Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG
Lehrstuhl für
INFORMATIK I
Algorithmen & Komplexität
Institut für Informatik

Wiederholung:

Grundlegende Algorithmen aus der Vorlesung ADS

Alexander Wolff

Sommersemester 2026

Übersicht

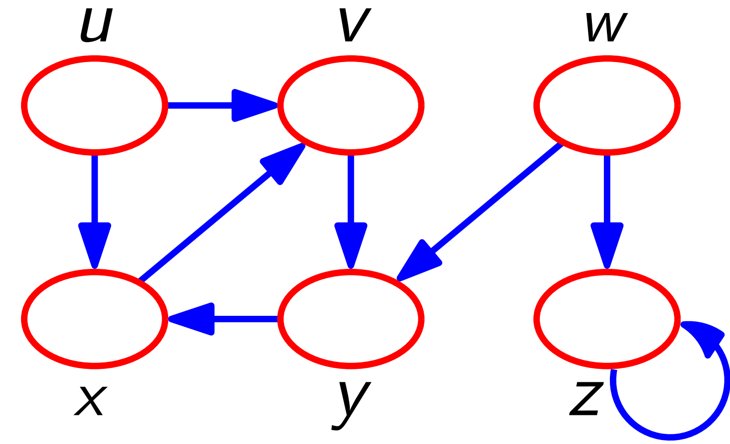
1. Graphdurchlaufstrategien
    1.1 Tiefensuche
        Beispiel
        Pseudocode
        Anwendung
    1.2 Breitensuche
2. Kürzeste Wege
3. Minimale Spannbäume

Übersicht

1. Graphdurchlaufstrategien
   - 1.1 Tiefensuche
     - Beispiel
     - Pseudocode
     - Anwendung
   - 1.2 Breitensuche
2. Kürzeste Wege
3. Minimale Spannbäume

Tiefensuche

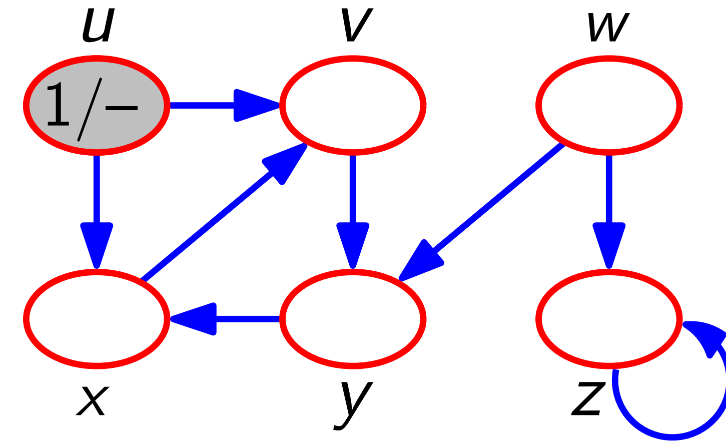
Eingabe: (un)gerichteter Graph G



Tiefensuche

Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d/u.f)$ *discovery time* / *finish time*

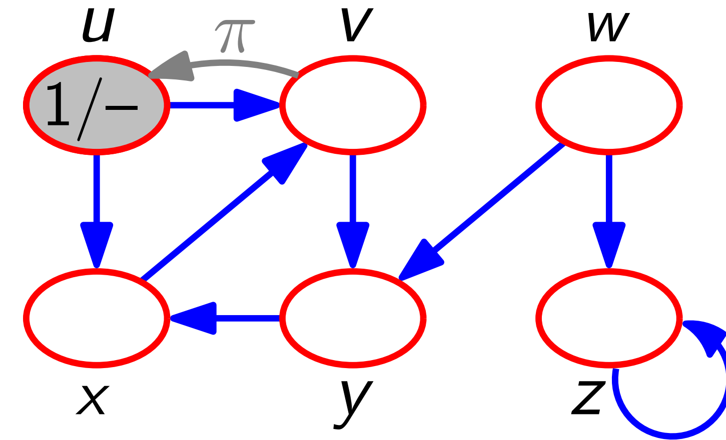


Tiefensuche

Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d/u.f)$ (*discovery time* / *finish time*)

– DFS-Wald ($\xleftarrow{\pi}$)

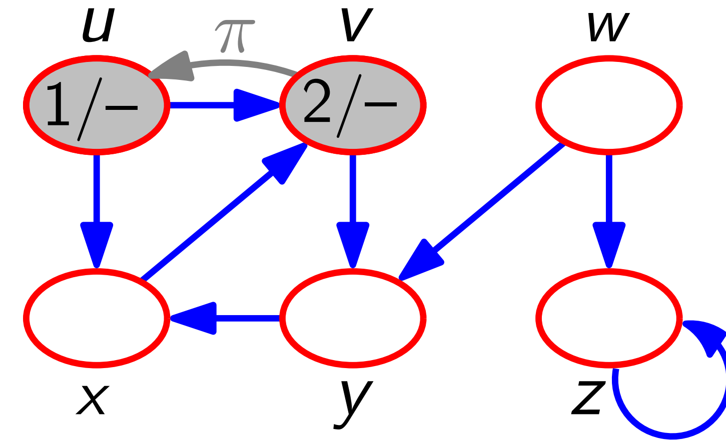


Tiefensuche

Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d / u.f)$ (*discovery time* / *finish time*)

– DFS-Wald $\left(\xleftarrow{\pi}\right)$

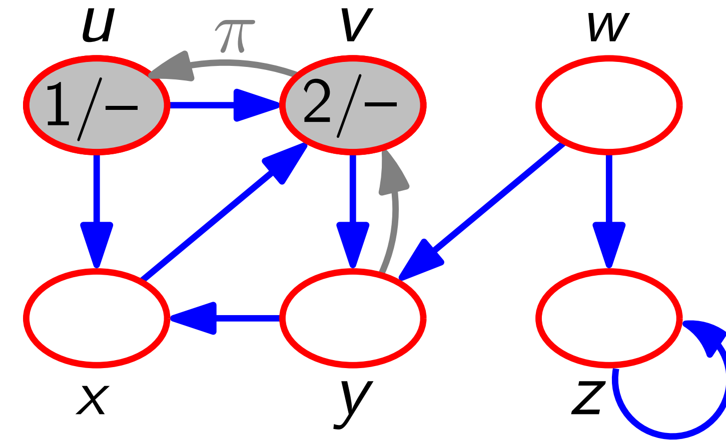


Tiefensuche

Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d\,/\,u.f)$ (*discovery time* / *finish time*)

– DFS-Wald ($\xleftarrow{\pi}$)

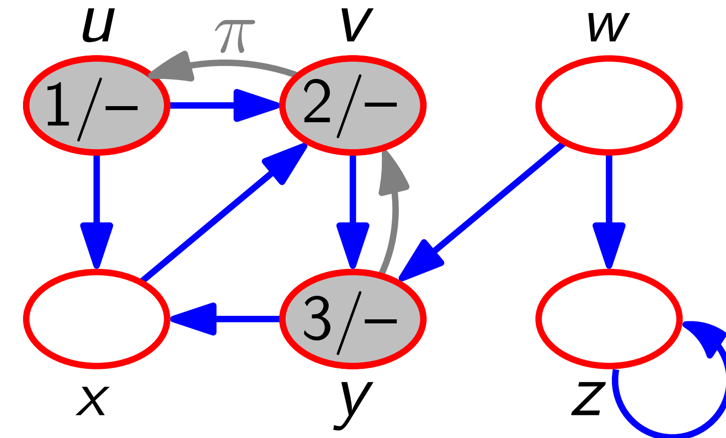


Tiefensuche

Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d/u.f)$ (discovery time / finish time)

– DFS-Wald $\left(\overset{\pi}{\longleftarrow}\right)$

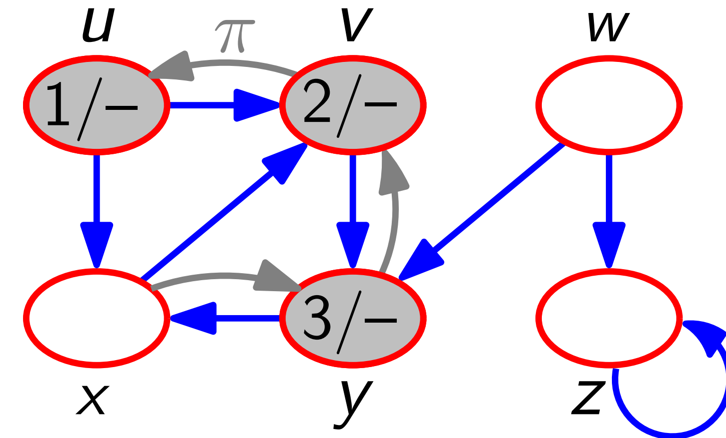


Tiefensuche

Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d\,/\,u.f)$ (*discovery time* / *finish time*)

– DFS-Wald (π)

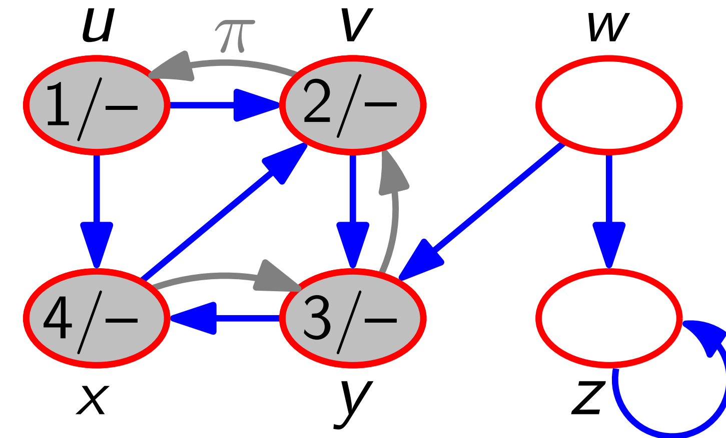


Tiefensuche

Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d/u.f)$ *discovery time* / *finish time*

– DFS-Wald $\left(\overset{\pi}{\longleftarrow}\right)$

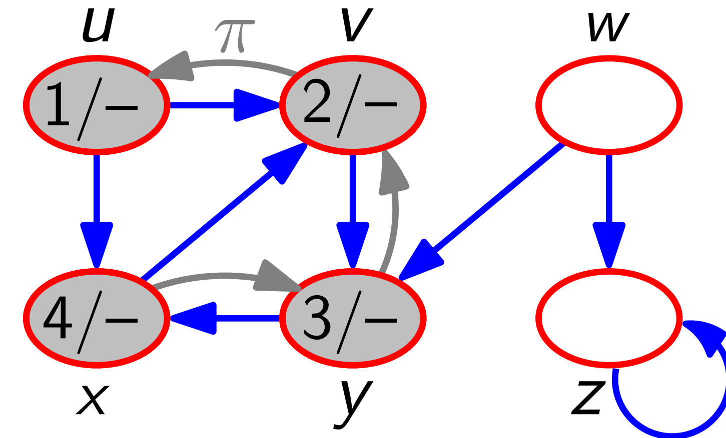


Tiefensuche

Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d/u.f)$ (*discovery time* / *finish time*)

– DFS-Wald $(\overset{\pi}{\leftarrow})$

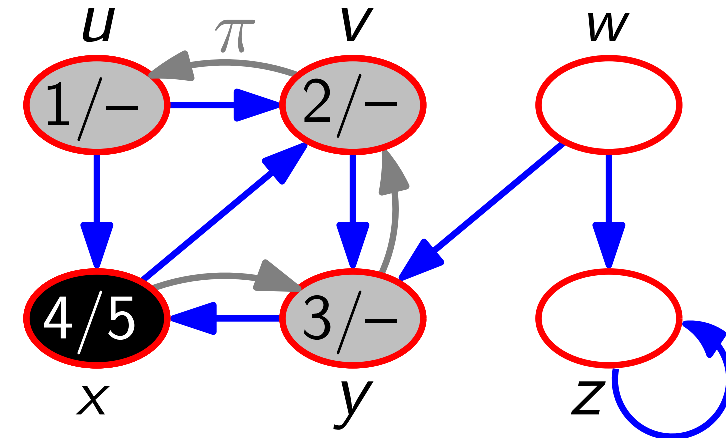


Tiefensuche

Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d / u.f)$ (*discovery time* / *finish time*)

– DFS-Wald ($\xleftarrow{\pi}$)

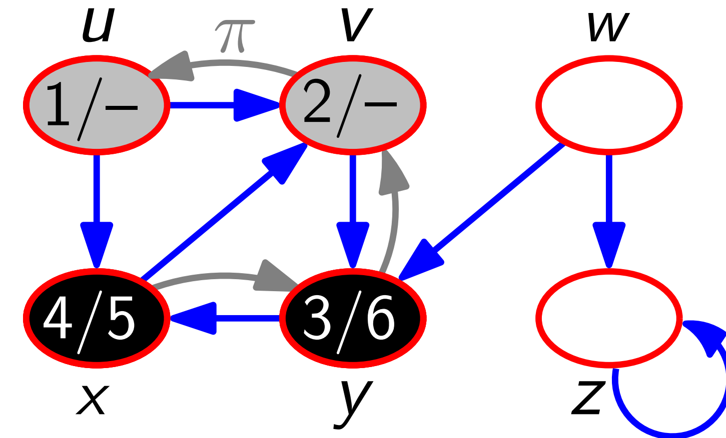


Tiefensuche

Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d/u.f)$ (*discovery time* / *finish time*)

– DFS-Wald $(\overset{\pi}{\longleftarrow})$

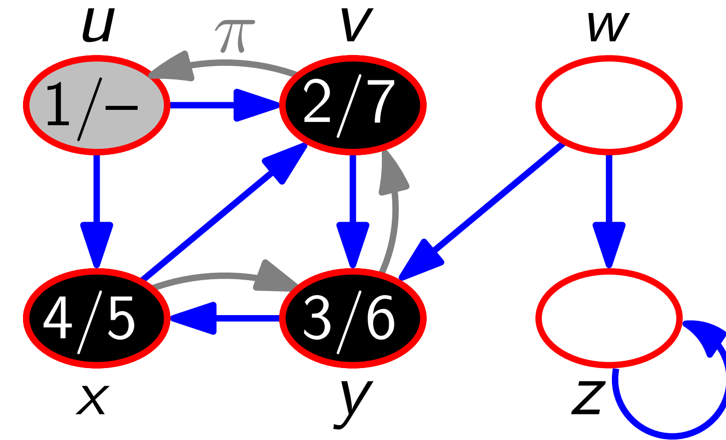


Tiefensuche

Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d/u.f)$ (*discovery time* / *finish time*)

– DFS-Wald $(\xleftarrow{\pi})$

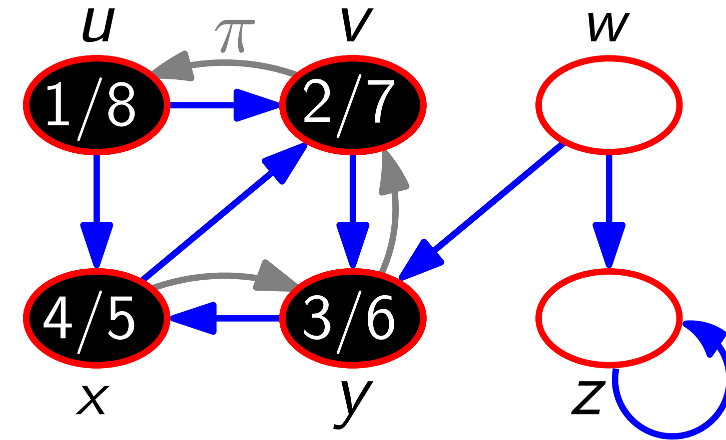


Tiefensuche

Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d / u.f)$ *discovery time* / *finish time*

