u,v)\} \subseteq E(T')$. Da $A \subseteq E(T)$ gilt und $(x,y) \notin A$ folgt $A \subseteq E(T')$ und somit auch $A \cup \{(u,v)\} \subseteq E(T')$. Damit ist alles gezeigt.

□

Eine wichtige Regel

Satz

Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph und $w : E \to \mathbb{R}^+$ eine Gewichtsfunktion. Sei $A \subseteq E$ Teilmenge eines minimalen Spannbaums von G, $(S, V - S)$ ein Schnitt von G, der A respektiert und (u, v) eine leichte Kante, die $(S, V - S)$ kreuzt. Dann ist die Kante (u, v) für A sicher.

Folgerungen für den Algorithmus

Zu jedem Zeitpunkt im Algorithmus `GenericMST` gilt

- A ist azyklisch,
- der Graph $G_A = (V, A)$ ist ein Wald, dessen Zusammenhangskomponenten Bäume sind (evtl. bestehend aus nur einem Knoten),
- jede für A sichere Kante verbindet unterschiedliche Komponenten von G_A.

Die Schleife wird nun $|V| - 1$ mal ausgeführt. Anfangs, wenn A leer ist, gibt es $|V|$ Bäume in G_A. Ihre Anzahl wird bei jeder Iteration um eins vermindert. Wenn G_A nur ein einzelner Baum ist, terminiert der Algorithmus.

Eine wichtige Folgerung

Satz

Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph und $w : E \to \mathbb{R}^+$ eine Gewichtsfunktion. Sei $A \subseteq E$ Teilmenge eines minimalen Spannbaums von G und $C = (V_C, E_C)$ eine Zusammenhangskomponente aus $G_A = (V, A)$. Falls (u, v) unter allen Kanten, die C mit einer anderen Komponente von G_A verbinden, das kleinste Gewicht hat, dann ist (u, v) für A sicher.

Beweis.

Der Schnitt $(V_C, V - V_C)$ respektiert A (sonst wäre V_C keine Zusammenhangskomponente) und (u, v) ist eine leichte Kante für diesen Schnitt. Nach dem eben bewiesenen Satz ist (u, v) dann sicher für A. □

Zur Nachbereitung

Anmerkung (zur Nachbereitung)

Zusammenhangskomponenten eines ungerichteten Graphen (wie eben die Komponente G_A) sind hier immer maximal zu wählen. D.h. möglichst viele Knoten sind in die Knotenmenge aufzunehmen (gerade all jene, die über Pfade erreicht werden können).

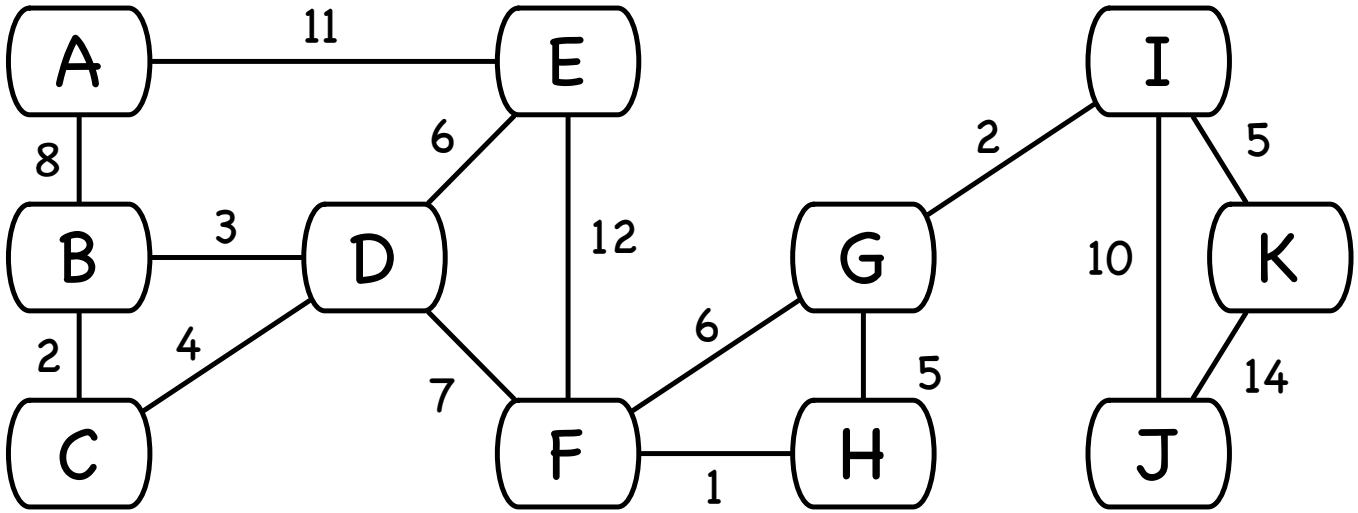
Kruskal - Die Idee

Der Algorithmus von Kruskal basiert auf eben bewiesenen Satz und erweitert den generischen Algorithmus um eine Vorschrift, wie die sichere Kante bestimmt werden soll. Die Idee ist sehr einfach:

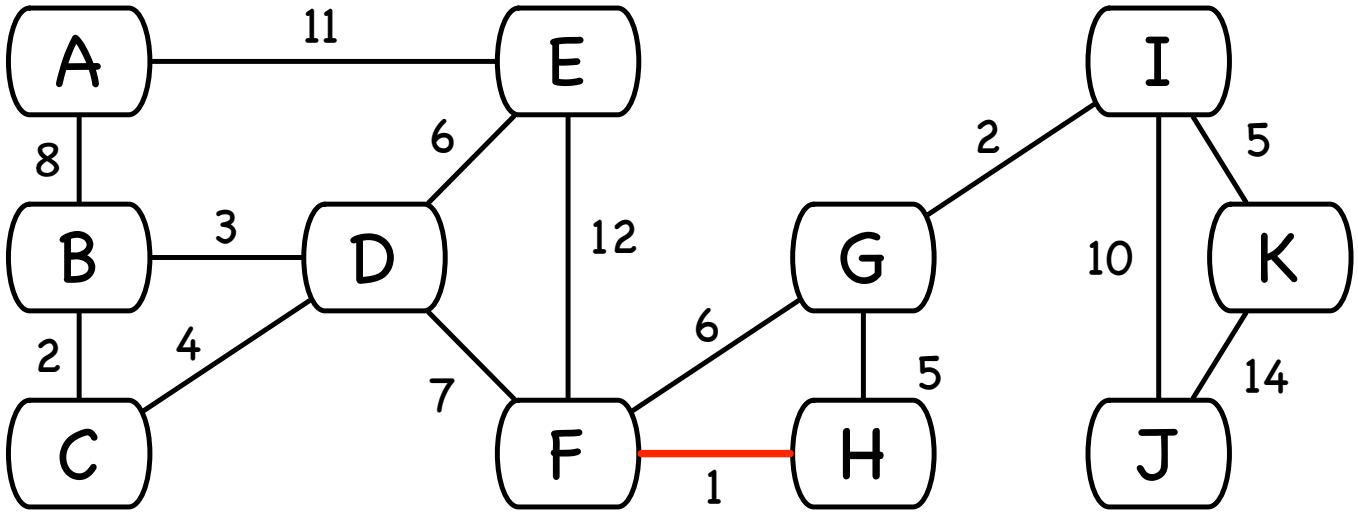
- Nimm unter allen Kanten jene mit dem kleinsten Gewicht, deren Hinzufügen nicht zu einem Kreis führt.

Die Menge A ist also ein Wald. Die hinzugefügte, sichere Kante ist immer eine Kante mit minimalen Gewicht, die zwei verschiedene Komponenten (von G_A) verbindet. Die Korrektheit dieses Verfahrens folgt sofort aus dem eben vorgestellten Satz.

