2.6 Graphen

Wir wollen einen Baum finden, der alle Knoten eines gegebenen Graphen beinhaltet und dessen Kantengewichtsumme minimal ist.

2.6.3 Minimal spannende Bäume

Minimal Spannende Bäume

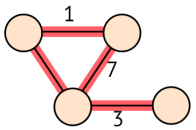
- Gegeben sei ein zusammenhängender, ungerichteter, gewichteter Graph G=(V,E) mit Gewichtsfunktion w:E→R.
- Gesucht ist ein Spannbaum S⊆E der

$$w(S) = \sum_{e \in S} w(e)$$

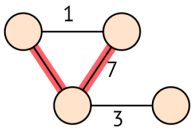
minimiert, der sogenannte MST.

Minimal Spannende Bäume

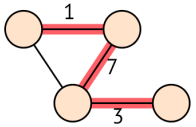
Kein Baum!



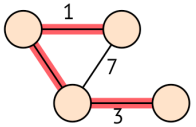
Nicht spannend!



Nicht minimal!



MST



Generischer Algorithmus

- Greedy-Strategie: Füge nach und nach Kanten zu einer anfangs leeren Kantenmenge A hinzu, und zwar unter Beachtung der folgenden Invariante:
  - A ist Teilmenge **eines** MST (*)

Ein Greedy-Algorithmus fällt in jedem Schritt eine Entscheidung, die lokal optimal ist und nie wieder revidiert wird.

Auch ein MST in nicht notwendigerweise eindeutig. Beispiel: Graph mit homogenen Kantengewichten.

Generischer Algorithmus

- Sei A eine Menge von Kanten, die die Invariante (*) erfüllt. Eine Kante e heißt sicher für A, falls $A \cup \{e\}$ ebenfalls die Invariante (*) erfüllt.

Generischer Algorithmus

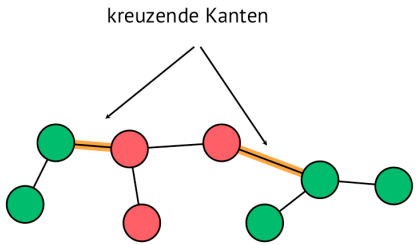
- GenericMst(G, w)
  A ← ∅
  while A ist kein MST do
    suche eine sichere Kante e für A
    A ← A ∪ {e}
  return A

Zu diesem Zeitpunkt weiß man natürlich noch nicht, wie man eine "sichere Kante" findet. Damit beschäftigen wir uns im Folgenden.

Generischer Algorithmus

- Ein Schnitt eines ungerichteten Graphen G=(V,E) ist eine Menge S⊆V
- Eine Kante (u,v) kreuzt den Schnitt S falls einer ihrer Endpunkte in S und der andere Endpunkt in V−S liegt.

Generischer Algorithmus



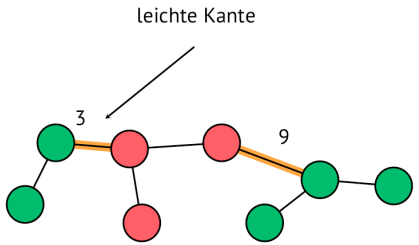
Rote Knoten: Schnitt. Grüne Knoten: Komplement von Schnitt.
Die Kanten, die die roten und grünen Knoten verbinden, heißen kreuzende Kanten.

Generischer Algorithmus

- Ein Schnitt respektiert A, falls keine Kante aus A den Schnitt kreuzt.
- Eine kreuzende Kante heißt leicht, falls ihr Gewicht minimal unter allen kreuzenden Kanten ist.

Es kann natürlich immer mehrere Kanten mit minimalem Gewicht geben, nämlich genau dann, wenn mehrere zu demselben Knoten adjazente Kanten dasselbe Gewicht haben.

Generischer Algorithmus



Eine leichte Kante ist diejenige, die das geringste Gewicht hat. In dem Falle steht nur jeweils eine Kante zur Auswahl.

Generischer Algorithmus

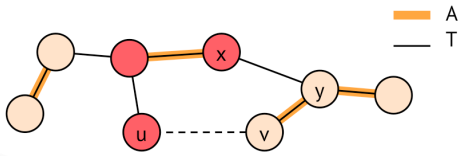
- Satz: A erfülle die Invariante (*). S sei ein Schnitt der A respektiert und (u,v) eine leichte Kante, die S kreuzt. Dann ist (u,v) sicher für A.