– DFS-Wald $(\overset{\pi}{\longleftarrow})$

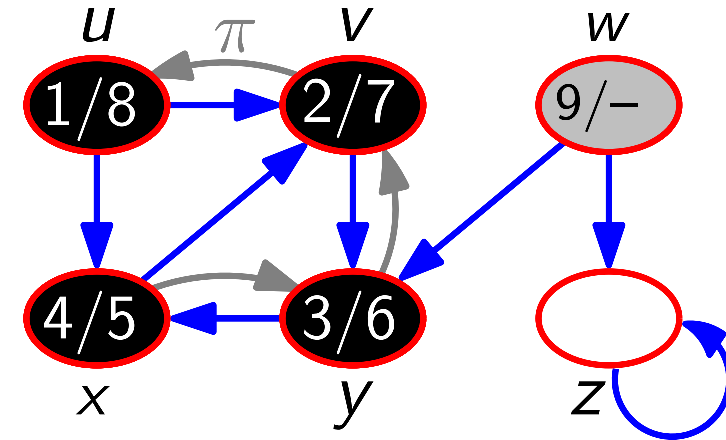


Tiefensuche

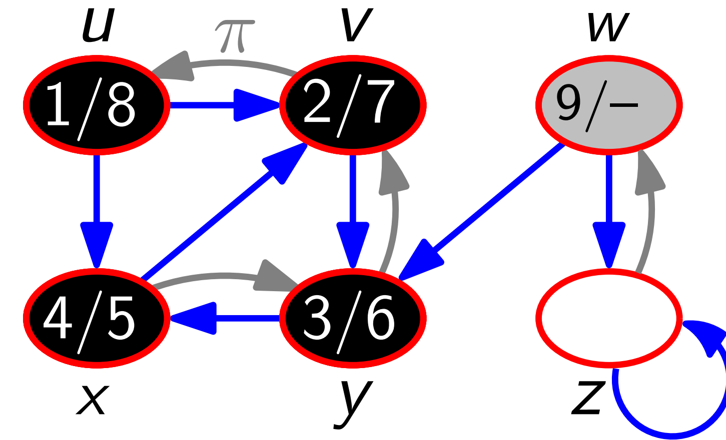
Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d / u.f)$ (*discovery time* / *finish time*)

– DFS-Wald $\left(\xleftarrow{\pi} \right)$



Tiefensuche



Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d / u.f)$ (*discovery time* / *finish time*)

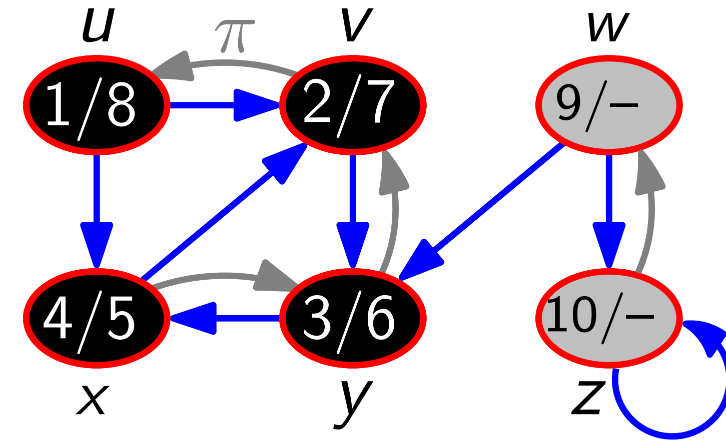
– DFS-Wald (π)

Tiefensuche

Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d/u.f)$ (discovery time / finish time)

– DFS-Wald $\left(\overset{\pi}{\longleftarrow}\right)$

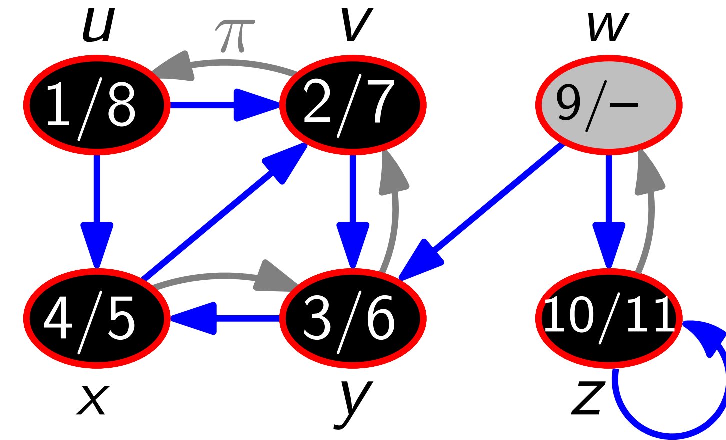


Tiefensuche

Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d / u.f)$ (discovery time / finish time)

– DFS-Wald $\left(\overset{\pi}{\longleftarrow}\right)$

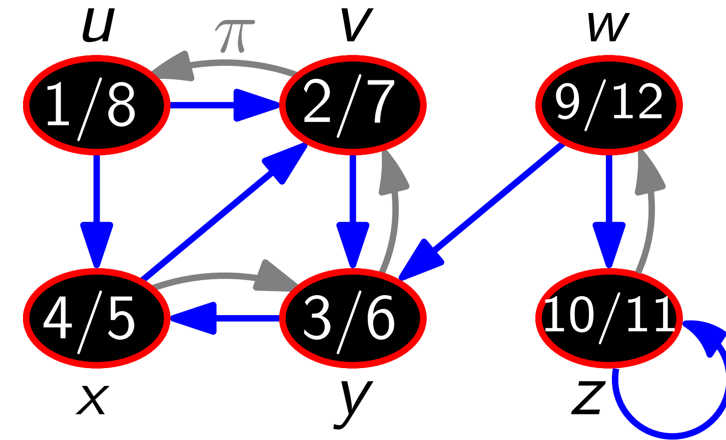


Tiefensuche

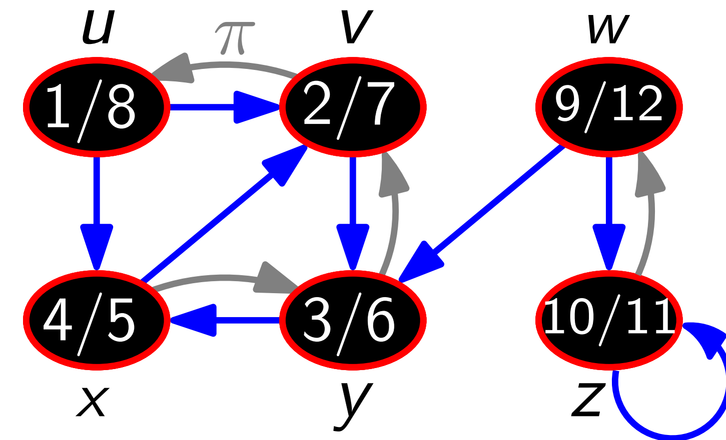
Eingabe: (un)gerichteter Graph G

Ausgabe: – Besuchsintervalle $(u.d\,/\,u.f)$ (*discovery time* / *finish time*)

– DFS-Wald $\left(\xleftarrow{\pi}\right)$



Tiefensuche



Eingabe: (un)gerichteter Graph G

Ausgabe:
– Besuchsintervalle $(u.d / u.f)$ (discovery time / finish time)
– DFS-Wald $\left(\overset{\pi}{\leftarrow}\right)$
– Klassifizierung der Graphkanten:

- Baumkanten (Kanten von G_π)

  Kanten des DFS-Waldes (entgegen π gerichtet)

- Rückwärtskanten (R)

  Nicht-Baumkanten zu einem Vorgängerknoten

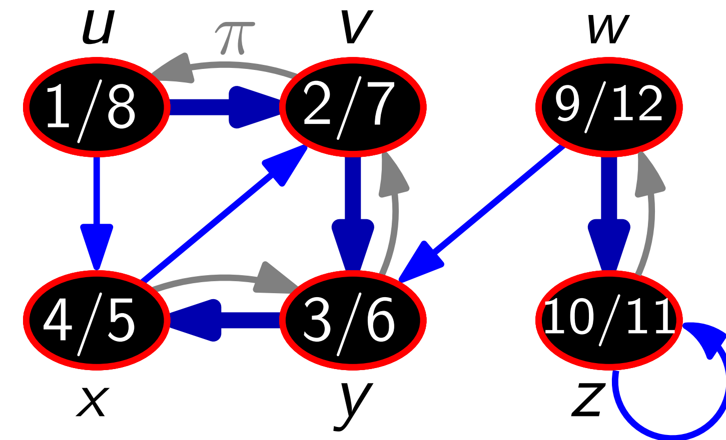
- Vorwärtskanten (V)

  Nicht-Baumkanten zu einem Nachfolgerknoten

- Kreuzkanten (K)

  Kanten, bei denen kein Endpunkt Vorgänger des anderen ist.

Tiefensuche



Eingabe: (un)gerichteter Graph G

Ausgabe:
– Besuchsintervalle ($u.d / u.f$) — *discovery time* / *finish time*
– DFS-Wald ($\overset{\pi}{\leftarrow}$)
– Klassifizierung der Graphkanten:

- Baumkanten (Kanten von G_π)

  Kanten des DFS-Waldes (entgegen π gerichtet)

- Rückwärtskanten (R)

  Nicht-Baumkanten zu einem Vorgängerknoten

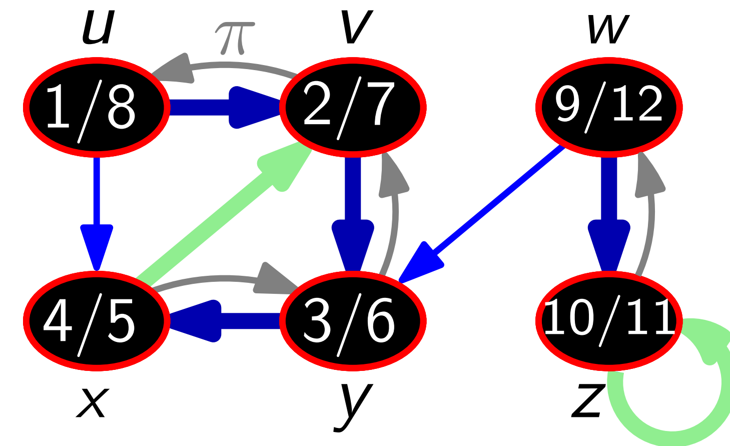
- Vorwärtskanten (V)

  Nicht-Baumkanten zu einem Nachfolgerknoten

- Kreuzkanten (K)

  Kanten, bei denen kein Endpunkt Vorgänger des anderen ist.

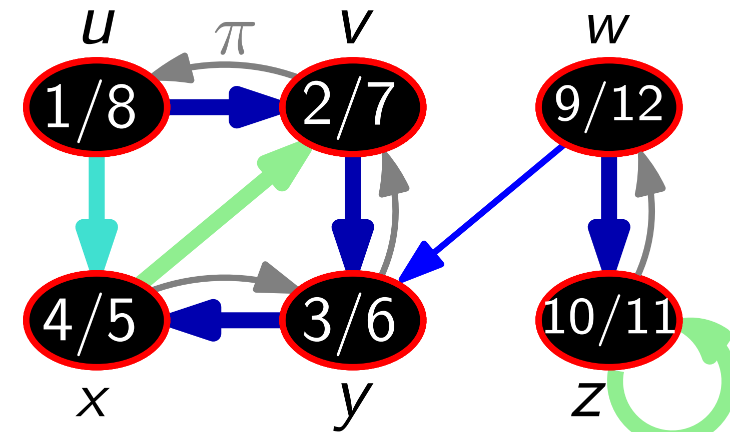
Tiefensuche



Eingabe: (un)gerichteter Graph G

Ausgabe:
- Besuchsintervalle $(u.d/u.f)$ (discovery time / finish time)
- DFS-Wald $\left(\overset{\pi}{\leftarrow}\right)$
- Klassifizierung der Graphkanten:
  - Baumkanten (Kanten von G_π)
    Kanten des DFS-Waldes (entgegen π gerichtet)
  - Rückwärtskanten (R)
    Nicht-Baumkanten zu einem Vorgängerknoten
  - Vorwärtskanten (V)
    Nicht-Baumkanten zu einem Nachfolgerknoten
  - Kreuzkanten (K)
    Kanten, bei denen kein Endpunkt Vorgänger des anderen ist.

Tiefensuche



Eingabe: (un)gerichteter Graph G

Ausgabe:
- Besuchsintervalle $(u.d / u.f)$ — *discovery time*, *finish time*
- DFS-Wald $(\overset{\pi}{\leftarrow})$
- Klassifizierung der Graphkanten:
  - Baumkanten (Kanten von G_π)

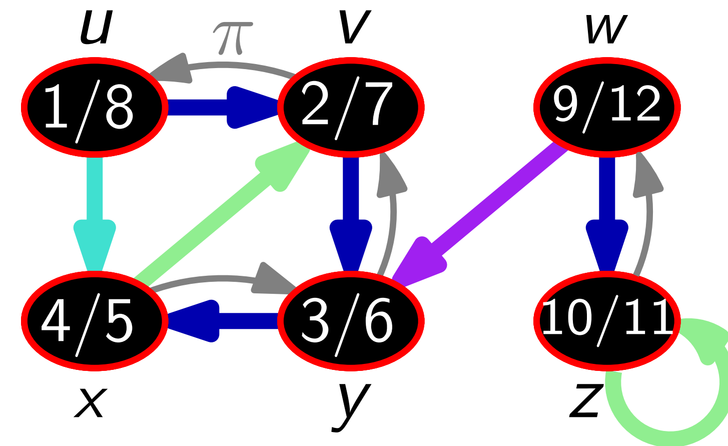
    Kanten des DFS-Waldes (entgegen π gerichtet)
  - Rückwärtskanten (R)

    Nicht-Baumkanten zu einem Vorgängerknoten
  - Vorwärtskanten (V)

    Nicht-Baumkanten zu einem Nachfolgerknoten
  - Kreuzkanten (K)

    Kanten, bei denen kein Endpunkt Vorgänger des anderen ist.

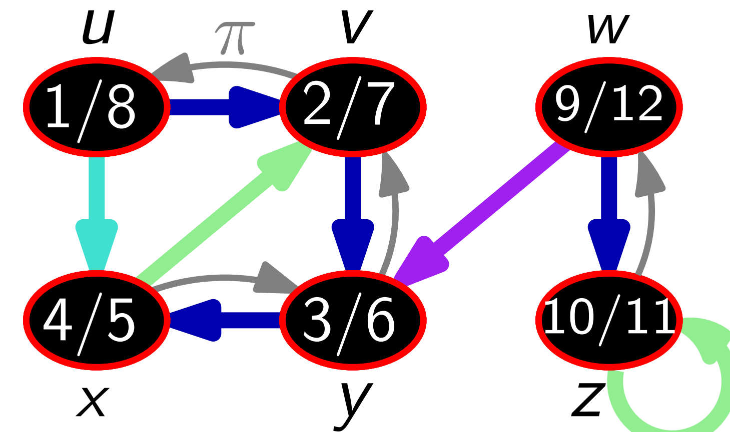
Tiefensuche



Eingabe: (un)gerichteter Graph G

Ausgabe:
- Besuchsintervalle $(u.d\,/\,u.f)$ (discovery time / finish time)
- DFS-Wald (π)
- Klassifizierung der Graphkanten:
  - Baumkanten (Kanten von G_π)
    Kanten des DFS-Waldes (entgegen π gerichtet)
  - Rückwärtskanten (R)
    Nicht-Baumkanten zu einem Vorgängerknoten
  - Vorwärtskanten (V)
    Nicht-Baumkanten zu einem Nachfolgerknoten
  - Kreuzkanten (K)
    Kanten, bei denen kein Endpunkt Vorgänger des anderen ist.