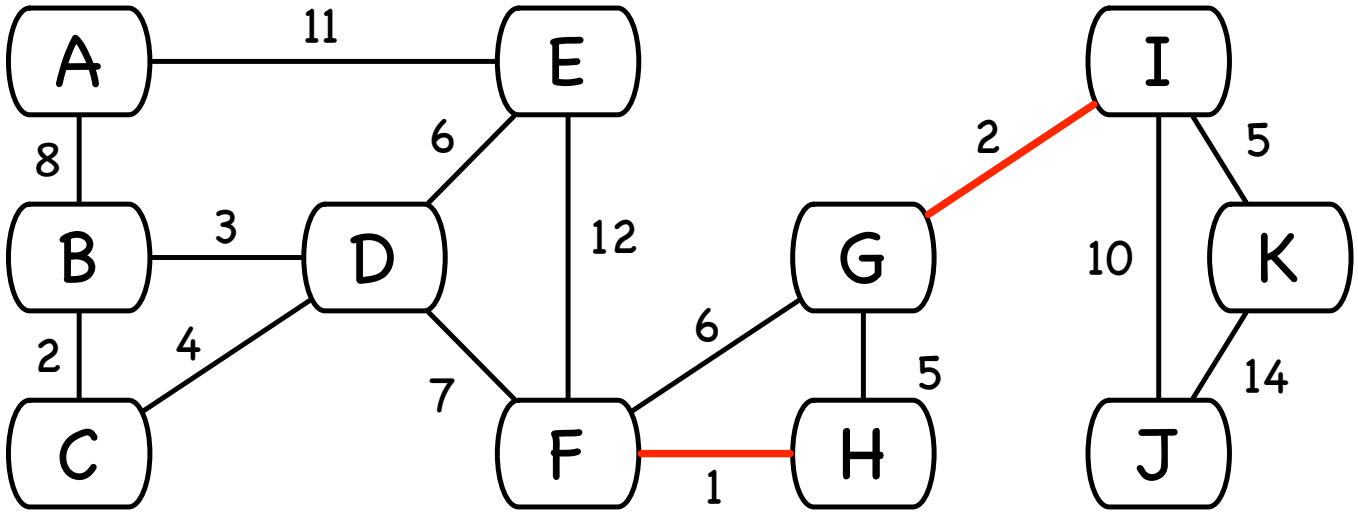
Kruskal - Beispiel



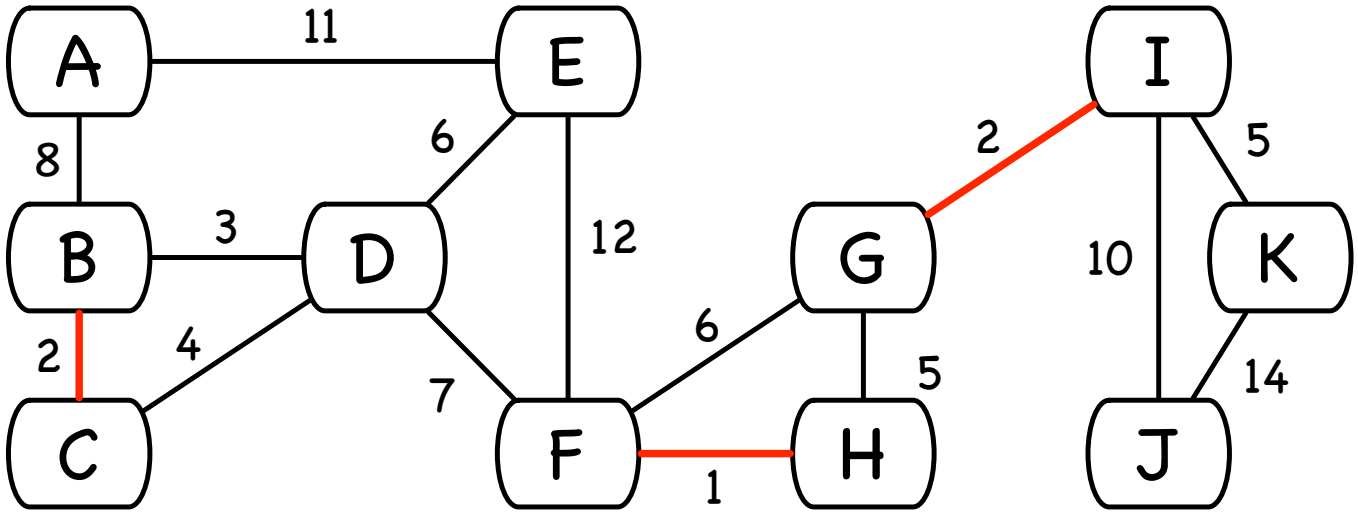
Kruskal - Beispiel



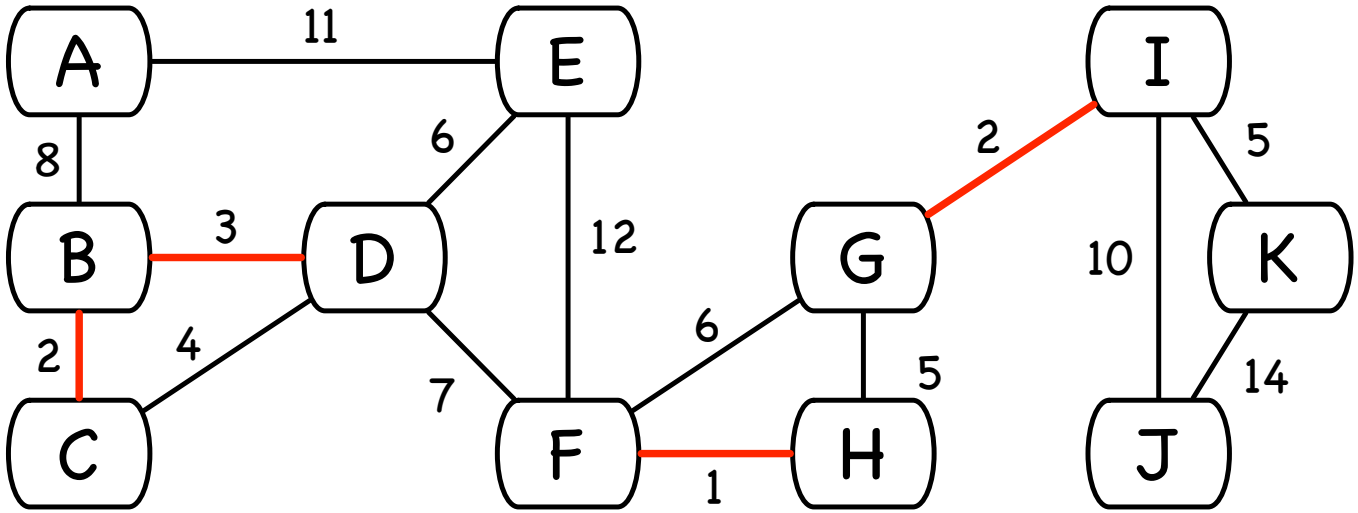
Kruskal - Beispiel



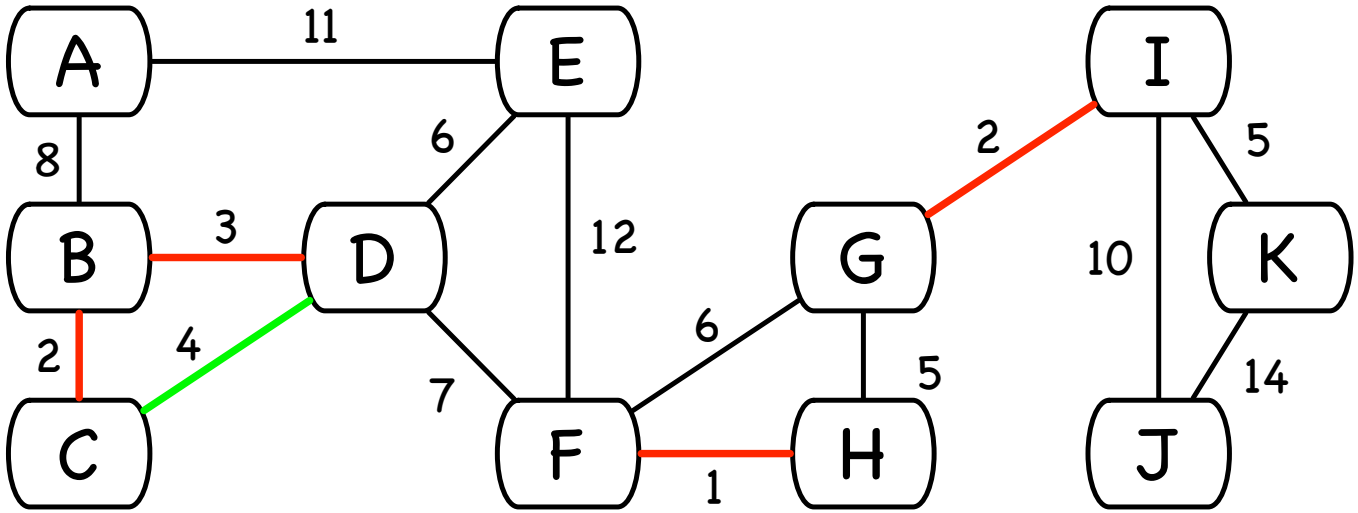
Kruskal - Beispiel



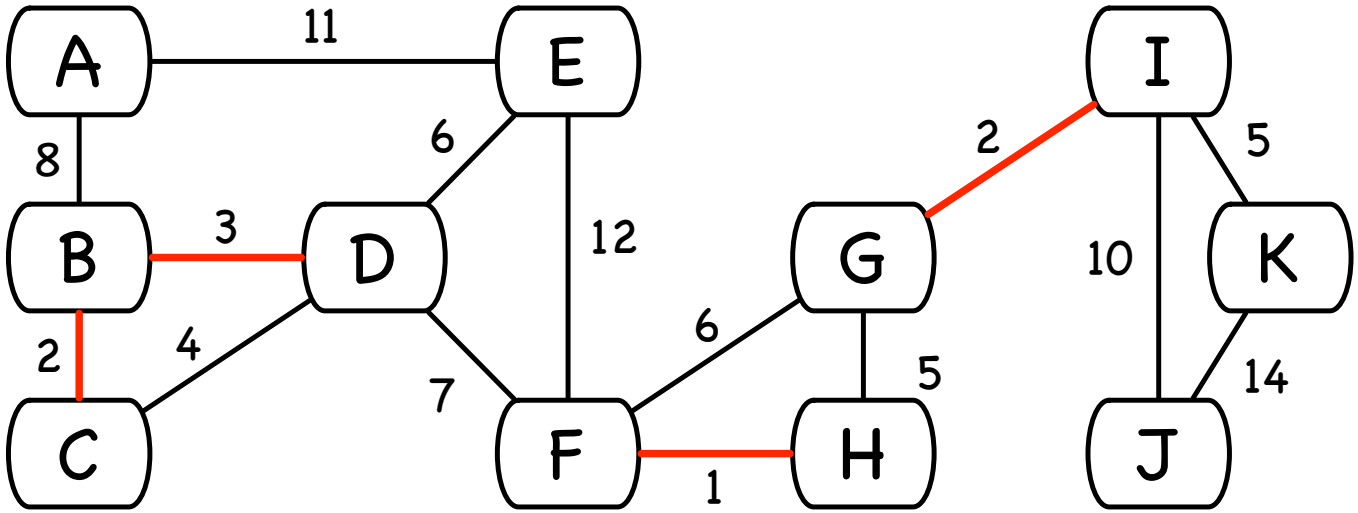
Kruskal - Beispiel



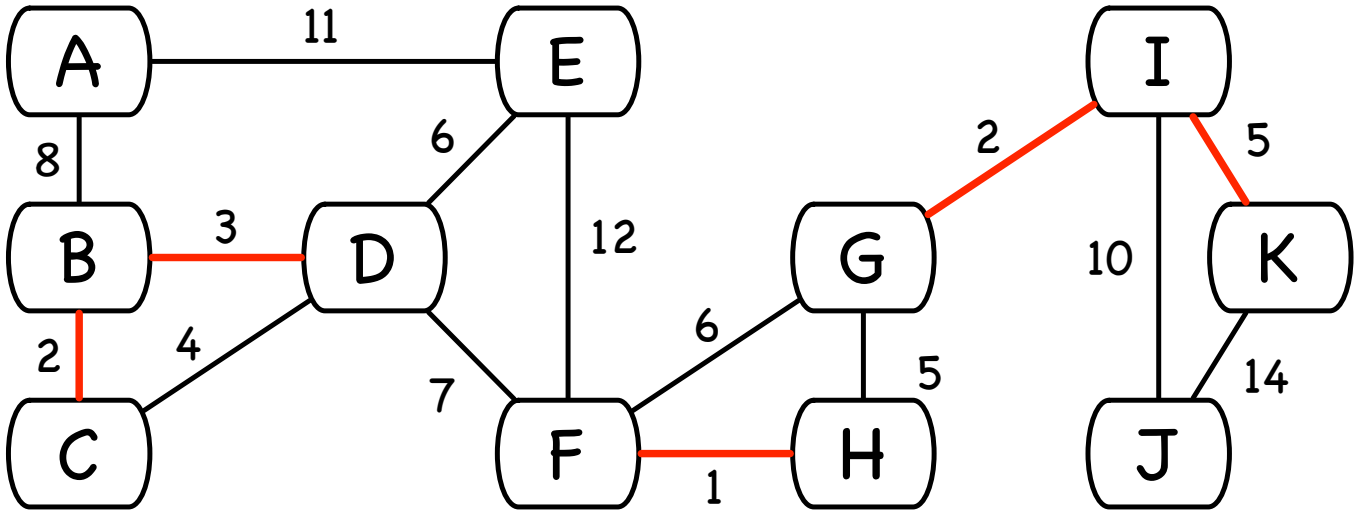
Kruskal - Beispiel



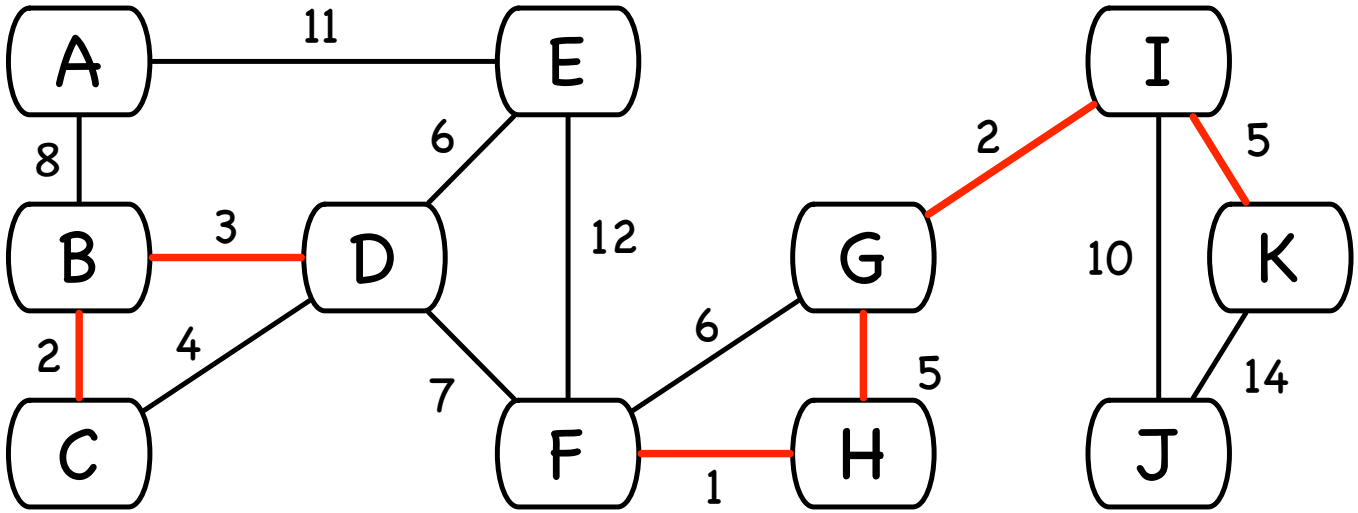
Kruskal - Beispiel



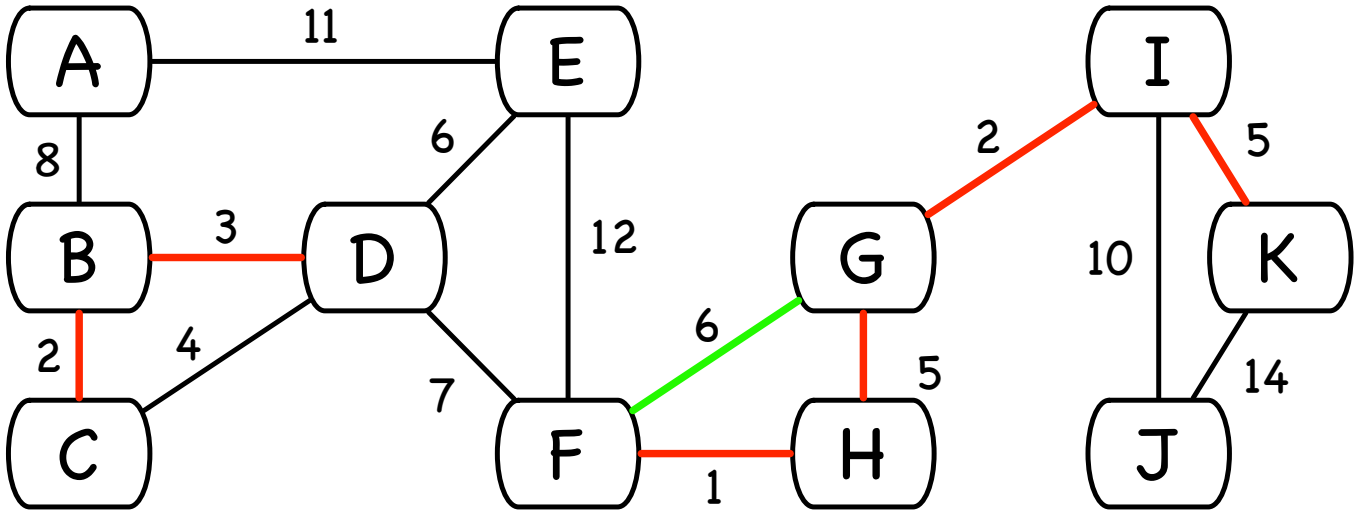
Kruskal - Beispiel



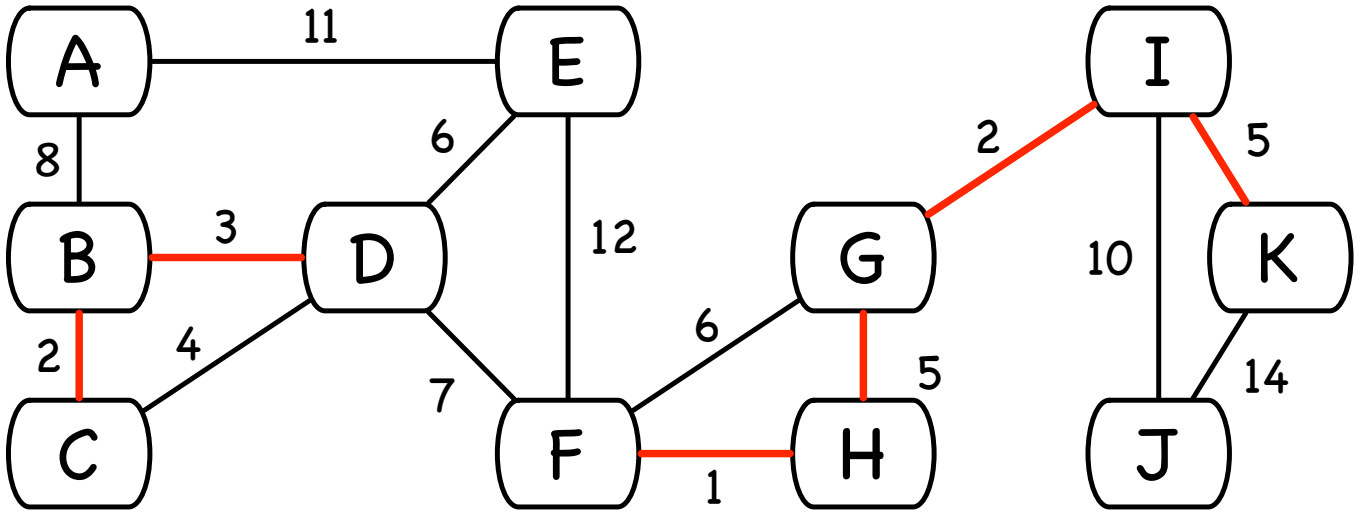
Kruskal - Beispiel



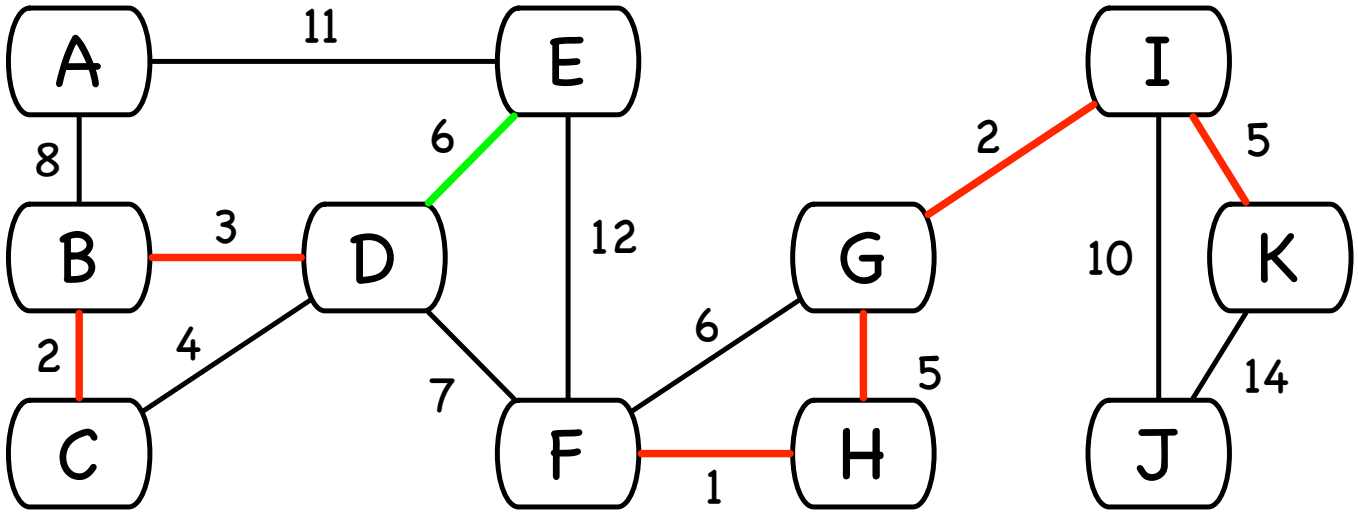
Kruskal - Beispiel



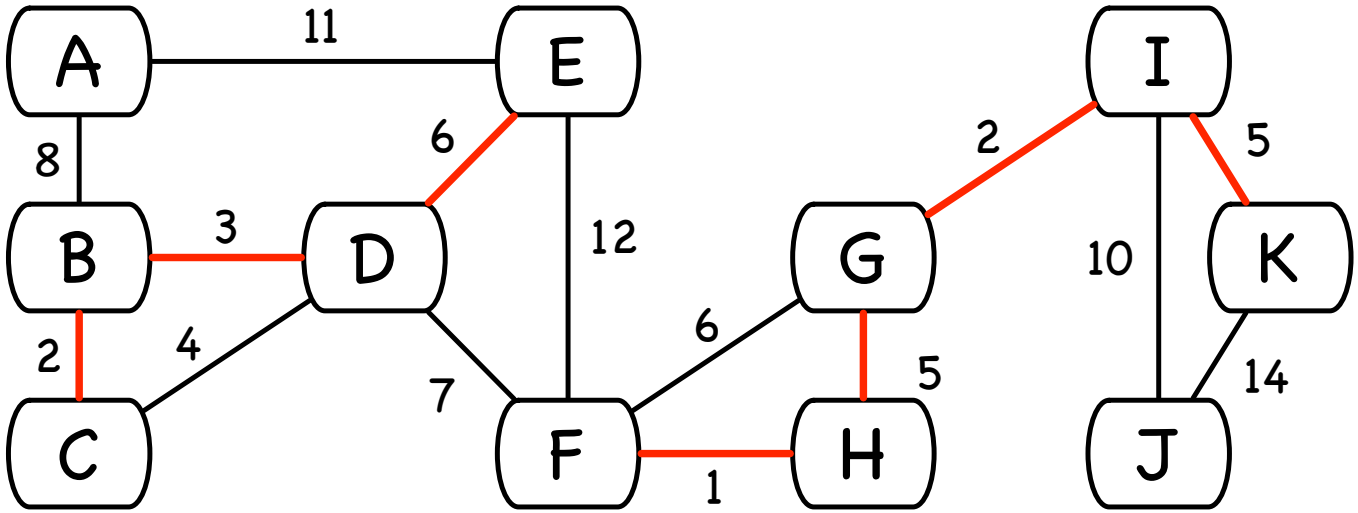
Kruskal - Beispiel



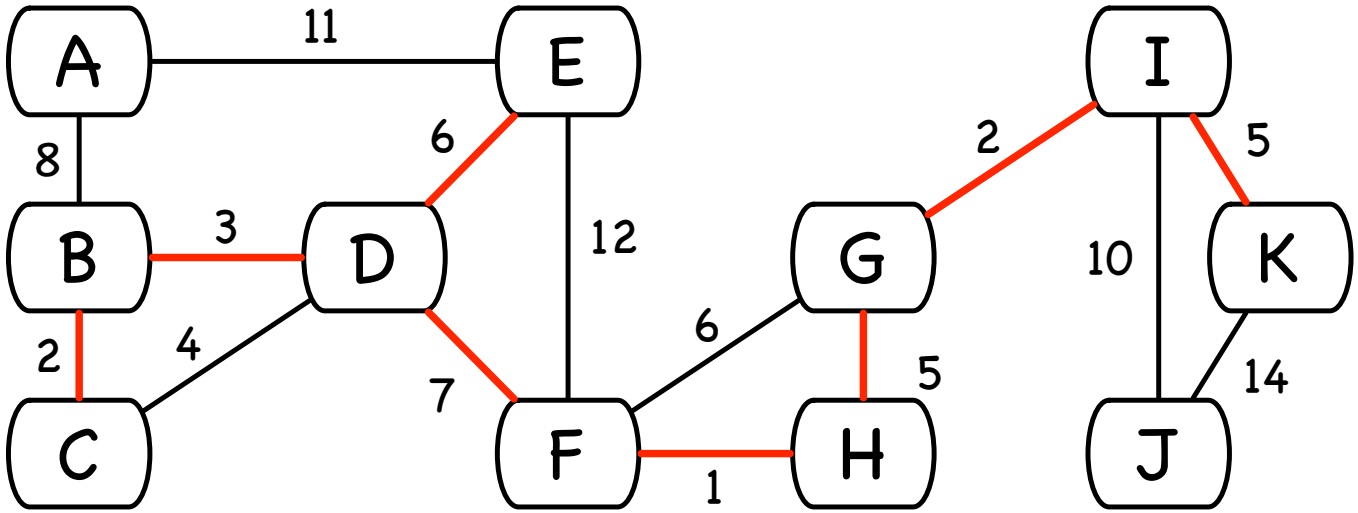
Kruskal - Beispiel



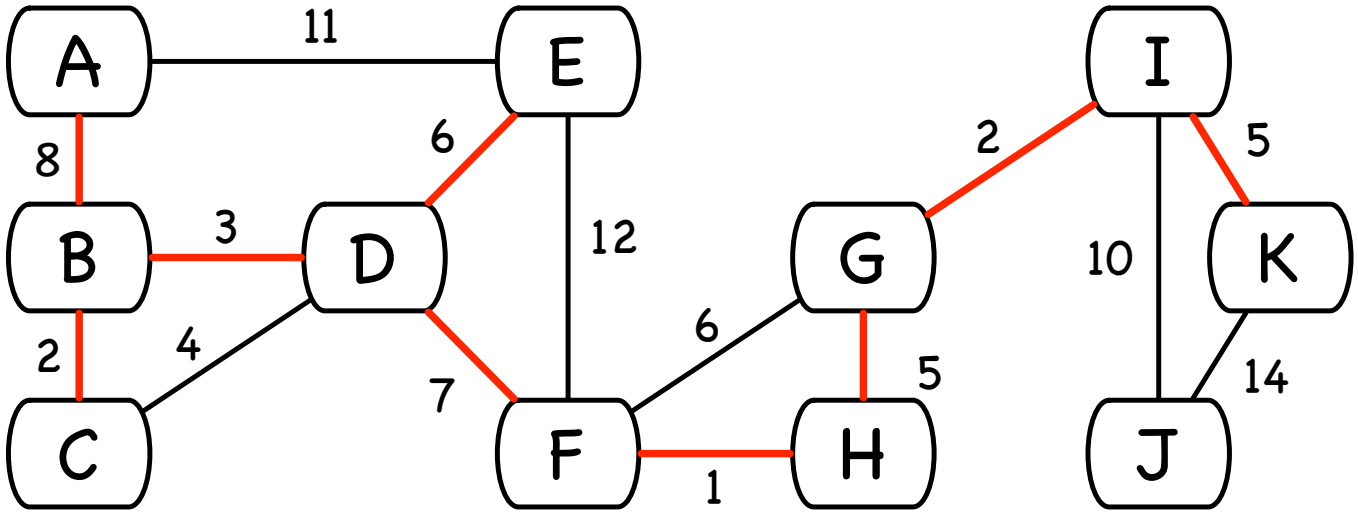
Kruskal - Beispiel



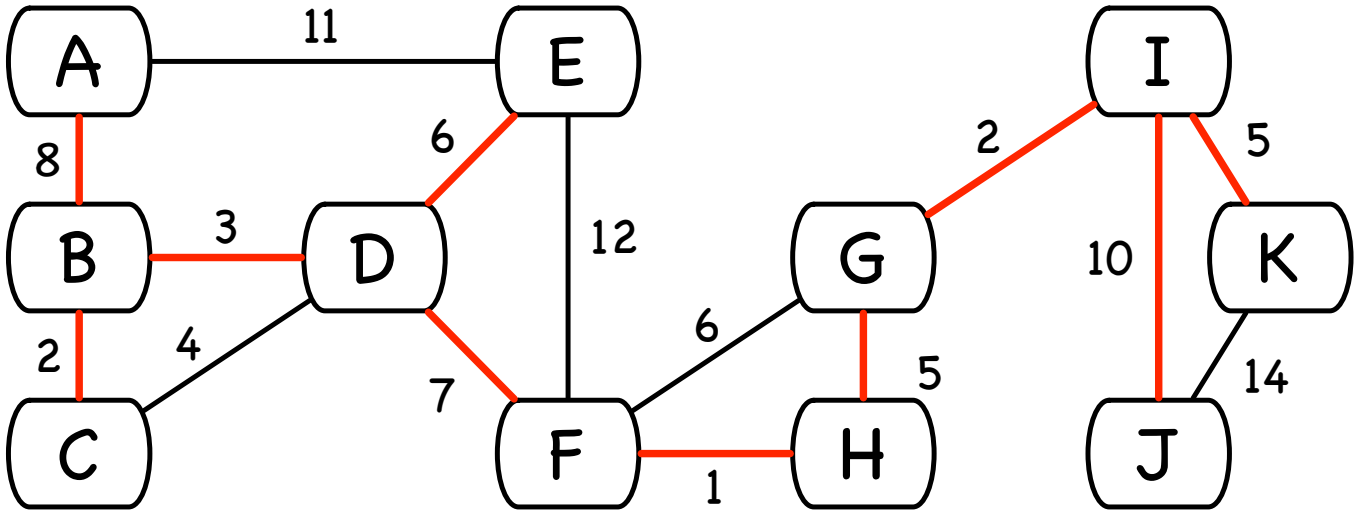
Kruskal - Beispiel



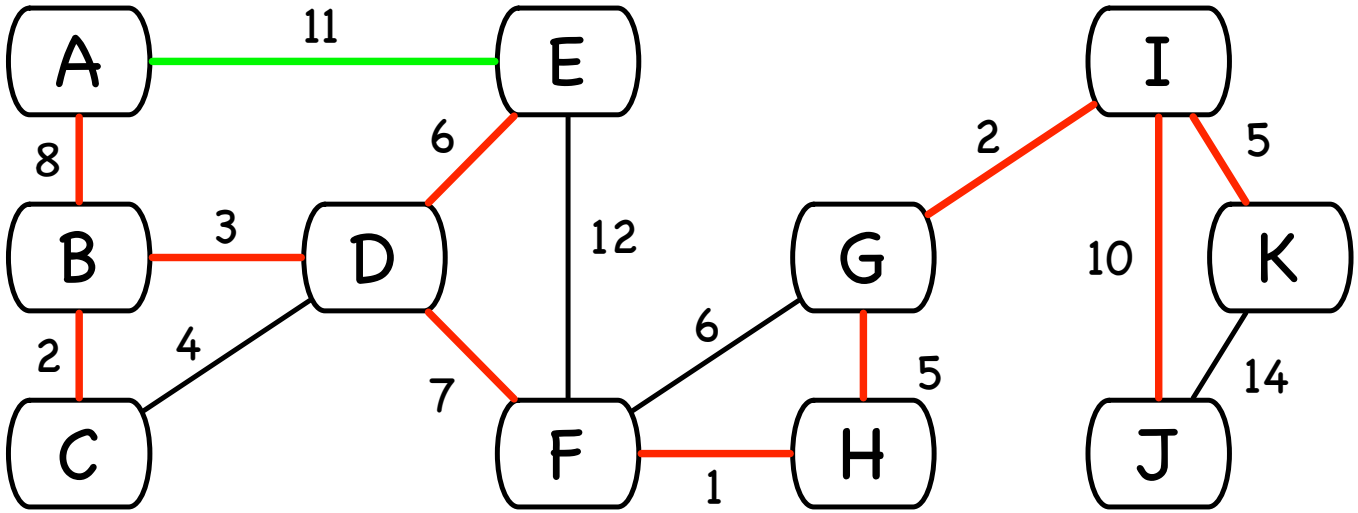
Kruskal - Beispiel



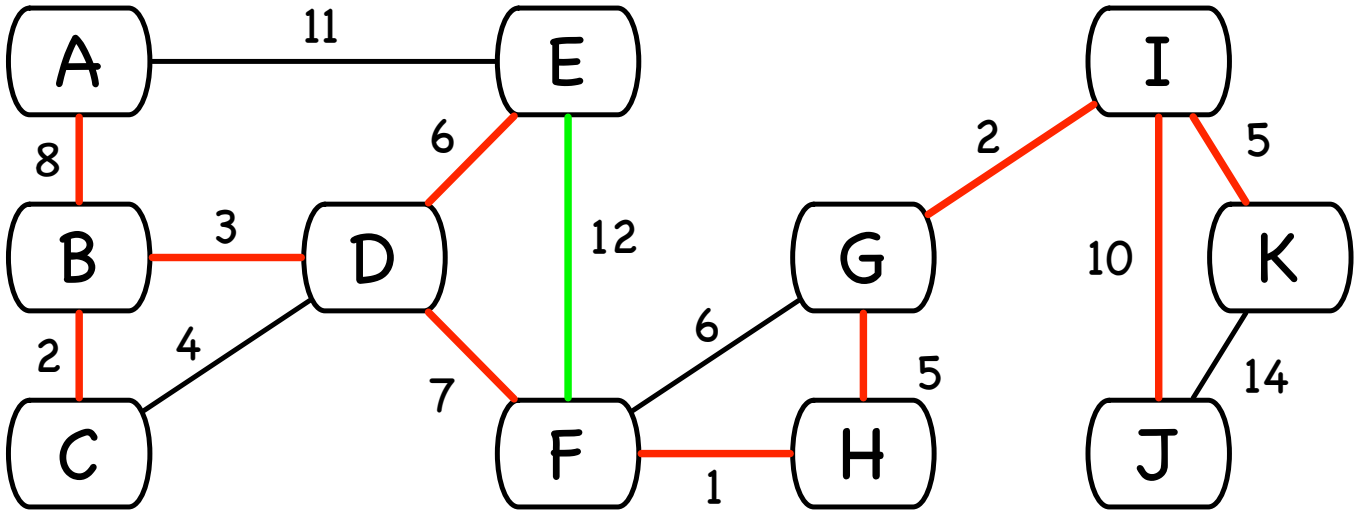
Kruskal - Beispiel



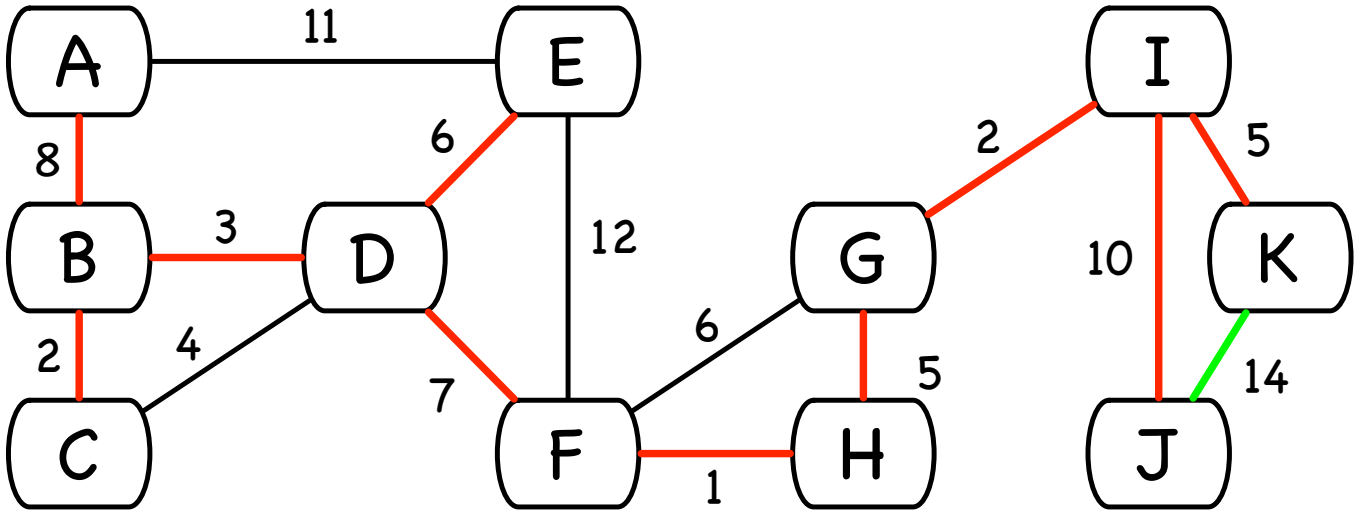
Kruskal - Beispiel



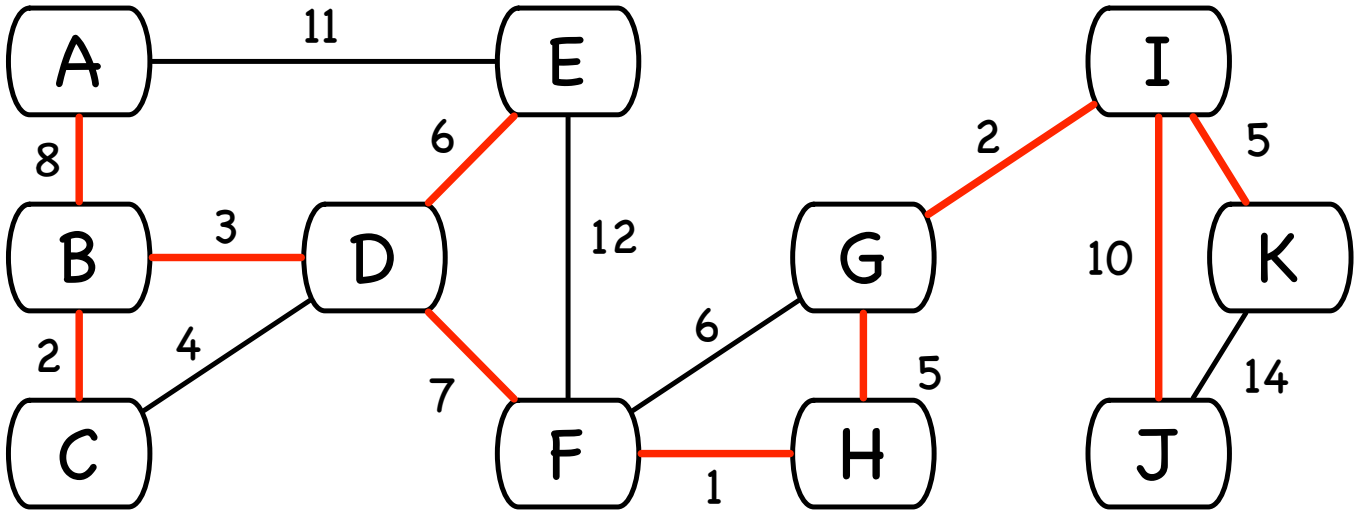
Kruskal - Beispiel



Kruskal - Beispiel



Kruskal - Beispiel



Kruskal - Datenstrukturen disjunkter Mengen

Die Implementierung ist nun recht anspruchsvoll. Wir benötigen eine Datenstruktur für disjunkte Mengen. Eine solche verwaltet eine Menge $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ von disjunkten dynamischen Mengen. Jede Menge wird durch einen Repräsentanten dargestellt. Wichtige Funktionen:

- `MakeSet`(v). Erzeugt eine neue Menge, deren einziges Element v ist. v ist dann Repräsentant dieser Menge.
- `FindSet`(v). Gibt einen Zeiger auf den Repräsentanten der (eindeutigen) Menge zurück, die v enthält.
- `Union`(u, v). Vereinigt die dynamischen Mengen, die u und v enthalten

Kruskal - Pseudocode

Algorithmus 3 KruskalMST(G, w)

1: $A = \emptyset$
2: **for** alle $v \in V(G)$ **do**
3: `MakeSet`(v)
4: **end for**
5: sortiere E in nichtfallender Reihenfolge nach dem Gewicht w
6: **for** alle $(u, v) \in E$ (sortiert) **do**
7: **if** `FindSet`$(u) \neq$ `FindSet`(v) **then**
8: $A = A \cup \{(u, v)\}$
9: `Union`(u, v)
10: **end if**
11: **end for**
12: **return** A

Kruskal - Ergebnis

- Die Korrektheit folgt aus den vorherigen Sätzen und aus der Korrektheit der Methoden der Datenstruktur für disjunkte Mengen (wäre noch zu zeigen).
- Die Laufzeit hängt auch stark von letztgenannten Methoden ab. Sie ist in $O(E \cdot \log V)$ möglich.

$\Rightarrow$ Idee des Verfahrens ist schnell erklärt, aber sowohl die Implementierung als auch die Beweisführung ist sehr anspruchsvoll.

Für Interessierte

Interessierte Leser finden mehr dazu in Kapitel 21 im [Cormen].

Prim - Die Idee

Ähnlich wie der Algorithmus von Kruskal basiert auch der Algorithmus von Prim auf einer einfachen Idee, um die sichere Kante auszuwählen:

- Wähle aus den Kanten eine minimale aus, die A mit einem noch isolierten Knoten von G_A verbindet. (Ganz zu Anfang startet man mit einem beliebigen Knoten.)

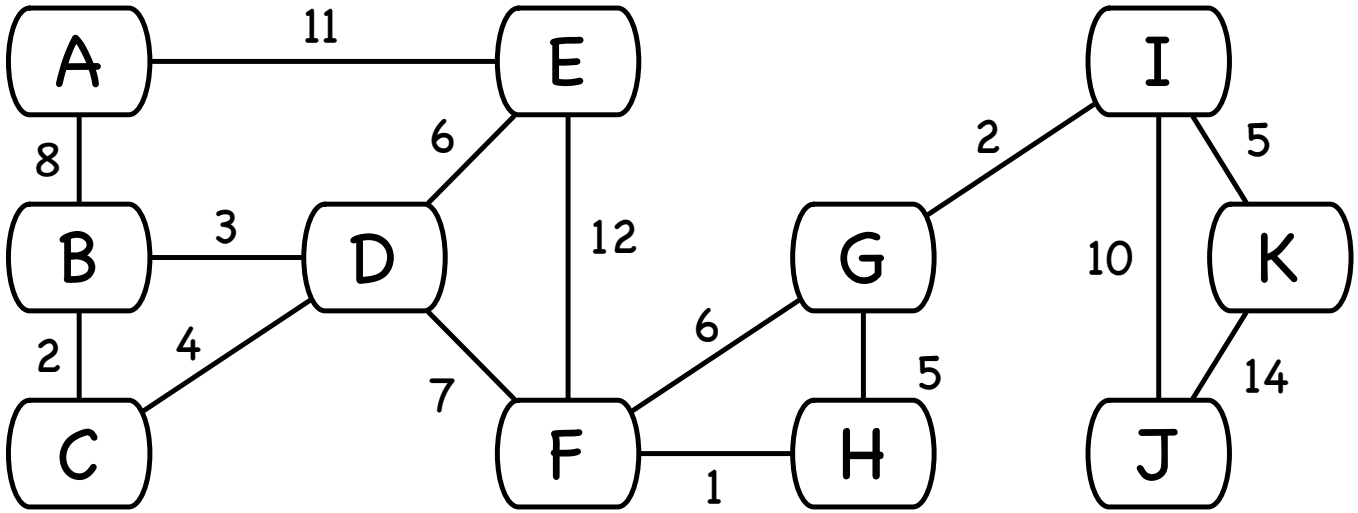
Die Menge A ist hier also ein einzelner Baum. Die hinzugefügte, sichere Kante ist immer eine Kante mit minimalen Gewicht, die den Baum mit einem Knoten verbindet, der noch nicht zum Baum gehörte. Die Korrektheit dieses Verfahrens folgt wieder aus dem vorhin vorgestellten Satz...