Die Invariante (*) bedeutet, dass A auch nach Hinzufügen der Kante (u,v) noch ein minimaler Spannbaum ist.

Generischer Algorithmus

- Sei T ein MST, der A enthält, aber nicht (u,v)

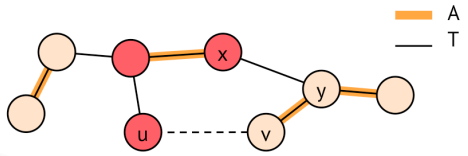
Würde er (u,v) enthalten, so wären wir schon fertig!



Generischer Algorithmus

- Sei T ein MST, der A enthält, aber nicht (u,v)
- Auf dem Weg von u nach v durch T gibt es eine Kante (x,y), die den Schnitt kreuzt

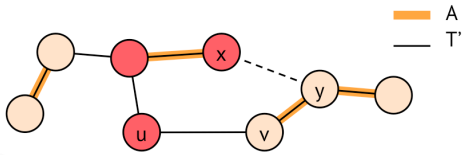
Der Weg existiert, da T ein spannend ist. Er ist eindeutig, da T ein Baum ist.



Beweis des Satzes in Folie 12.

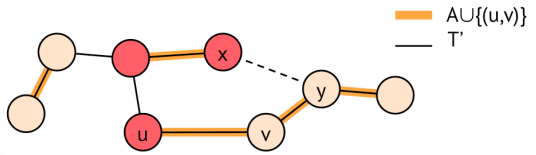
Generischer Algorithmus

- Sei T ein MST, der A enthält, aber nicht (u,v)
- Auf dem Weg von u nach v durch T gibt es eine Kante (x,y), die den Schnitt kreuzt
- T' = T − {(x,y)} ∪ {(u,v)} ist ein MST, denn w(T') = w(T) − w(x,y) + w(u,v) ⩽ w(T)



Generischer Algorithmus

- Sei T ein MST, der A enthält, aber nicht (u,v)
- Auf dem Weg von u nach v durch T gibt es eine Kante (x,y), die den Schnitt kreuzt
- T' = T − {(x,y)} ∪ {(u,v)} ist ein MST
- Wegen A ⊆ T und (x,y) ∉ A folgt A ∪ {(u,v)} ⊆ T', also ist (u,v) sicher für A



2.6.3 Minimal spannende Bäume

2.6.3.1 Generischer Algorithmus
2.6.3.2 Prims Algorithmus
2.6.3.3 Kruskals Algorithmus

Prims und Kruskals Algorithmen sind zwei Instanzen des generischen Algorithmus.

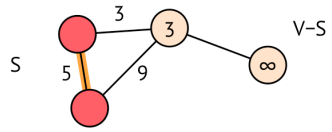
Prims Algorithmus

- Die Kanten in A bilden einen Baum. Jeder Knoten u erhält einen Zeiger p[u] auf seinen Vater im MST.
- Der A respektierende Schnitt S ist gegeben durch $S = \{ u : (u,v) \in A \}$

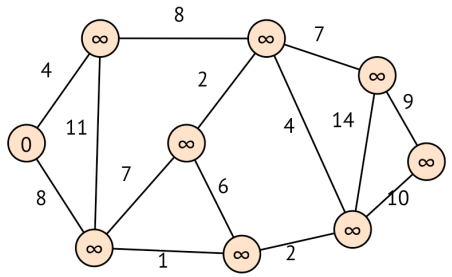
Man geht immer ausgehend von einem Knoten zu seinem Nachbarn und "erobert" ihn. Der neu eroberte Knoten
bekommt dann einen Zeiger auf seinen Vorgänger p[u].

