

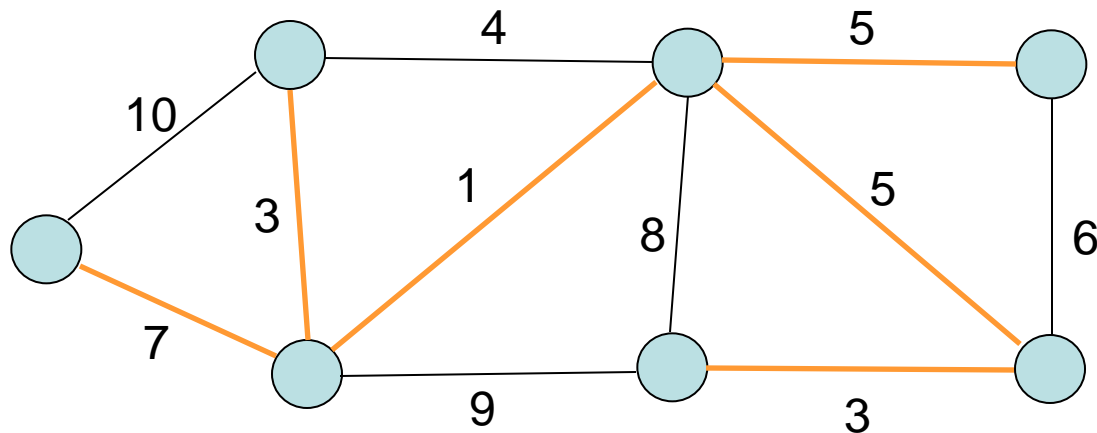


## Datenstrukturen, Algorithmen und Programmierung 2 (DAP2)

## Graphalgorithmen

### Minimale Spannbäume

- Gegeben: Gewichteter, ungerichteter, zusammenhängender Graph  $G=(V,E)$
- Gesucht: Ein aufspannender Baum mit minimalem Gewicht
- Aufspannender Baum: Inklusionsmaximaler kreisfreier Teilgraph mit Knotenmenge  $V$



## Graphalgorithmen

### *Berechnung von minimalen Spannäumen*

- Gieriger Algorithmus
- Invariante: Menge A von Kanten, die immer Untermenge eines minimalen Spannbaums ist
- Algorithmus: Finde Kante, die zu A hinzugefügt werden kann, ohne dass Invariante verletzt wird

### *Definition*

Eine Kante, die zu A hinzugefügt werden kann, ohne die Invariante zu verletzen, heißt **sicher**.

## Graphalgorithmen

### GenerischerMSTAlgorithmus(G)

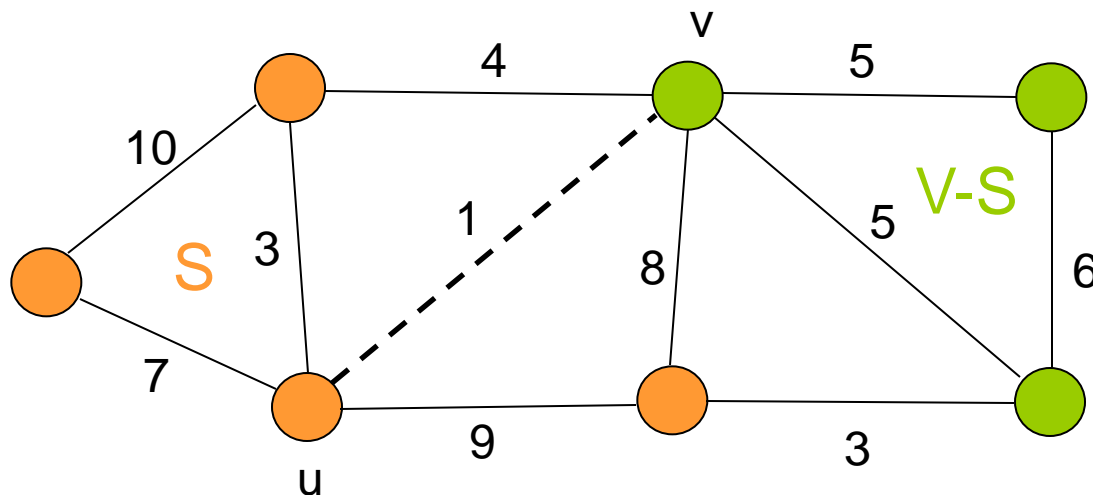
1.  $A \leftarrow \emptyset$
2. **while** A ist kein minimaler Spannbaum **do**
3.     finde Kante  $(u,v)$ , die sicher für A ist
4.      $A \leftarrow A \cup \{(u,v)\}$
5. **return** A

Wie findet man  
sichere Kante?

## Graphalgorithmen

### Definition

- Ein **Schnitt**  $(S, V-S)$  in einem ungerichteten Graph  $G=(V,E)$  ist eine Partition von  $V$ .
- Eine Kante  $(u,v) \in E$  **kreuzt** den Schnitt  $(S, V-S)$ , wenn  $u \in S$  und  $v \in V-S$  liegt.

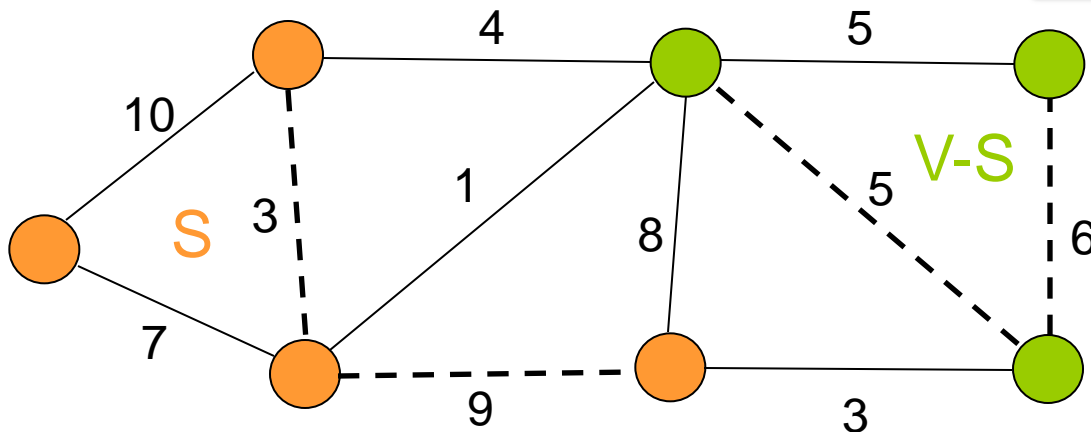


## Graphalgorithmen

### Definition

- Ein Schnitt **respektiert** eine Menge A von Kanten, wenn keine der Kanten den Schnitt kreuzt.