Eine wichtige Folgerung - Wiederholung

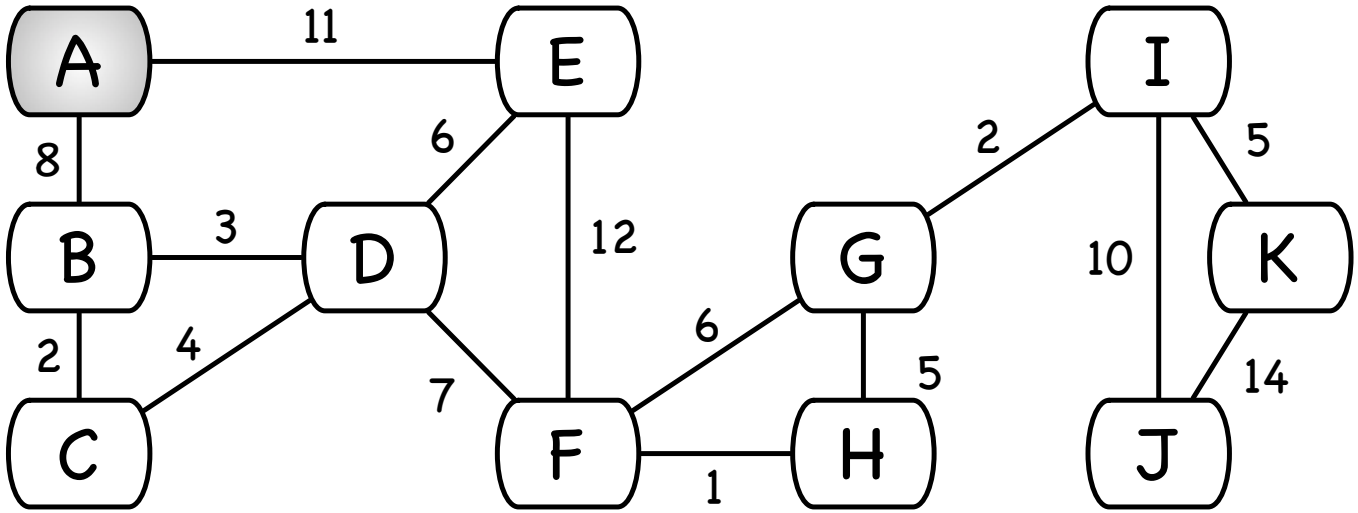
Satz

Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph und $w : E \to \mathbb{R}^+$ eine Gewichtsfunktion. Sei $A \subseteq E$ Teilmenge eines minimalen Spannbaums von G und $C = (V_C, E_C)$ eine Zusammenhangskomponente aus $G_A = (V, A)$. Falls (u, v) unter allen Kanten, die C mit einer anderen Komponente von G_A verbinden, das kleinste Gewicht hat, dann ist (u, v) für A sicher.

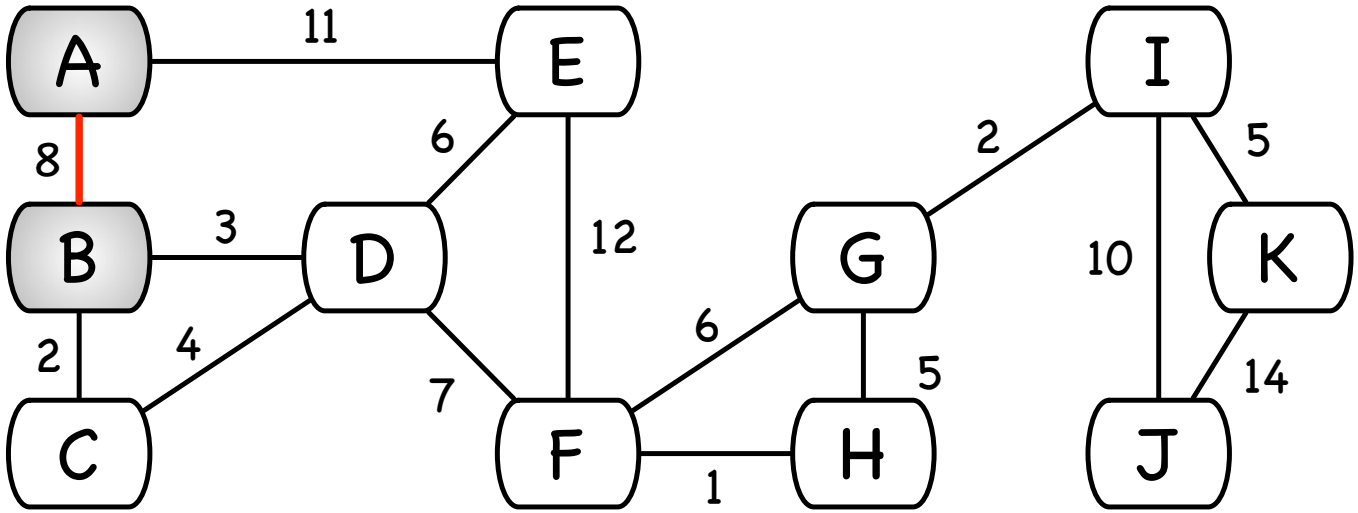
Prim - Ein Beispiel



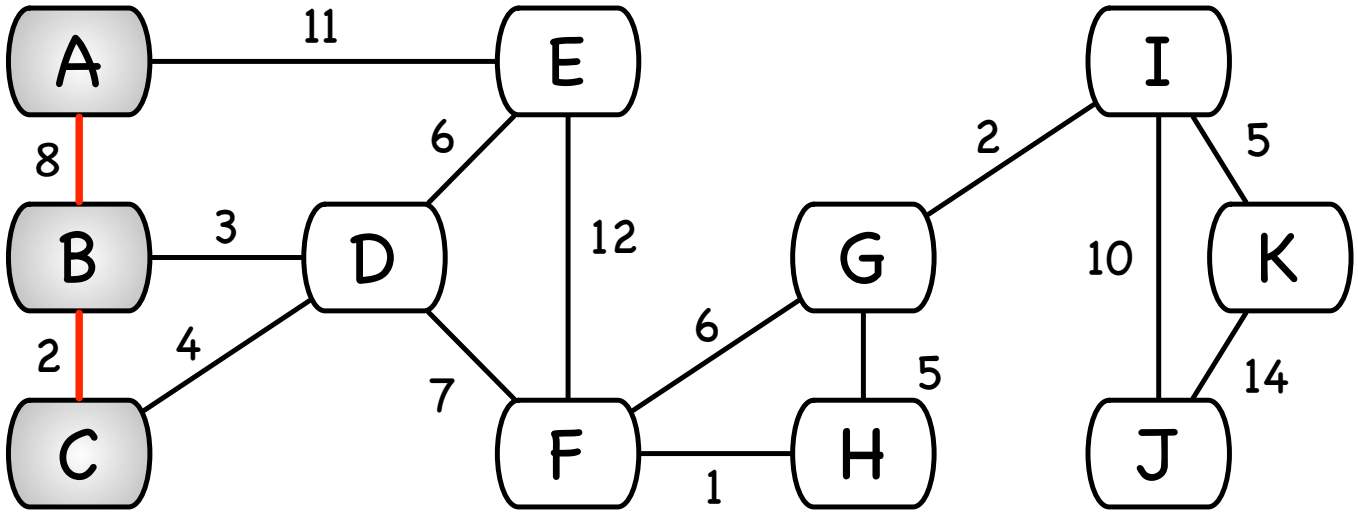
Prim - Ein Beispiel



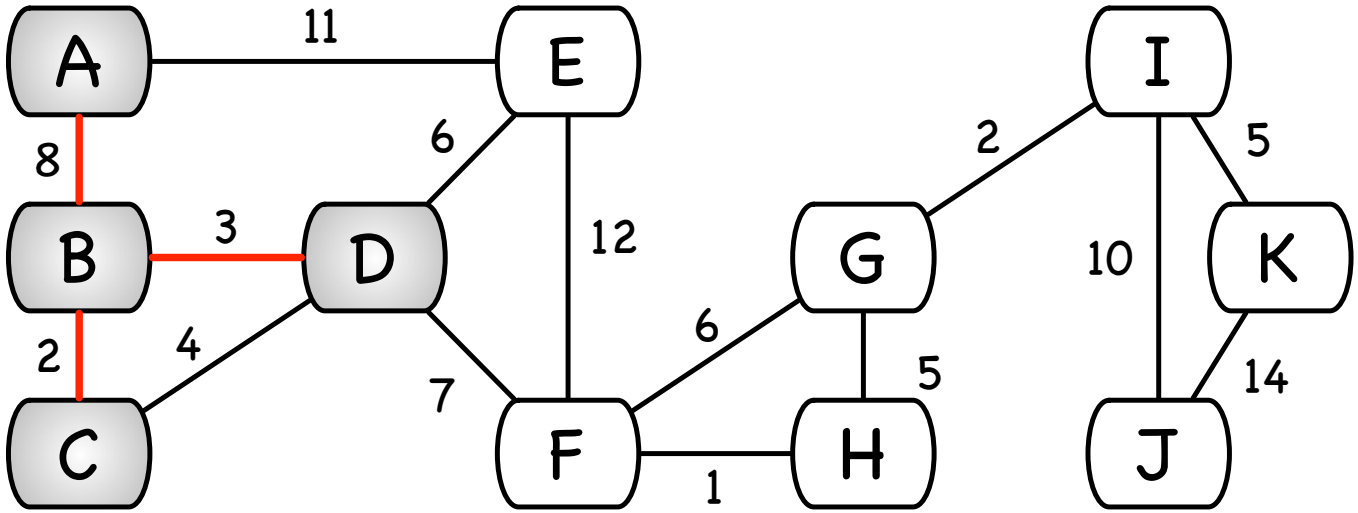
Prim - Ein Beispiel



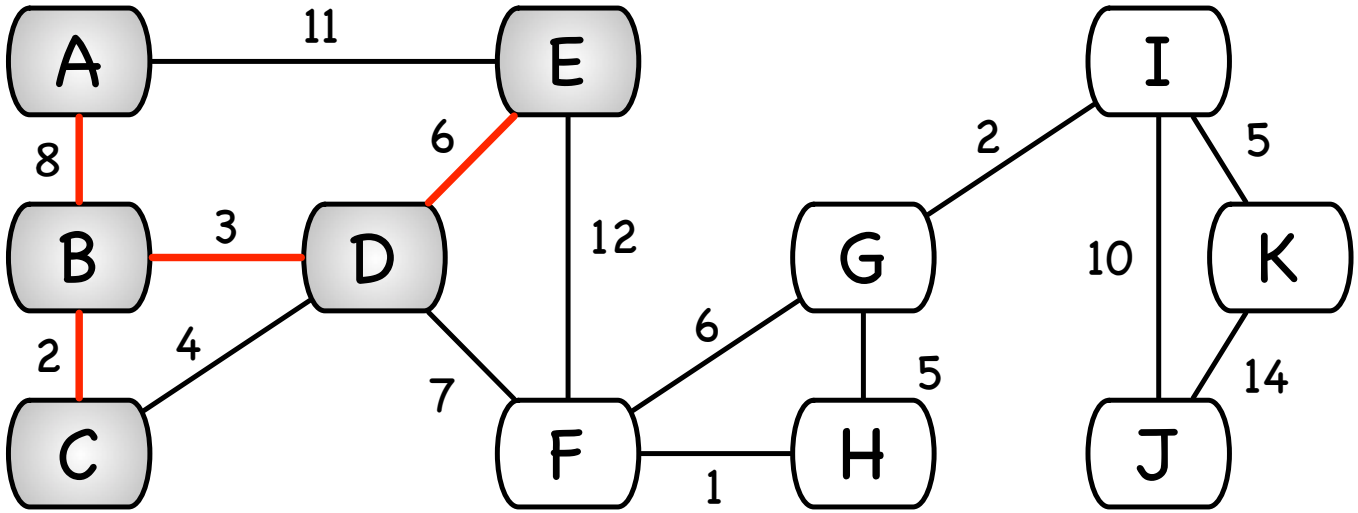
Prim - Ein Beispiel



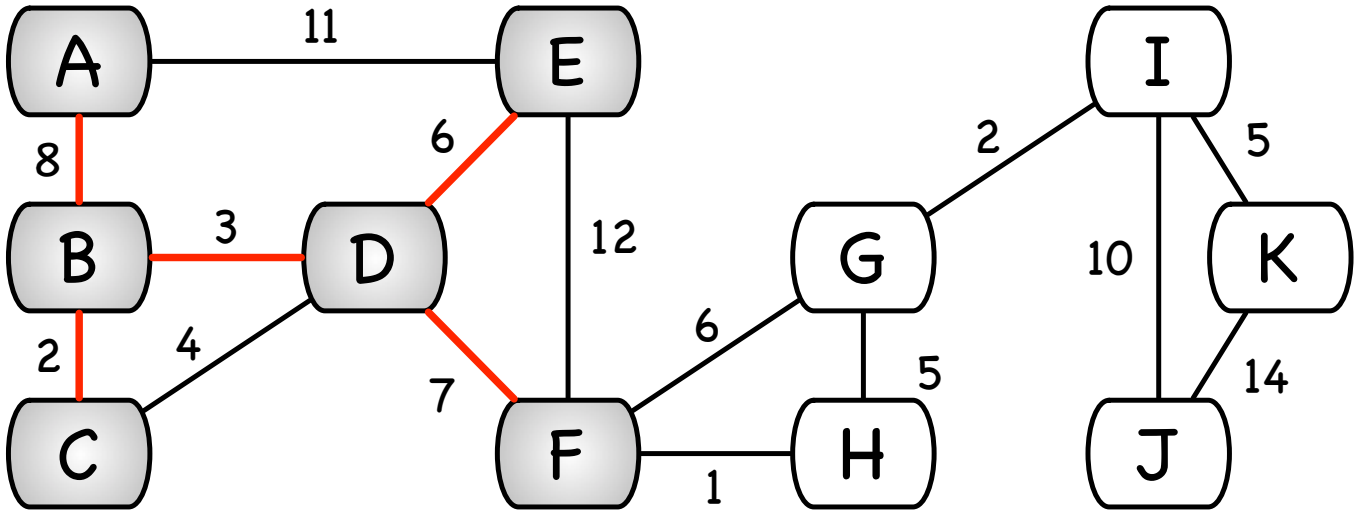
Prim - Ein Beispiel



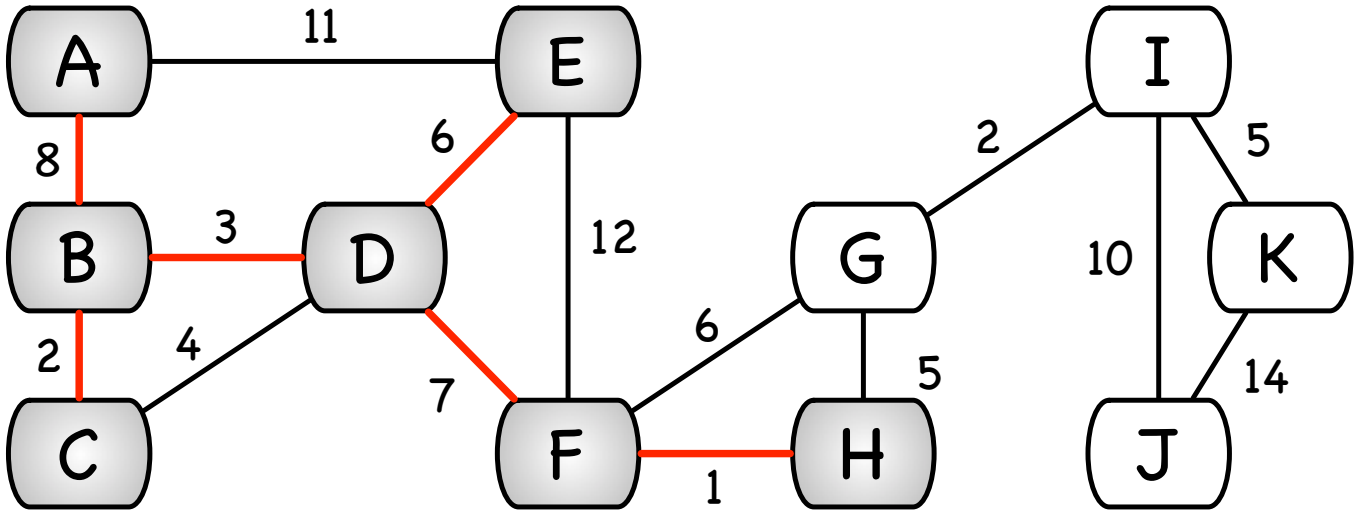
Prim - Ein Beispiel



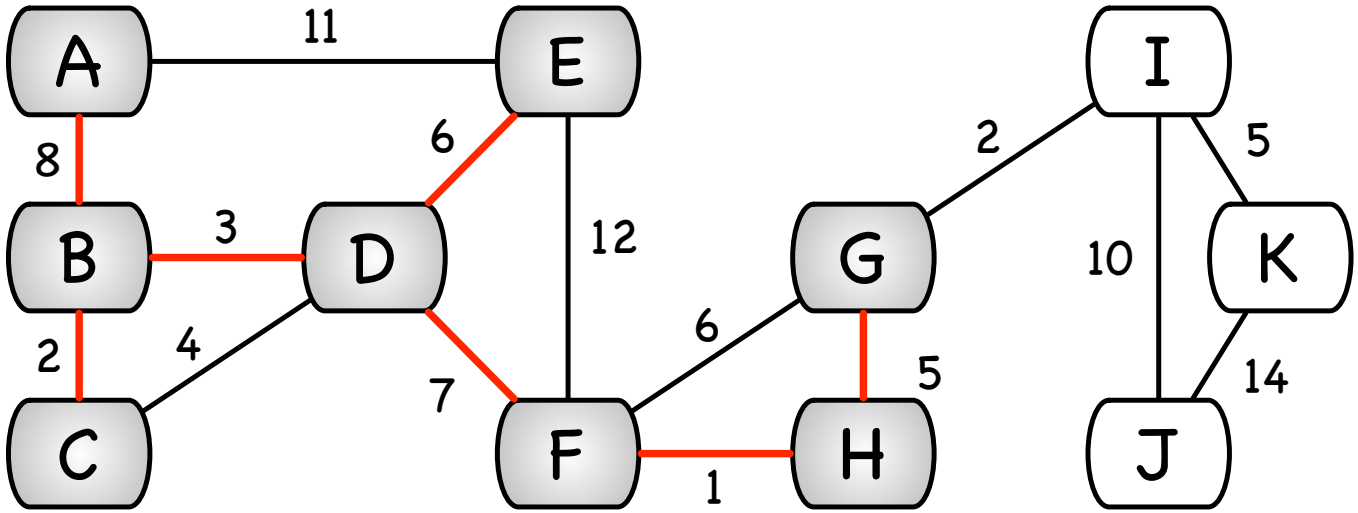
Prim - Ein Beispiel



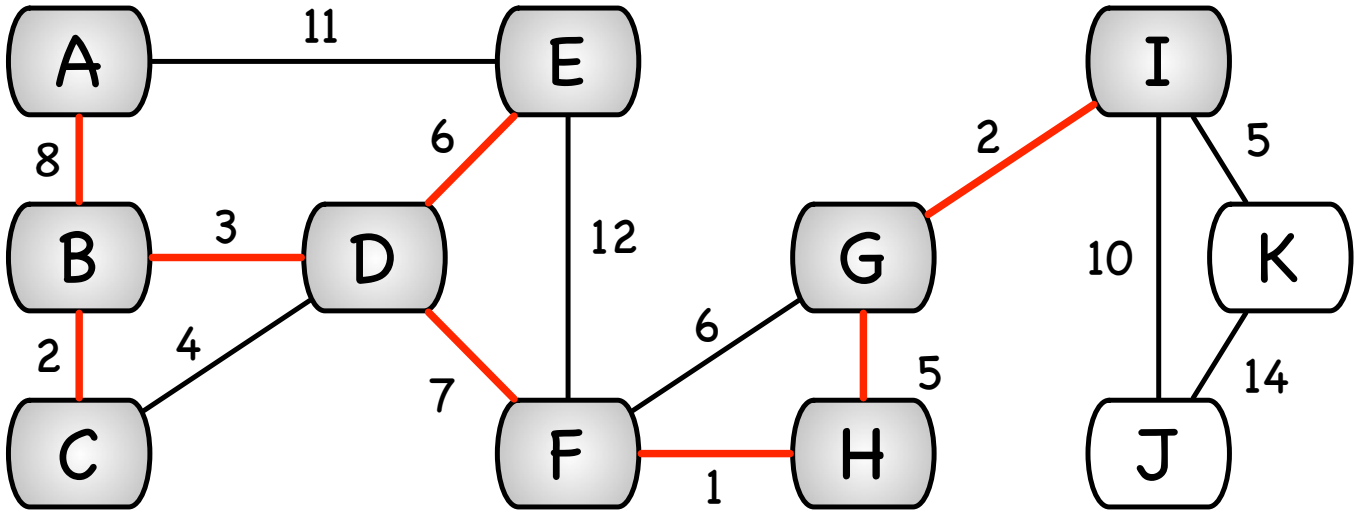
Prim - Ein Beispiel



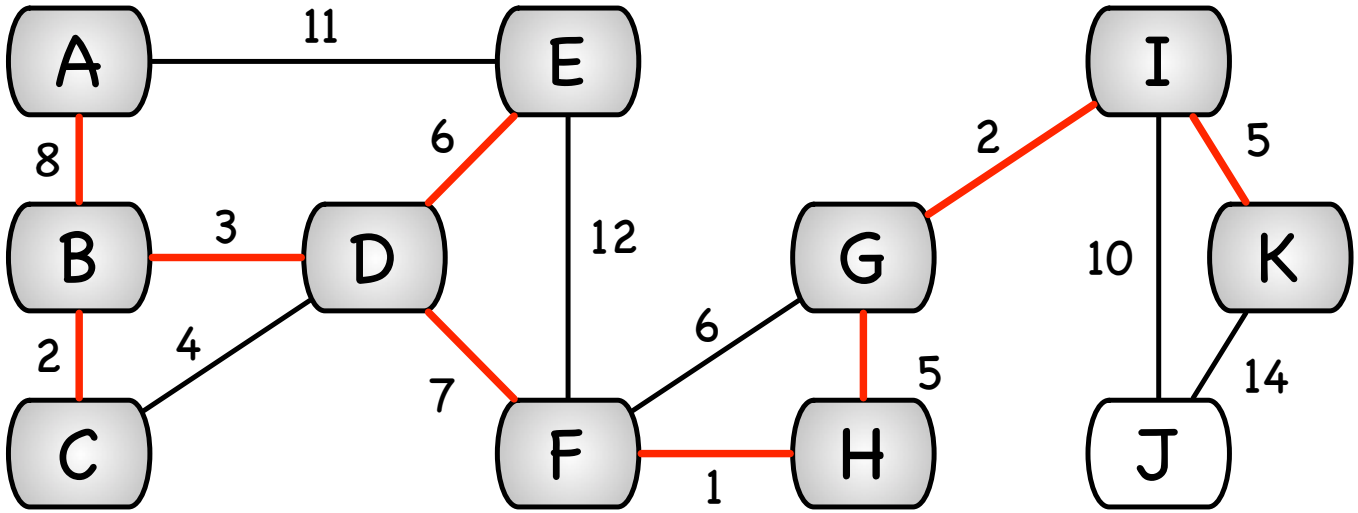
Prim - Ein Beispiel



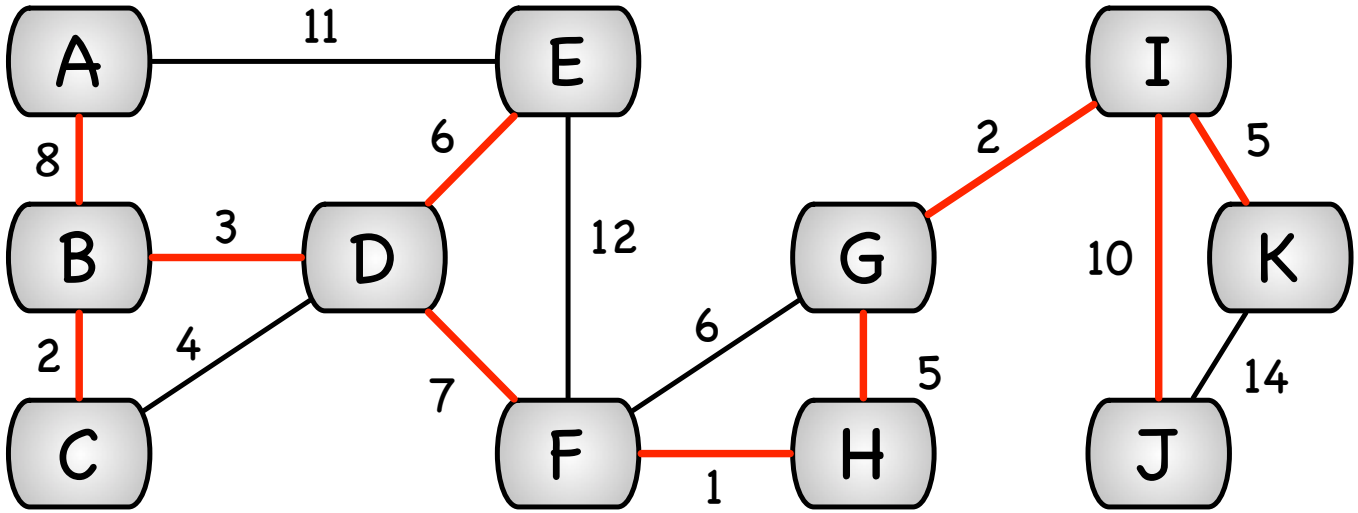
Prim - Ein Beispiel



Prim - Ein Beispiel



Prim - Ein Beispiel



Implementierung

Die Schwierigkeit bei der Implementierung besteht darin, die Auswahl der Kante, die zu A hinzugefügt werden soll, möglichst einfach (d.h. effizient) zu gestalten.

Im nachfolgenden Algorithmus wird mit einer Min-Prioritätswarteschlange gearbeitet (die man z.B. mit binären Min-Heaps implementieren könnte). In dieser werden alle nicht zum Baum gehörenden Knoten gespeichert.

schlüssel[v] ist das kleinste Gewicht aller Kanten, die v mit einem Knoten des (bisherigen) Baumes verbinden (und muss daher u.U. im Verlauf angepasst werden).

Die Menge A wird implizit als $A = \{(v, \pi[v]) \mid v \in V - \{r\} - Q\}$ gehalten.