Prims Algorithmus

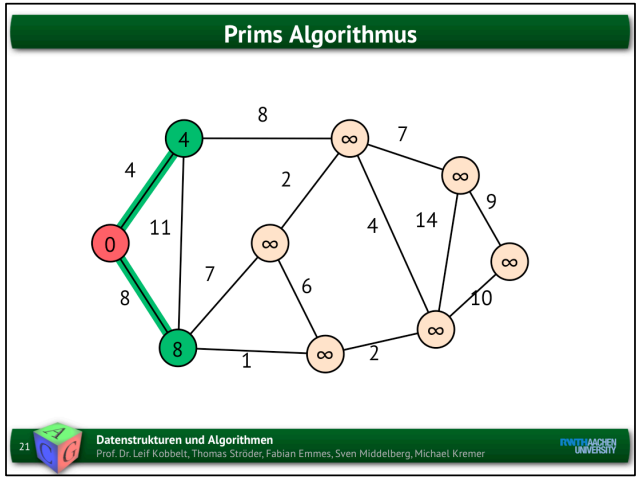
- Knoten u außerhalb von S erhalten einen Schlüssel, der das minimale Gewicht einer Kante angibt, die u mit S verbindet (∞ falls es keine solche Kante gibt)



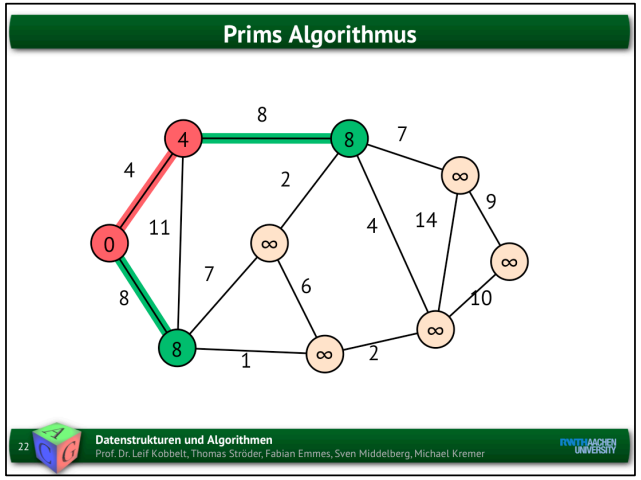
Prims Algorithmus



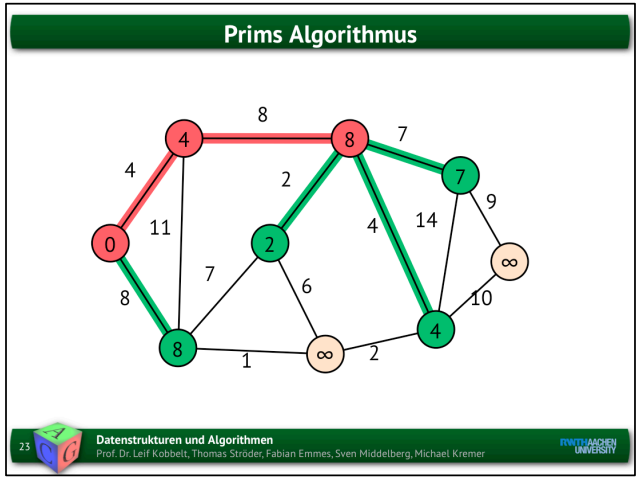
Initialkonfiguration des Algorithmus.

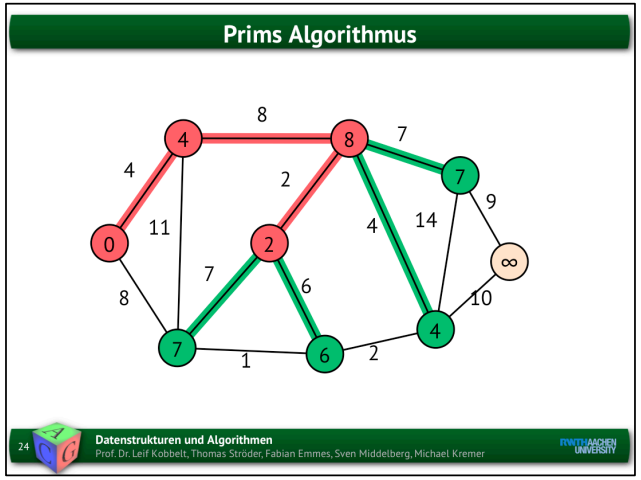


Knoten 0 wird erobert. Dieser hat die beiden Nachbarn 4 und 8.