Tiefensuche



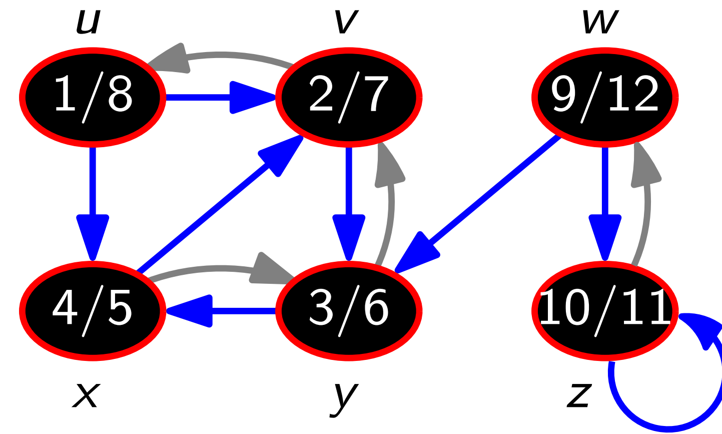
Eingabe: (un)gerichteter Graph G

Ausgabe:
- Besuchsintervalle $(u.d / u.f)$ (discovery time / finish time)
- DFS-Wald $(\xleftarrow{\pi})$
- Klassifizierung der Graphkanten:

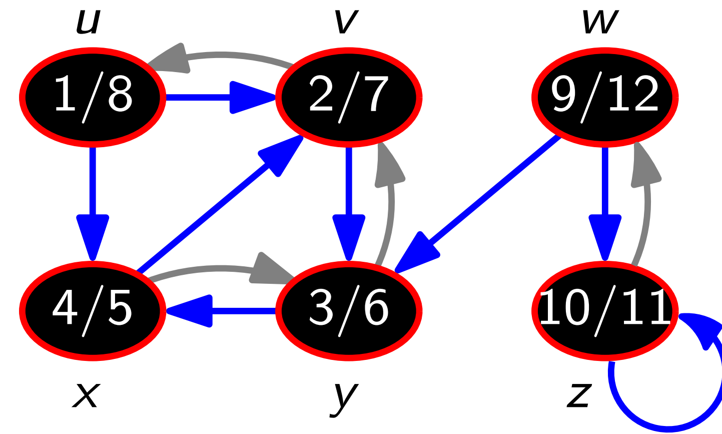
	Farbe Zielknoten:
● Baumkanten (Kanten von G_π) Kanten des DFS-Waldes (entgegen π gerichtet)	*weiss*
● Rückwärtskanten (R) Nicht-Baumkanten zu einem Vorgängerknoten	*grau*
● Vorwärtskanten (V) Nicht-Baumkanten zu einem Nachfolgerknoten	*schwarz und* start.$d <$ ziel.d
● Kreuzkanten (K) Kanten, bei denen kein Endpunkt Vorgänger des anderen ist.	*schwarz und* start.$d >$ ziel.d

Tiefensuche – Pseudocode

DFS(Graph $G = (V, E)$)
foreach $u \in V$ **do**
  $u.color = white$
  $u.\pi = nil$
$time = 0$ // globale Variable!
foreach $u \in V$ **do**
  if $u.color == white$ **then** DFSVisit(G, u)



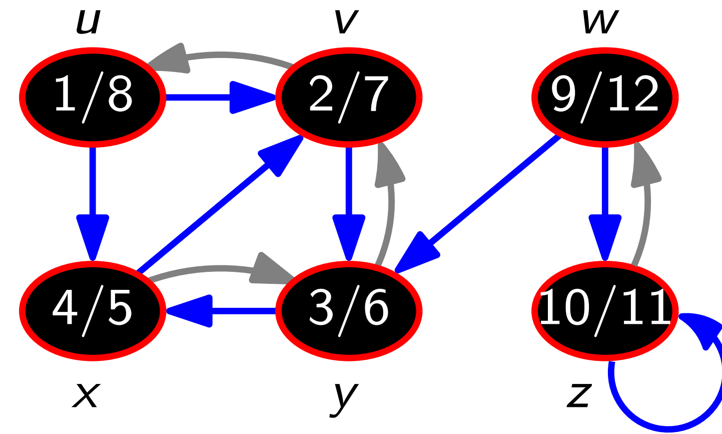
Tiefensuche – Pseudocode



DFS(Graph $G = (V, E)$)
  foreach $u \in V$ **do**
    $u.color = white$
    $u.\pi = nil$
  $time = 0$ // globale Variable!
  foreach $u \in V$ **do**
    if $u.color == white$ **then** DFSVisit(G, u)

DFSVisit(Graph G, Vertex u)
  $time = time + 1$
  $u.d = time$; $u.color = gray$

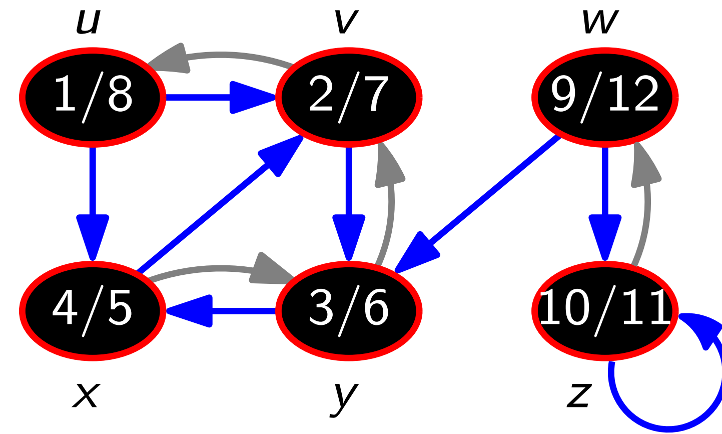
Tiefensuche – Pseudocode



DFS(Graph $G = (V, E)$)
  foreach $u \in V$ **do**
    $u.color = white$
    $u.\pi = nil$
  $time = 0$ // globale Variable!
  foreach $u \in V$ **do**
    if $u.color == white$ **then** DFSVisit(G, u)

DFSVisit(Graph G, Vertex u)
  $time = time + 1$
  $u.d = time$; $u.color = gray$
  foreach $v \in$ Adj[u] **do**

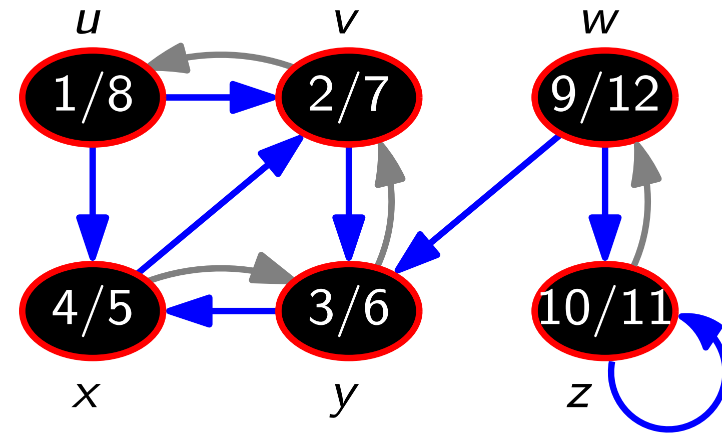
Tiefensuche – Pseudocode




```
DFS(Graph G = (V, E))
  foreach u ∈ V do
    u.color = white
    u.π = nil
  time = 0    // globale Variable!
  foreach u ∈ V do
    if u.color == white then DFSVisit(G, u)
```

```
DFSVisit(Graph G, Vertex u)
  time = time + 1
  u.d = time; u.color = gray
  foreach v ∈ Adj[u] do
    if v.color == white then
      v.π = u; DFSVisit(G, v)
```

Tiefensuche – Pseudocode



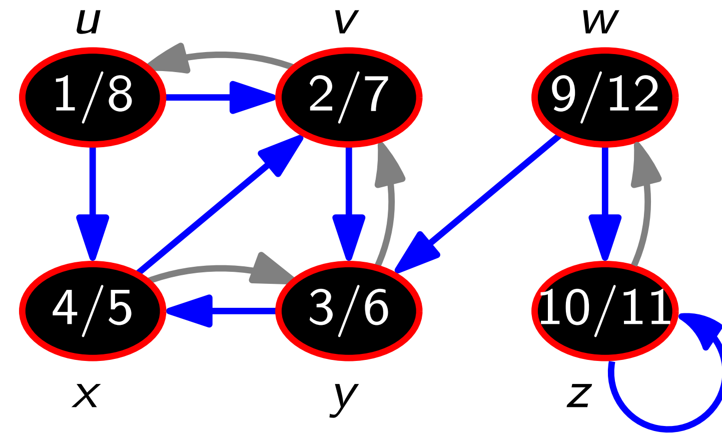
DFS(Graph $G = (V, E)$)
  foreach $u \in V$ **do**
    $u.color = white$
    $u.\pi = nil$
  $time = 0$ // globale Variable!
  foreach $u \in V$ **do**
    if $u.color == white$ **then** DFSVisit(G, u)

DFSVisit(Graph G, Vertex u)
  $time = time + 1$
  $u.d = time$; $u.color = gray$
  foreach $v \in \text{Adj}[u]$ **do**
    if $v.color == white$ **then**
      $v.\pi = u$; DFSVisit(G, v)
  $time = time + 1$
  $u.f = time$; $u.color = black$

Tiefensuche – Pseudocode

```
DFS(Graph G = (V, E))
  foreach u ∈ V do
    u.color = white
    u.π = nil
  time = 0      // globale Variable!
  foreach u ∈ V do
    if u.color == white then DFSVisit(G, u)
```

```
DFSVisit(Graph G, Vertex u)
  time = time + 1
  u.d = time; u.color = gray
  foreach v ∈ Adj[u] do
    if v.color == white then
      v.π = u; DFSVisit(G, v)
  time = time + 1
  u.f = time; u.color = black
```

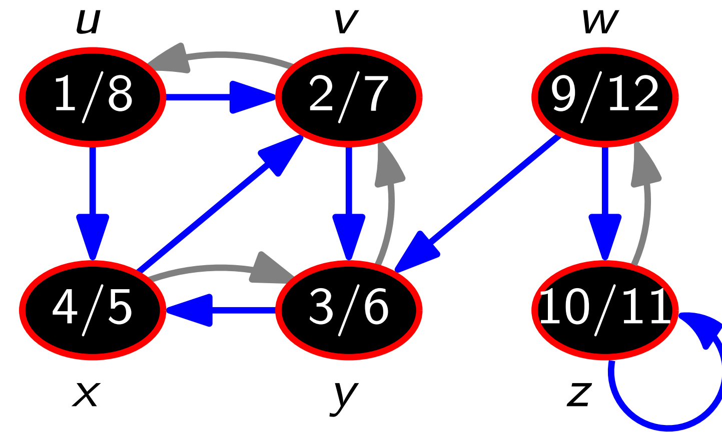


Laufzeit?

Tiefensuche – Pseudocode

DFS(Graph $G = (V, E)$)
  foreach $u \in V$ **do**
    $u.color = white$
    $u.\pi = nil$
  $time = 0$ // globale Variable!
  foreach $u \in V$ **do**
    if $u.color == white$ **then** DFSVisit(G, u)

DFSVisit(Graph G, Vertex u)
  $time = time + 1$
  $u.d = time$; $u.color = gray$
  foreach $v \in \text{Adj}[u]$ **do**
    if $v.color == white$ **then**
      $v.\pi = u$; DFSVisit(G, v)
  $time = time + 1$
  $u.f = time$; $u.color = black$



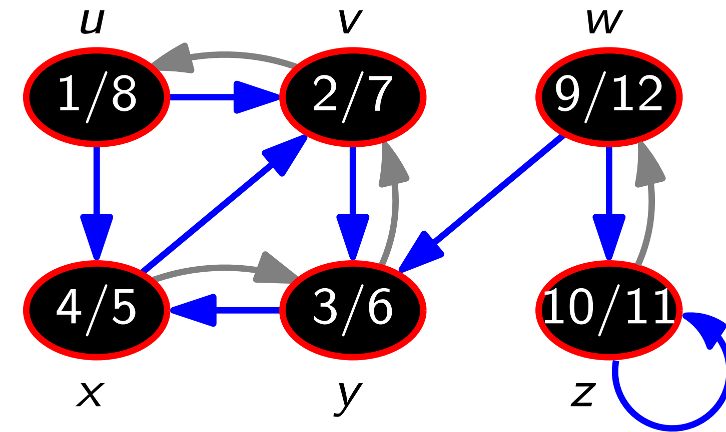
Laufzeit?

- DFSVisit wird nur für weiße Knoten aufgerufen.

Tiefensuche – Pseudocode

DFS(Graph $G = (V, E)$)
  foreach $u \in V$ **do**
    $u.color = white$
    $u.\pi = nil$
  $time = 0$ // globale Variable!
  foreach $u \in V$ **do**
    if $u.color == white$ **then** DFSVisit(G, u)

DFSVisit(Graph G, Vertex u)
  $time = time + 1$
  $u.d = time$; $u.color = gray$
  foreach $v \in \text{Adj}[u]$ **do**
    if $v.color == white$ **then**
      $v.\pi = u$; DFSVisit(G, v)
  $time = time + 1$
  $u.f = time$; $u.color = black$



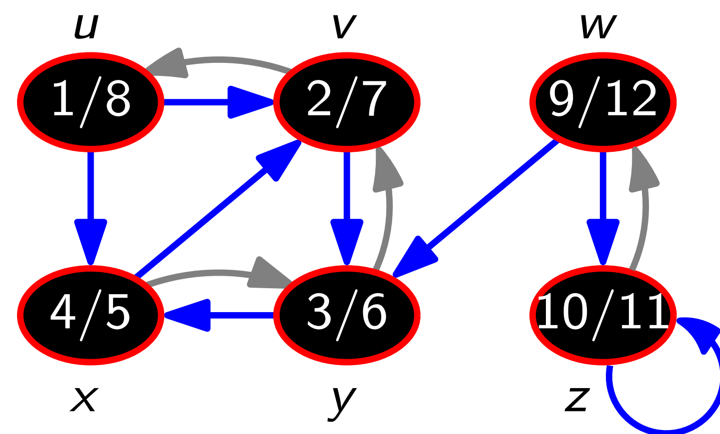
Laufzeit?

- DFSVisit wird nur für weiße Knoten aufgerufen.
- In DFSVisit wird der neue Knoten sofort gefärbt.

Tiefensuche – Pseudocode

DFS(Graph $G = (V, E)$)
  foreach $u \in V$ **do**
    $u.color = white$
    $u.\pi = nil$
  $time = 0$ // globale Variable!
  foreach $u \in V$ **do**
    if $u.color == white$ **then** DFSVisit(G, u)

DFSVisit(Graph G, Vertex u)
  $time = time + 1$
  $u.d = time$; $u.color = gray$
  foreach $v \in \text{Adj}[u]$ **do**
    if $v.color == white$ **then**
      $v.\pi = u$; DFSVisit(G, v)
  $time = time + 1$
  $u.f = time$; $u.color = black$



Laufzeit?