Prim - Pseudocode

Algorithmus 4 PrimMST(G, w, r)

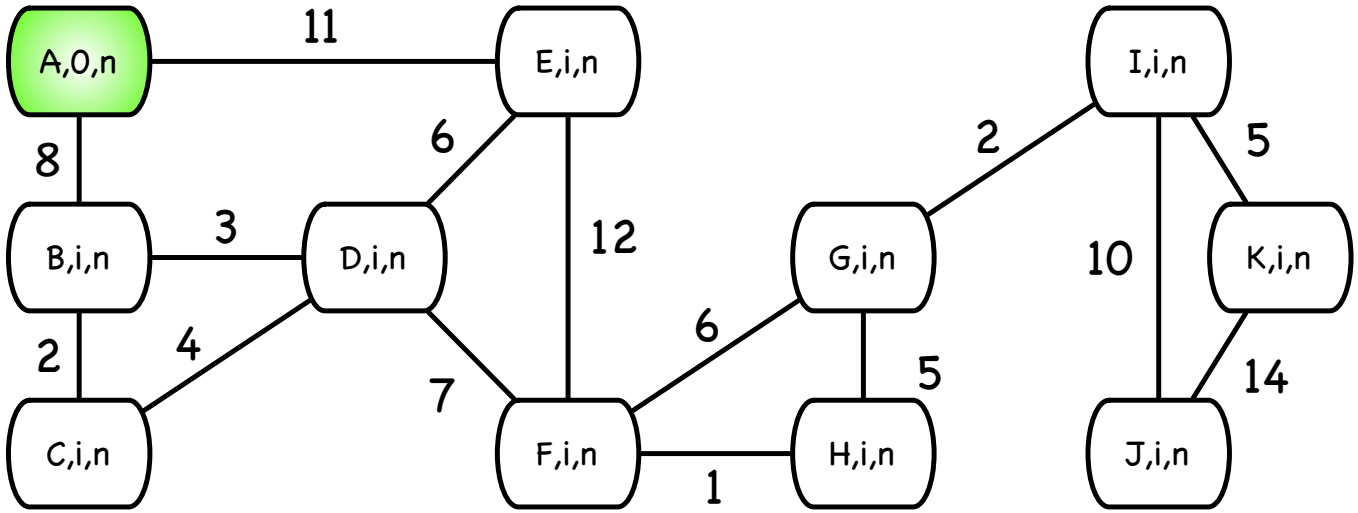
```
1:  for alle v ∈ V(G) do
2:      schlüssel[v] = ∞, π[v] = nil
3:  end for
4:  schlüssel[r] = 0, Q = V(G)
5:  while Q ≠ ∅ do
6:      u = ExtractMin(Q)
7:      for alle v ∈ Adj[u] do
8:          if v ∈ Q und w(u, v) < schlüssel[v] then
9:              π[v] = u
10:             schlüssel[v] = w(u, v)
11:         end if
12:     end for
13: end while
```

Prim - Anmerkung

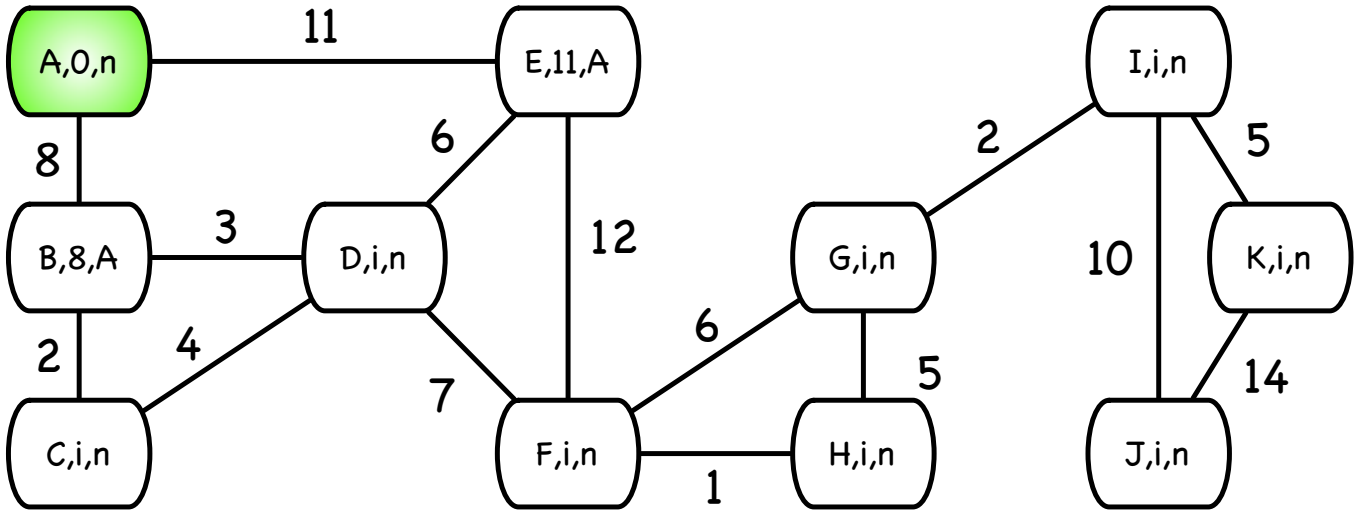
Anmerkung (zur Nachbereitung)

Man beachte: Der Algorithmus arbeitet etwas anders als im Beispiel oben, da ja $\pi[v]$ und schlüssel$[v]$ u.U. mehrmals gesetzt werden. (Dies ist nötig, um den Spannbaum zu konstruieren und geschah vorher implizit indem z.B. immer wieder alle Kanten betrachtet wurden.)
Im nachfolgenden Beispiel ist das erste Element im Tupel der Knotenname, das zweite ist schlüssel$[v]$ und das dritte ist $\pi[v]$.