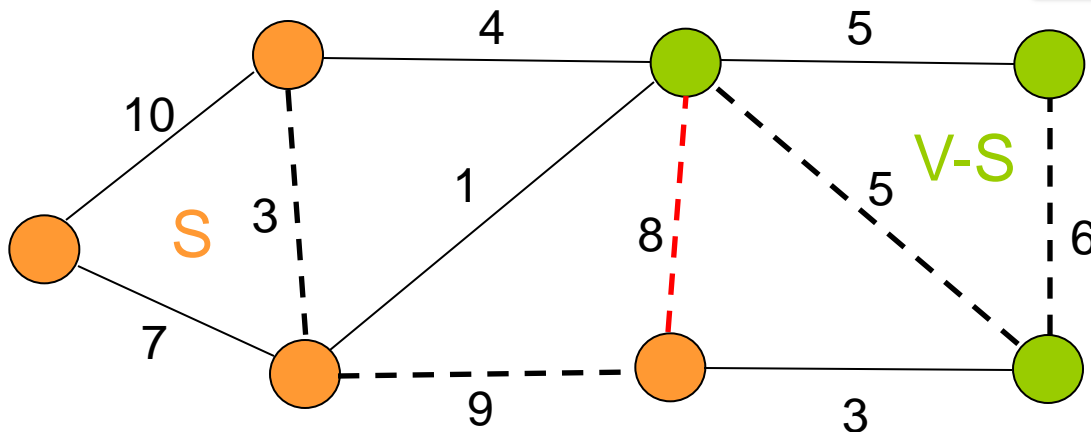
Beispiel:  
Der Schnitt respektiert die gestrichelten Kanten.



## Graphalgorithmen

### Definition

- Ein Schnitt **respektiert** eine Menge A von Kanten, wenn keine der Kanten den Schnitt kreuzt.

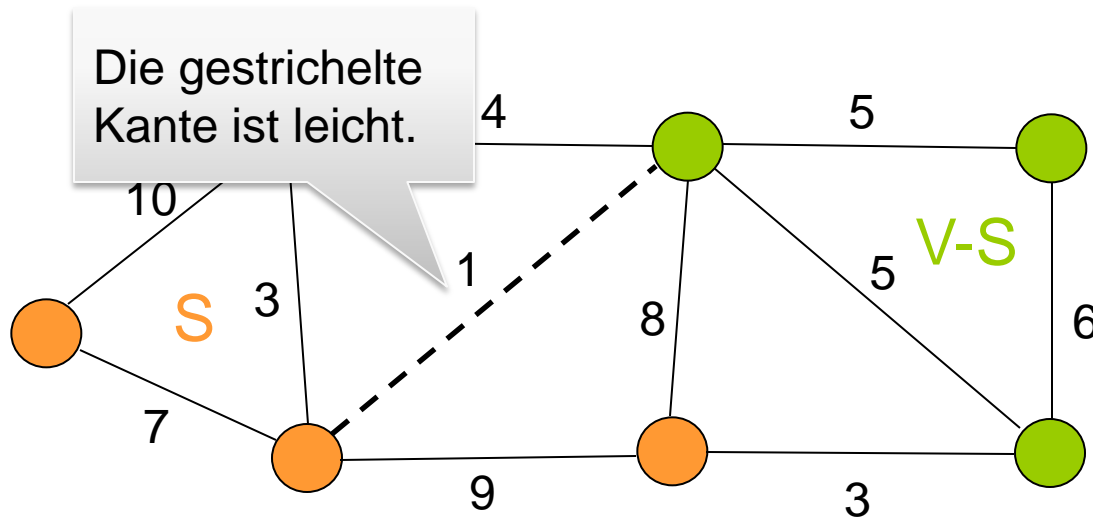


Zweites Beispiel:  
Der Schnitt respektiert die gestrichelten Kanten nicht.

## Graphalgorithmen

### Definition

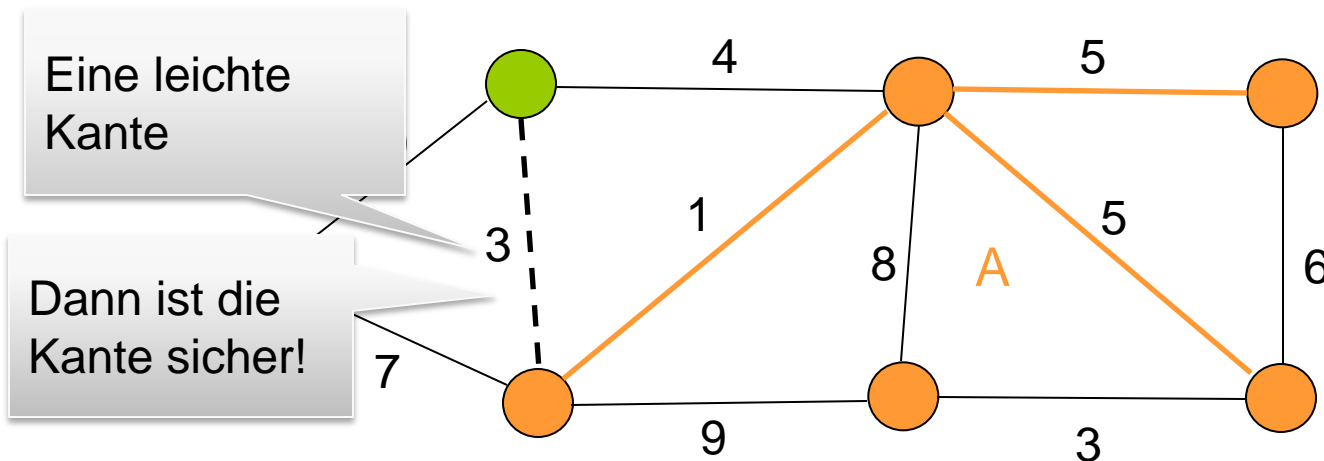
- Eine Kante, die einen Schnitt kreuzt, heißt **leicht**, wenn sie minimales Gewicht unter allen Kanten hat, die den Schnitt kreuzen.