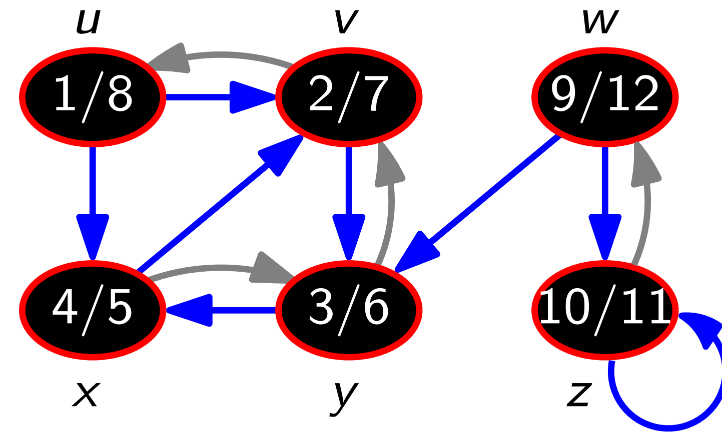
- DFSVisit wird nur für weiße Knoten aufgerufen.
- In DFSVisit wird der neue Knoten sofort gefärbt.

$\Rightarrow$ DFSVisit wird für jeden Knoten genau $1\times$ aufgerufen.

Tiefensuche – Pseudocode

```
DFS(Graph G = (V, E))
   foreach u ∈ V do
      u.color = white
      u.π = nil
   time = 0      // globale Variable!
   foreach u ∈ V do
      if u.color == white then DFSVisit(G, u)
```

```
DFSVisit(Graph G, Vertex u)
   time = time + 1
   u.d = time; u.color = gray
   foreach v ∈ Adj[u] do
      if v.color == white then
         v.π = u; DFSVisit(G, v)
   time = time + 1
   u.f = time; u.color = black
```



Laufzeit?

- DFSVisit wird nur für weiße Knoten aufgerufen.
- In DFSVisit wird der neue Knoten sofort gefärbt.

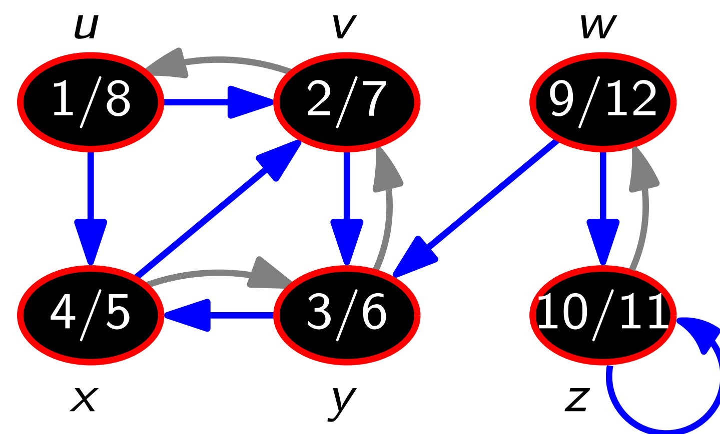
$\Rightarrow$ DFSVisit wird für jeden Knoten genau $1\times$ aufgerufen.

- Jede Kante wird *insgesamt* höchstens $2\times$ betrachtet.

Tiefensuche – Pseudocode

DFS(Graph $G = (V, E)$)
  foreach $u \in V$ **do**
    $u.color = white$
    $u.\pi = nil$
  $time = 0$ // globale Variable!
  foreach $u \in V$ **do**
    if $u.color == white$ **then** DFSVisit(G, u)

DFSVisit(Graph G, Vertex u)
  $time = time + 1$
  $u.d = time$; $u.color = gray$
  foreach $v \in \text{Adj}[u]$ **do**
    if $v.color == white$ **then**
      $v.\pi = u$; DFSVisit(G, v)
  $time = time + 1$
  $u.f = time$; $u.color = black$



Laufzeit?

- DFSVisit wird nur für weiße Knoten aufgerufen.
- In DFSVisit wird der neue Knoten sofort gefärbt.

$\Rightarrow$ DFSVisit wird für jeden Knoten genau $1\times$ aufgerufen.

- Jede Kante wird *insgesamt* höchstens $2\times$ betrachtet.

DFS gesamt $O(V + E)$ Zeit

Tiefensuche – Anwendung

Topologische Sortierung: Lineare Ordnung der Knoten, so dass aus $(u, v) \in E$ folgt: u kommt vor v.

Tiefensuche – Anwendung

Topologische Sortierung: Lineare Ordnung der Knoten, so dass aus $(u, v) \in E$ folgt: u kommt vor v.

$\Rightarrow$ kreisfrei

Tiefensuche – Anwendung

Topologische Sortierung: Lineare Ordnung der Knoten, so dass aus $(u, v) \in E$ folgt: u kommt vor v.

$\Rightarrow$ kreisfrei

TopologicalSort(DirectedGraph G)
  $L =$ **new** List()
  DFS(G) mit folgender Änderung:
    Wenn ein Knoten schwarz gefärbt wird,
    häng ihn *vorne* an die Liste L an.
  return L

Tiefensuche – Anwendung

Topologische Sortierung: Lineare Ordnung der Knoten, so dass aus $(u, v) \in E$ folgt: u kommt vor v.

$\Rightarrow$ kreisfrei

TopologicalSort(DirectedGraph G)
  L = **new** List()
  DFS(G) mit folgender Änderung:
    Wenn ein Knoten schwarz gefärbt wird,
    häng ihn *vorne* an die Liste L an.
  return L

Laufzeit?
$O(V + E)$

Tiefensuche – Anwendung

Topologische Sortierung: Lineare Ordnung der Knoten, so dass aus $(u, v) \in E$ folgt: u kommt vor v.

$\Rightarrow$ kreisfrei

TopologicalSort(DirectedGraph G)
  L = **new** List()
  DFS(G) mit folgender Änderung:
    Wenn ein Knoten schwarz gefärbt wird,
    häng ihn *vorne* an die Liste L an.
  return L

Laufzeit?
$O(V + E)$

Einen Graphen topologisch zu sortieren ist ein wichtiger Vorverarbeitungsschritt bei der Lösung vieler Probleme –

Tiefensuche – Anwendung

Topologische Sortierung: Lineare Ordnung der Knoten, so dass aus $(u, v) \in E$ folgt: u kommt vor v.

$\Rightarrow$ kreisfrei

TopologicalSort(DirectedGraph G)
  $L =$ **new** List()
  DFS(G) mit folgender Änderung:
    Wenn ein Knoten schwarz gefärbt wird,
    häng ihn *vorne* an die Liste L an.
  return L

Laufzeit?
$O(V + E)$

Einen Graphen topologisch zu sortieren ist ein wichtiger Vorverarbeitungsschritt bei der Lösung vieler Probleme – z.B. in der Ablaufplanung, wo gerichtete Kanten Abhängigkeiten von Aufträgen ausdrücken.

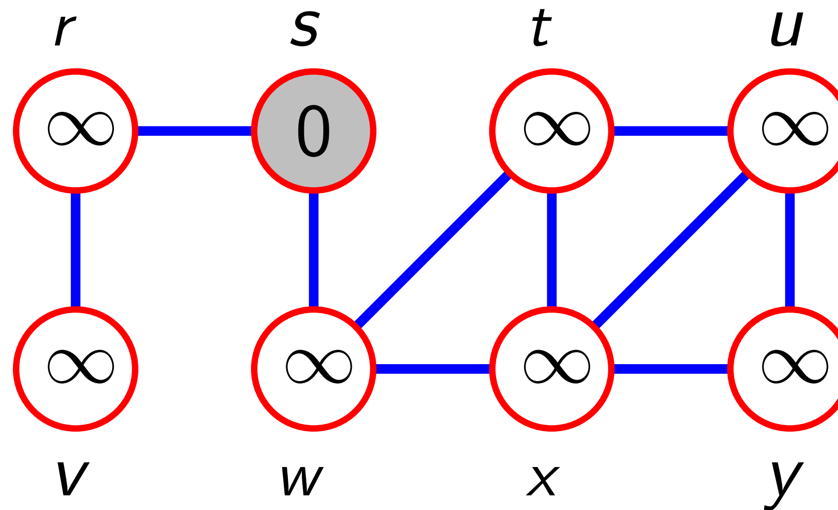
Übersicht

1. Graphdurchlaufstrategien
   1.1 Tiefensuche
   1.2 Breitensuche
       Beispiel
       Pseudocode
       Anwendung
2. Kürzeste Wege
3. Minimale Spannbäume

Breitensuche – Beispiel

Eingabe: (un)gerichteter Graph G

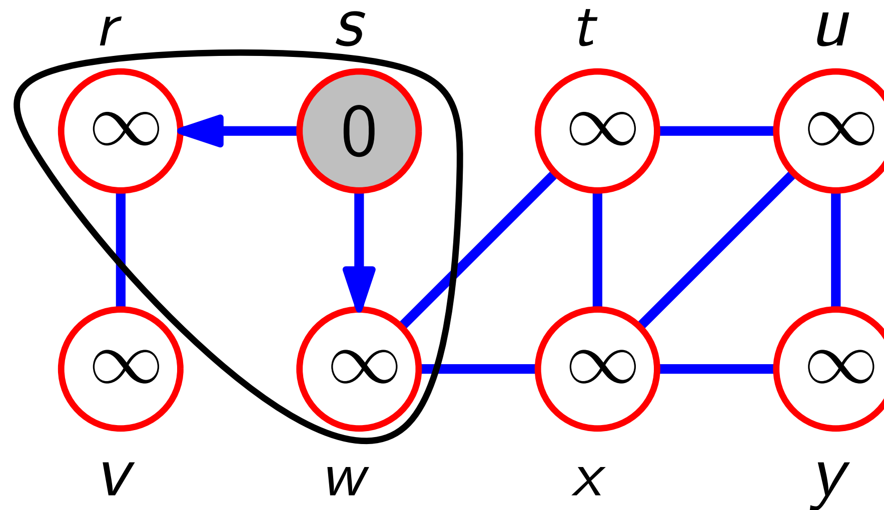
Ausgabe: – Abstand vom Startknoten
– BFS-Wald