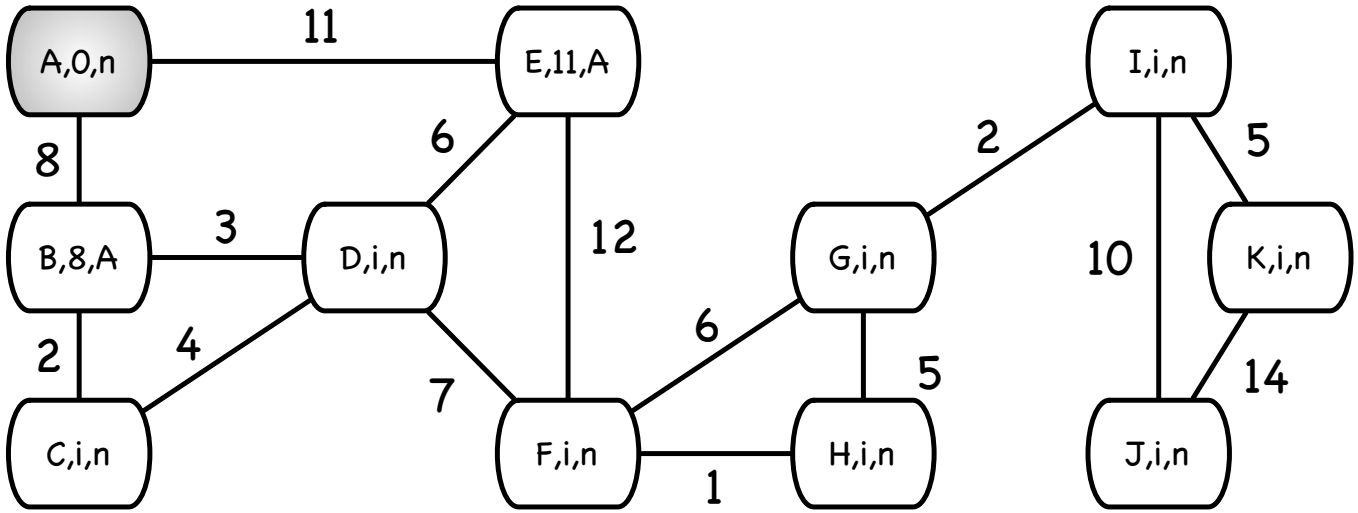
PrimMST - Ein Beispiel



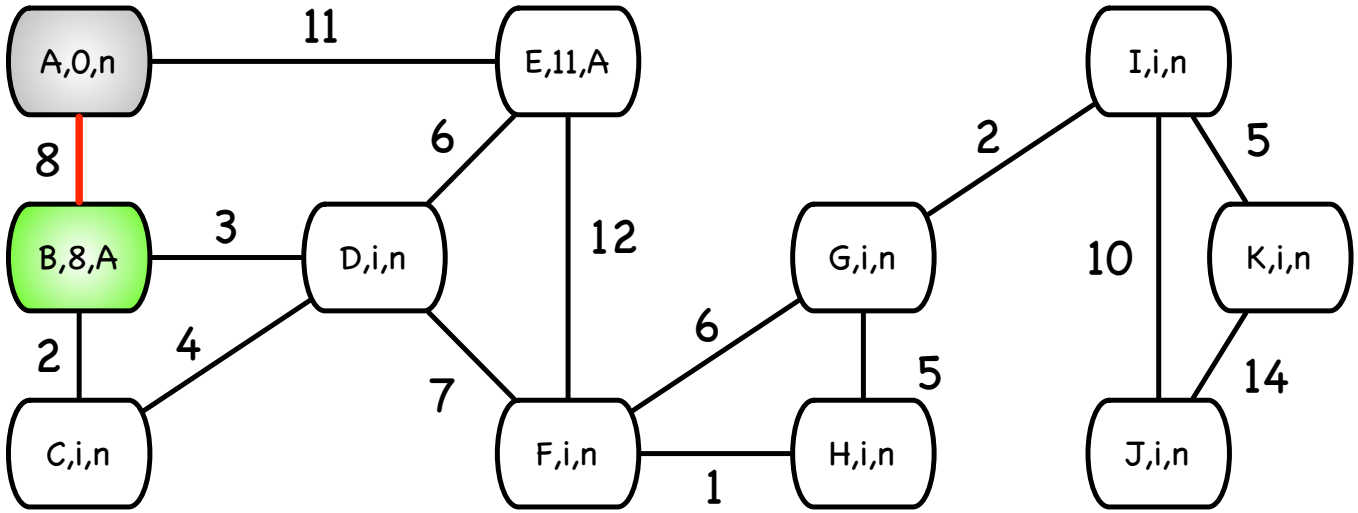
PrimMST - Ein Beispiel



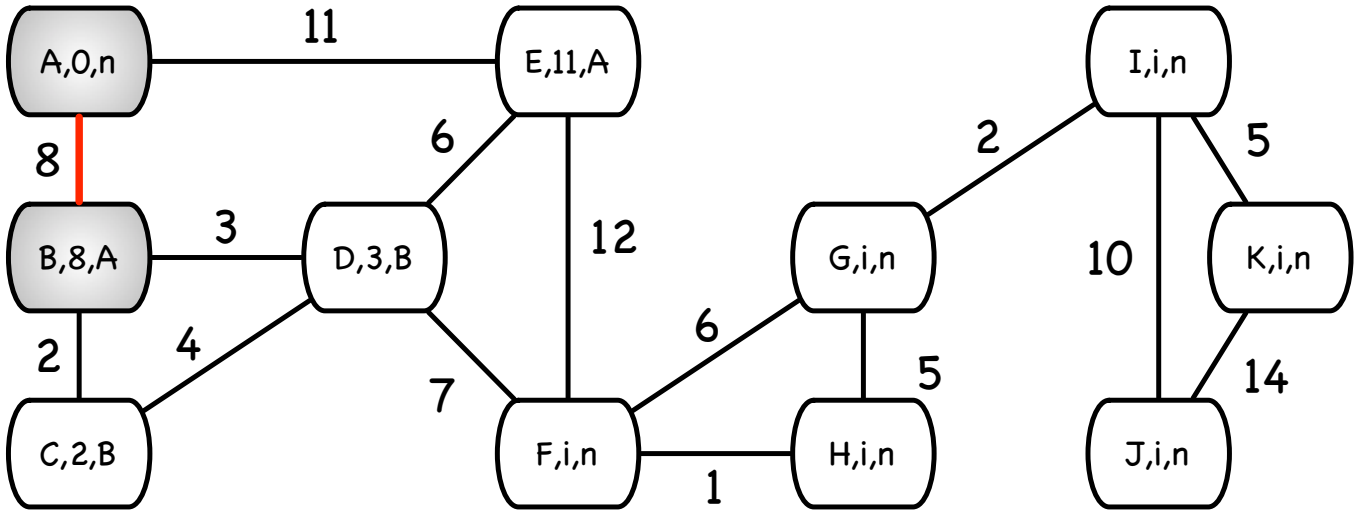
PrimMST - Ein Beispiel



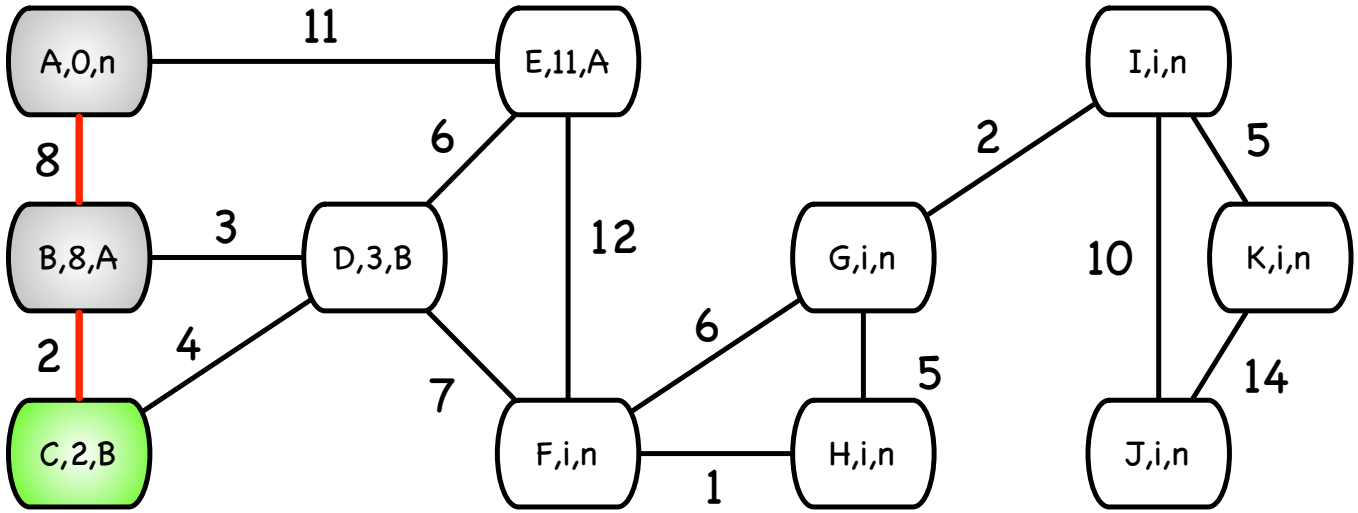
PrimMST - Ein Beispiel



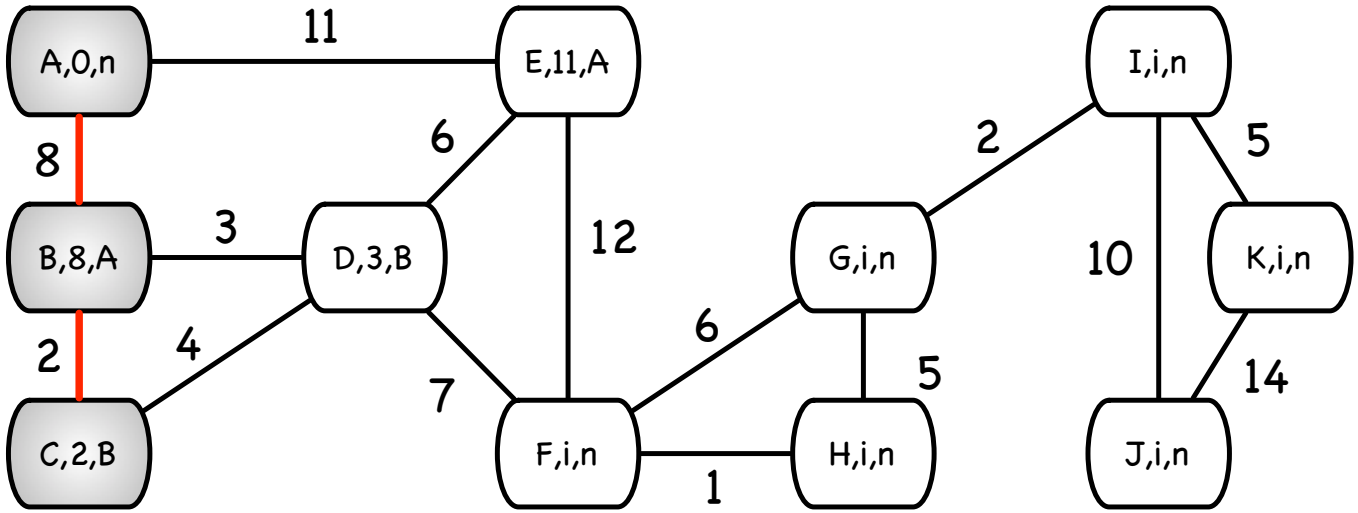
PrimMST - Ein Beispiel



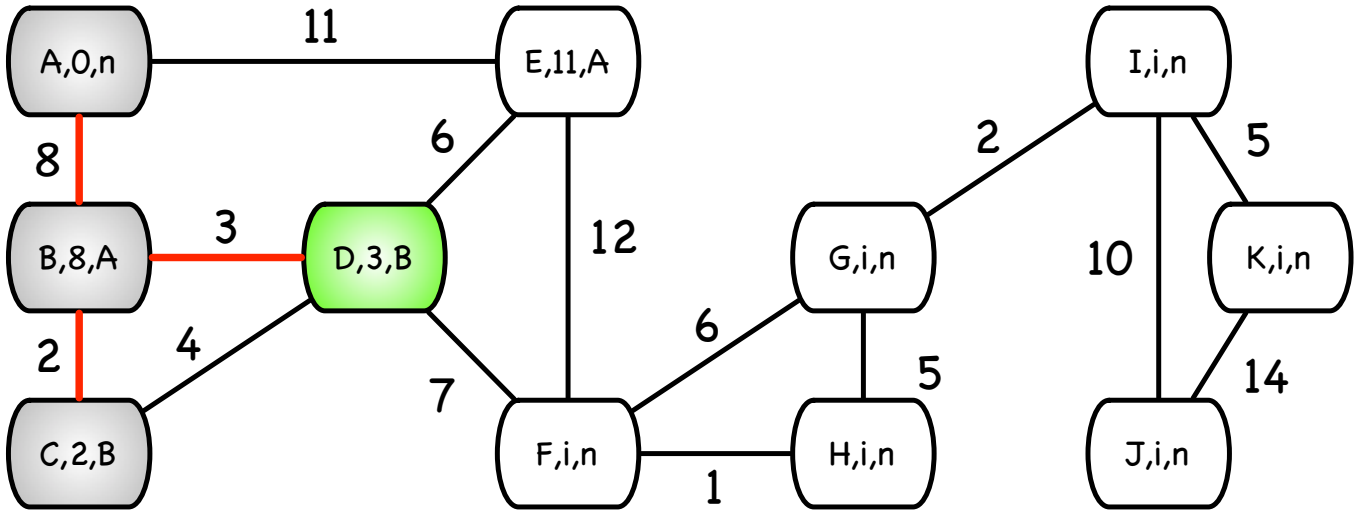
PrimMST - Ein Beispiel



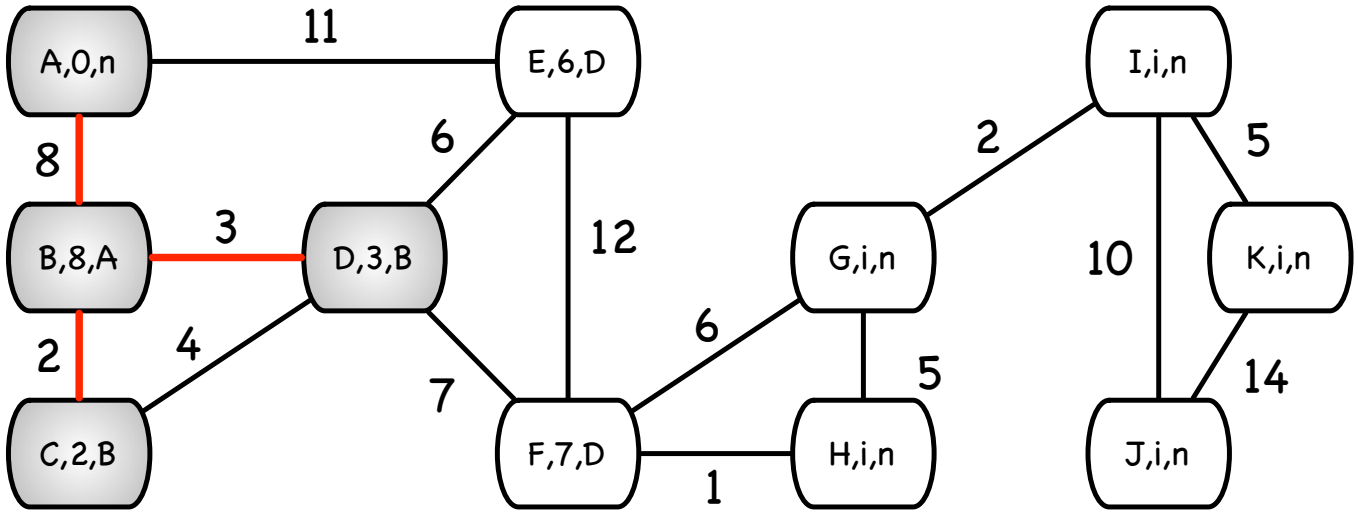
PrimMST - Ein Beispiel



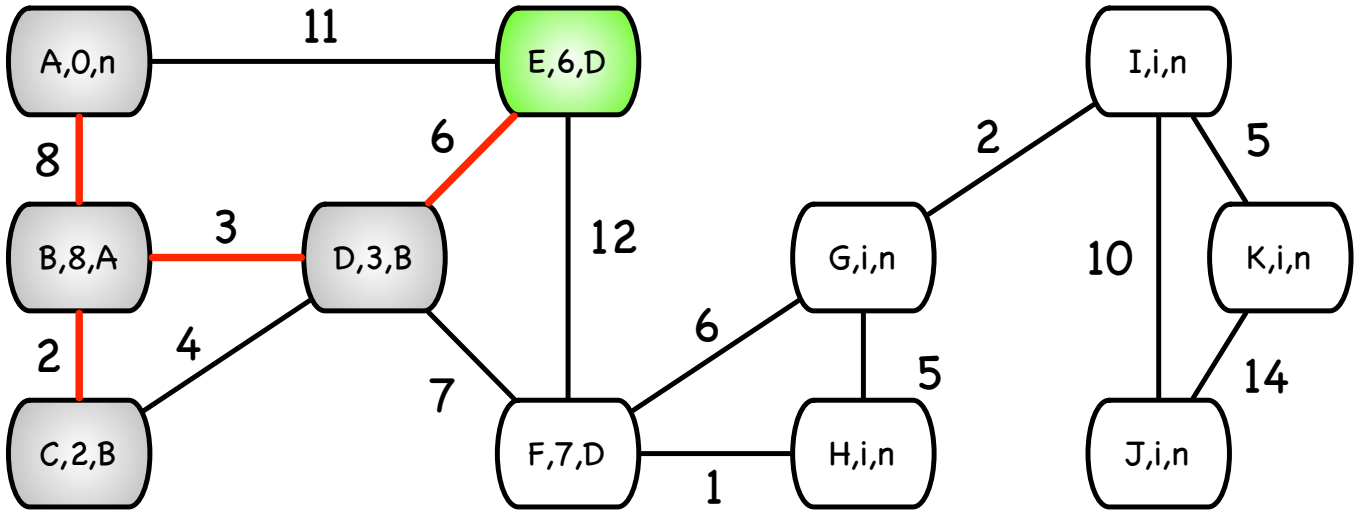
PrimMST - Ein Beispiel



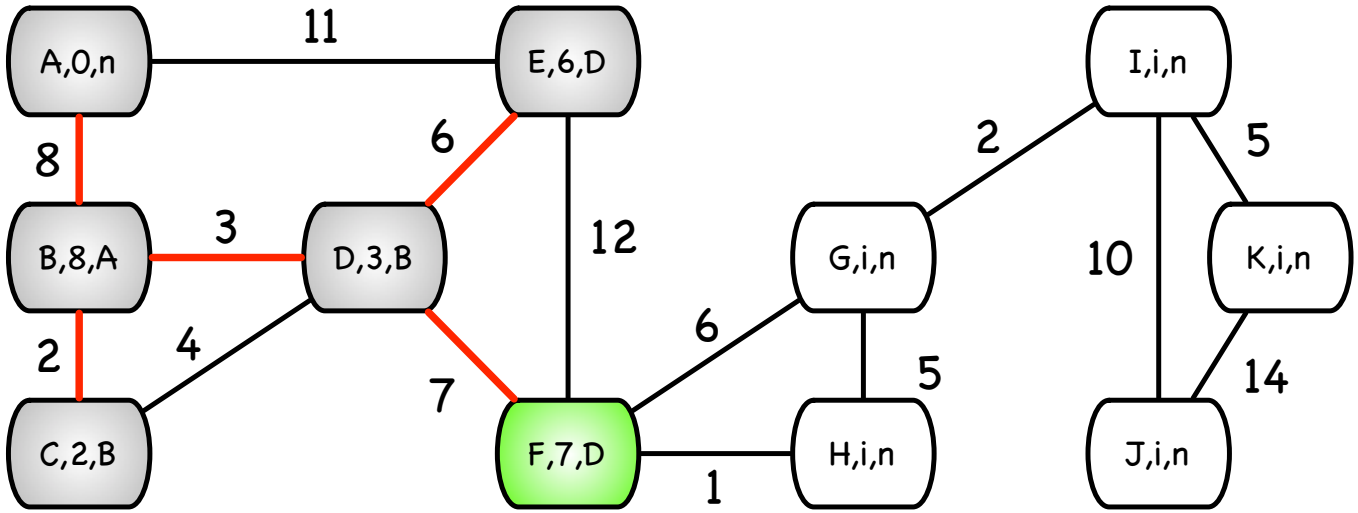
PrimMST - Ein Beispiel



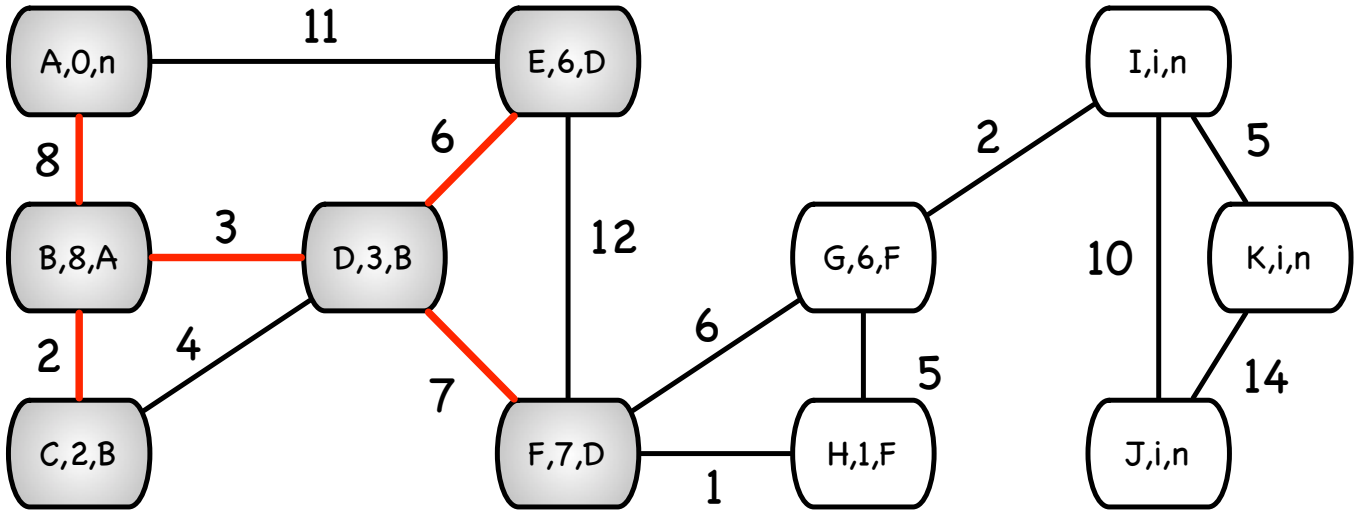
PrimMST - Ein Beispiel



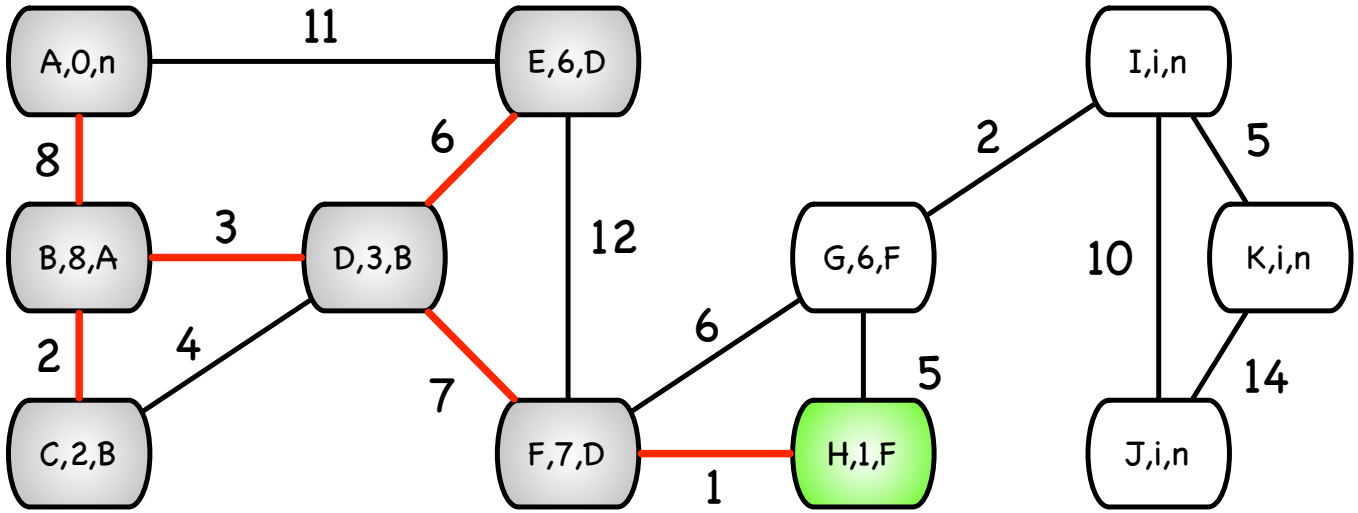
PrimMST - Ein Beispiel



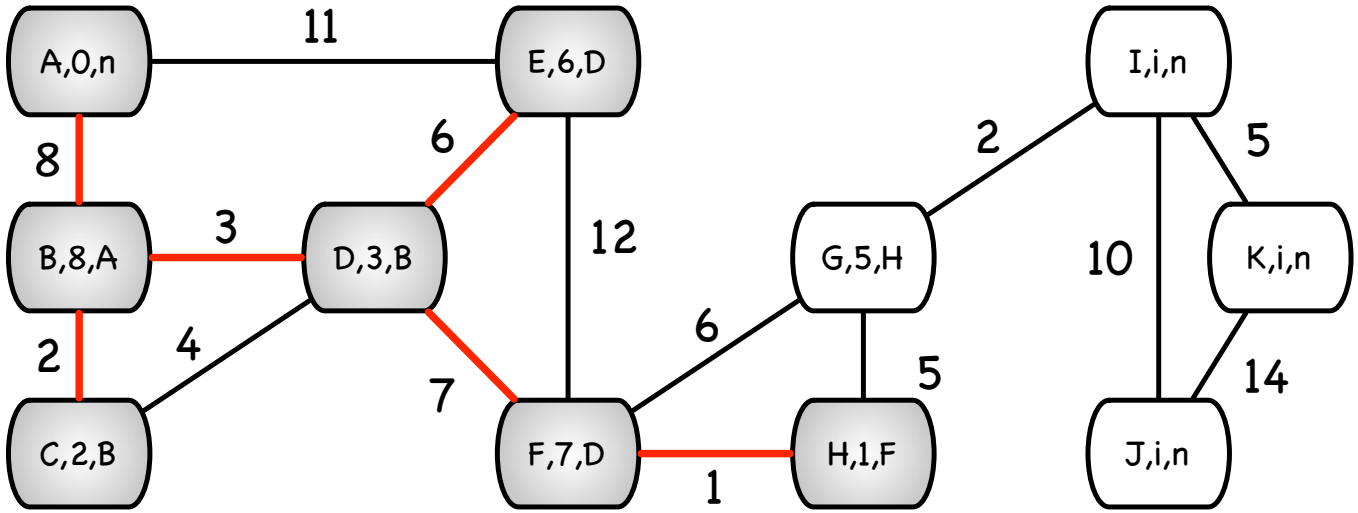
PrimMST - Ein Beispiel



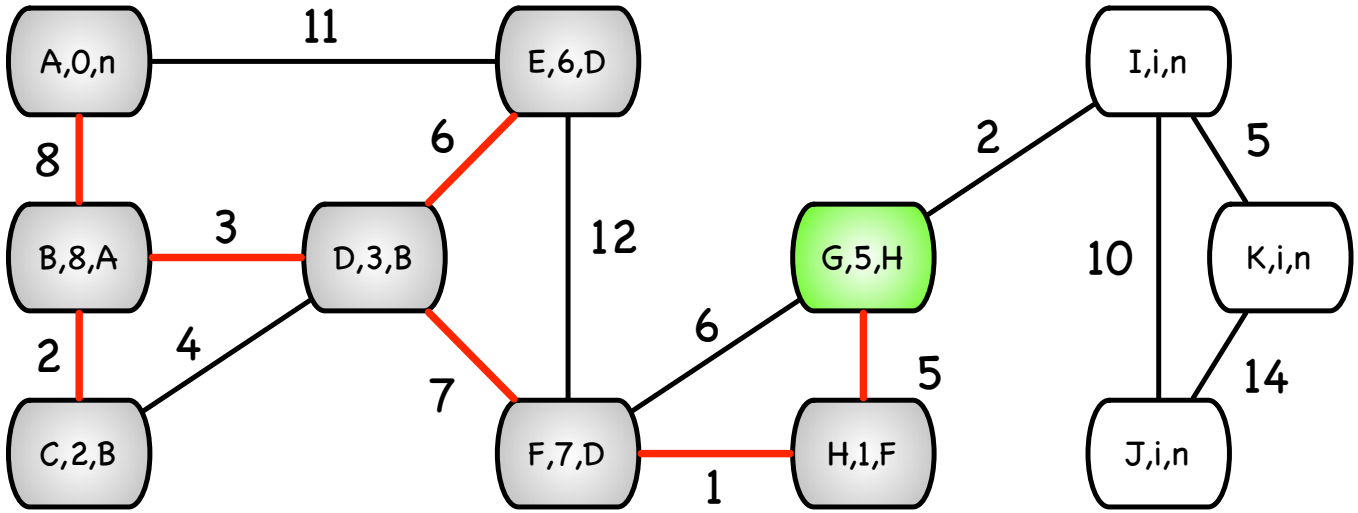
PrimMST - Ein Beispiel



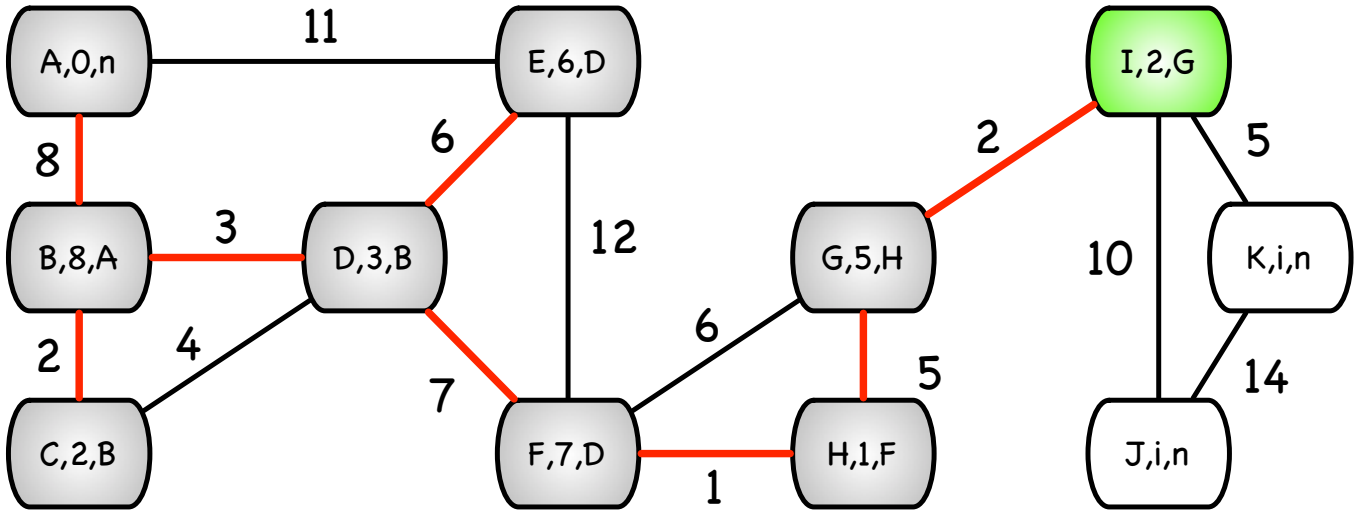
PrimMST - Ein Beispiel



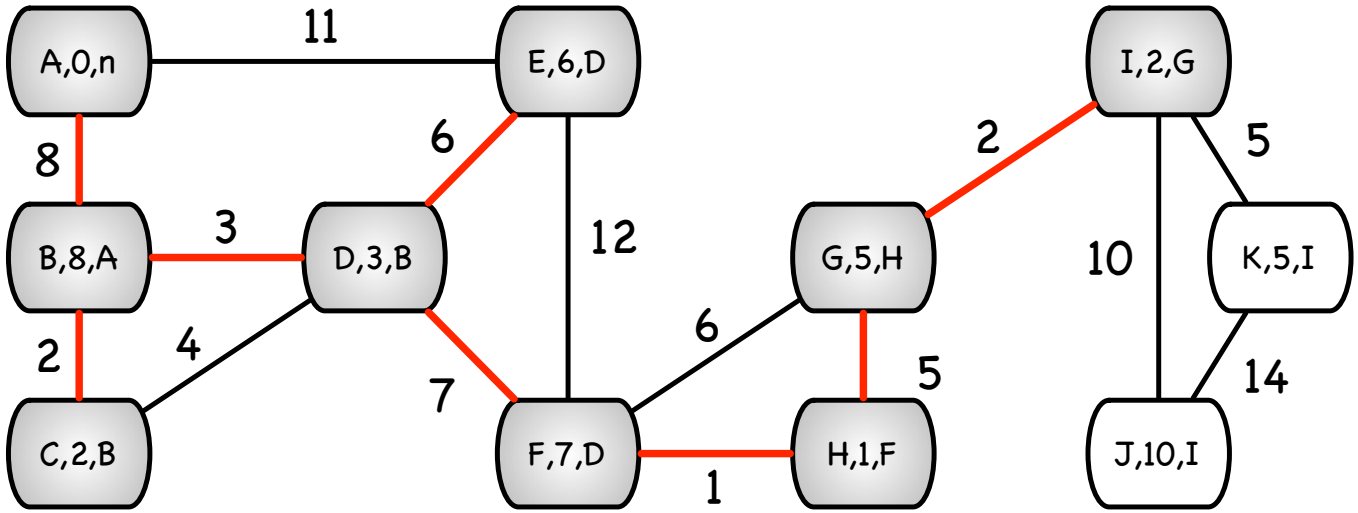
PrimMST - Ein Beispiel



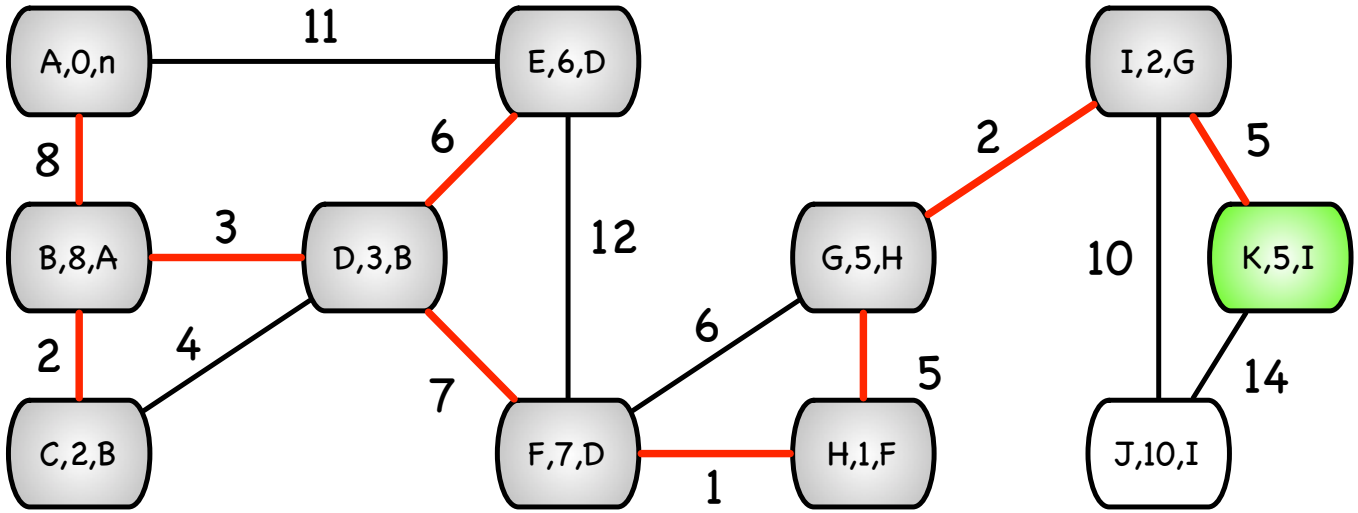
PrimMST - Ein Beispiel



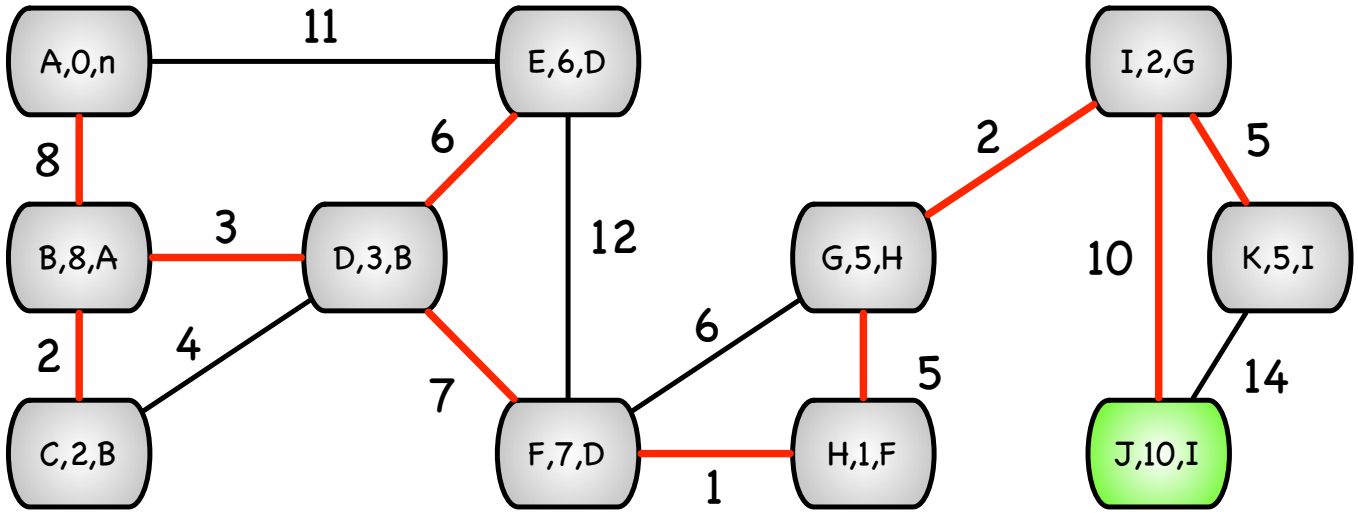
PrimMST - Ein Beispiel



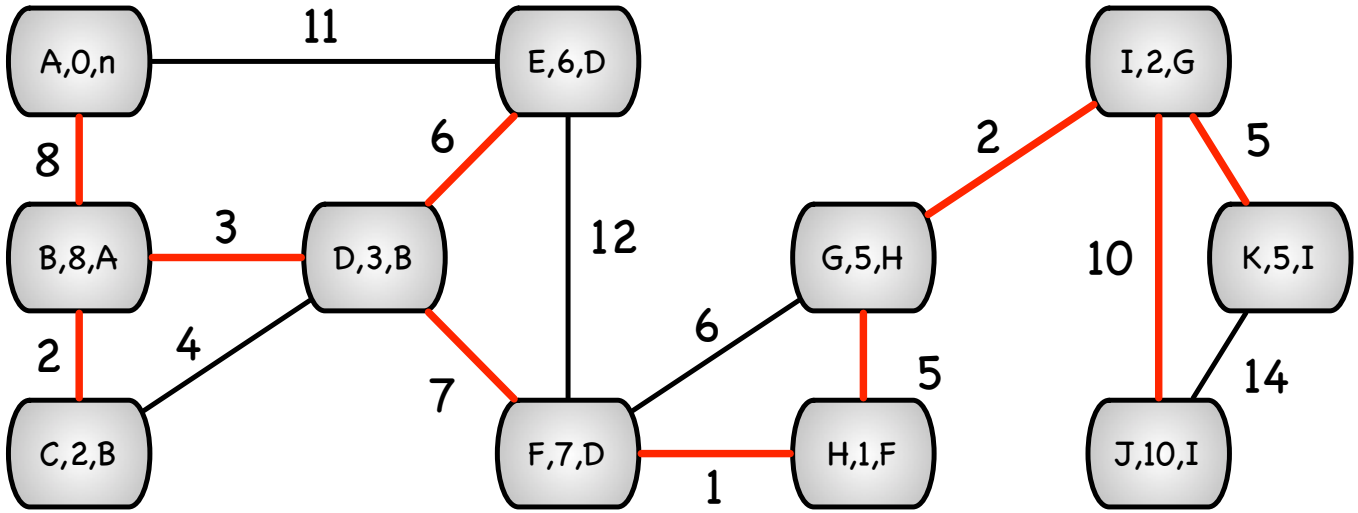
PrimMST - Ein Beispiel



PrimMST - Ein Beispiel



PrimMST - Ein Beispiel



Prim - Schleifeninvariante

Wir müssen nun noch die Korrektheit des Algorithmus zeigen. Wir wussten vorher nur, dass die *Idee* (diese Kante zu nehmen) korrekt ist. Nachweisen müssen wir aber, dass der Algorithmus dies auch stets tut und somit die Idee von Prim korrekt implementiert.

Vor jeder Iteration der while-Schleife gilt:

1. $A = \{(v, \pi[v]) \mid v \in V - \{r\} - Q\}$
2. *Die Knoten, die sich bereit im MST befinden, sind jene aus* $V - Q$.
3. *Ist* $v \in Q$ *mit* $\pi[v] \neq \texttt{nil}$, *so ist* schlüssel$[v]$ *das Gewicht einer Kante* $(v, \pi[v])$, *die* v *mit einem Knoten verbindet, der bereits zum minimalen Spannbaum gehört und die unter allen Kanten mit dieser Eigenschaft minimal ist.*

Zur Nachbereitung

Anmerkung (zur Nachbereitung)

Bei Kruskal war dies nicht nötig, da die Implementierung bereits sehr dicht an der Idee war.

Anmerkung (zur Nachbereitung)

Die Schleifeninvariante wäre nun formal zu zeigen (und daraus die Korrektheit zu folgern). Wir diskutieren dies nachfolgend nur kurz anhand des Pseudocodes. (Wer nicht in der Vorlesung war: Gilt die Schleifeninvariante bei der Initialisierung? Klappt's mit der Fortsetzung? Terminiert der Algorithmus und liefert die Schleifeninvariante dann etwas, womit die Korrektheit des Algorithmus gefolgert werden kann?)

Prim - Pseudocode (Wdh.)

Algorithmus 5 PrimMST(G, w, r)

```
1:  for alle v ∈ V(G) do
2:      schlüssel[v] = ∞, π[v] = nil
3:  end for
4:  schlüssel[r] = 0, Q = V(G)
5:  while Q ≠ ∅ do
6:      u = ExtractMin(Q)
7:      for alle v ∈ Adj[u] do
8:          if v ∈ Q und w(u, v) < schlüssel[v] then
9:              π[v] = u
10:             schlüssel[v] = w(u, v)
11:         end if
12:     end for
13: end while
```

Prim - Laufzeitanalyse

Wir benutzen einen binären MinHeap. Damit folgt:

- Die Initialisierung geht in $O(V)$.
- Der Rumpf der while-Schleife wird $|V|$-mal ausgeführt. Jede `ExtractMin`-Operation benötigt $O(\log V)$ Zeit, also benötigen wir für diese Operation insgesamt $O(V \cdot \log V)$ Zeit.
- Die for-Schleife wird $O(E)$-mal ausgeführt. In ihrem Rumpf ist der Test auf Enthaltensein in $O(1)$ möglich, indem man jeden Knoten v mit einem zusätzlichen Bit versieht, das aussagt, ob v in Q ist oder nicht. Die Zuweisung beinhaltet eine `DecreaseKey`-Operation, die in $O(\log V)$ möglich ist. Insgesamt brauchen wir hier also $O(E \cdot \log V)$.
- Insgesamt ist die Laufzeit damit in $O(V \cdot \log V + E \cdot \log V) = O(E \cdot \log V)$.

Kruskal und Prim - Zusammenfassung

Beide vorgestellten Algorithmen (Kruskal und Prim) verwenden eine bestimmte Regel, um die sichere Kante zu bestimmen.

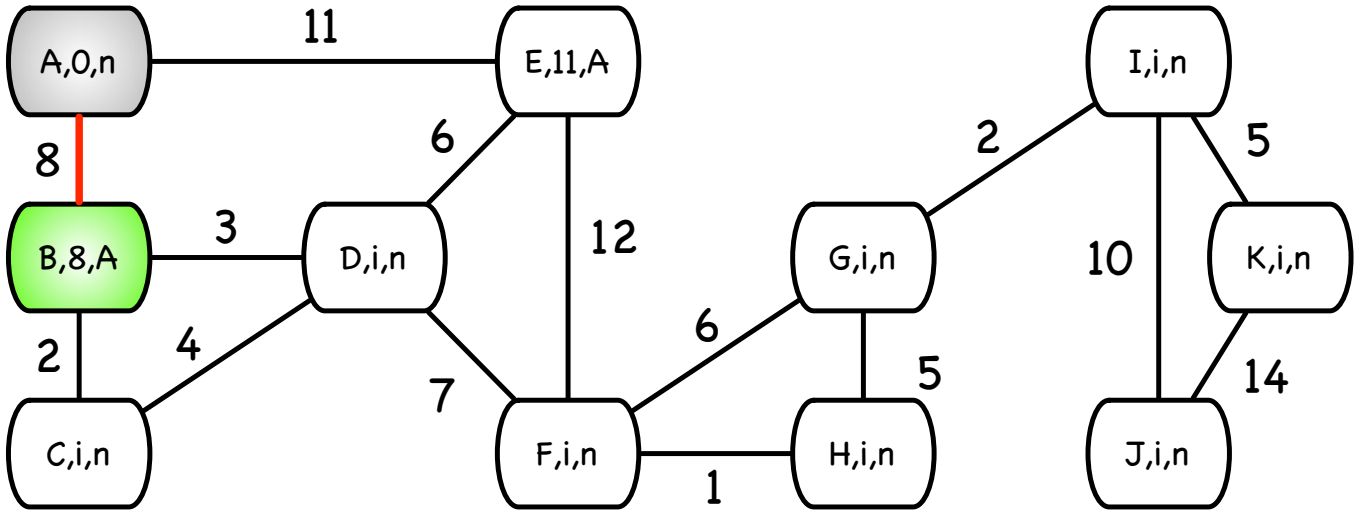
- Kruskal: Die Menge A ist ein Wald. Die hinzugefügte, sichere Kante ist immer eine Kante mit minimalen Geweicht, die zwei verschiedene Komponenten (von G_A) verbindet.
- Prim: Die Menge A ist ein einzelner Baum. Die hinzugefügte, sichere Kante ist immer eine Kante mit minimalen Gewicht, die den Baum mit einem Knoten verbindet, der noch nicht zum Baum gehörte.
- Beide Algorithmen sind *nicht eindeutig*.
- Die Algorithmen können verschiedene Ergebnisse liefern.

Mit guten Datenstrukturen ist Kruskal in $O(E \cdot \log V)$ möglich, Prim ist in $O(E \cdot \log V)$ bei Nutzung von Min-Heaps und sogar in $O(E + V \cdot \log V)$ bei Nutzung von Fibonacci-Heaps.

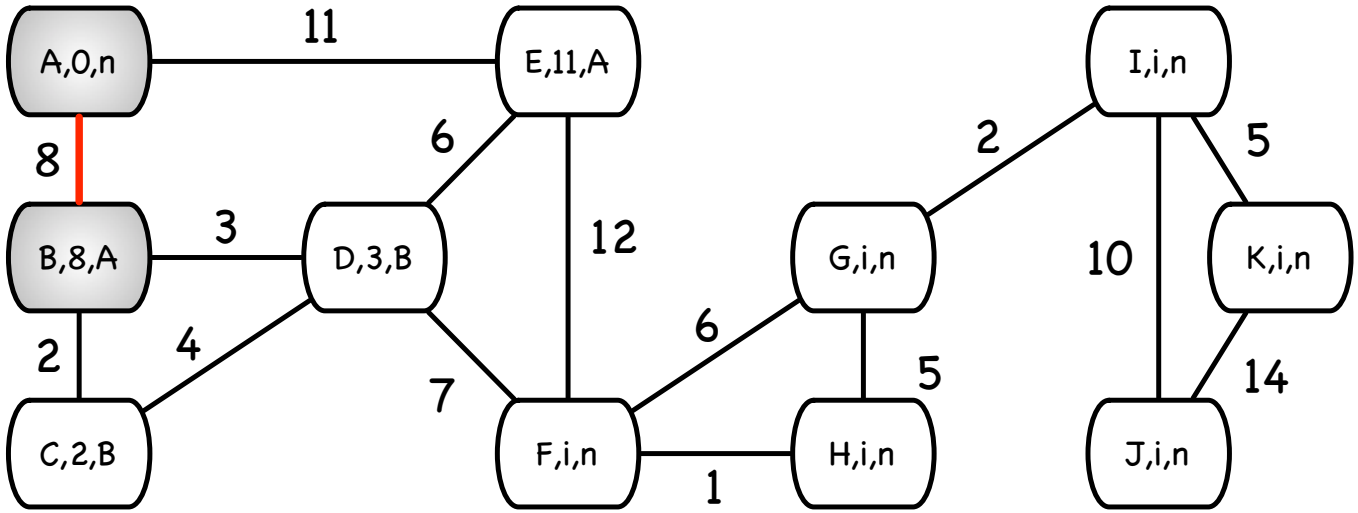
Von Prim zu Dijkstra...

Mit einer kleinen Änderung kann man das Problem der kürzesten Pfade bei einem einzigen Startknoten auf einem gewichteten (gerichteten oder ungerichteten) Graphen lösen...