## Graphalgorithmen

### Satz 72

Sei  $G=(V,E)$  ein zusammenhängender, ungerichteter und gewichteter Graph  $G=(V,E)$ . Sei  $A \subseteq E$  Teilmenge eines minimalen Spannbaums von  $G$  ist. Sei  $(S, V-S)$  ein Schnitt von  $G$ , der  $A$  respektiert und sei  $(u,v)$  eine leichte Kanten, die diesen Schnitt kreuzt. Dann ist  $(u,v)$  sicher für  $A$ .



Ein Schnitt, der  $A$  respektiert.

## Graphalgorithmen

### Satz 72

Sei  $G=(V,E)$  ein zusammenhängender, ungerichteter und gewichteter Graph  $G=(V,E)$ . Sei  $A \subseteq E$  Teilmenge eines minimalen Spannbaums von  $G$  ist. Sei  $(S, V-S)$  ein Schnitt von  $G$ , der  $A$  respektiert und sei  $(u,v)$  eine leichte Kanten, die diesen Schnitt kreuzt. Dann ist  $(u,v)$  sicher für  $A$ .

### Beweis

- Sei  $T$  min. Spannbaum, der  $A$  enthält
- Annahme:  $T$  enthält nicht leichte Kante  $(u,v)$
- Wir konstruieren min. Spannbaum  $T'$ , der  $A$  und  $(u,v)$  enthält
- Kante  $(u,v)$  bildet Kreis mit Pfad  $p$  von  $u$  nach  $v$  in  $T$
- Da  $u$  und  $v$  auf gegenüberliegenden Seiten des Schnitts  $(S, V-S)$  liegen, gibt es mind. eine Kante aus  $p$ , die auch den Schnitt kreuzt
- Sei  $(x,y)$  eine solche Kante

## Graphalgorithmen

### Satz 72

Sei  $G=(V,E)$  ein zusammenhängender, ungerichteter und gewichteter Graph  $G=(V,E)$ . Sei  $A \subseteq E$  Teilmenge eines minimalen Spannbaums von  $G$  ist. Sei  $(S, V-S)$  ein Schnitt von  $G$ , der  $A$  respektiert und sei  $(u,v)$  eine leichte Kanten, die diesen Schnitt kreuzt. Dann ist  $(u,v)$  sicher für  $A$ .

### Beweis

- $(x,y)$  ist nicht in  $A$ , da der Schnitt  $A$  respektiert
- Da  $(x,y)$  auf dem eindeutig bestimmten Pfad von  $u$  nach  $v$  in  $T$  ist, wird  $T$  durch Entfernen von  $(x,y)$  in zwei Komponenten aufgeteilt.
- Hinzunahme von  $(u,v)$  verbindet diese Komponenten wieder
- Definiere:  $T' = T - \{(x,y)\} \cup \{(u,v)\}$
- Wir zeigen, dass  $T'$  min. Spannbaum ist

## Graphalgorithmen

### Satz 72

Sei  $G=(V,E)$  ein zusammenhängender, ungerichteter und gewichteter Graph  $G=(V,E)$ . Sei  $A \subseteq E$  Teilmenge eines minimalen Spannbaums von  $G$  ist. Sei  $(S, V-S)$  ein Schnitt von  $G$ , der  $A$  respektiert und sei  $(u,v)$  eine leichte Kanten, die diesen Schnitt kreuzt. Dann ist  $(u,v)$  sicher für  $A$ .

### Beweis

- Da  $(u,v)$  leichte Kante ist, die  $(S, V-S)$  kreuzt und  $(x,y)$  ebenfalls  $(S, V-S)$  kreuzt, gilt  $w(u,v) \leq w(x,y)$ . Daher
- $w(T') = w(T) - w(x,y) + w(u,v) \leq w(T)$
- $T$  ist aber min. Spannbaum und somit gilt  $w(T) \leq w(T')$
- Daher muss  $T'$  ebenfalls min. Spannbaum sein
- Es bleibt zu zeigen, dass  $(u,v)$  sicher für  $A$  ist.

## Graphalgorithmen

### Satz 72

Sei  $G=(V,E)$  ein zusammenhängender, ungerichteter und gewichteter Graph  $G=(V,E)$ . Sei  $A \subseteq E$  Teilmenge eines minimalen Spannbaums von  $G$  ist. Sei  $(S, V-S)$  ein Schnitt von  $G$ , der  $A$  respektiert und sei  $(u,v)$  eine leichte Kanten, die diesen Schnitt kreuzt. Dann ist  $(u,v)$  sicher für  $A$ .

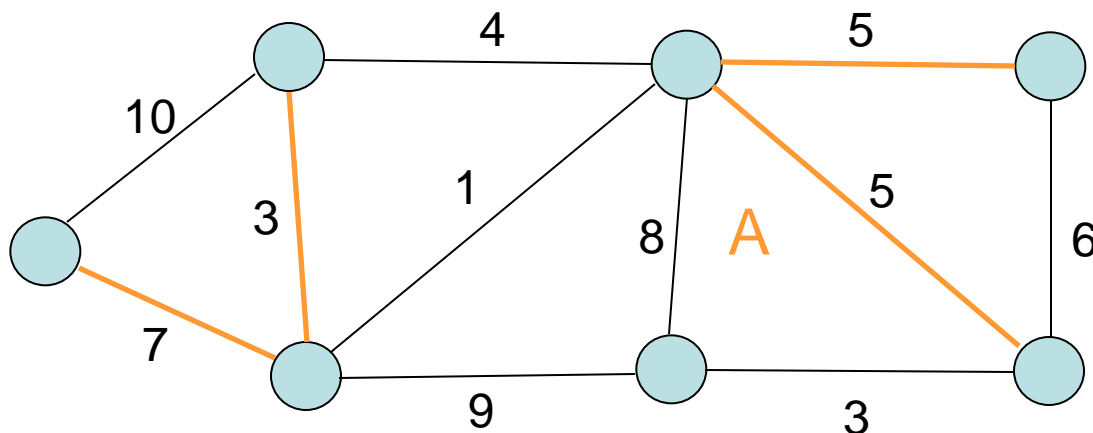
### Beweis

- Es bleibt zu zeigen, dass  $(u,v)$  sicher für  $A$  ist.
- $A \subseteq T'$ , da  $A \subseteq T$  und  $(x,y) \notin A$
- Also ist  $A \cup \{(u,v)\} \subseteq T'$  und da  $T'$  min. Spannbaum ist, ist  $(u,v)$  sicher für  $A$