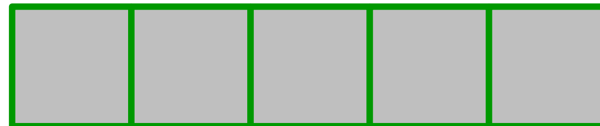


Breitensuche – Beispiel

Eingabe: (un)gerichteter Graph G

Ausgabe: – Abstand vom Startknoten
– BFS-Wald

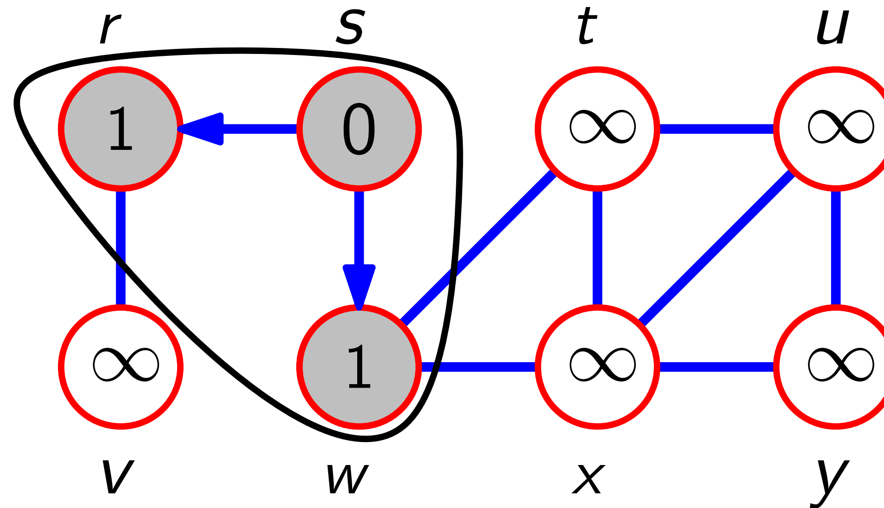


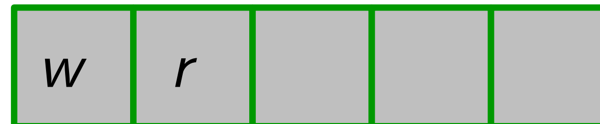


Breitensuche – Beispiel

Eingabe: (un)gerichteter Graph G

Ausgabe: – Abstand vom Startknoten
– BFS-Wald

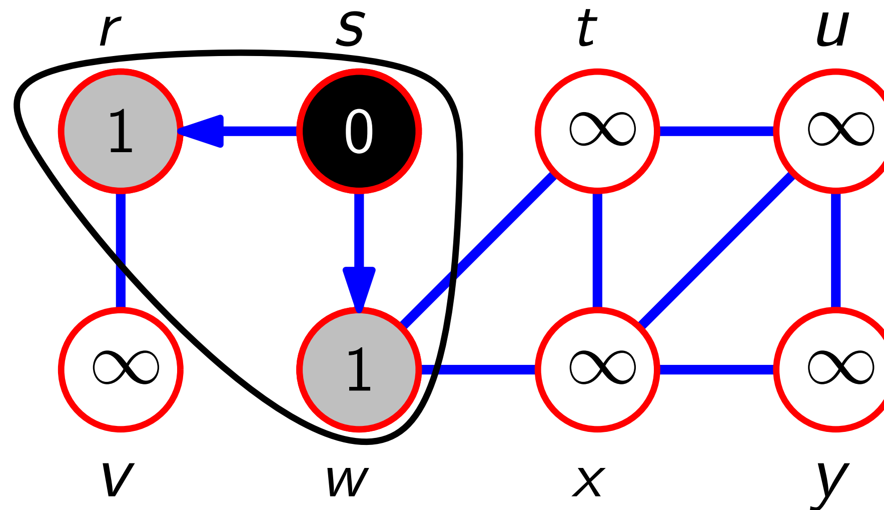


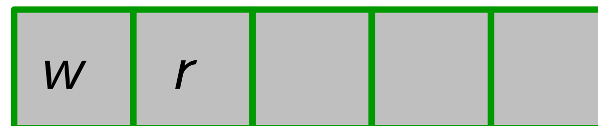


Breitensuche – Beispiel

Eingabe: (un)gerichteter Graph G

Ausgabe: – Abstand vom Startknoten
– BFS-Wald



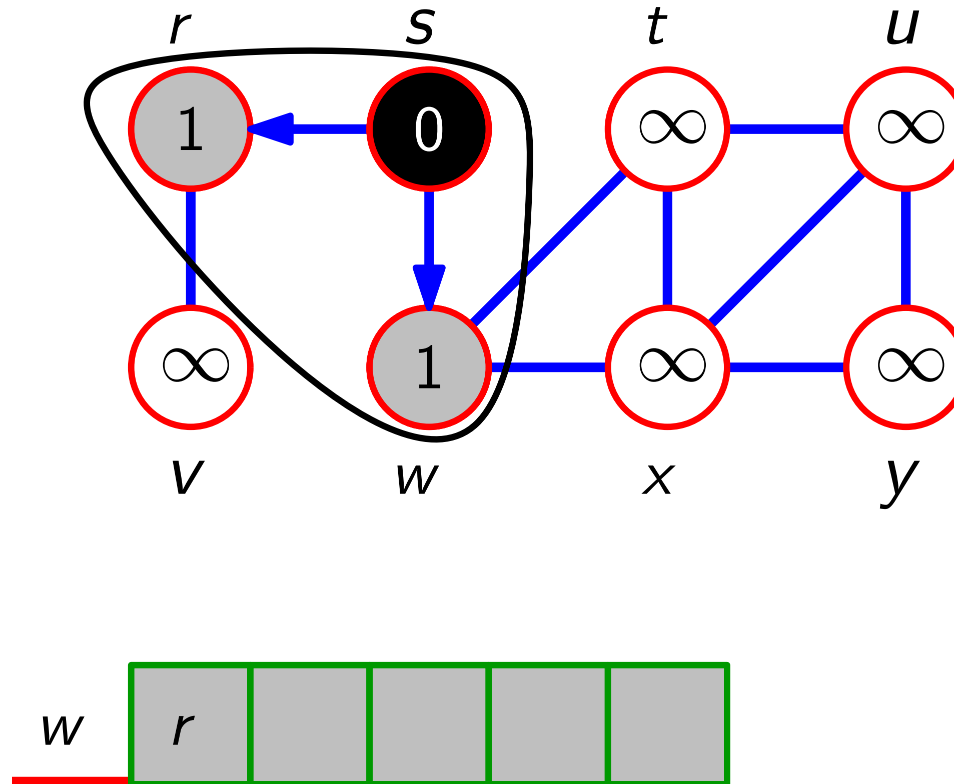


Breitensuche – Beispiel

Eingabe: (un)gerichteter Graph G

Ausgabe: – Abstand vom Startknoten

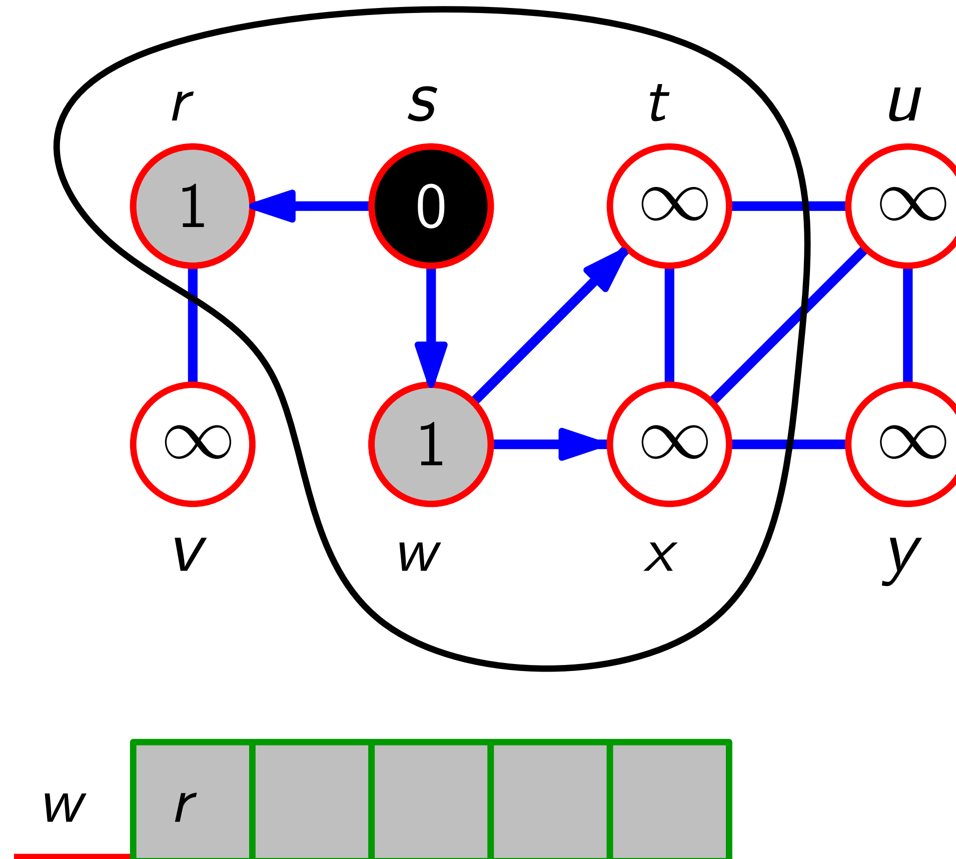
– BFS-Wald



Breitensuche – Beispiel

Eingabe: (un)gerichteter Graph G

Ausgabe: – Abstand vom Startknoten
– BFS-Wald

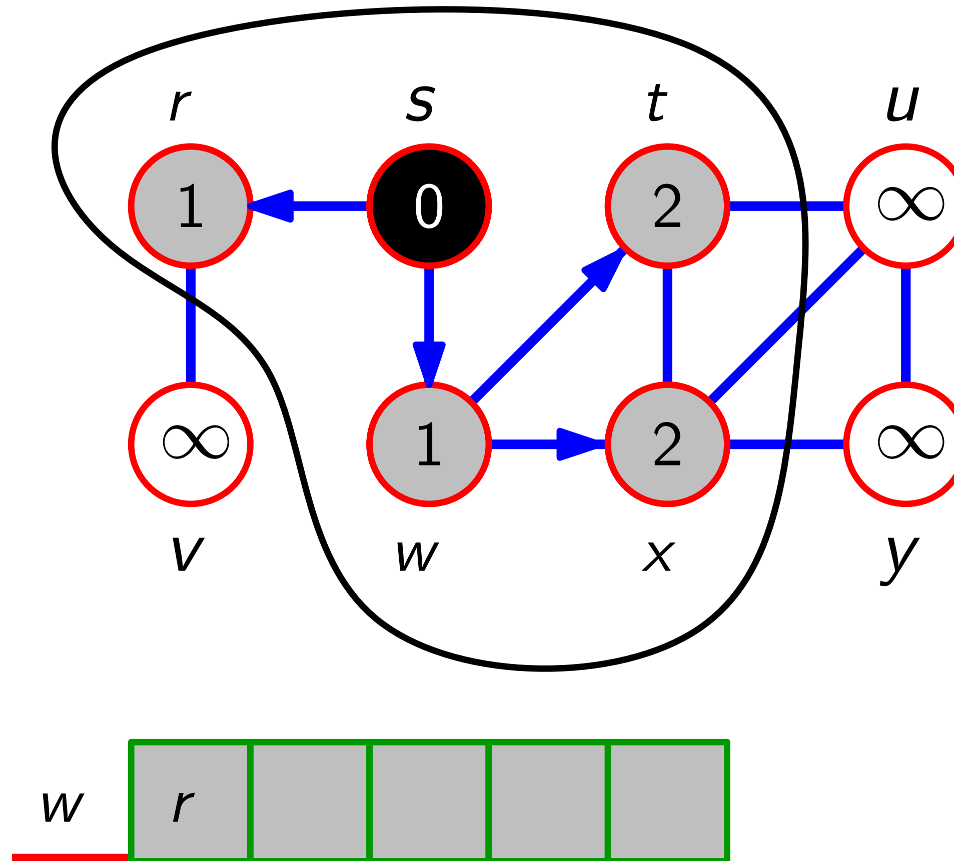


Breitensuche – Beispiel

Eingabe: (un)gerichteter Graph G

Ausgabe: – Abstand vom Startknoten

– BFS-Wald

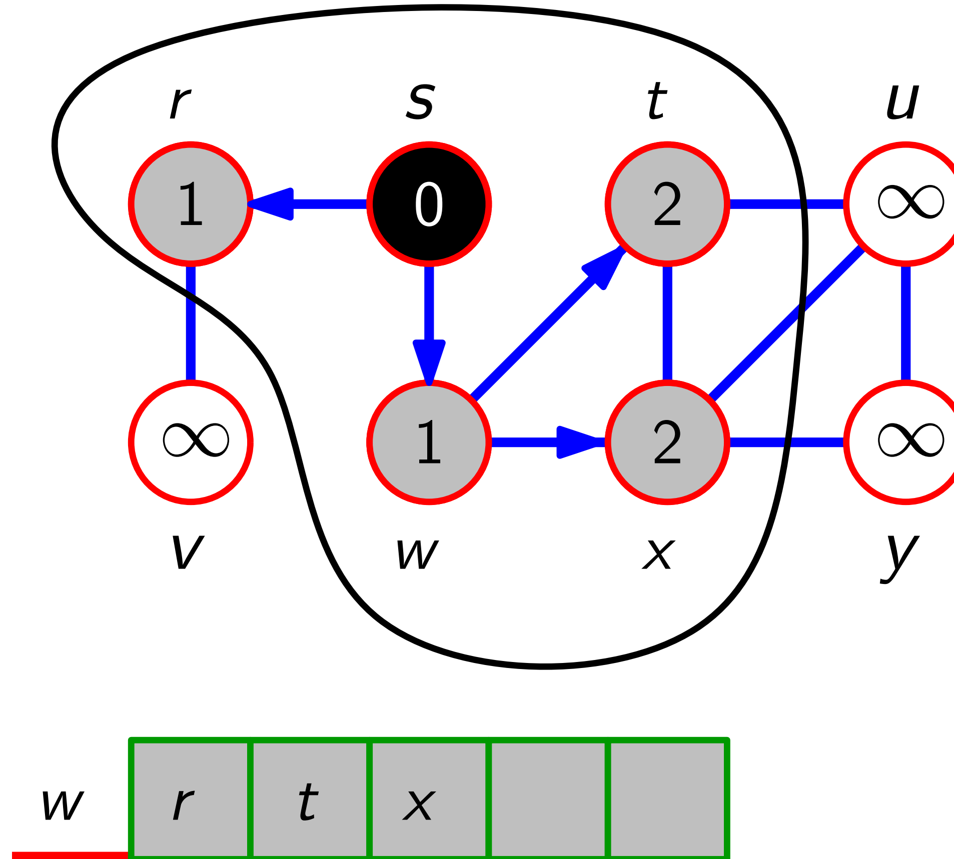


Breitensuche – Beispiel

Eingabe: (un)gerichteter Graph G

Ausgabe: – Abstand vom Startknoten

– BFS-Wald

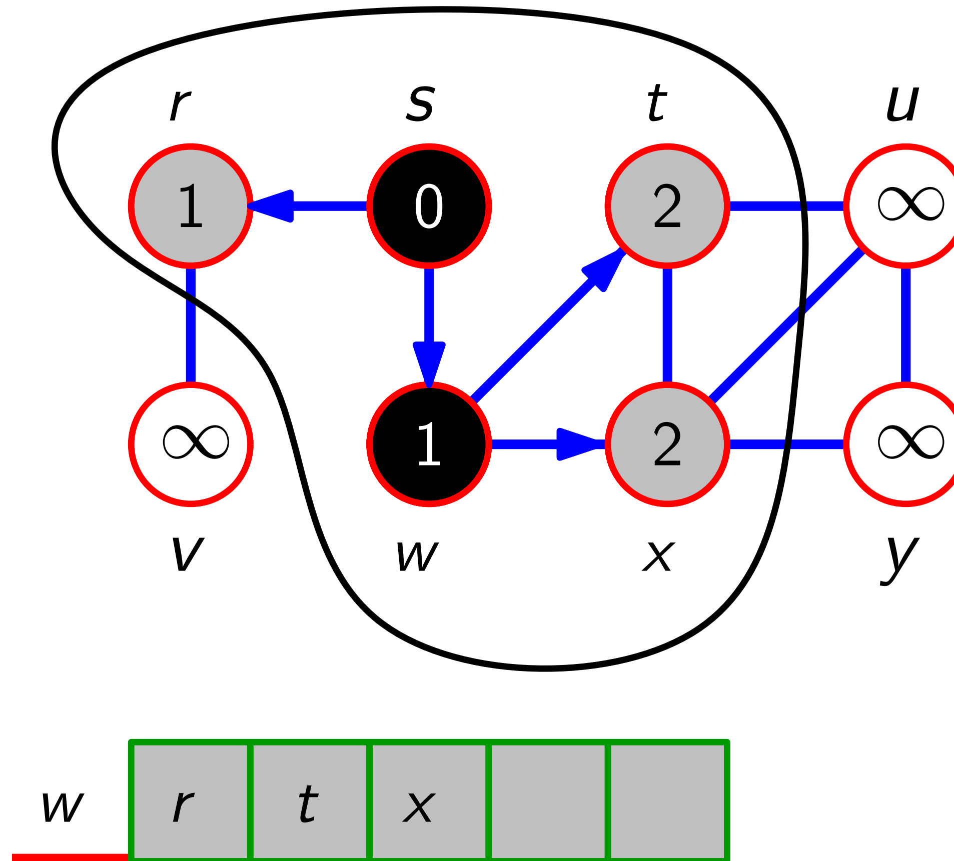


Breitensuche – Beispiel

Eingabe: (un)gerichteter Graph G

Ausgabe: – Abstand vom Startknoten

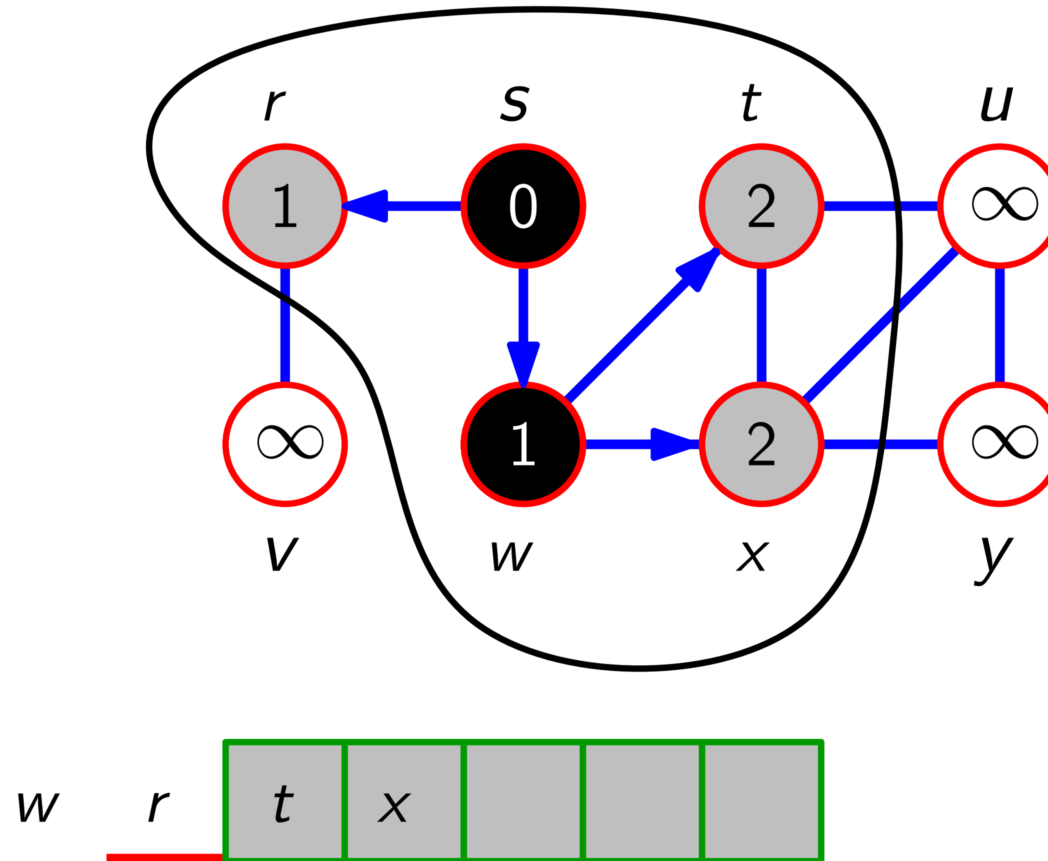
– BFS-Wald



Breitensuche – Beispiel

Eingabe: (un)gerichteter Graph G

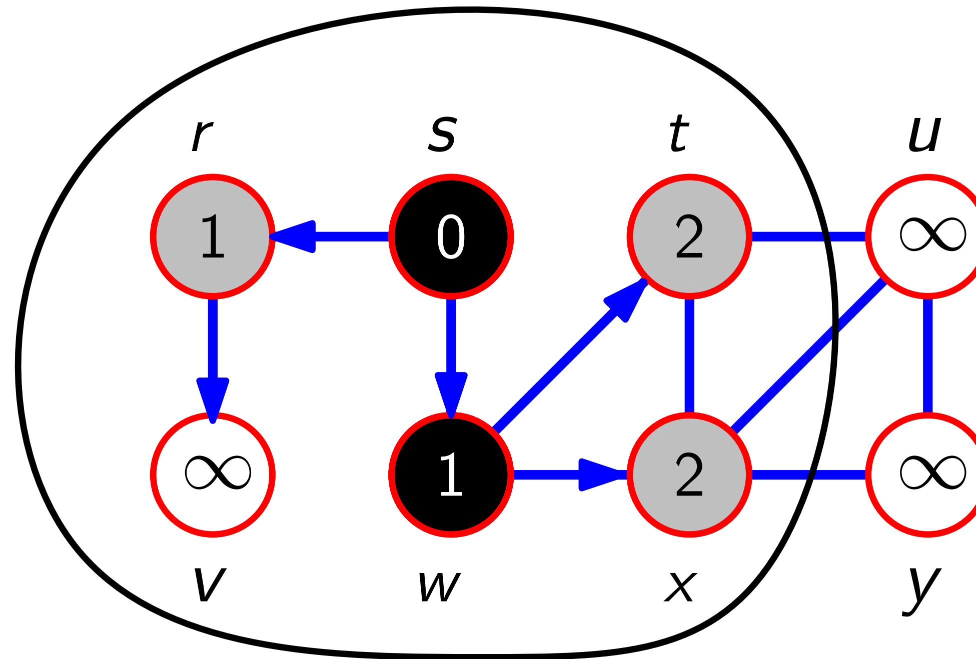
Ausgabe: – Abstand vom Startknoten
– BFS-Wald