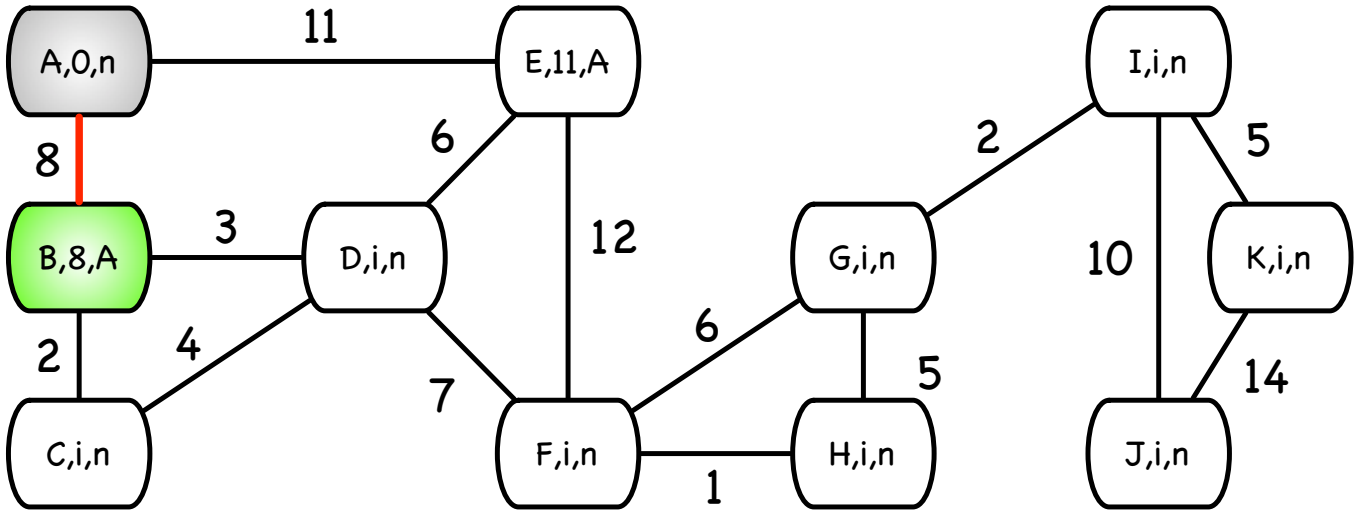
Von Prim zu Dijkstra - Beispiel



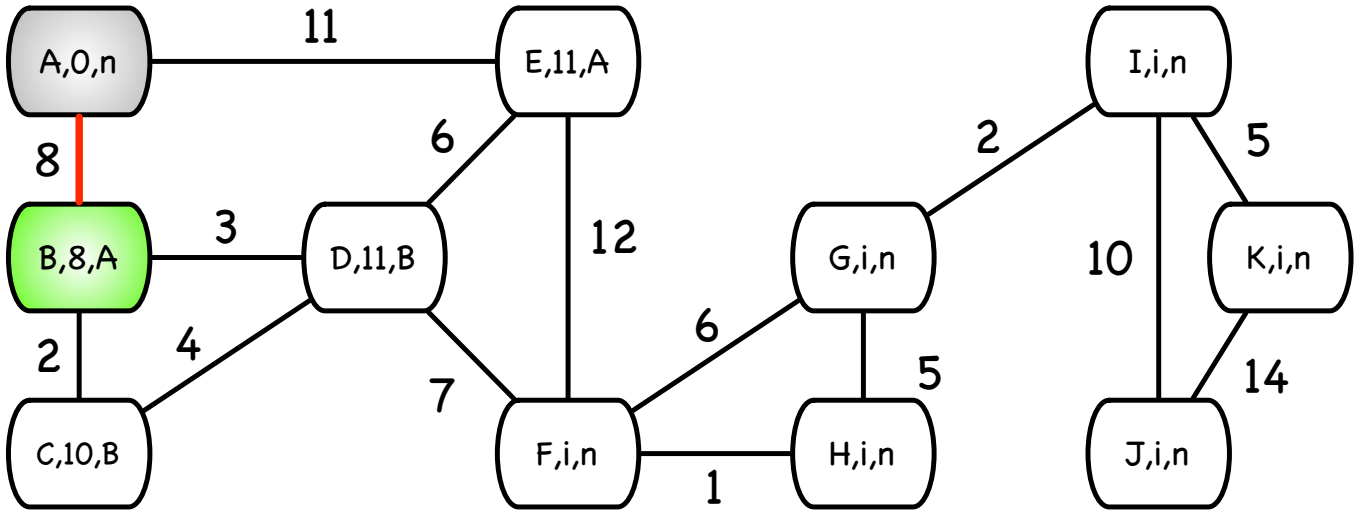
Von Prim zu Dijkstra - Beispiel



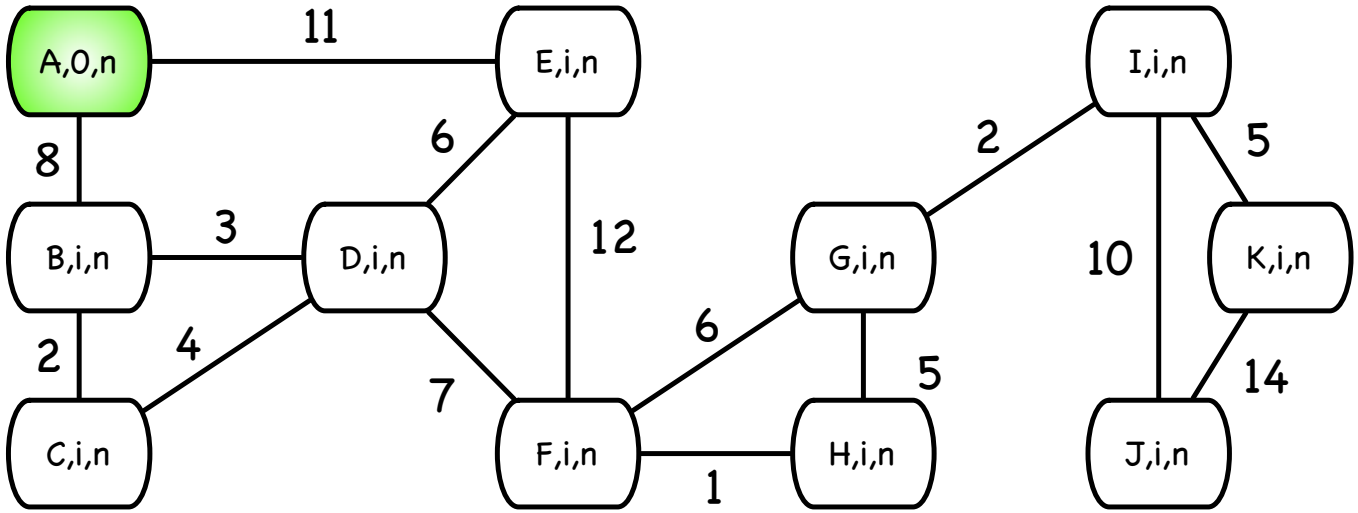
Von Prim zu Dijkstra - Beispiel



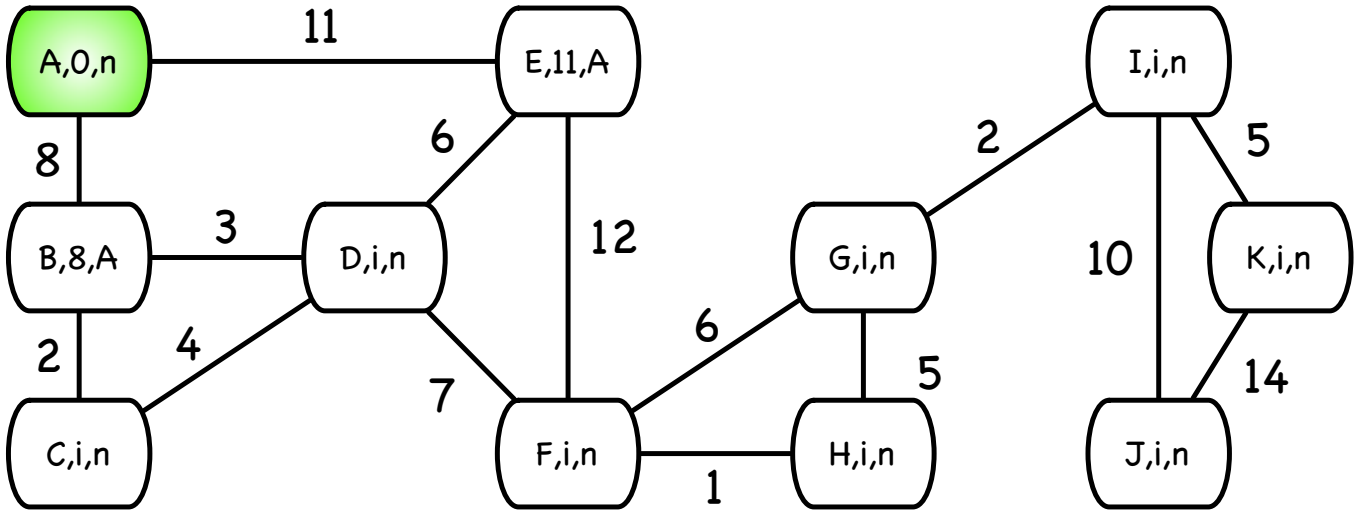
Von Prim zu Dijkstra - Beispiel



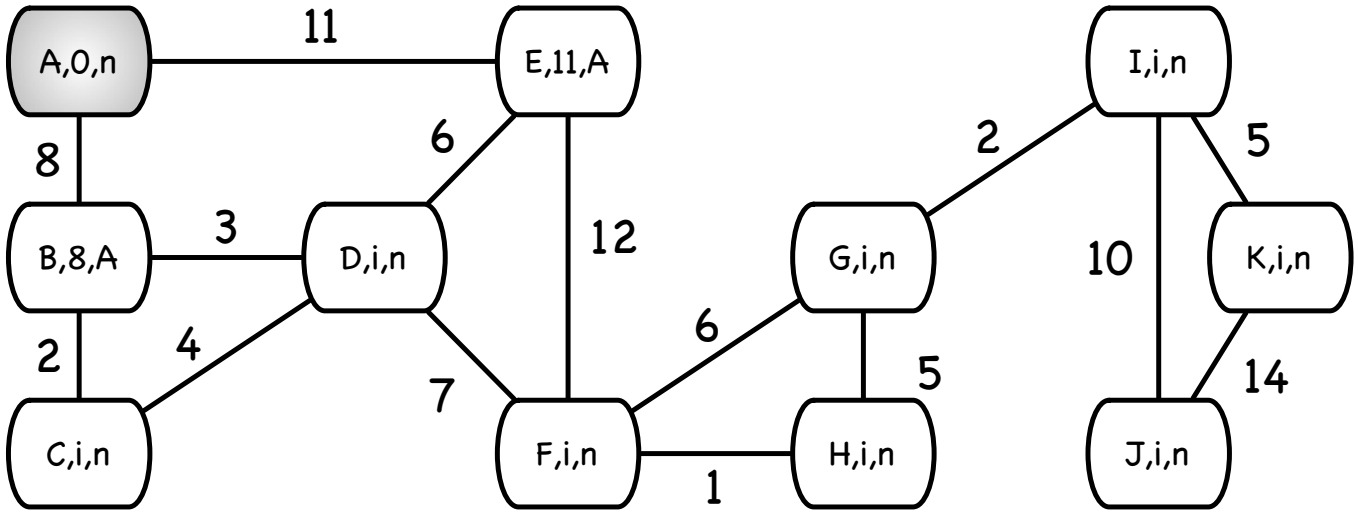
Von Prim zu Dijkstra - Beispiel



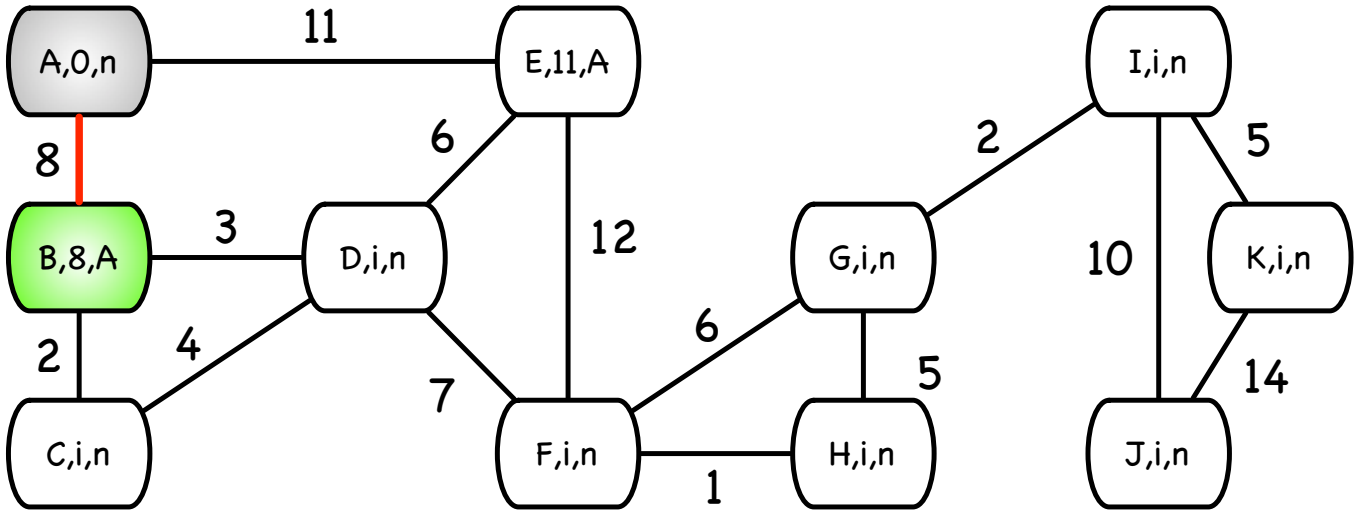
Von Prim zu Dijkstra - Beispiel



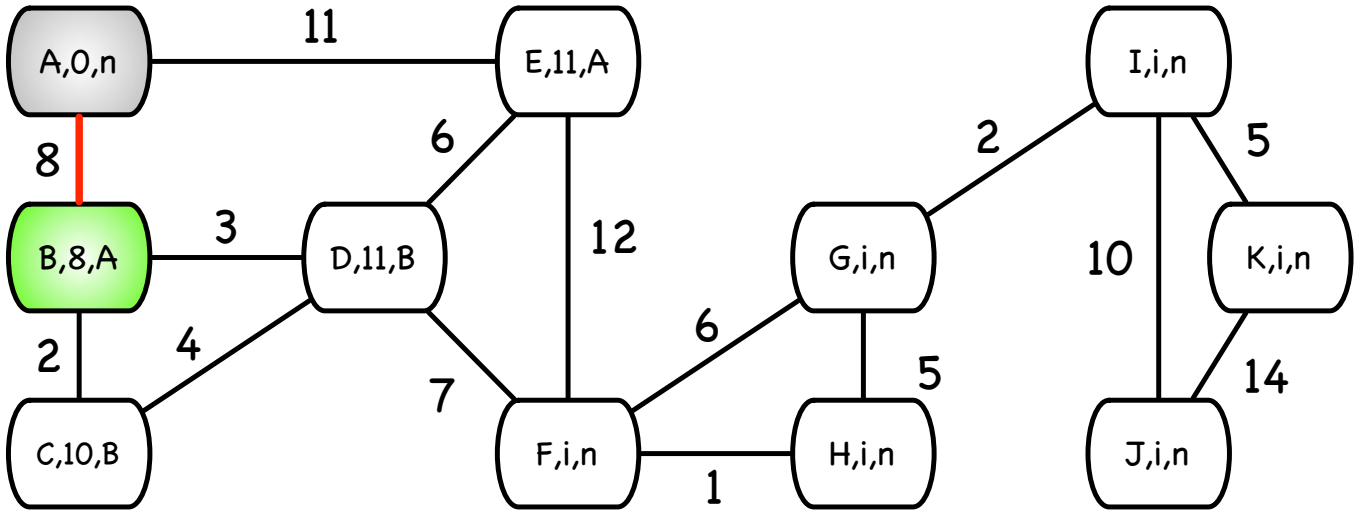
Von Prim zu Dijkstra - Beispiel



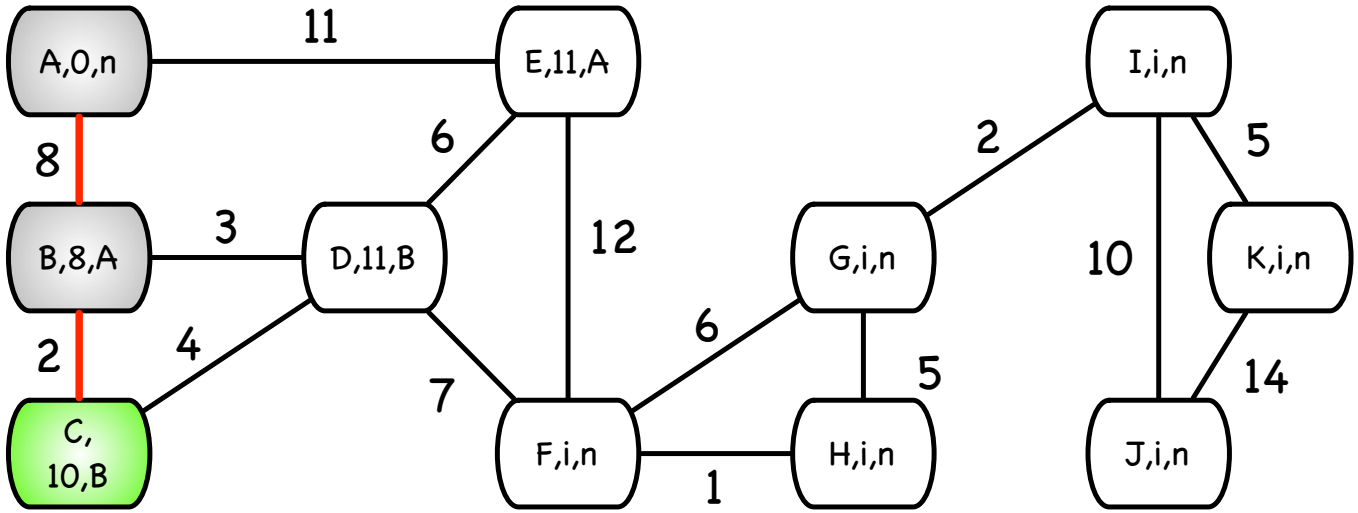
Von Prim zu Dijkstra - Beispiel



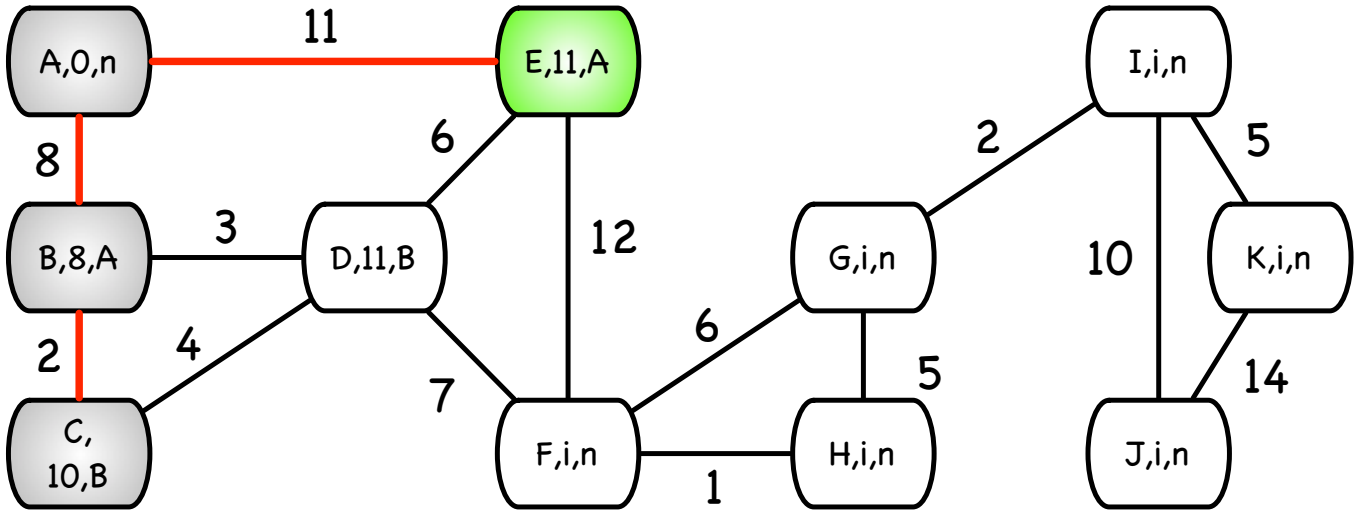
Von Prim zu Dijkstra - Beispiel



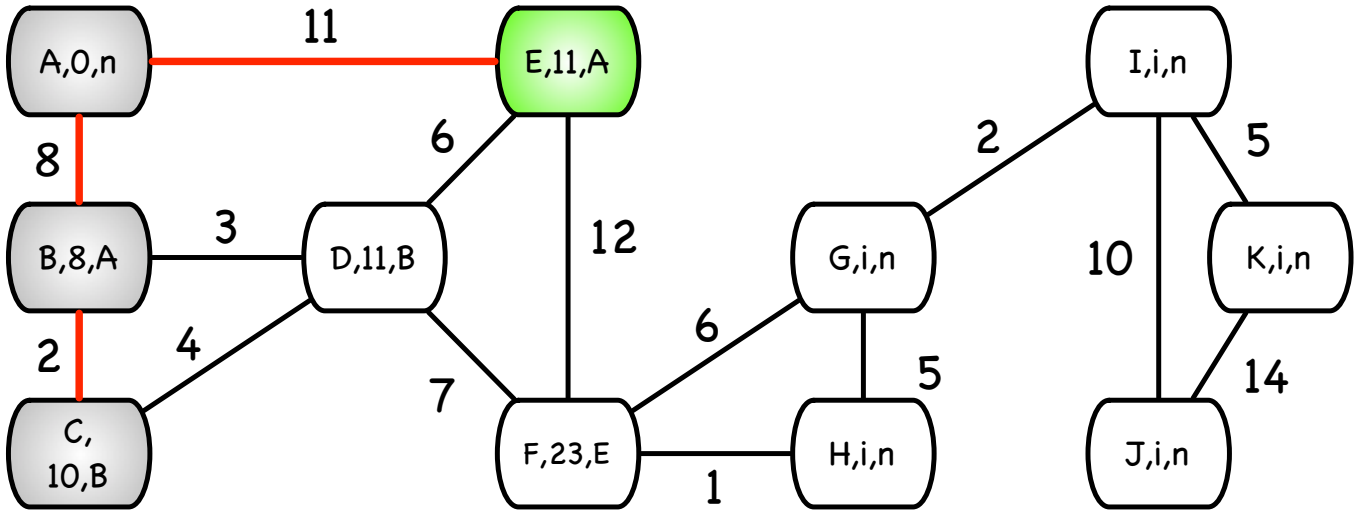
Von Prim zu Dijkstra - Beispiel



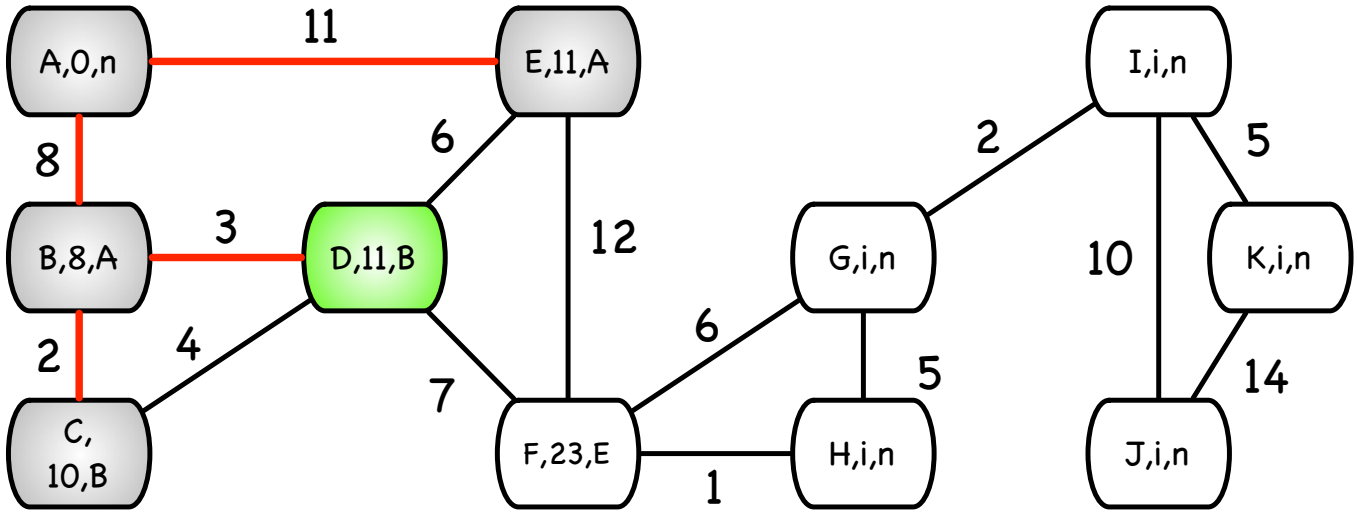
Von Prim zu Dijkstra - Beispiel



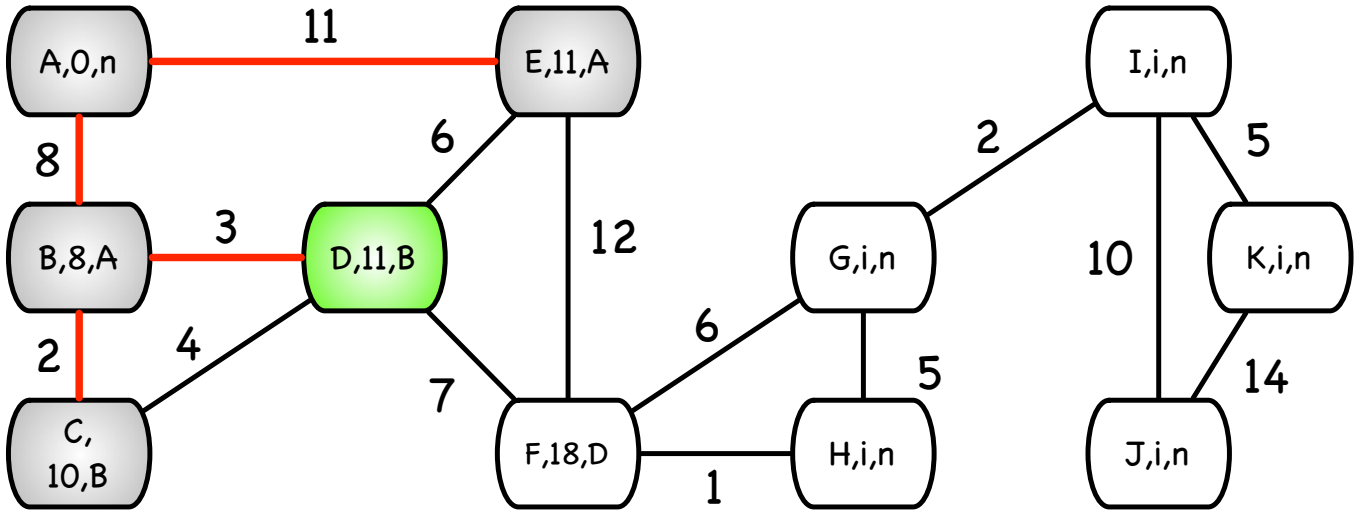
Von Prim zu Dijkstra - Beispiel



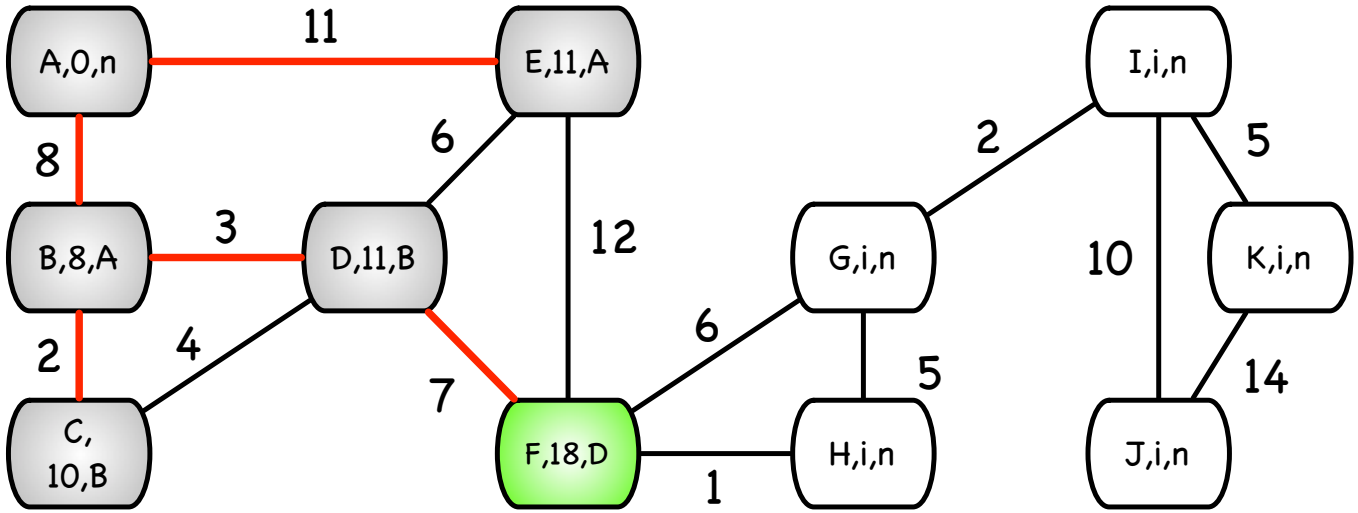
Von Prim zu Dijkstra - Beispiel



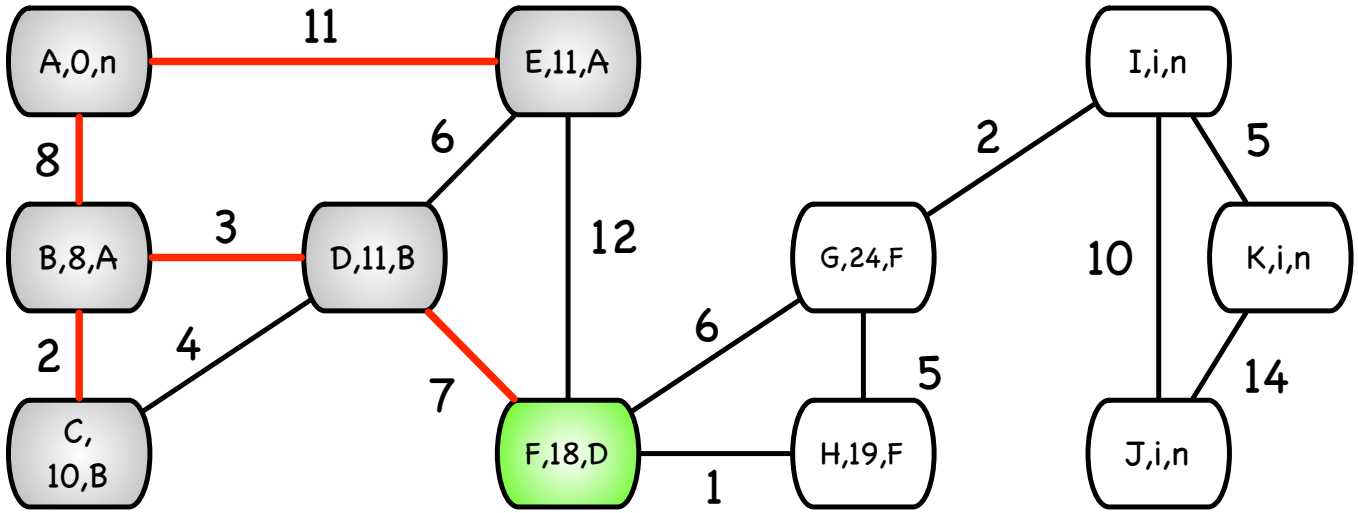
Von Prim zu Dijkstra - Beispiel



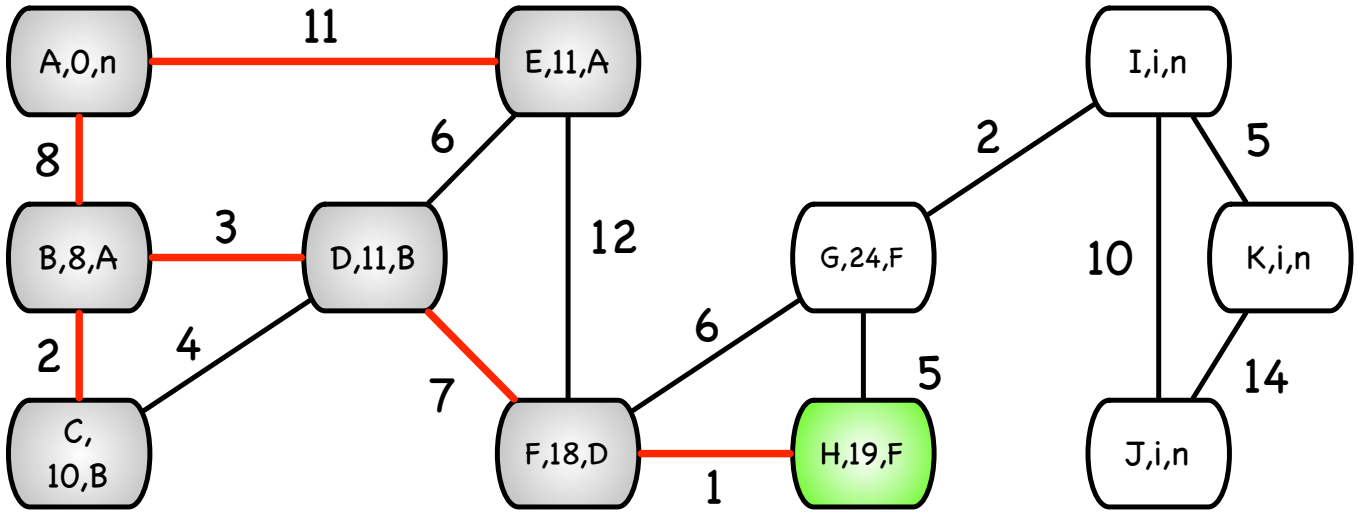
Von Prim zu Dijkstra - Beispiel



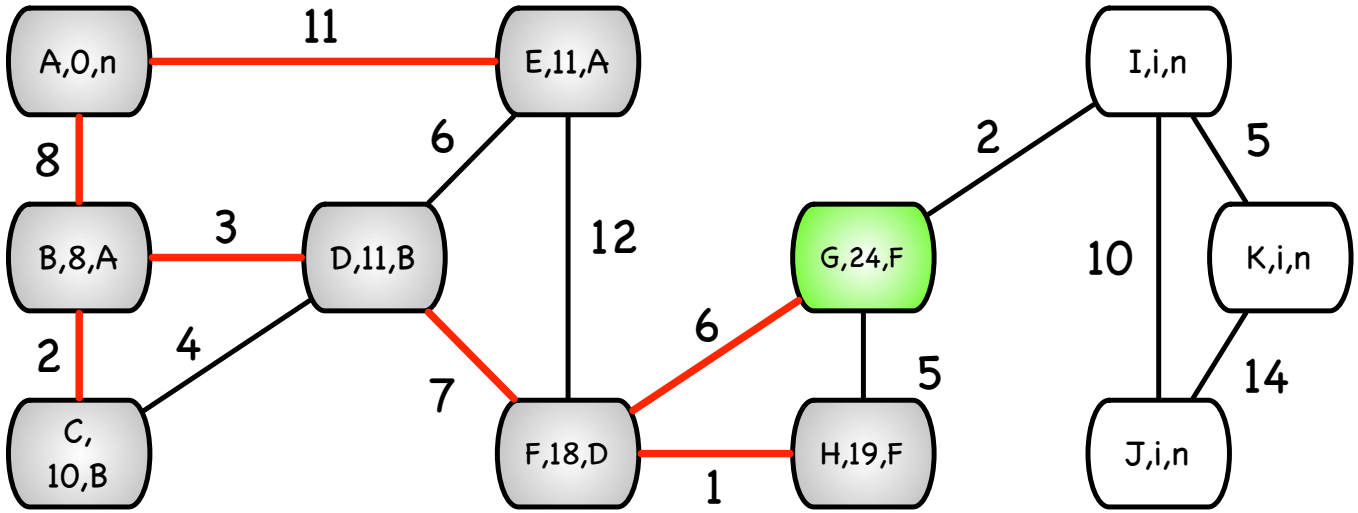
Von Prim zu Dijkstra - Beispiel



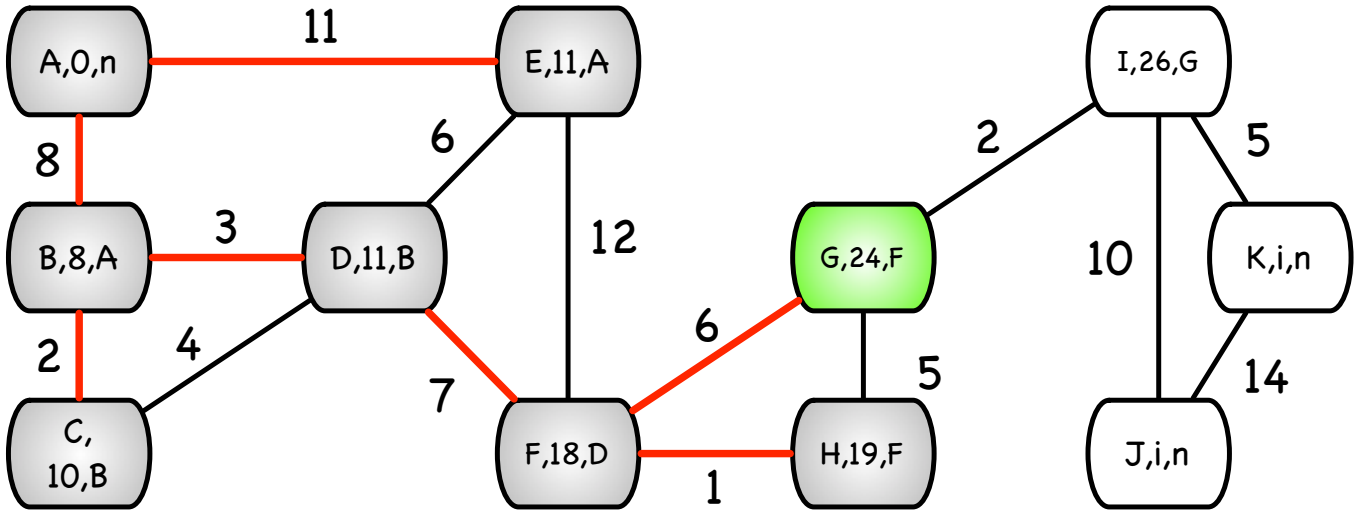
Von Prim zu Dijkstra - Beispiel



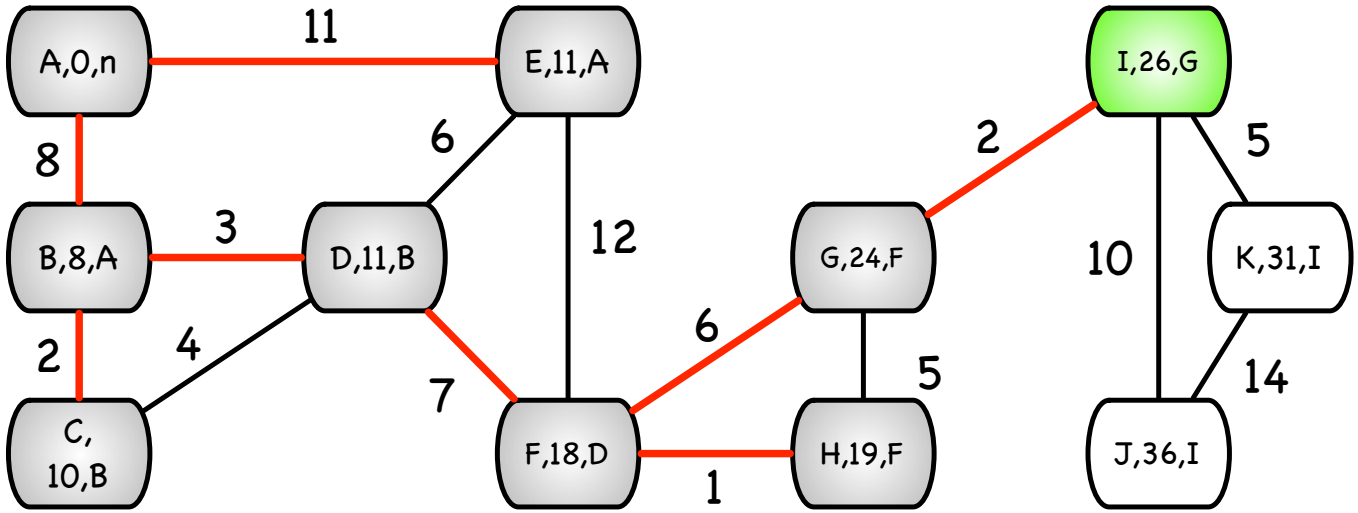
Von Prim zu Dijkstra - Beispiel



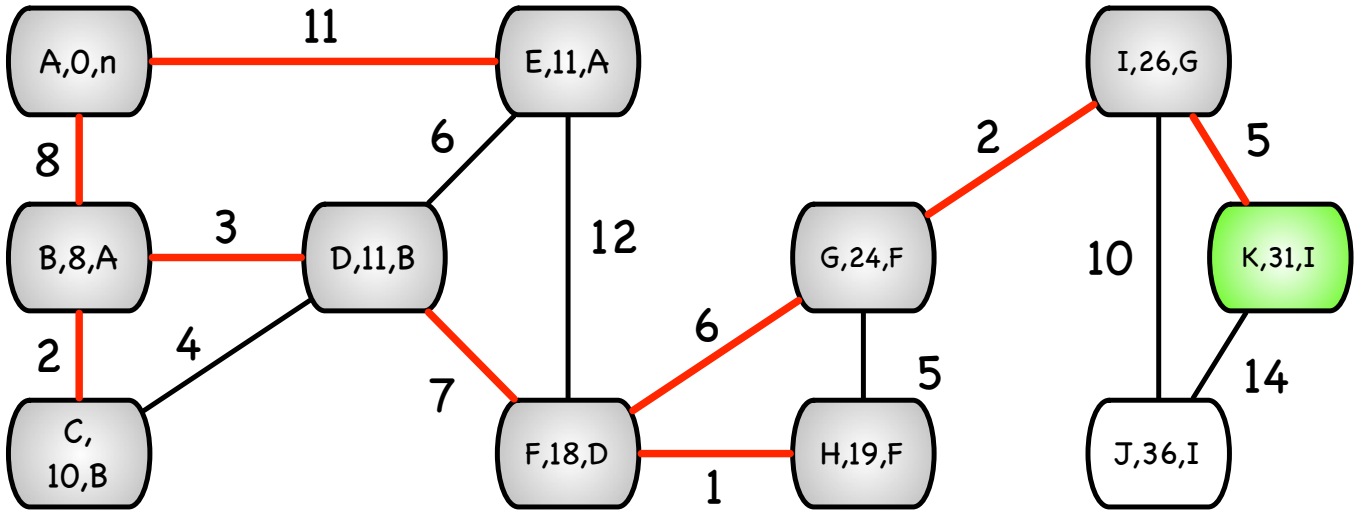
Von Prim zu Dijkstra - Beispiel



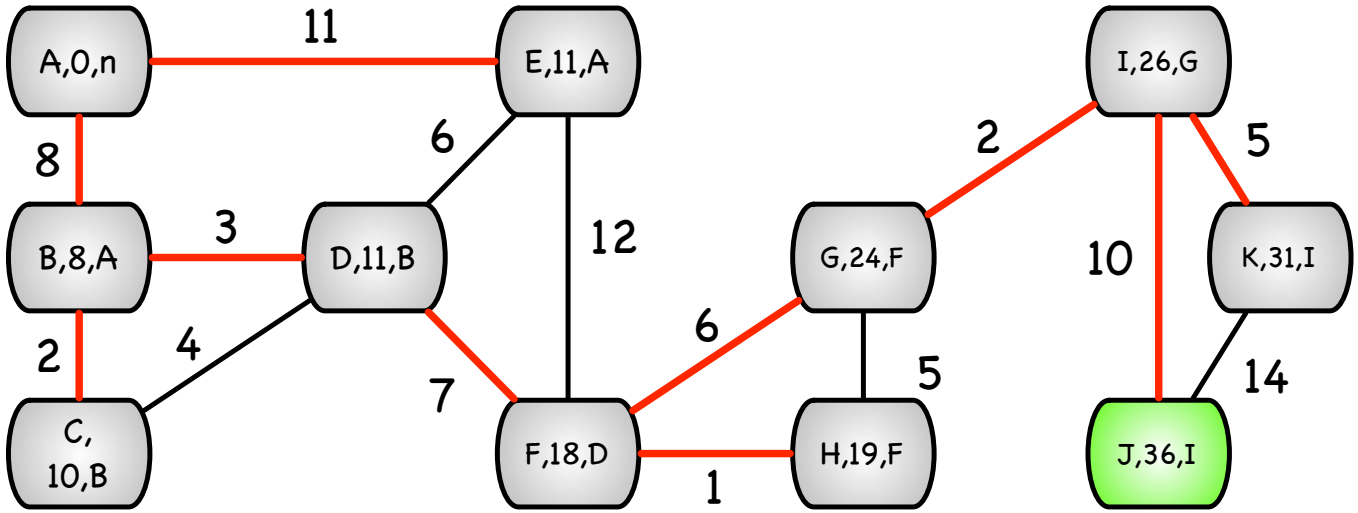
Von Prim zu Dijkstra - Beispiel



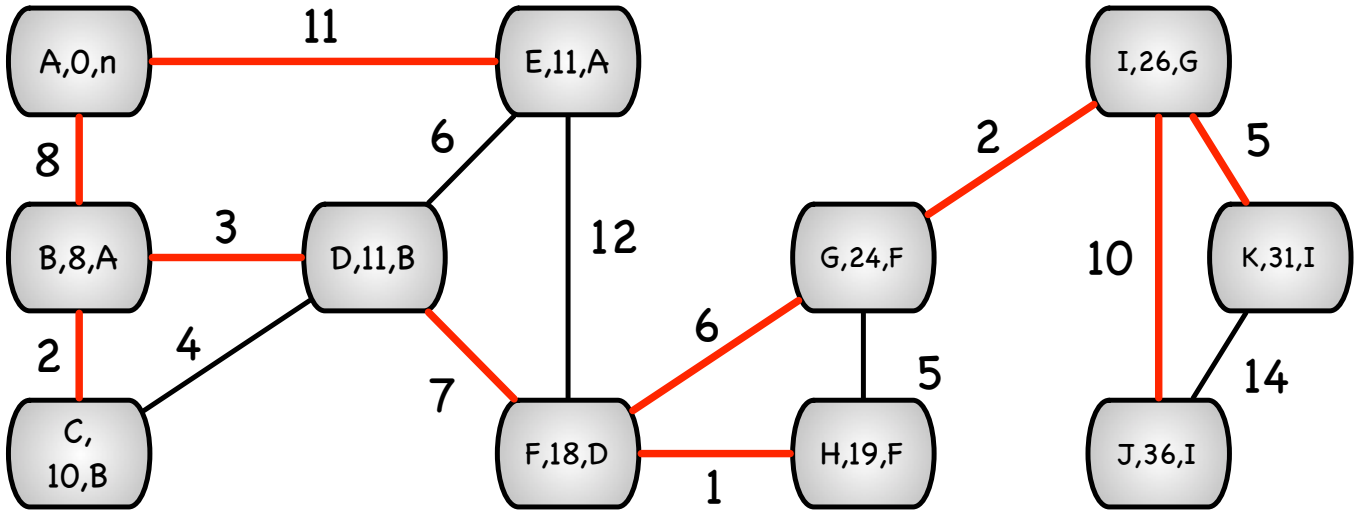
Von Prim zu Dijkstra - Beispiel



Von Prim zu Dijkstra - Beispiel



Von Prim zu Dijkstra - Beispiel



Problemstellung

Definition (Problem der kürzesten Pfade)

Eingabe: Gegeben ein gerichteter Graph $G = (V, E)$, eine Gewichtsfunktion $w : E \to \mathbb{R}^+$ und ein Knoten s.
Gesucht: Für jeden Knoten $v \in V$ ein Pfad P_v, der in s beginnt, in v endet und der unter allen Pfaden mit dieser Eigenschaft das geringste Gewicht hat.

Anmerkung

Das Gewicht eine Pfades $p = [v_0, v_1, \ldots, v_k]$ ist dabei definiert als $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

Varianten

- Variante 1: Für jeden Knoten $v \in V$ suchen wir einen kürzesten Pfad von *einem Startknoten s* aus (*single-source*).
- Variante 2: Finde von allen Knoten einen kürzesten Pfad zu einem gegebenen Zielknoten.