## Graphalgorithmen

### Korollar 73

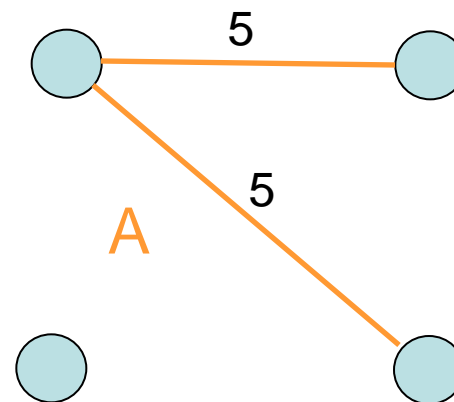
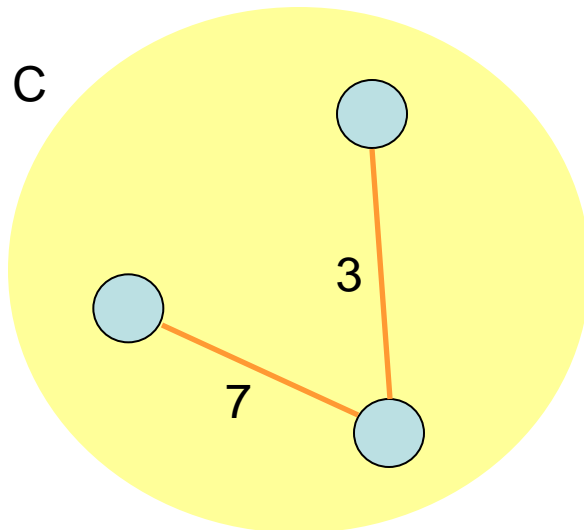
Sei  $G=(V,E)$  ein zusammenhängender, ungerichteter, gewichteter Graph. Sei  $A$  eine Teilmenge von  $E$ , die in einem minimalen Spannbaum von  $G$  enthalten ist und sei  $C$  eine Zusammenhangskomponente (Baum) im Wald  $H=(V,A)$ . Wenn  $(u,v)$  eine leichte Kante ist, die  $C$  mit einer anderen Zusammenhangskomponente in  $H$  verbindet, dann ist  $(u,v)$  sicher für  $A$ .



## Graphalgorithmen

### Korollar 73

Sei  $G=(V,E)$  ein zusammenhängender, ungerichteter, gewichteter Graph. Sei  $A$  eine Teilmenge von  $E$ , die in einem minimalen Spannbaum von  $G$  enthalten ist und sei  $C$  eine Zusammenhangskomponente (Baum) im Wald  $H=(V,A)$ . Wenn  $(u,v)$  eine leichte Kante ist, die  $C$  mit einer anderen Zusammenhangskomponente in  $H$  verbindet, dann ist  $(u,v)$  sicher für  $A$ .

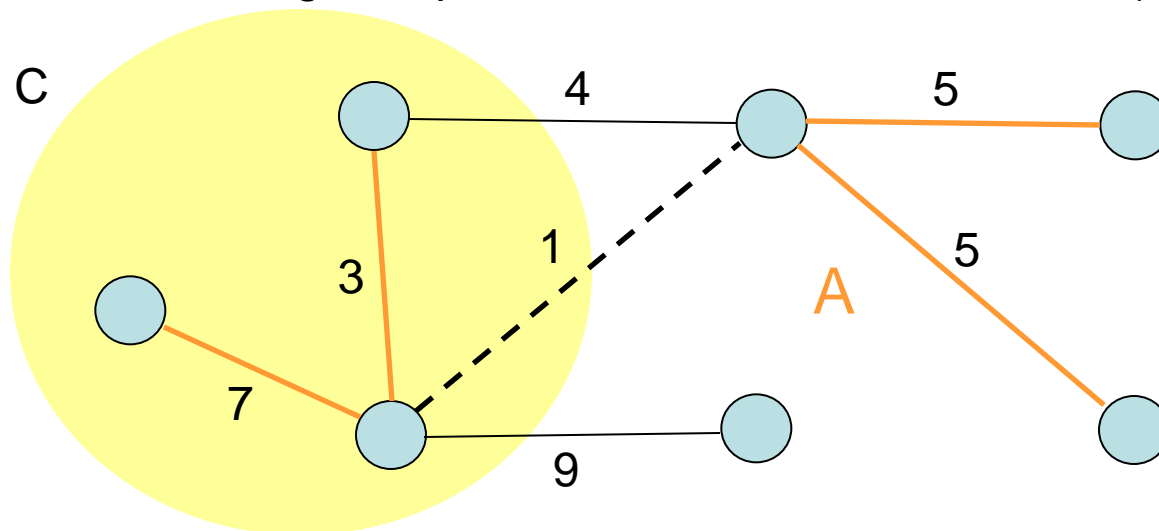


Wald H und Komponente C.

## Graphalgorithmen

### Korollar 73

Sei  $G=(V,E)$  ein zusammenhängender, ungerichteter, gewichteter Graph. Sei  $A$  eine Teilmenge von  $E$ , die in einem minimalen Spannbaum von  $G$  enthalten ist und sei  $C$  eine Zusammenhangskomponente (Baum) im Wald  $H=(V,A)$ . Wenn  $(u,v)$  eine leichte Kante ist, die  $C$  mit einer anderen Zusammenhangskomponente in  $H$  verbindet, dann ist  $(u,v)$  sicher für  $A$ .