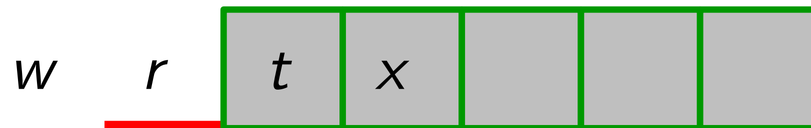


Breitensuche – Beispiel

Eingabe: (un)gerichteter Graph G

Ausgabe: – Abstand vom Startknoten
– BFS-Wald

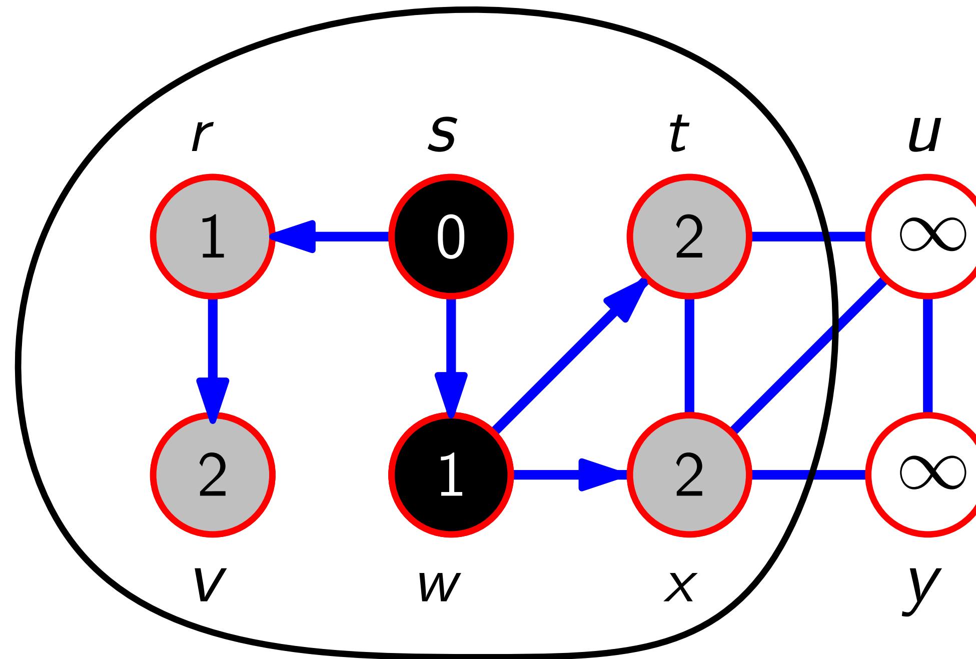


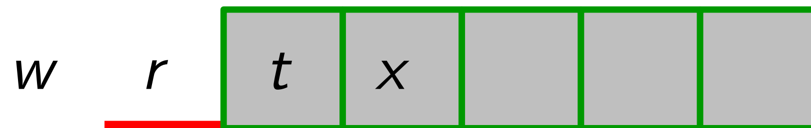


Breitensuche – Beispiel

Eingabe: (un)gerichteter Graph G

Ausgabe: – Abstand vom Startknoten
– BFS-Wald

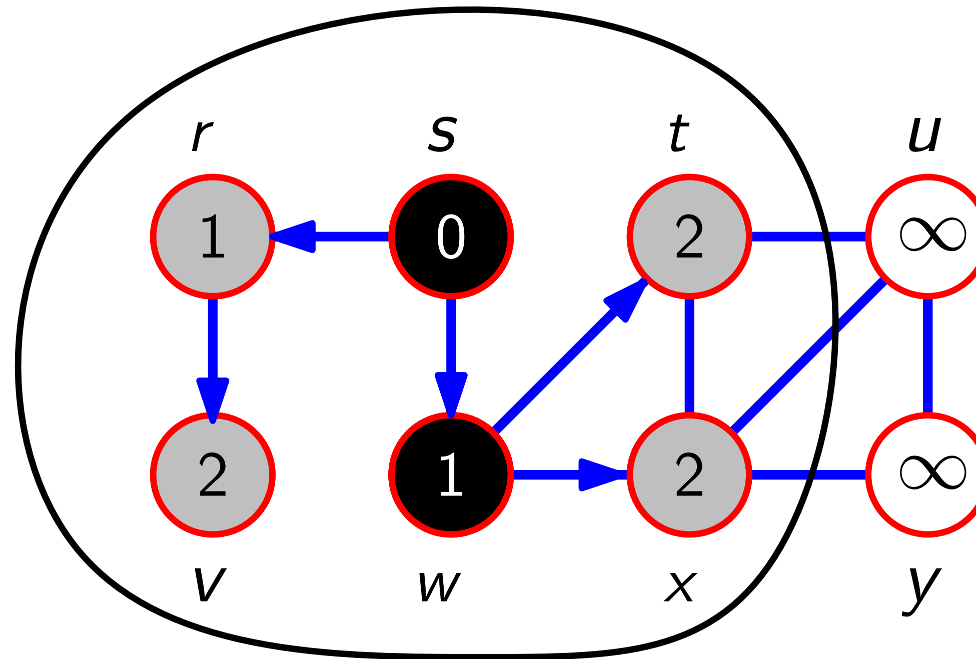


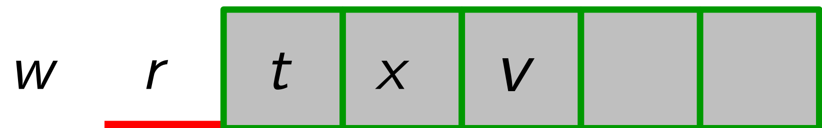


Breitensuche – Beispiel

Eingabe: (un)gerichteter Graph G

Ausgabe: – Abstand vom Startknoten
– BFS-Wald



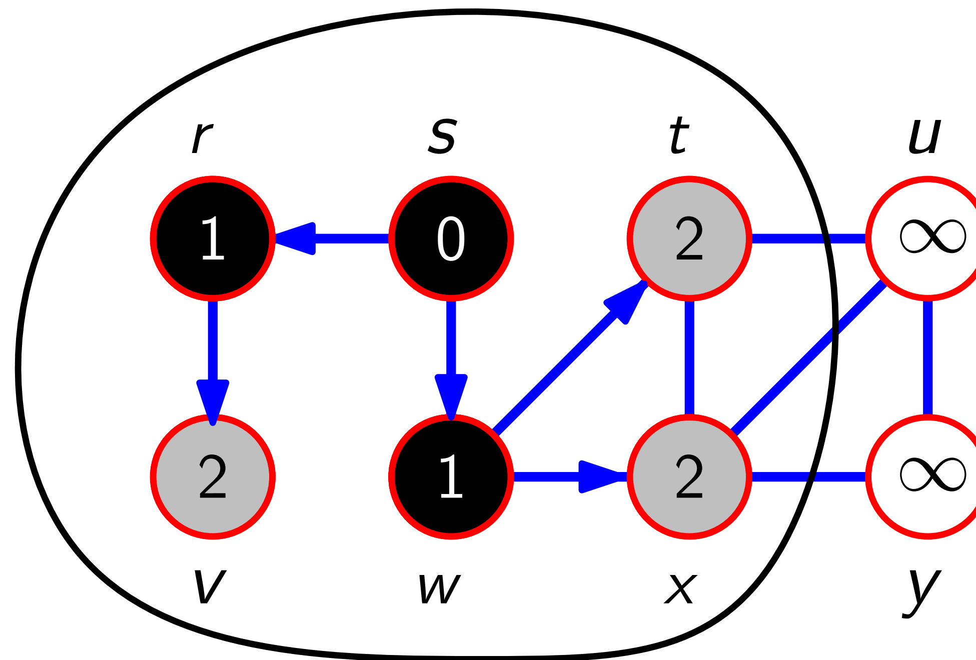


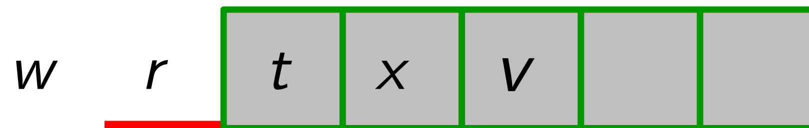
Breitensuche – Beispiel

Eingabe: (un)gerichteter Graph G

Ausgabe: – Abstand vom Startknoten

– BFS-Wald

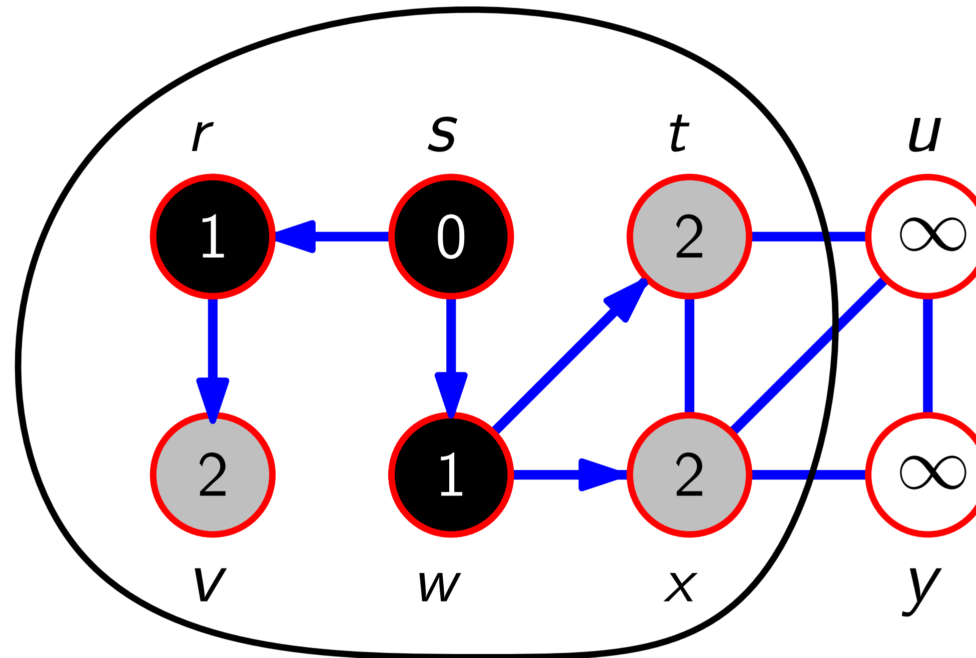


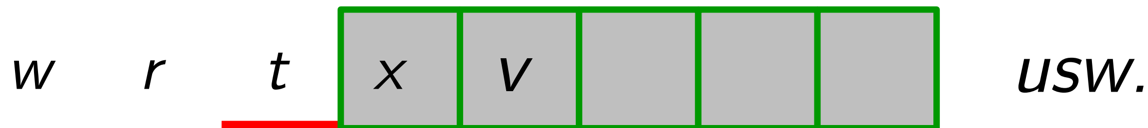


Breitensuche – Beispiel

Eingabe: (un)gerichteter Graph G

Ausgabe: – Abstand vom Startknoten
– BFS-Wald





usw.

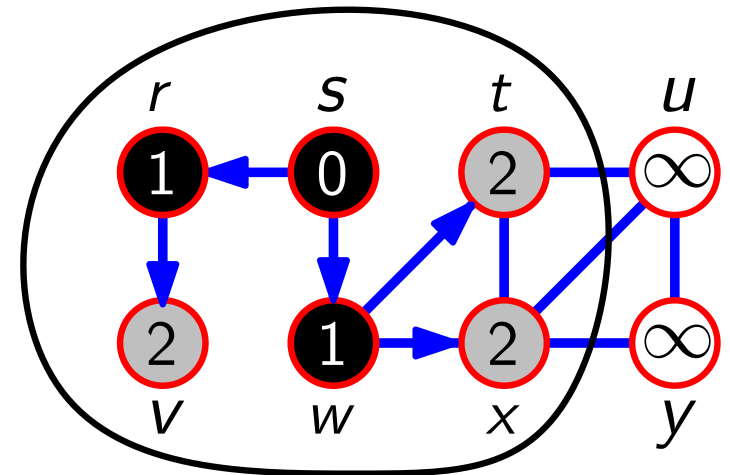
Breitensuche – Pseudocode

r s t u
1 0 2 ∞
2 1 2 ∞
v w x y

w r | t | x | v | | | usw.

Initialize(Graph G, Vertex s)
foreach $u \in V$ **do**
$u.color = white$
$u.d = \infty$
$u.\pi = nil$
$s.color = gray$
$s.d = 0$

Breitensuche – Pseudocode



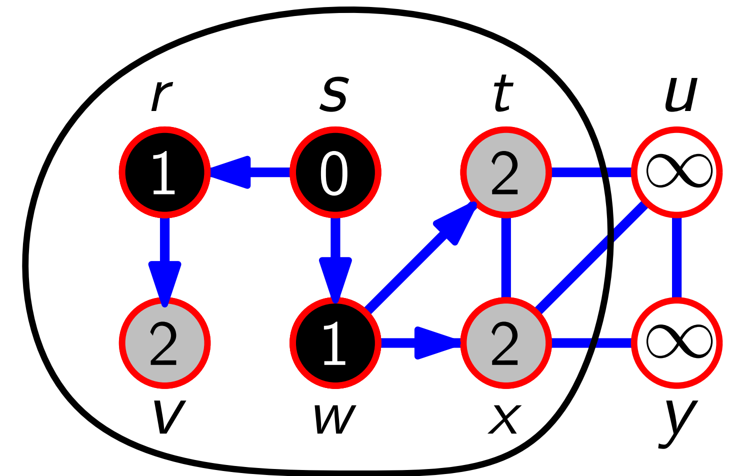
w r | t | x | v | | | usw.