- Variante 3: Finde einen kürzesten Pfad für einen gegebenen Start- und einen gegebenen Zielknoten.
- Variante 4: Finde für jedes Knotenpaar u, v einen kürzesten Pfad von u nach v.

Varianten

- Variante 1: Für jeden Knoten $v \in V$ suchen wir einen kürzesten Pfad von *einem Startknoten s* aus (*single-source*).
- Variante 2: Finde von allen Knoten einen kürzesten Pfad zu einem gegebenen Zielknoten.

Anmerkung

Dreht man die Richtungen aller Kanten um, kann man 2 auf 1 zurückführen.

Varianten

- Variante 1: Für jeden Knoten $v \in V$ suchen wir einen kürzesten Pfad von *einem Startknoten s* aus (*single-source*).
- Variante 3: Finde einen kürzesten Pfad für einen gegebenen Start- und einen gegebenen Zielknoten.

Anmerkung

1 löst 3 mit und es ist kein Verfahren für 3 bekannt, das im schlechtesten Fall wirklich asymptotisch schneller wäre (als der beste single-source-Algorithmus).

Varianten

- Variante 1: Für jeden Knoten $v \in V$ suchen wir einen kürzesten Pfad von *einem Startknoten* s aus (*single-source*).
- Variante 4: Finde für jedes Knotenpaar u, v einen kürzesten Pfad von u nach v.

Anmerkung

4 geht im Prinzip, indem man 1 für jeden Knoten löst, aber dies geht schneller ($\Rightarrow$ Floyd-Warshall-Algorithmus).

Wichtige Eigenschaft kürzester Pfade

Satz

Sei $G = (V, E)$ gewichteter, gerichteter Graph. $w : E \to \mathbb{R}^+$ eine Gewichtsfunktion. Sei $p = [v_1, v_2, \ldots, v_k]$ ein kürzester Pfad von v_1 nach v_k und p_{ij} der Teilpfad von v_i nach v_j $(1 \leq i \leq j \leq k)$. Dann ist p_{ij} ein kürzester Pfad von v_i nach v_j.

Beweis.

Beweisidee: Nehmen wir an dies wäre nicht so, dann gäbe es einen kürzeren Pfad p'_{ij} von v_i nach v_j. Ersetzt man p_{ij} in p durch diesen, erhält man einen kürzeren v_1-v_k-Pfad im Widerspruch dazu, dass p ein kürzester Pfad ist. □

Negative Gewichte und Zyklen

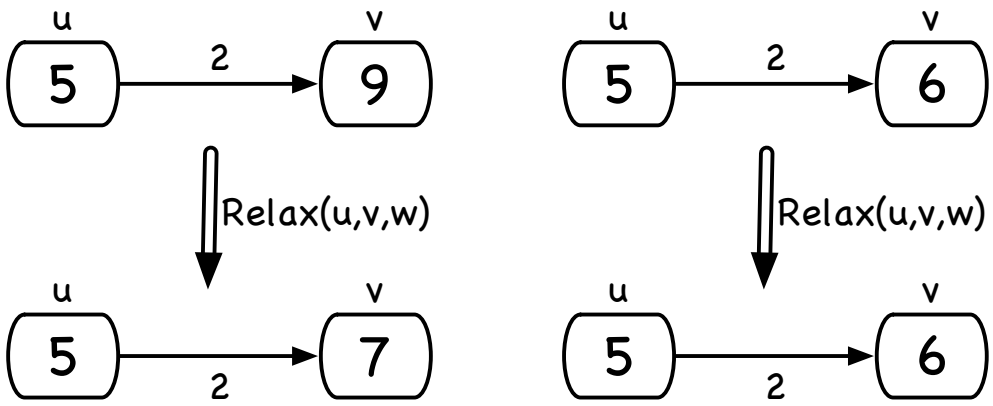
- Kanten mit negativen Gewichten sind möglich, auch wenn nicht alle Algorithmen damit arbeiten können. Negative Zyklen verhindern aber kürzeste Pfade!
- Der gleich folgenden Algorithmus von Dijkstra erlaubt keine negativen Gewichte.
- Der Bellman-Ford-Algorithmus erlaubt negative Gewichte und kann negative Zyklen entdecken.
- Normale Zyklen können vorkommen, werden aber bei einem kürzesten Pfad nie gebraucht (im Gegenteil!).
- Die Darstellung/Speicherung kürzester Pfade geschieht wieder mit der Vorgängerfunktion $\pi[v]$.