Leichte Kante  
ist sicher.

## Graphalgorithmen

### *Korollar 73*

Sei  $G=(V,E)$  ein zusammenhängender, ungerichteter, gewichteter Graph. Sei  $A$  eine Teilmenge von  $E$ , die in einem minimalen Spannbaum von  $G$  enthalten ist und sei  $C$  eine Zusammenhangskomponente (Baum) im Wald  $H=(V,A)$ . Wenn  $(u,v)$  eine leichte Kante ist, die  $C$  mit einer anderen Zusammenhangskomponente in  $H$  verbindet, dann ist  $(u,v)$  sicher für  $A$ .

### *Beweis*

- Der Schnitt  $(C,V-C)$  respektiert  $A$  und  $(u,v)$  ist damit leichte Kante für diesen Schnitt.

## Graphalgorithmen

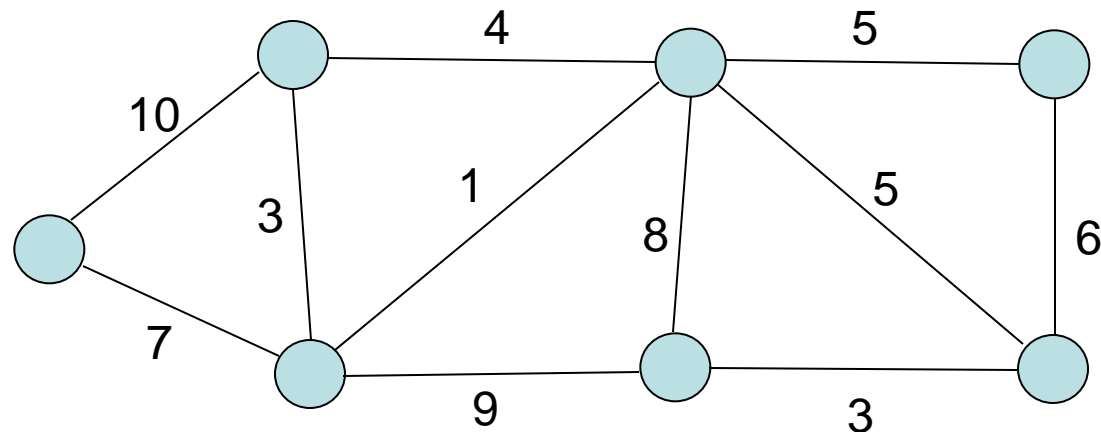
### *Idee des Algorithmus von Kruskal*

- Verwende generischen Algorithmus
- Nimm immer die Kante mit geringstem Gewicht, die zwei Bäume im aktuellen aufspannenden Wald verbindet und füge diese zu A hinzu
- Seien C und D diese beiden Bäume
- Die Kante ist eine leichte Kante, die C mit einem anderen Baum verbindet
- Damit ist sie sicher für C

## Graphalgorithmen (siehe Vollversion)

### Kruskal(G)

1.  $A \leftarrow \emptyset$
2. Sortiere Kanten nach Gewicht
3. **for each**  $(u,v) \in E$  geordnet nach aufsteigendem Gewicht **do**
4. **if**  $u$  und  $v$  sind nicht in der selben Zusammenhangskomponente in Graph  $H=(V,A)$  **then**
5.  $A \leftarrow A \cup \{(u,v)\}$
6. **return**  $A$



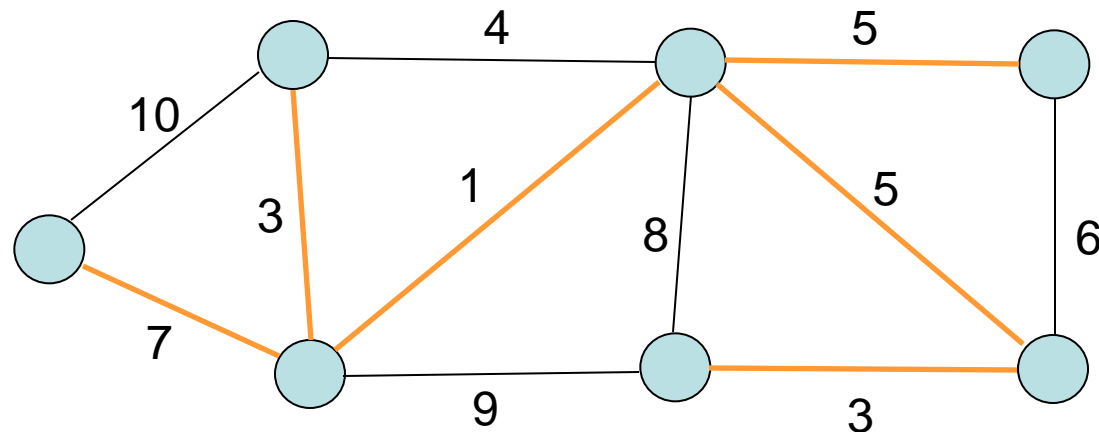
## Graphalgorithmen

Wie kann man Zeile 4 implementieren?

Laufzeit:  $O(|E|\log|E|+|E|)$  „Zeit für Zeile 4“

### Kruskal(G)

1.  $A \leftarrow \emptyset$
2. Sortiere Kanten nach Gewicht
3. **for each**  $(u,v) \in E$  geordnet nach aufsteigendem Gewicht **do**
4. **if**  $u$  und  $v$  sind nicht in der selben Zusammenhangskomponente in Graph  $H=(V,A)$  **then**
5.      $A \leftarrow A \cup \{(u,v)\}$
6. **return**  $A$