BFS(Graph G, Vertex s)
  Initialize(G, s)
  Q = new Queue()

Initialize(Graph G, Vertex s)
  foreach $u \in V$ **do**
    $u.color = white$
    $u.d = \infty$
    $u.\pi = nil$
  $s.color = gray$
  $s.d = 0$

Breitensuche – Pseudocode

BFS(Graph G, Vertex s)
  Initialize(G, s)
  Q = new Queue()
  Q.Enqueue(s)



w r t x v usw.

Initialize(Graph G, Vertex s)
  foreach $u \in V$ **do**
    $u.color = white$
    $u.d = \infty$
    $u.\pi = nil$
  $s.color = gray$
  $s.d = 0$

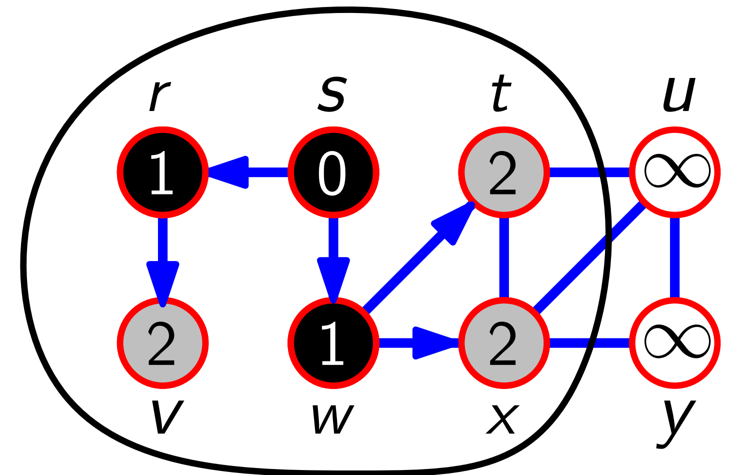
Breitensuche – Pseudocode

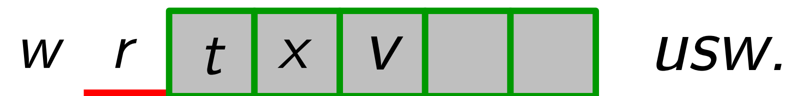
BFS(Graph G, Vertex s)
  Initialize(G, s)
  Q = new Queue()
  Q.Enqueue(s)
  while $Q \neq \emptyset$ **do**

Initialize(Graph G, Vertex s)
  foreach $u \in V$ **do**
    $u.color = white$
    $u.d = \infty$
    $u.\pi = nil$
  $s.color = gray$
  $s.d = 0$

Breitensuche – Pseudocode

BFS(Graph G, Vertex s)
  Initialize(G, s)
  Q = new Queue()
  Q.Enqueue(s)
  while $Q \neq \emptyset$ **do**
    $u = Q$.Dequeue()





Initialize(Graph G, Vertex s)
  foreach $u \in V$ **do**
    $u.color = white$
    $u.d = \infty$
    $u.\pi = nil$
  $s.color = gray$
  $s.d = 0$

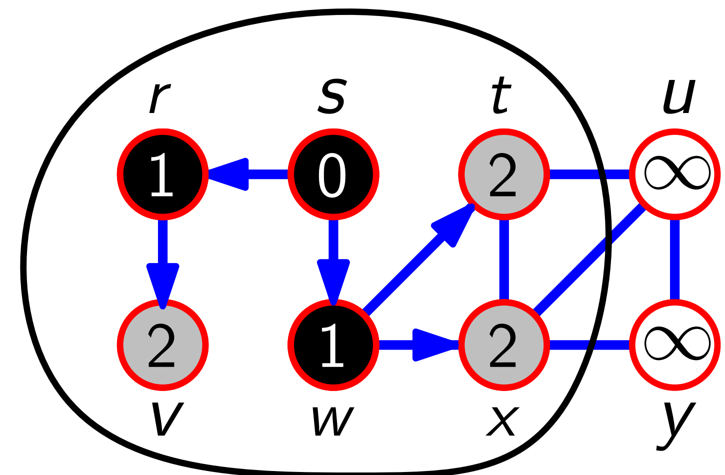
Breitensuche – Pseudocode

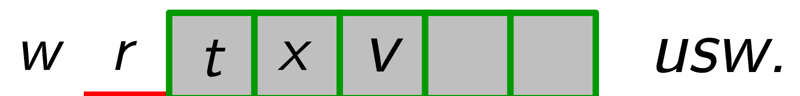
```
BFS(Graph G, Vertex s)
   Initialize(G, s)
   Q = new Queue()
   Q.Enqueue(s)
   while Q ≠ ∅ do
      u = Q.Dequeue()
      foreach v ∈ Adj[u] do
```



```
Initialize(Graph G, Vertex s)
   foreach u ∈ V do
      u.color = white
      u.d = ∞
      u.π = nil
   s.color = gray
   s.d = 0
```

Breitensuche – Pseudocode

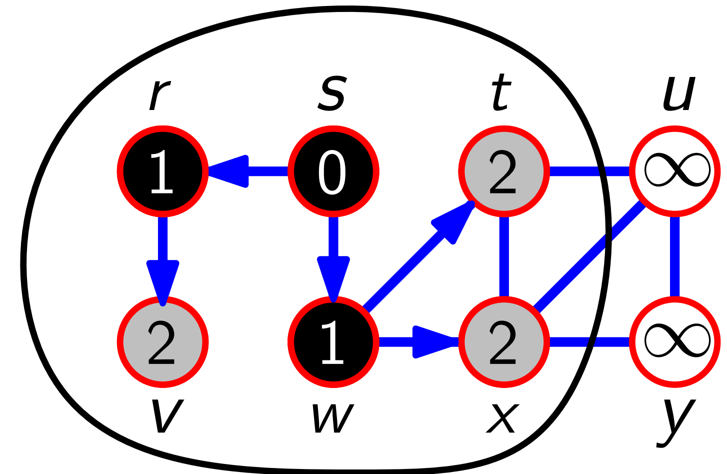





```
BFS(Graph G, Vertex s)
  Initialize(G, s)
  Q = new Queue()
  Q.Enqueue(s)
  while Q ≠ ∅ do
      u = Q.Dequeue()
      foreach v ∈ Adj[u] do
          if v.color == white then
              v.color = gray
              v.d = u.d + 1
              v.π = u
              Q.Enqueue(v)
```

```
Initialize(Graph G, Vertex s)
  foreach u ∈ V do
      u.color = white
      u.d = ∞
      u.π = nil
  s.color = gray
  s.d = 0
```

Breitensuche – Pseudocode

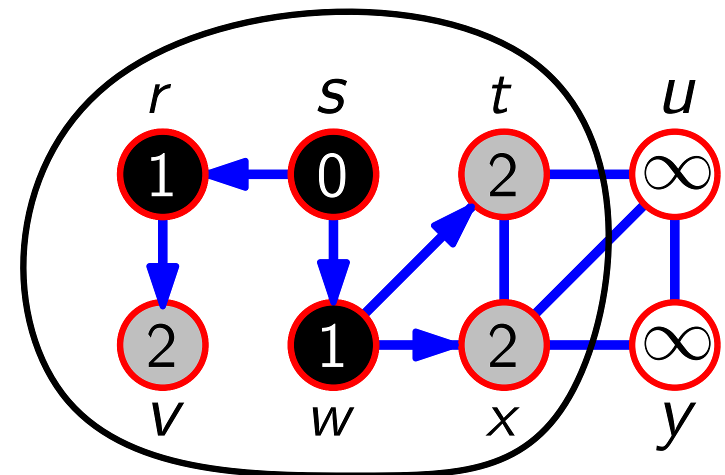


w r | t | x | v | | | usw.

```
BFS(Graph G, Vertex s)
  Initialize(G, s)
  Q = new Queue()
  Q.Enqueue(s)
  while Q ≠ ∅ do
      u = Q.Dequeue()
      foreach v ∈ Adj[u] do
          if v.color == white then
              v.color = gray
              v.d = u.d + 1
              v.π = u
              Q.Enqueue(v)
      u.color = black
```

```
Initialize(Graph G, Vertex s)
  foreach u ∈ V do
      u.color = white
      u.d = ∞
      u.π = nil
  s.color = gray
  s.d = 0
```

Breitensuche – Pseudocode



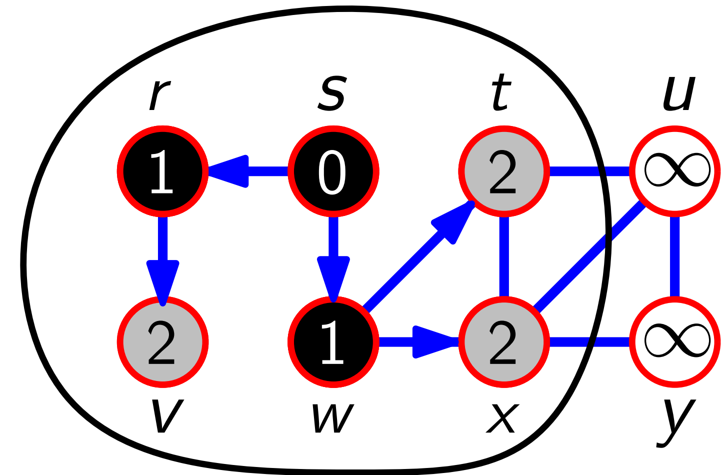
w r | t | x | v | | | usw.

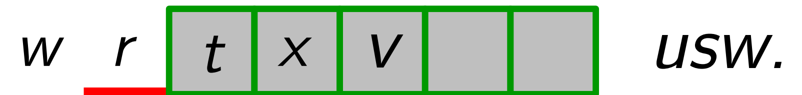
```
BFS(Graph G, Vertex s)
  Initialize(G, s)
  Q = new Queue()
  Q.Enqueue(s)
  while Q ≠ ∅ do
      u = Q.Dequeue()
      foreach v ∈ Adj[u] do
          if v.color == white then
              v.color = gray
              v.d = u.d + 1
              v.π = u
              Q.Enqueue(v)
      u.color = black
```

```
Initialize(Graph G, Vertex s)
  foreach u ∈ V do
      u.color = white
      u.d = ∞
      u.π = nil
  s.color = gray
  s.d = 0
```

Laufzeit?

Breitensuche – Pseudocode





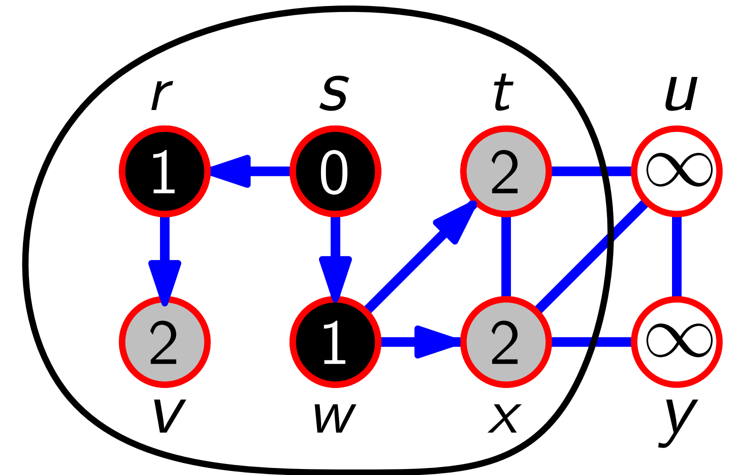

```
BFS(Graph G, Vertex s)
   Initialize(G, s)
   Q = new Queue()
   Q.Enqueue(s)
   while Q ≠ ∅ do
      u = Q.Dequeue()
      foreach v ∈ Adj[u] do
         if v.color == white then
            v.color = gray
            v.d = u.d + 1
            v.π = u
            Q.Enqueue(v)
      u.color = black
```

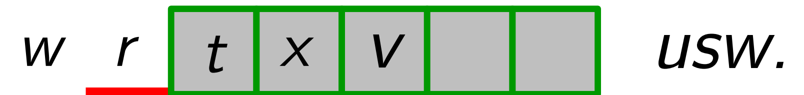
```
Initialize(Graph G, Vertex s)
   foreach u ∈ V do
      u.color = white
      u.d = ∞
      u.π = nil
   s.color = gray
   s.d = 0
```

Initialize

Laufzeit?

Breitensuche – Pseudocode





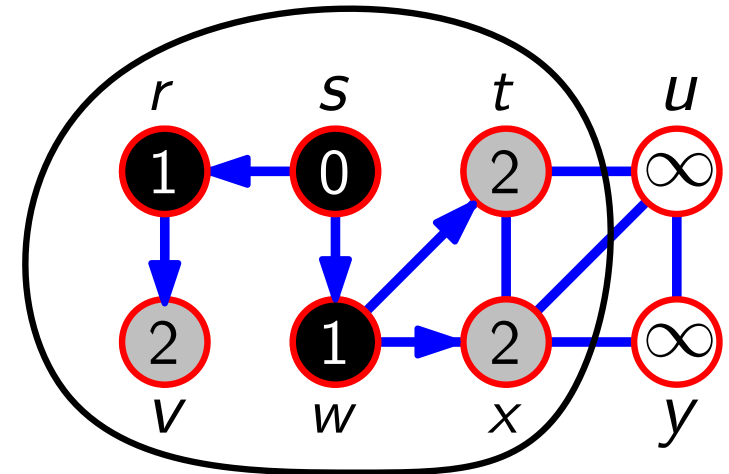

```
BFS(Graph G, Vertex s)
  Initialize(G, s)
  Q = new Queue()
  Q.Enqueue(s)
  while Q ≠ ∅ do
      u = Q.Dequeue()
      foreach v ∈ Adj[u] do
          if v.color == white then
              v.color = gray
              v.d = u.d + 1
              v.π = u
              Q.Enqueue(v)
      u.color = black
```

```
Initialize(Graph G, Vertex s)
  foreach u ∈ V do
      u.color = white
      u.d = ∞
      u.π = nil
  s.color = gray
  s.d = 0
```

Laufzeit? Initialize $O(V)$

Breitensuche – Pseudocode

```
BFS(Graph G, Vertex s)
  Initialize(G, s)
  Q = new Queue()
  Q.Enqueue(s)
  while Q ≠ ∅ do
    u = Q.Dequeue()
    foreach v ∈ Adj[u] do
      if v.color == white then
        v.color = gray
        v.d = u.d + 1
        v.π = u
        Q.Enqueue(v)
    u.color = black
```



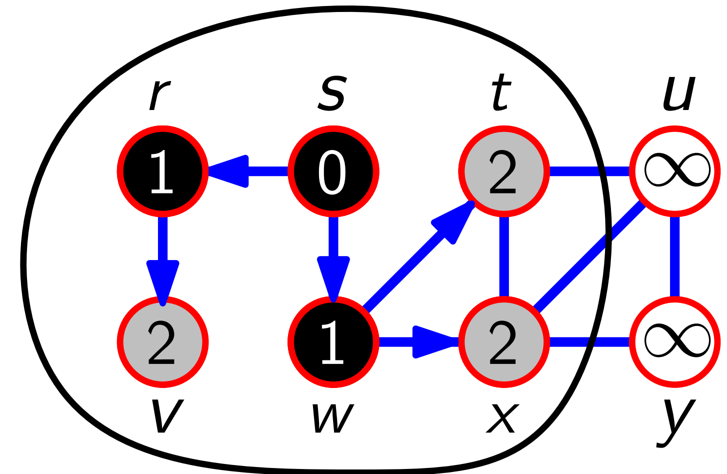
w r | t | x | v | | | usw.