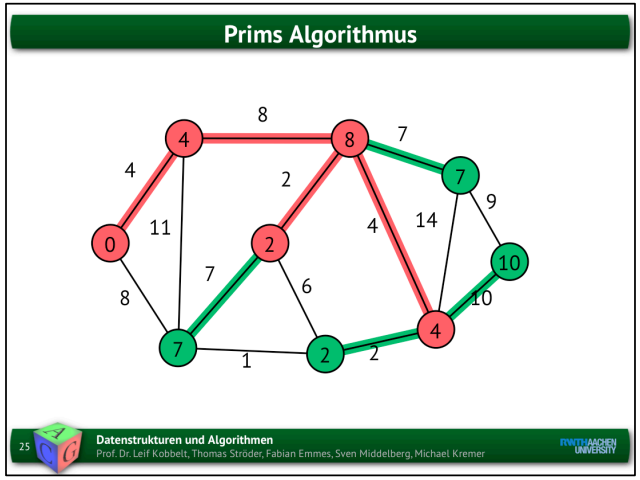


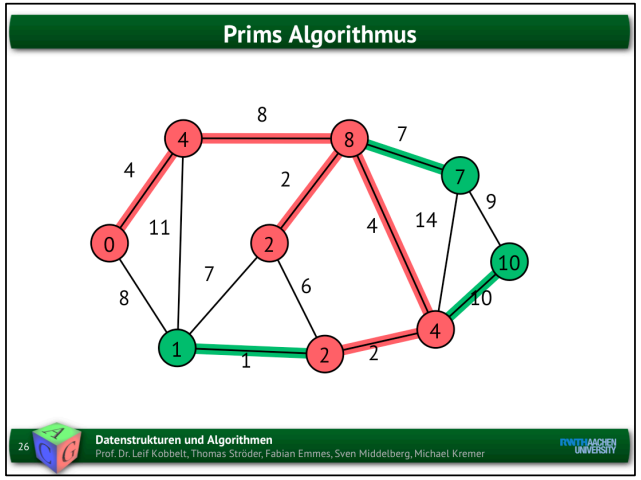
Vom Knoten 4 aus wählen wir den Nachbarknoten mit dem geringsten Gewicht. Die Kante 11 wird nicht
betrachtet, weil wir den Knoten 8 bereits durch eine Kante mit Gewicht 8 dem Spannbaum hinzugefügt haben.
Die roten Knoten/Kanten sind diejenigen, die dem MST hinzugefügt wurden. Die grünen stehen für den
Schnitt.





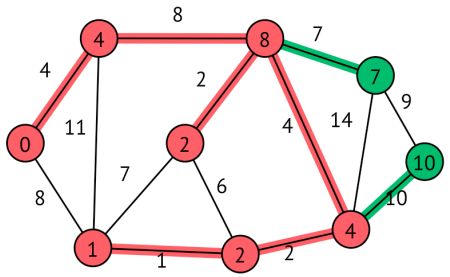
Knoten 8 (unten) kann nun über eine Kante mit Gewicht 7 erobert werden.



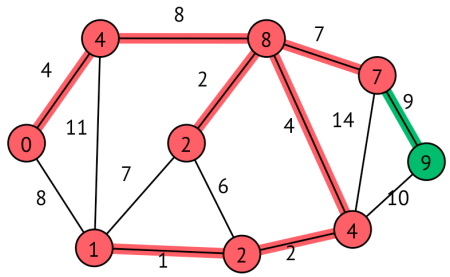


Die Kanten werden nur dann zum Schnitt (potentiell leichte Kanten) hinzugefügt,
wenn ihr Gewicht jeweils geringer ist,
als das, was vorher in den adjazenten Knoten gespeichert ist.