## Graphalgorithmen

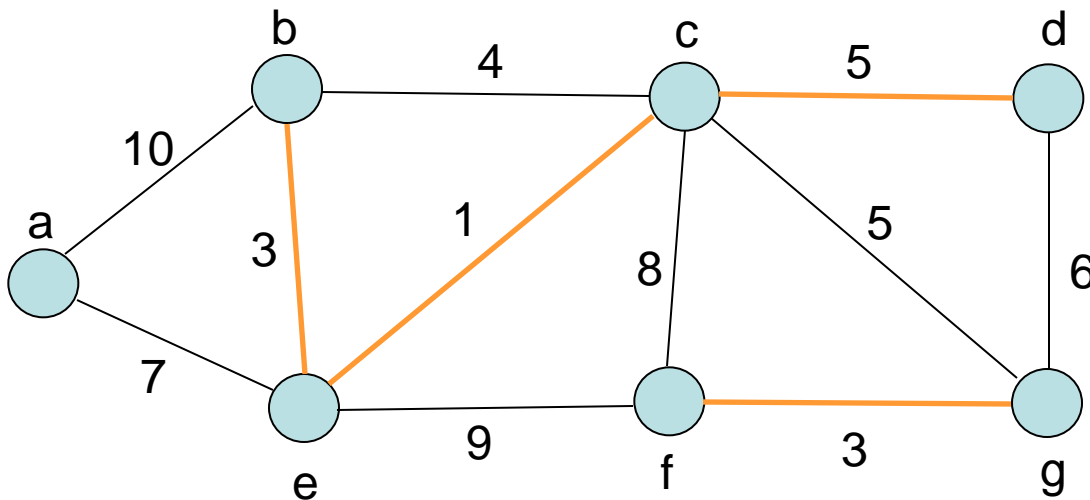
### *Union-Find Datenstrukturen*

- Familie von *disjunkten* Mengen  $\mathcal{S} = \{S_1, \dots, S_k\}$
- Für jede Menge gibt es einen Repräsentanten
- Make-Set(x): Erzeuge neue Menge, die nur x enthält
- Union(x,y): Vereinigung der Mengen, die x bzw. y enthalten
- Find-Set(x): Gibt Referenz auf den Repräsentanten der Menge, die x enthält

## Graphalgorithmen

### Idee

- Disjunkte Mengen bei Union-Find sind Knoten des Graphen bei Kruskal



Im Beispiel:  
 $\{a\}, \{b,c,d,e\}, \{f,g\}$

## Graphalgorithmen

### Kruskal(G)

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

Zu Beginn ist jeder Knoten eine Zusammenhangskomponente in  $H=(V,A)$

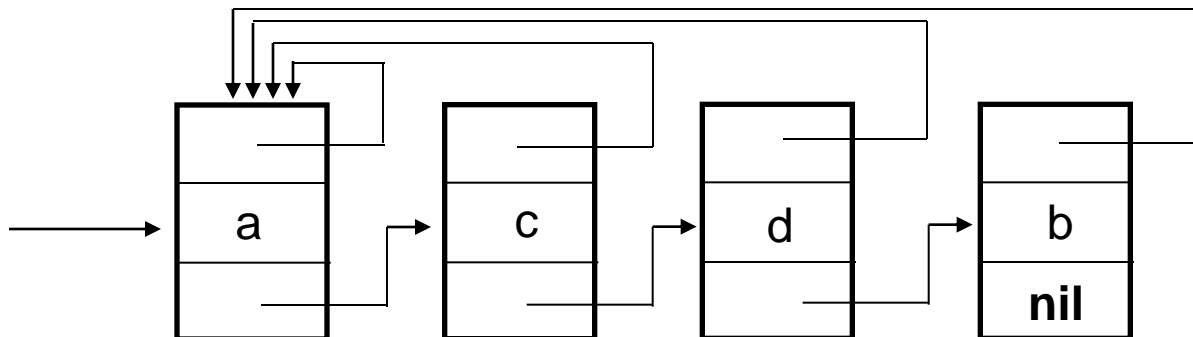
Sind  $u$  und  $v$  in derselben Zusammenhangskomponente?

Wenn ja, dann müssen die Zusammenhangskomponenten vereinigt werden

## Graphalgorithmen

### *Eine einfache Union-Find Datenstruktur*

- Jede Menge ist Liste
- Erstes Element ist Repräsentant
- Jedes Listenelement enthält Zeiger auf den Repräsentanten



## Graphalgorithmen

### *Implementierung*

- Make-Set in  $O(1)$  Zeit einfach
- Find in  $O(1)$  Zeit einfach
- Union: Hänge die eine Liste hinter die andere und aktualisiere alle Zeiger

### *Laufzeit*

- Betrachte Sequenz von  $m$  Operationen aus Make-Set, Find, und Union
- Laufzeit für Union  $O(m)$  (da wir höchstens  $m$  Elemente haben)

## Graphalgorithmen

### *Beobachtung*

- Wir hängen evtl. immer eine sehr Länge Liste an eine sehr kurze
- Wenn wir immer die kurze hinter die lange hängen, müssen wir nur die Referenzen in der kurzen Liste aktualisieren
- Aber bringt das etwas (mehr als Konstanten)?



Ja!

## Graphalgorithmen

### Satz 74

Wenn wir verkettete Listen als Union-Find Datenstruktur benutzen und bei einer Union Operation immer die kürzere hinter die längere Liste hängen und entsprechend aktualisieren, dann benötigt eine Sequenz von  $m$  Operationen aus Make-Set, Union und Find, von denen  $n$  Operationen Make-Set sind,  $O(m + n \log n)$  Zeit.

### Beweis

- Wir analysieren zunächst, wie oft der Repräsentantenzeiger eines Elements einer Menge der Größe  $k$  maximal aktualisiert wurde
- Betrachte Element  $x$
- Jedes mal, wenn der Repräsentantenzeiger von  $x$  aktualisiert wurde, war  $x$  in der kleineren der vereinigten Mengen
- Damit hat sich die Größe der Menge mindestens verdoppelt

## Graphalgorithmen

### Satz 74

Wenn wir verkettete Listen als Union-Find Datenstruktur benutzen und bei einer Union Operation immer die kürzere hinter die längere Liste hängen und entsprechend aktualisieren, dann benötigt eine Sequenz von  $m$  Operationen aus Make-Set, Union und Find, von denen  $n$  Operationen Make-Set sind,  $O(m + n \log n)$  Zeit.