Zwei wichtige Hilfsfunktionen

Algorithmus 6 InitSingleSource(G, s)

1: **for** alle $v \in V(G)$ **do**
2: $\quad d[v] = \infty$
3: $\quad \pi[v] = \texttt{nil}$
4: **end for**
5: $d[s] = 0$

Zwei wichtige Hilfsfunktionen



Algorithmus 7 Relax(u, v, w)

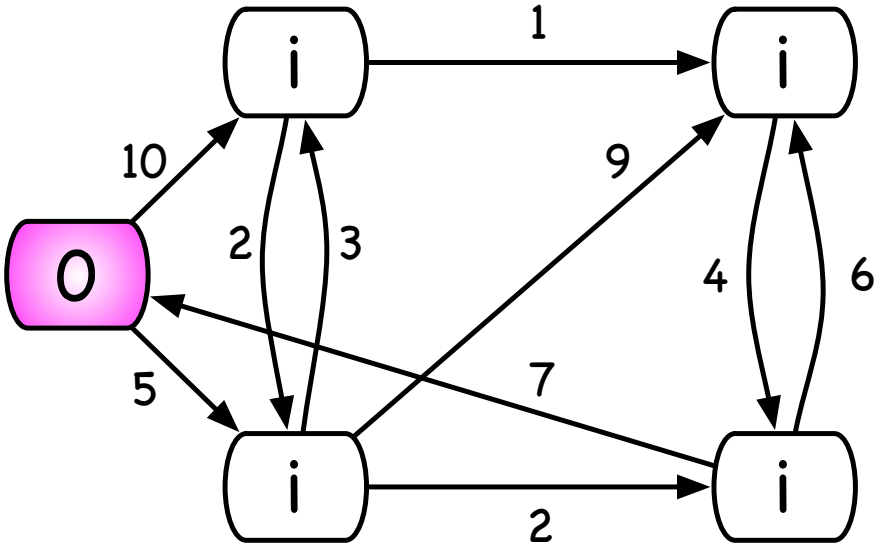
1: **if** $d[v] > d[u] + w(u, v)$ **then**
2: $\quad d[v] = d[u] + w(u, v)$
3: $\quad \pi[v] = u$
4: **end if**

Der Dijkstra-Algorithmus

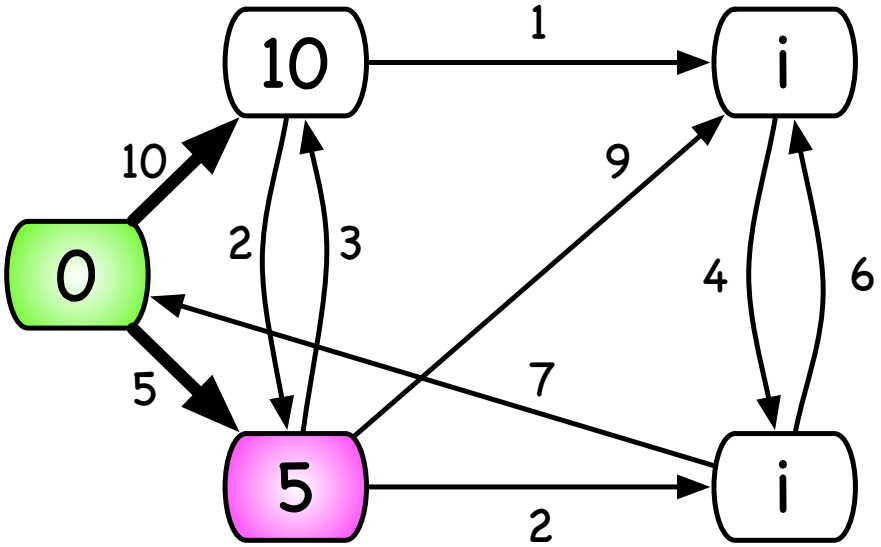
Algorithmus 8 Dijkstra(G, w, s)

```
1:  InitSingleSource(G, s)
2:  S = ∅
3:  Q = V(G)
4:  while Q ≠ ∅ do
5:      u = ExtractMin(Q)
6:      S = S ∪ {u}
7:      for alle v ∈ Adj[u] do
8:          if d[v] > d[u] + w(u, v) then
9:              d[v] = d[u] + w(u, v)
10:             π[v] = u
11:         end if
12:     end for
13: end while
```

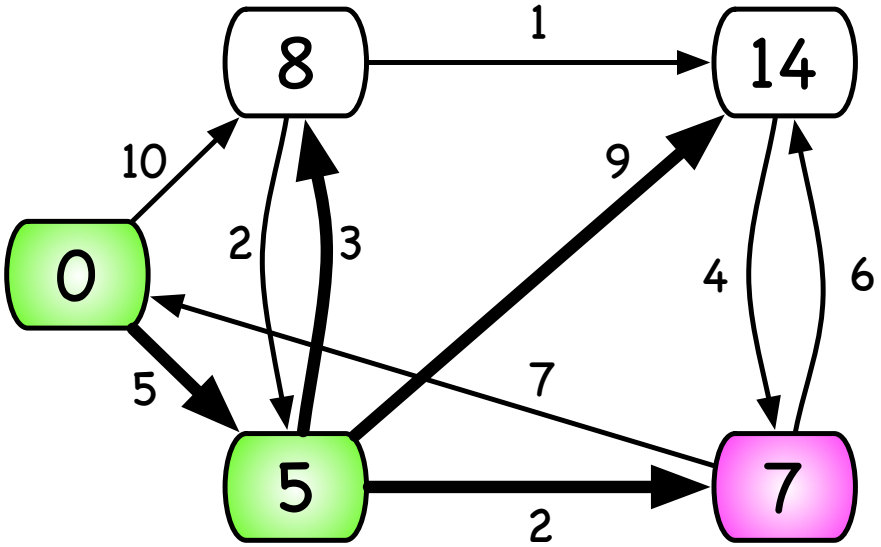
Beispiel



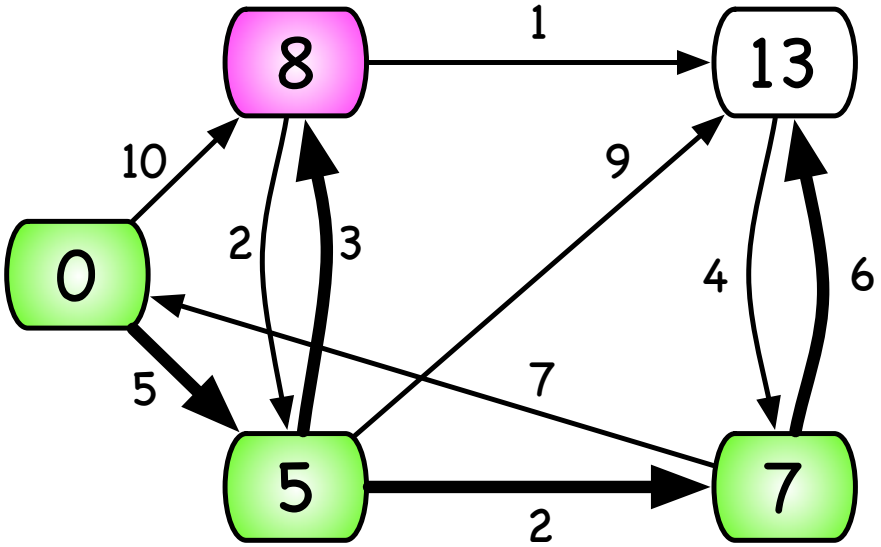
Beispiel



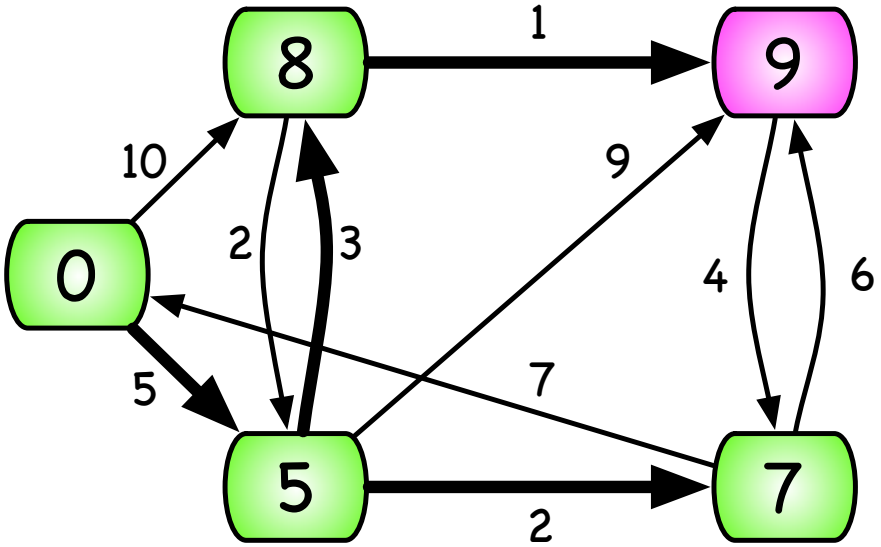
Beispiel



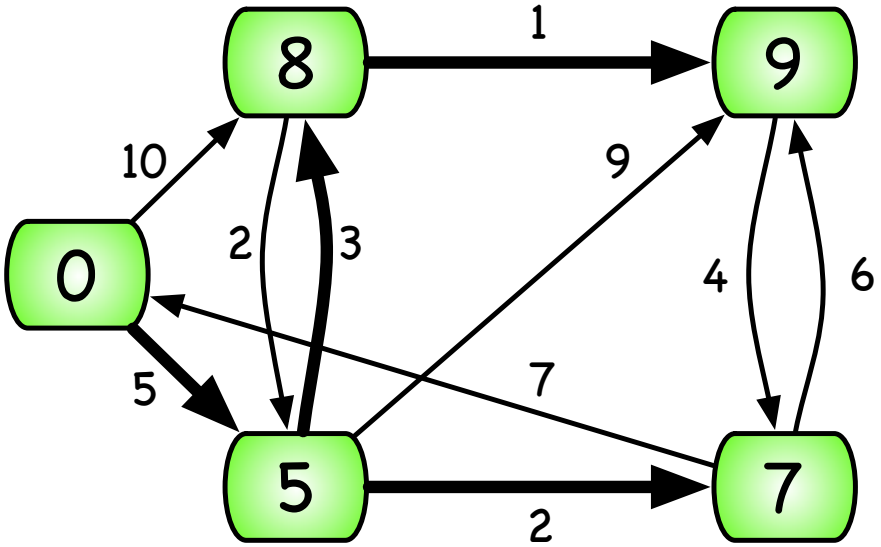
Beispiel



Beispiel



Beispiel



Einführung

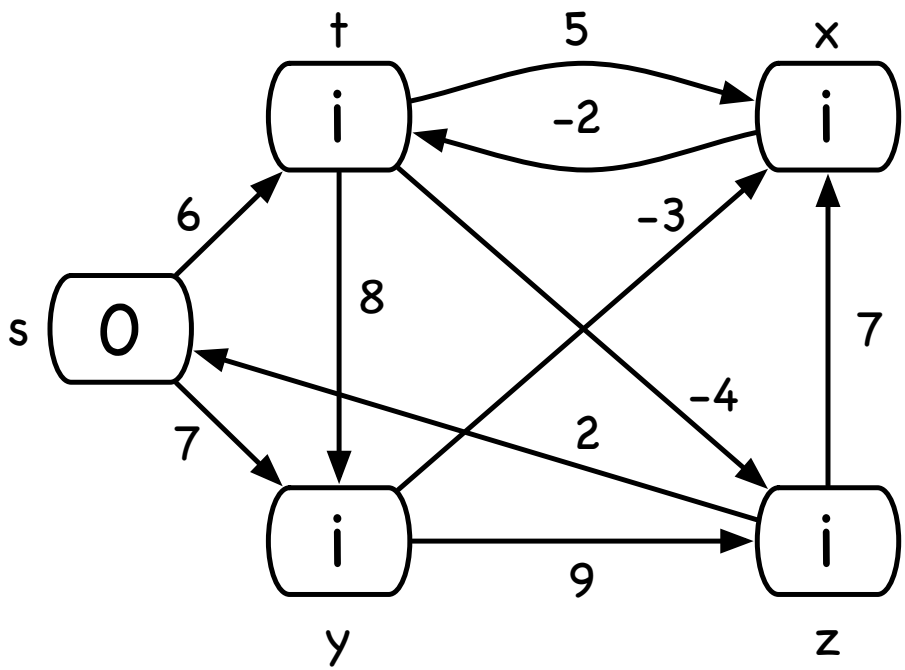
Der Bellman-Ford-Algorithmus löst wie Dijkstra das Problem der kürzesten Pfade bei einem Startknoten. Anders als Dijkstra dürfen Kantengewichte aber auch negativ sein. Der Algorithmus liefert `false` zurück, sollte ein Zyklus mit negativem Gewicht existieren.

Der Bellman-Ford-Algorithmus

Algorithmus 9 BellmanFord(G, w, s)

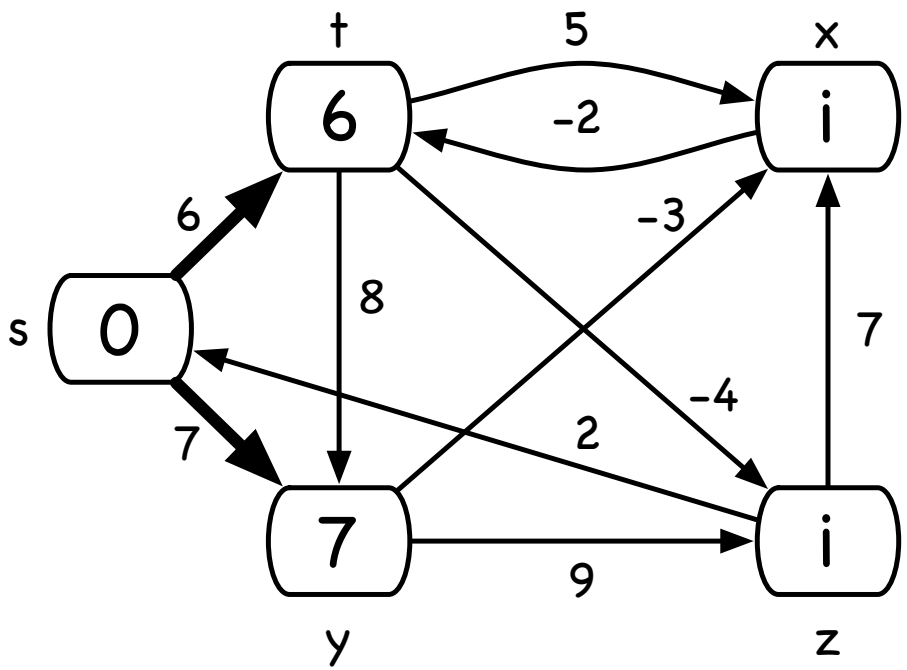
```
 1: InitSingleSource(G, s)
 2: for i = 1 to |V(G)| − 1 do
 3:     for alle (u, v) ∈ E(G) do
 4:         Relax(u, v, w)
 5:     end for
 6: end for
 7: for alle (u, v) ∈ E(G) do
 8:     if d[v] > d[u] + w(u, v) then
 9:         return false
10:     end if
11: end for
```

Beispiel



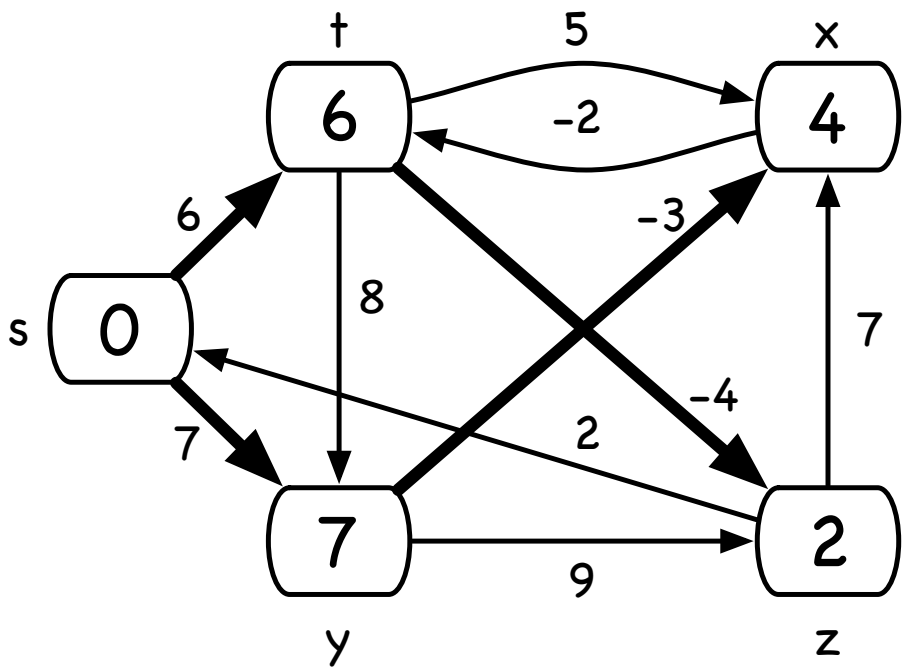
Kantendurchlauf:
$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$

Beispiel



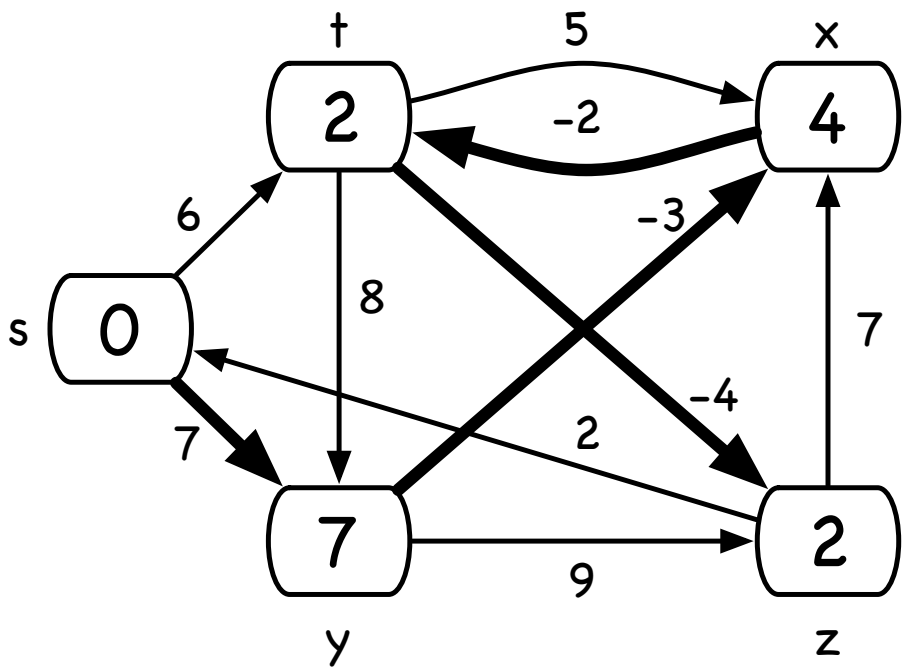
Kantendurchlauf:
$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$

Beispiel



Kantendurchlauf:
$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$

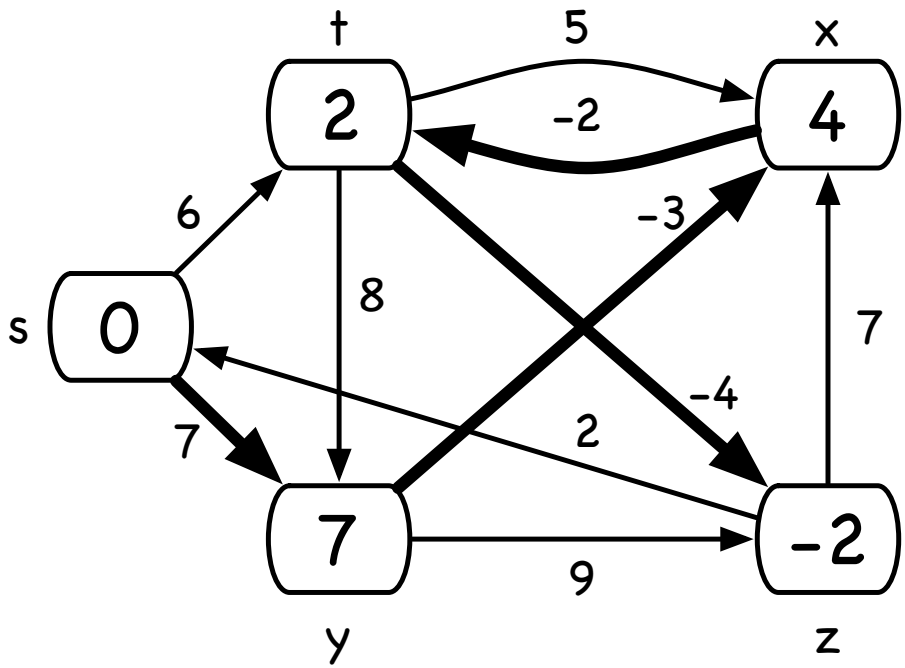
Beispiel



Kantendurchlauf:

$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$

Beispiel



Kantendurchlauf:
$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$

Anmerkungen zum Beispiel

Anmerkung

- Wir hatten im Beispiel vier (also $|V(G)| - 1$) Iterationen. Im Test der unteren for-Schleife finden wir nun nichts (keine Kante führt zu einer weiteren Relaxtion), also sind wir fertig.
- Wäre die Kante von z nach s mit z.B. 1 gewichtet, dann würde der Test in der untern for-Schleife nun false zurückliefern. Dies bedeutet auch, dass ein negativer Zyklus (von s nach s) gefunden wurde.
- Oben sind nur $|V(G)| - 1$ Iterationen nötig, da ansonsten schon maximale Zyklen (Zyklen, deren Länge der Anzahl der Knoten entspricht) gefunden werden.
- Laufzeit: $O(V \cdot E)$.

Einführung

Der Floyd-Warshall-Algorithmus löst das Problem in der vierten Variante, d.h. er bestimmt zu jedem Paar zweier Knoten einen kürzesten Pfad zwischen ihnen. Kanten mit negativem Gewicht sind zugelassen. Der nachfolgende Algorithmus arbeitet aber nicht, sollten auch Zyklen mit negativem Gewicht vorhanden sein!

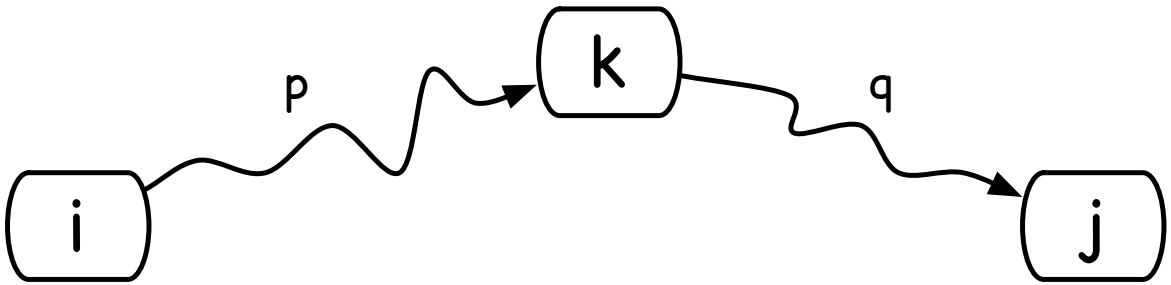
Wir wollen dies nachfolgend nur kurz anreißen...

Anmerkung (für Interessierte)

Die Idee des Algorithmus basiert auf einer ähnlichen Idee wie die Vorschrift, bei der Konstruktion eines regulären Ausdrucks zu einem Automaten (Kleene-Konstruktion, R_{ij}^k).

Zur Illustration der Idee

$$d_{ij}^{k} = \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$$



Die Pfade p und q haben nur Zwischenknoten aus $\{1, 2, \ldots, k-1\}$. Der ganze Pfad von i nach j enthält auch einmal den Knoten aus k.

Der Floyd-Warshall-Algorithmus

Algorithmus 10 FloydWarhsall(W)

1: $n = \text{zeilen}[W]$
2: $D^0 = W$
3: **for** $k = 1$ to n **do**
4: **for** $i = 1$ to n **do**
5: **for** $j = 1$ to n **do**
6: $d_{ij}^k = \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$
7: **end for**
8: **end for**
9: **end for**

Kürzeste Pfade - Zusammenfassung

- Dijkstra. Für gewichtete, gerichtete Graphen, *ohne* negative Gewichte. Laufzeit: $O(V^2)$ und sogar $O(V \cdot \log V + E)$.
- Bellman-Ford(-Moore). Für gewichtete, gerichtete Graphen. Negative Gewichte sind erlaubt. Laufzeit: $O(V \cdot E)$.
- Floyd-Warshall. Für gewichtete, gerichtete Graphen. Negative Gewichte sind erlaubt. Laufzeit: $O(V^3)$. Ermittelt anders als die obigen beiden, kürzeste Pfade *für alle Knotenpaare*!

Anmerkung

Alle obigen Verfahren gehen auch mit ungerichteten Graphen (da eine ungerichtete Kante durch zwei gerichtete Kanten ersetzt werden kann), sowie für ungewichtete Graphen (allen Kanten Gewicht 1 geben).

Zusammenfassung und Ausblick

- Graphen
  - Grundlagen, Adjazenzmatrix, Adjazenzliste
  - Breiten- & Tiefensuche
  - Anwendung der Tiefensuche
    - Topologisches Sortieren
    - Bestimmung starker Zusammenhangskomponenten
  - Minimale Spannbäume
    - Definition
    - Algorithmus von Kruskal
    - Algorithmus von Prim
  - Bestimmen der kürzesten Pfade
    - Definition und Varianten
    - Algorithmus von Dijkstra
    - Algorithmus von Bellman-Ford
    - Algorithmus von Floyd-Warshall
- Nächstes Mal:
  - Flüße

Literaturhinweis

Literatur

Die letzte und die heutige Vorlesung behandeln Teile der Kapitel 22 (Elementare Graphalgorithmen), 23 (Minimale Spannbäume), 24 (Das Problem der kürzesten Pfade bei einem einzigen Startknoten) und 25 (Das Problem der kürzesten Pfade für alle Kontenpaare) aus dem [Cormen].
Insb. ist in Kapitel 24 der Beweis der Korrektheit von Dijkstras-Algorithmus zu finden.

Problemstellung

Definition (Bestimmung eines minimalen Spannbaums)

Eingabe: Gegeben ein **ungerichteter** Graph $G = (V, E)$ und eine Gewichtsfunktion $w : E \to \mathbb{R}^+$.
Gesucht: Ein Spannbaum T derart, dass

$$w(T) = \sum_{(u,v) \in E(T)} w(u, v)$$

minimal ist. Mit $E(T)$ werden dabei die Kanten von T bezeichnet. T wird minimaler Spannbaum oder MST genannt.

Ein Spannbaum von G ist ein zusammenhängender und azyklischer Teilgraph (also ein Baum) von G, der alle Knoten von G enthält.

Kruskal und Prim

Sei A die bisher gewählte Menge von Kanten für den MST. Die Algorithmen von Kruskal und Prim arbeiten wie folgt:

- Kruskal: Nimm unter allen Kanten (aus $E - A$) jene mit dem kleinsten Gewicht, deren Hinzufügen nicht zu einem Kreis führt.
- Prim: Wähle aus den Kanten eine minimale aus, die A mit einem noch isolierten Knoten von G_A verbindet.

Korrektheit ist kompliziert zu zeigen. Bei Prim muss zusätzlich Aufwand betrieben werden, um die Implementation als Korrekt nachzuweisen.

Zur Laufzeit: Beide sind in $O(E \cdot \log V)$. Prim ist sogar in $O(E + V \cdot \log V)$ möglich.

Problemstellung

Definition (Problem der kürzesten Pfade)

Eingabe: Gegeben ein gerichteter Graph $G = (V, E)$, eine Gewichtsfunktion $w : E \to \mathbb{R}^+$ und ein Knoten s.
Gesucht: Für jeden Knoten $v \in V$ ein Pfad P_v, der in s beginnt, in v endet und der unter allen Pfaden mit dieser Eigenschaft das geringste Gewicht hat.

Anmerkung

Das Gewicht eine Pfades $p = [v_0, v_1, \ldots, v_k]$ ist dabei definiert als $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

Kürzeste Pfade - Zusammenfassung

- Dijkstra. Für gewichtete, gerichtete Graphen, *ohne* negative Gewichte. Laufzeit: $O(V^2)$ und sogar $O(V \cdot \log V + E)$.
- Bellman-Ford(-Moore). Für gewichtete, gerichtete Graphen. Negative Gewichte sind erlaubt. Laufzeit: $O(V \cdot E)$.
- Floyd-Warshall. Für gewichtete, gerichtete Graphen. Negative Gewichte sind erlaubt. Laufzeit: $O(V^3)$. Ermittelt anders als die obigen beiden kürzeste Pfade *für alle Knotenpaare*!

Anmerkung

Alle obigen Verfahren gehen auch mit ungerichteten Graphen (da eine ungerichtete Kante durch zwei gerichtete Kanten ersetzt werden kann), sowie für ungewichtete Graphen (allen Kanten Gewicht 1 geben).