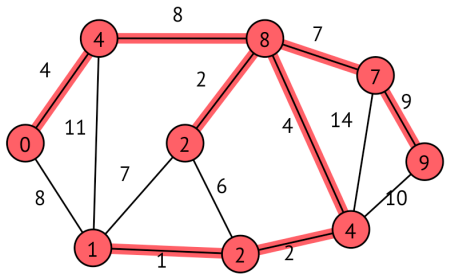
Prims Algorithmus



Prims Algorithmus



Prims Algorithmus



Prims Algorithmus

- PRIM(G,w,r)
  - for each u ∈ V do
    - key[u] ← ∞
    - p[u] ← NIL
  - key[r] ← 0
  - Q ← V
  - while Q not empty do
    - u ← ExtractMinimum(Q)
    - for each v ∈ Adj[u] do
      - if v ∈ Q and w(u,v) < key[v] then
        - p[v] ← u
        - key[v] ← w(u,v)

Prims Algorithmus

- PRIM(G,w,r)
  - for each u ∈ V do
    - key[u] ← ∞
    - p[u] ← NIL

    } O(V)
  - key[r] ← 0
  - Q ← V
  - while Q not empty do
    - u ← ExtractMinimum(Q)
    - for each v ∈ Adj[u] do
      - if v ∈ Q and w(u,v) < key[v] then
        - p[v] ← u
        - key[v] ← w(u,v)

Prims Algorithmus

- PRIM(G,w,r)
  - for each u ∈ V do
    - key[u] ← ∞
    - p[u] ← NIL
  - key[r] ← 0
  - Q ← V

  ] O(V) Heap Aufbau!
  - while Q not empty do
    - u ← ExtractMinimum(Q)
    - for each v ∈ Adj[u] do
      - if v ∈ Q and w(u,v) < key[v] then
        - p[v] ← u
        - key[v] ← w(u,v)

Wir benutzen einen Heap, um die Knoten (sortiert anhand ihrer assoziierten Gewichte) zu speichern.

Prims Algorithmus

```
• PRIM(G,w,r)
    for each u ∈ V do
      key[u] ← ∞
      p[u] ← NIL
    key[r] ← 0
    Q ← V
    while Q not empty do                          ⎤
      u ← ExtractMinimum(Q)                       ⎥
      for each v ∈ Adj[u] do                      ⎥
        if v ∈ Q and w(u,v) < key[v] then         ⎥  V Läufe
          p[v] ← u                                ⎥
          key[v] ← w(u,v)                         ⎦
```


Prims Algorithmus

```
• PRIM(G,w,r)
    for each u ∈ V do
      key[u] ← ∞
      p[u] ← NIL
    key[r] ← 0                                    Entfernen eines
    Q ← V                                         Elements aus dem
    while Q not empty do                          Heap
      u ← ExtractMinimum(Q)                       ⎤ O(log V)
      for each v ∈ Adj[u] do                      ⎦
        if v ∈ Q and w(u,v) < key[v] then
          p[v] ← u
          key[v] ← w(u,v)
```

Prims Algorithmus

- PRIM(G,w,r)

```
for each u ∈ V do
    key[u] ← ∞
    p[u] ← NIL
key[r] ← 0
Q ← V
while Q not empty do                    Durchläufe insgesamt
    u ← ExtractMinimum(Q)
    for each v ∈ Adj[u] do
        if v ∈ Q and w(u,v) < key[v] then
            p[v] ← u                    O(E)
            key[v] ← w(u,v)
```


Prims Algorithmus

- PRIM(G,w,r)

```
for each u ∈ V do
    key[u] ← ∞
    p[u] ← NIL
key[r] ← 0
Q ← V
while Q not empty do                    Heap muss aktualisiert werden
    u ← ExtractMinimum(Q)
    for each v ∈ Adj[u] do
        if v ∈ Q and w(u,v) < key[v] then
            p[v] ← u
            key[v] ← w(u,v)             O(log V)
```

Prims Algorithmus

```
• PRIM(G,w,r)
    for each u ∈ V do                                  ┐
      key[u] ← ∞                                       │
      p[u] ← NIL                                       │ O(V)
    key[r] ← 0                                         │
    Q ← V                                              ┘
    while Q not empty do                               ┐
      u ← ExtractMinimum(Q)                            │
      for each v ∈ Adj[u] do                           │ O(V×log V
        if v ∈ Q and w(u,v) < key[v] then              │ + E×log V)
          p[v] ← u                                     │
          key[v] ← w(u,v)                              ┘
```