### Beweis

- Damit gilt für jedes  $k \leq n$ , dass nach  $\lceil \log k \rceil$  Aktualisierungen des Repräsentantenzeigers von  $x$ , die Menge, die  $x$  enthält, mindestens  $k$  Elemente besitzt
- Da die größte Menge maximal  $n$  Elemente besitzt, wurde jeder Repräsentantenzeiger maximal  $O(\log n)$  mal aktualisiert (über alle Union-Operationen)
- Damit ist die Gesamtlaufzeit für die Aktualisierungen der  $n$  Objekte  $O(n \log n)$

## Graphalgorithmen

### Satz 74

Wenn wir verkettete Listen als Union-Find Datenstruktur benutzen und bei einer Union Operation immer die kürzere hinter die längere Liste hängen und entsprechend aktualisieren, dann benötigt eine Sequenz von  $m$  Operationen aus Make-Set, Union und Find, von denen  $n$  Operationen Make-Set sind,  $O(m + n \log n)$  Zeit.

### Beweis

- Jedes MakeSet und Find benötigt  $O(1)$  Zeit und es gibt  $O(m)$  davon
- Damit ist die gesamte Laufzeit für die Sequenz  $O(m + n \log n)$

## Graphalgorithmen

Kruskal(G)

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

### Satz 75

Der Algorithmus von Kruskal berechnet in  $O(|E| \log |E|)$  einen minimalen Spannbaum eines gewichteten, zusammenhängenden, ungerichteten Graphen  $G=(V,E)$ .

## Graphalgorithmen

### Satz 75

Der Algorithmus von Kruskal berechnet in  $O(|E| \log |E|)$  einen minimalen Spannbaum eines gewichteten, zusammenhängenden, ungerichteten Graphen  $G=(V,E)$ .

### Beweis

- Der Algorithmus hält die Invariante aufrecht, dass die Mengen in der Union-Find Datenstruktur den Zusammenhangskomponenten des durch die bisher ausgewählten Kanten  $A$  definierten Graphen entsprechen.
- Da die Kanten in aufsteigender Reihenfolge ihrer Gewichte betrachtet werden, ist jede Kante, die zwei solche Zusammenhangskomponenten verbindet, eine leichte Kante
- Somit ist sie nach Korollar 73 auch sicher
- Wir wissen, dass der Algorithmus nur sichere Kanten in  $A$  einfügt
- Es bleibt zu zeigen, dass  $A$  am Ende des Algorithmus ein aufspannender Baum ist

## Graphalgorithmen

### Satz 75

Der Algorithmus von Kruskal berechnet in  $O(|E| \log |E|)$  einen minimalen Spannbaum eines gewichteten, zusammenhängenden, ungerichteten Graphen  $G=(V,E)$ .

### Beweis

- Annahme:  $A$  ist am Ende des Algorithmus kein aufspannender Baum
- Da  $A$  nur aus sicheren Kanten besteht, ist  $A$  dann Teilmenge eines min. Spannbaums  $T$
- Betrachte eine Kante  $(u,v)$ , aus  $T-A$
- Das Entfernen von  $(u,v)$  aus  $T$  definiert einen Schnitt  $(S,V-S)$
- Da  $A$  keine Kante enthält, die  $(S,V-S)$  kreuzt ist jede Zusammenhangskomponente von  $A$  entweder Teilmenge von  $S$  oder von  $V-S$
- Wenn nun der Algorithmus  $(u,v)$  betrachtet, so liegt  $u$  in einer Komponente mit Knoten aus  $S$  und  $v$  in einer Komponente mit Knoten aus  $V-S$

## Graphalgorithmen

### *Satz 75*

Der Algorithmus von Kruskal berechnet in  $O(|E| \log |E|)$  einen minimalen Spannbaum eines gewichteten, zusammenhängenden, ungerichteten Graphen  $G=(V,E)$ .

### *Beweis*

- Damit liegen  $u$  und  $v$  in unterschiedlichen Komponenten und somit ist  $\text{Find}(u) \neq \text{Find}(v)$
- Dann hätte der Algorithmus aber  $(u,v)$  in  $A$  aufgenommen. Widerspruch!
- Somit ist  $A$  am Ende des Algorithmus ein aufspannender Baum und auch ein min. Spannbaum, da alle Kanten sicher waren

## Graphalgorithmen

### Satz 75

Der Algorithmus von Kruskal berechnet in  $O(|E| \log |E|)$  einen minimalen Spannbaum eines gewichteten, zusammenhängenden, ungerichteten Graphen  $G=(V,E)$ .

### Beweis

- Laufzeit:
- Der Algorithmus benötigt  $O(|E| \log |E|)$  Zeit zum Sortieren
- Er führt  $O(|E| + |V|)$  Operationen mit der Union-Find-Datenstruktur durch
- Davon sind  $|V|$  Operationen MakeSet
- Somit ist die Laufzeit für die Operationen der Union-Find-Datenstruktur  $O(|E| + |V| \log |V|)$
- Diese dominieren der Laufzeit der zweiten for-Schleife
- Insgesamt ist daher die Laufzeit  $O(|E| \log |E|)$   
(da  $|V| = O(|E|)$  für zusammenhängende Graphen)

## Graphalgorithmen

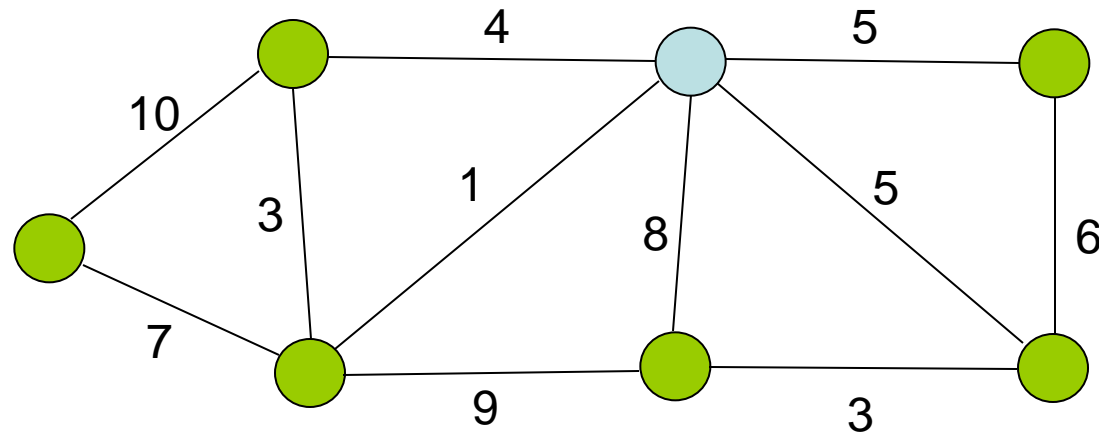
### *Idee des Algorithmus von Prim*

- Verwende generischen Algorithmus
- Nimm immer eine Kante mit minimalem Gewicht, die einen Knoten in Baum A mit einem Knoten verbindet, der nicht in Baum A ist und füge diese zu A hinzu
- Die Kante ist eine leichte Kante, die Baum A mit einem weiteren Knoten verbindet
- Damit ist sie sicher für A