Prims Algorithmus

- Nach der Ausführung von Prim(G,w,r) ist der MST gegeben durch
  $A = \{ (u,p[u]) : u \in V - \{r\} \}$
- Implementierung von Prims Algorithmus z.B. mit binärem Heap

Prims Algorithmus

- Implementiert man Prims Algorithmus mittels eines binären Heaps, so ist der Aufwand

  $O((V+E)\times \log V)$ $O(E\times \log V)$

  (Graph zusammenhängend)

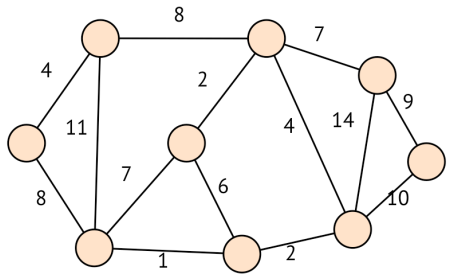
2.6.3 Minimal spannende Bäume

Kruskals Algorithmus

- Die Kanten in A bilden einen Wald. Jeder Knoten u in A erhält einen Zeiger p[u] auf seinen Vater im MST.
- Der A respektierende Schnitt S ist gegeben durch S = { u : (u,v) ∈ A }
- In jedem Schritt wird die Kante mit kleinstem Gewicht gesucht, die zwei Bäume aus A verbindet. Diese Kante wird zu A hinzugefügt.

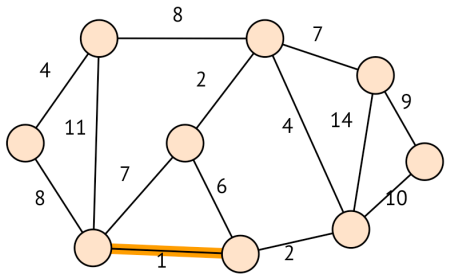
Hier wird nicht der Graph schrittweise abgelaufen und geguckt, was jeweils lokal optimal ist. Sondern es werden einfach
die global minimalgewichteten Kanten zu einem Wald hinzugefügt, so dass MST-Eigenschaft nicht verletzt wird.

Kruskals Algorithmus

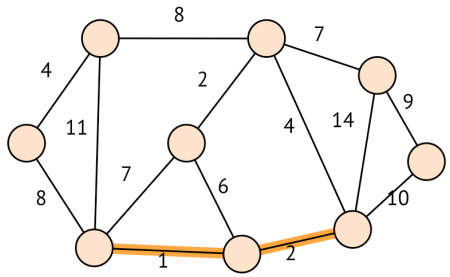


Wir suchen uns die "billigste" aller verfügbaren Kanten heraus, die zwei Komponenten verbindet (und die Spannbaumeigenschaft nicht verletzt).
Initial haben wir |V| Komponenten (= MSTs).

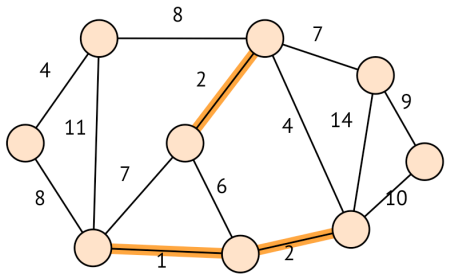
Kruskals Algorithmus



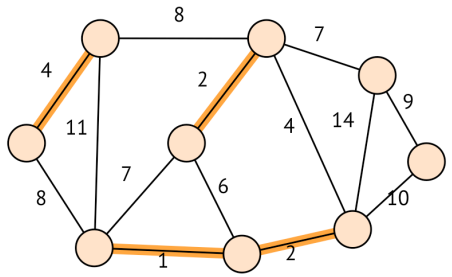
Kruskals Algorithmus



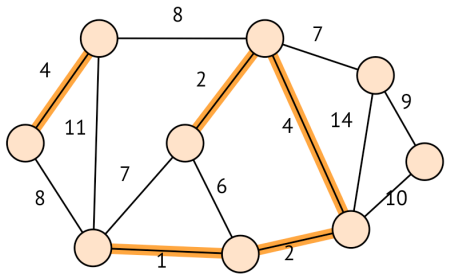
Kruskals Algorithmus



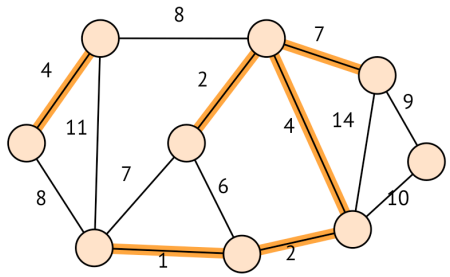
Kruskals Algorithmus



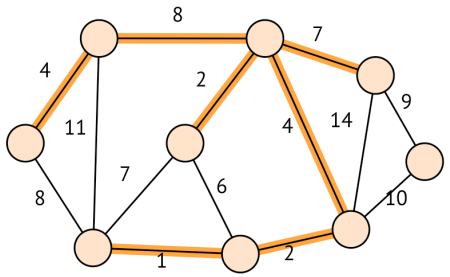
Kruskals Algorithmus



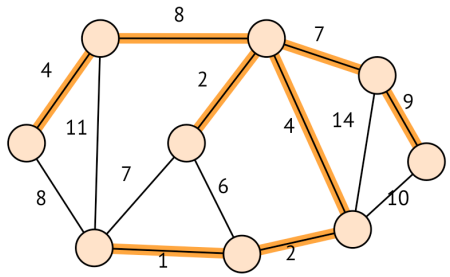
Kruskals Algorithmus



Kruskals Algorithmus



Kruskals Algorithmus



Kruskals Algorithmus

- Kruskal(G,w)

```
A ← ∅
for all v ∈ V do
   MakeSet(v)
sort E into non-decreasing order
for each (u,v) ∈ E do
   if Find(u) ≠ Find(v) then
      A ← A ∪ {(u,v)}
      Union(u,v)
return A
```


Kruskals Algorithmus

- Kruskal(G,w)

```
A ← ∅
for all v ∈ V do
   MakeSet(v)
sort E into non-decreasing order
for each (u,v) ∈ E do
   if Find(u) ≠ Find(v) then
      A ← A ∪ {(u,v)}
      Union(u,v)
return A
```

V × MakeSet()

O(E×log E)

O(E) × Find()
O(V) × Union()

Kruskals Algorithmus

$E < V^2 \rightarrow \log E = O(\log V)$

- Aufwand
  - Sortieren: O(E×log E) = O(E×log V)
  - Union-Find
    - V × MakeSet()
    - O(E) × Find()
    - O(V) × Union()

    ➡O((V+E)×a(V)) = O(E×logV)

  ➡Gesamtaufwand: O(E×log V)

log (V^2) = 2 log V

Kruskals Algorithmus

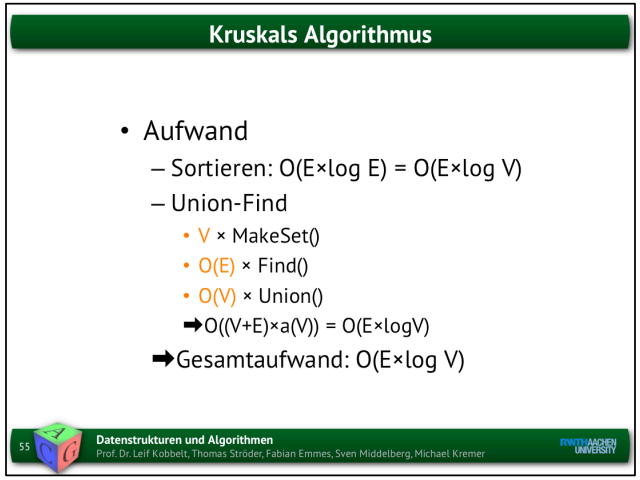
- Aufwand
  - Sortieren: O(E×log E) = O(E×log V)
  - Union-Find

    E > V−1 und a(V) = O(log V)
    - V × MakeSet()
    - O(E) × Find()
    - O(V) × Union()

    ➡O((V+E)×a(V)) = O(E×logV)

  ➡Gesamtaufwand: O(E×log V)

Aufwand für Union-Find



Der Aufwand ist genauso groß wie bei Prims Algorithmus, aber die Reihenfolge, in der die Kanten dem Spannbaum hinzugefügt werden, ist verschieden.