## Graphalgorithmen (siehe Vollversion)

Prim(G,r)

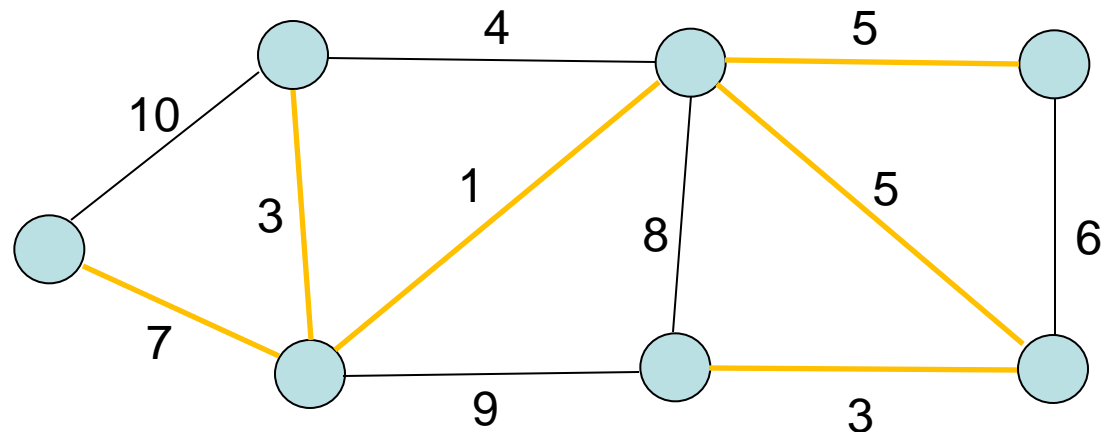
1.  $Q \leftarrow V - \{r\}$
2.  $A \leftarrow \emptyset$
3. **while**  $Q \neq \emptyset$
4. Finde Kante  $(u,v)$  mit minimalem Gewicht, die den Schnitt  $(Q, V-Q)$  kreuzt, wobei  $u \in Q$
5.  $Q \leftarrow Q - \{u\}$
6.  $A \leftarrow A \cup \{(u,v)\}$
7. **return**  $A$



## Graphalgorithmen (siehe Vollversion)

Prim(G,r)

1.  $Q \leftarrow V - \{r\}$
2.  $A \leftarrow \emptyset$
3. **while**  $Q \neq \emptyset$
4. Finde Kante  $(u,v)$  mit minimalem Gewicht, die den Schnitt  $(Q, V-Q)$  kreuzt, wobei  $u \in Q$
5.  $Q \leftarrow Q - \{u\}$
6.  $A \leftarrow A \cup \{(u,v)\}$
7. **return** A



## Graphalgorithmen

### *Prioritätenschlange*

- Alle Knoten, die noch nicht zum Baum gehören, werden in Prioritätenschlange Q abgespeichert
- $\text{key}[v]$ : minimales Gewicht einer Kante, die  $v$  mit Baum verbindet
- $\text{parent}[v]$ : Vorgänger von  $v$  im Baum
- Menge  $A$  implizit gegeben durch  $A = \{(v, \text{parent}[v]) \mid v \in V - \{r\} - Q\}$

## Graphalgorithmen

Prim(G,r)

1.  $Q \leftarrow V$
2. **For each** vertex  $u \in Q$  **do**  $\text{key}[u] \leftarrow \infty$
3.  $\text{key}[r] \leftarrow 0, \text{parent}[r] \leftarrow \text{nil}$
4. **While**  $Q \neq \emptyset$  **do**
5.  $u \leftarrow \text{Extract-Min}(Q)$
6. **For each**  $v \in \text{Adj}[u]$  **do**
7. **If**  $v \in Q$  **and**  $w(u,v) < \text{key}[v]$  **then**
8.  $\text{key}[v] \leftarrow w(u,v), \text{parent}[v] \leftarrow u$

Zu Beginn ist jeder Knoten in der Schlange Q

r bildet die Wurzel des Spannbaums

u ist inzident zu leichter Kante, die Schnitt  $(Q, V-Q)$  kreuzt

Wenn ja, dann ist u neuer Vorgänger von v und  $\text{key}[v]$  gibt die neuen Kosten für v an

Gehört v noch nicht zum Baum und könnte über u billiger erreicht werden als bisher bekannt?

## Graphalgorithmen

### *Implementierung*

- Q als binäre Halde

### *Laufzeit*

- Initialisierung von Q:  $O(|V|)$
- Ausführung der while-Schleife:  $O(|V|)$ -mal
- Extract-Min:  $O(\log |V|)$
- Länge aller Adjazenzlisten:  $O(|E|)$
- Test  $v \in Q$ :  $O(1)$
- Änderung eines Schlüsselwertes:  $O(\log |V|)$
- Gesamtlaufzeit:  $O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$

## Graphalgorithmen

### *Satz 76*

Der Algorithmus von Prim berechnet einen minimalen Spannbaum eines gewichteten, zusammenhängenden, ungerichteten Graphen in  $O(|E| \log |V|)$  Zeit.

### *Beweis*

- Die Laufzeit haben wir bereits analysiert.
- Die Korrektheit folgt aus der Tatsache, dass wir nur sichere Kanten verwenden und dass der Algorithmus einen Baum berechnet