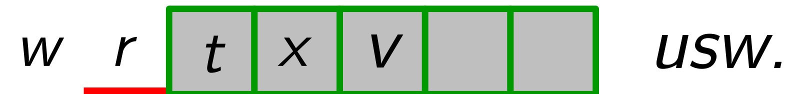
```
Initialize(Graph G, Vertex s)
  foreach u ∈ V do
    u.color = white
    u.d = ∞
    u.π = nil
  s.color = gray
  s.d = 0
```

Initialize En-/Dequeues

Laufzeit? $O(V)$

Breitensuche – Pseudocode



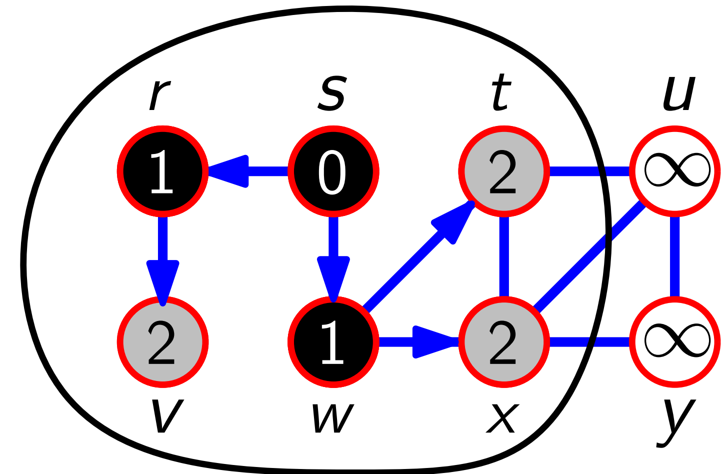



```
BFS(Graph G, Vertex s)
   Initialize(G, s)
   Q = new Queue()
   Q.Enqueue(s)
   while Q ≠ ∅ do
      u = Q.Dequeue()
      foreach v ∈ Adj[u] do
         if v.color == white then
            v.color = gray
            v.d = u.d + 1
            v.π = u
            Q.Enqueue(v)
      u.color = black
```

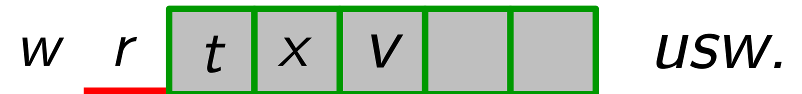
```
Initialize(Graph G, Vertex s)
   foreach u ∈ V do
      u.color = white
      u.d = ∞
      u.π = nil
   s.color = gray
   s.d = 0
```

Laufzeit? Initialize $O(V)$ + En-/Dequeues $O(V)$

Breitensuche – Pseudocode




```
BFS(Graph G, Vertex s)
  Initialize(G, s)
  Q = new Queue()
  Q.Enqueue(s)
  while Q ≠ ∅ do
  |   u = Q.Dequeue()
  |   foreach v ∈ Adj[u] do
  |   |   if v.color == white then
  |   |   |   v.color = gray
  |   |   |   v.d = u.d + 1
  |   |   |   v.π = u
  |   |   |   Q.Enqueue(v)
  |   u.color = black
```



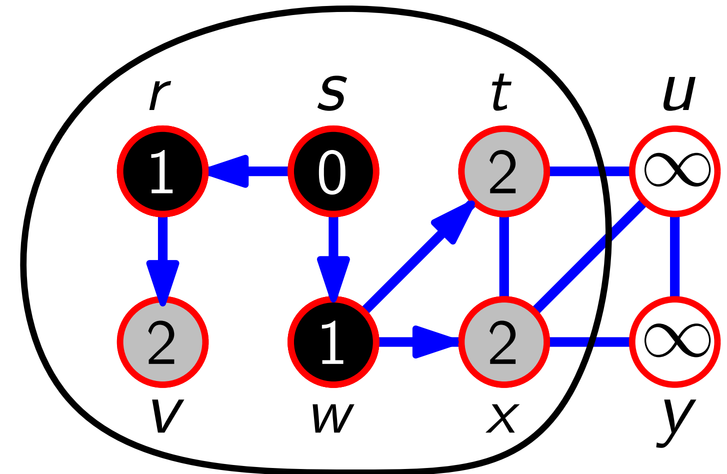

```
Initialize(Graph G, Vertex s)
  foreach u ∈ V do
  |   u.color = white
  |   u.d = ∞
  |   u.π = nil
  s.color = gray
  s.d = 0
```

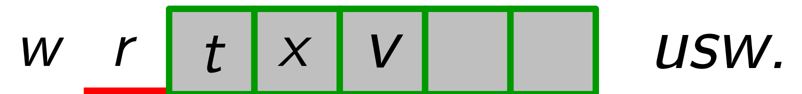
Initialize — En-/Dequeues — Adjazenzlisten (foreach-Schleifen)

Laufzeit? $O(V) + O(V)$

Breitensuche – Pseudocode

```
BFS(Graph G, Vertex s)
  Initialize(G, s)
  Q = new Queue()
  Q.Enqueue(s)
  while Q ≠ ∅ do
    u = Q.Dequeue()
    foreach v ∈ Adj[u] do
      if v.color == white then
        v.color = gray
        v.d = u.d + 1
        v.π = u
        Q.Enqueue(v)
    u.color = black
```



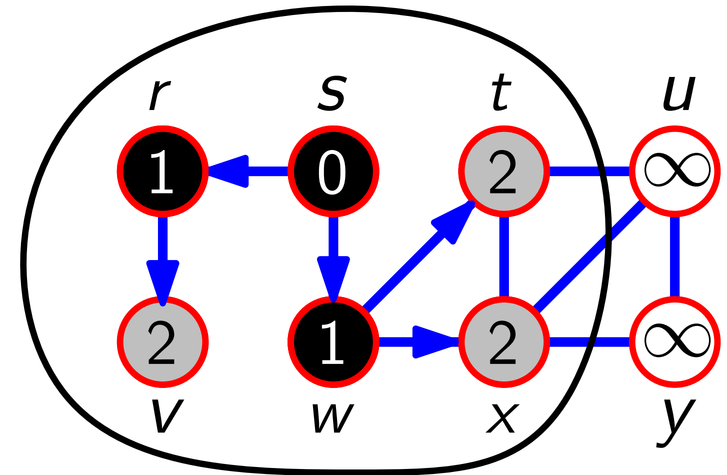



```
Initialize(Graph G, Vertex s)
  foreach u ∈ V do
    u.color = white
    u.d = ∞
    u.π = nil
  s.color = gray
  s.d = 0
```

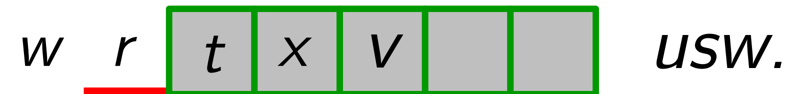
Initialize En-/Dequeues Adjazenzlisten (foreach-Schleifen)

Laufzeit? $O(V) + O(V) + O(E)$

Breitensuche – Pseudocode




```
BFS(Graph G, Vertex s)
  Initialize(G, s)
  Q = new Queue()
  Q.Enqueue(s)
  while Q ≠ ∅ do
      u = Q.Dequeue()
      foreach v ∈ Adj[u] do
          if v.color == white then
              v.color = gray
              v.d = u.d + 1
              v.π = u
              Q.Enqueue(v)
      u.color = black
```




```
Initialize(Graph G, Vertex s)
  foreach u ∈ V do
      u.color = white
      u.d = ∞
      u.π = nil
  s.color = gray
  s.d = 0
```

Laufzeit?

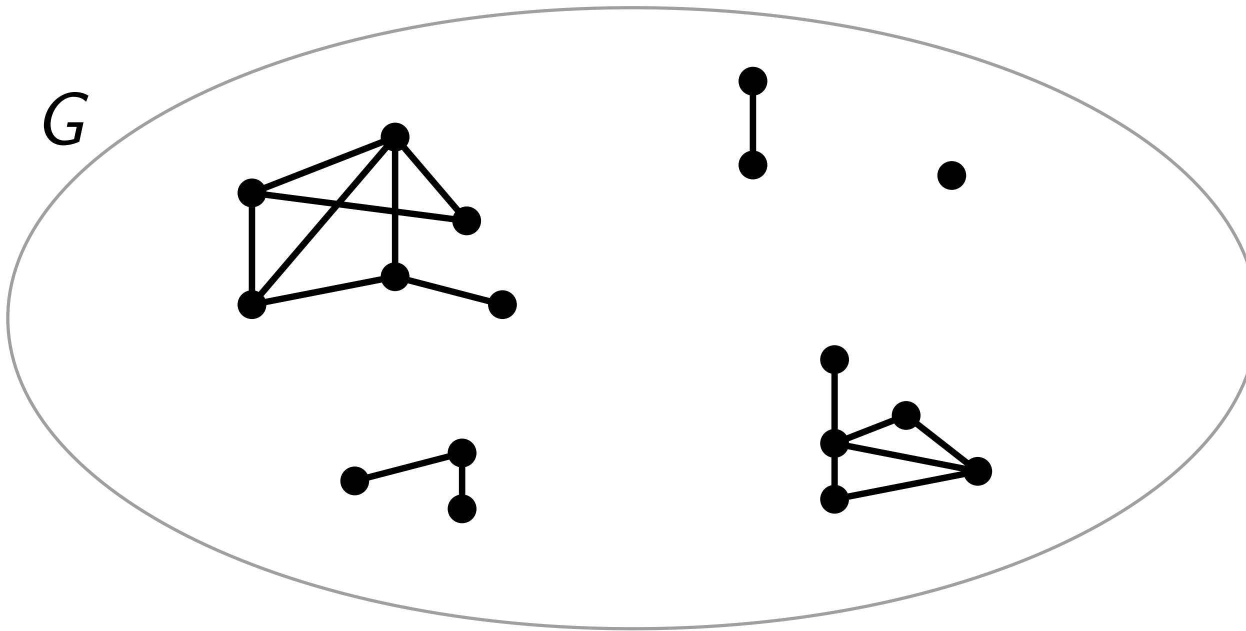
	Initialize		En-/Dequeues		Adjazenzlisten (foreach-Schleifen)		
	$O(V)$	$+$	$O(V)$	$+$	$O(E)$	$=$	$O(V+E)$

Breitensuche – Anwendung

Zusammenhangskomponente:
Maximale Teilmenge von Knoten, die über Wege miteinander verbunden sind.

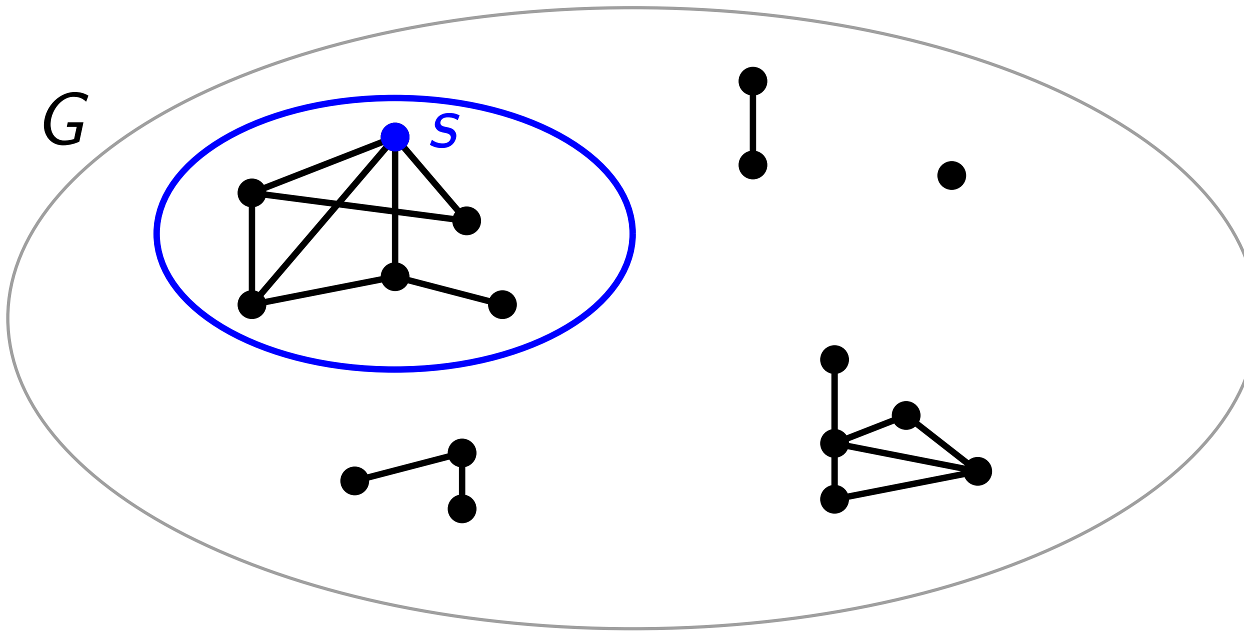
Breitensuche – Anwendung

Zusammenhangskomponente:
Maximale Teilmenge von Knoten, die über Wege miteinander verbunden sind.



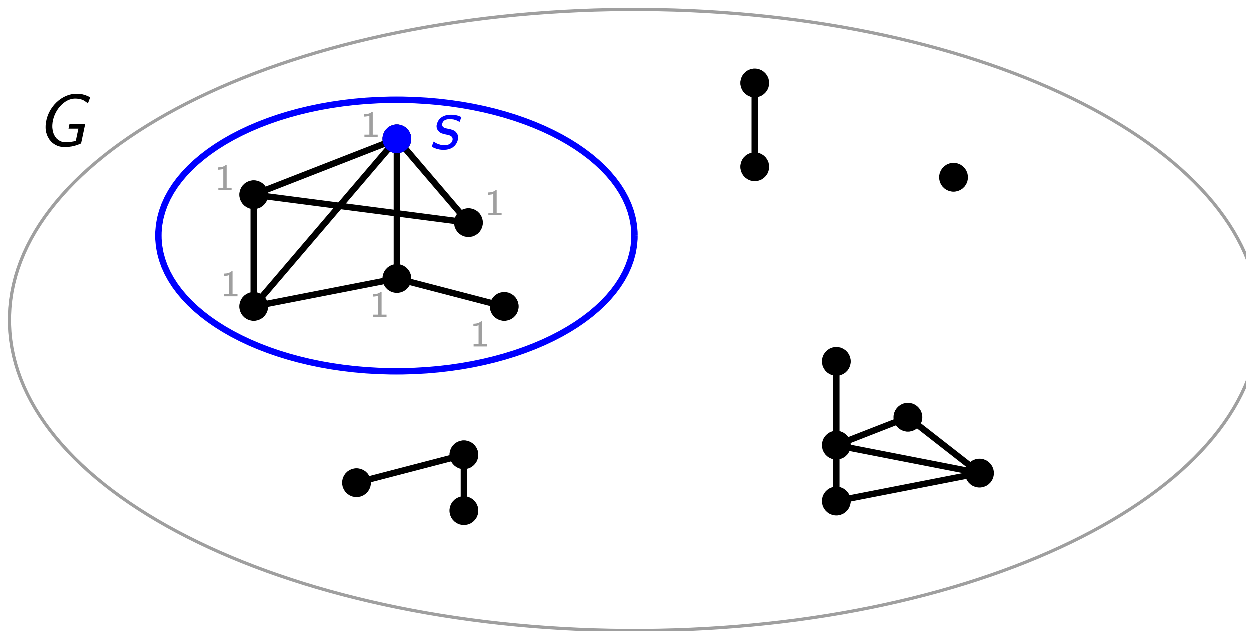
Breitensuche – Anwendung

Zusammenhangskomponente:
Maximale Teilmenge von Knoten, die über Wege miteinander verbunden sind.



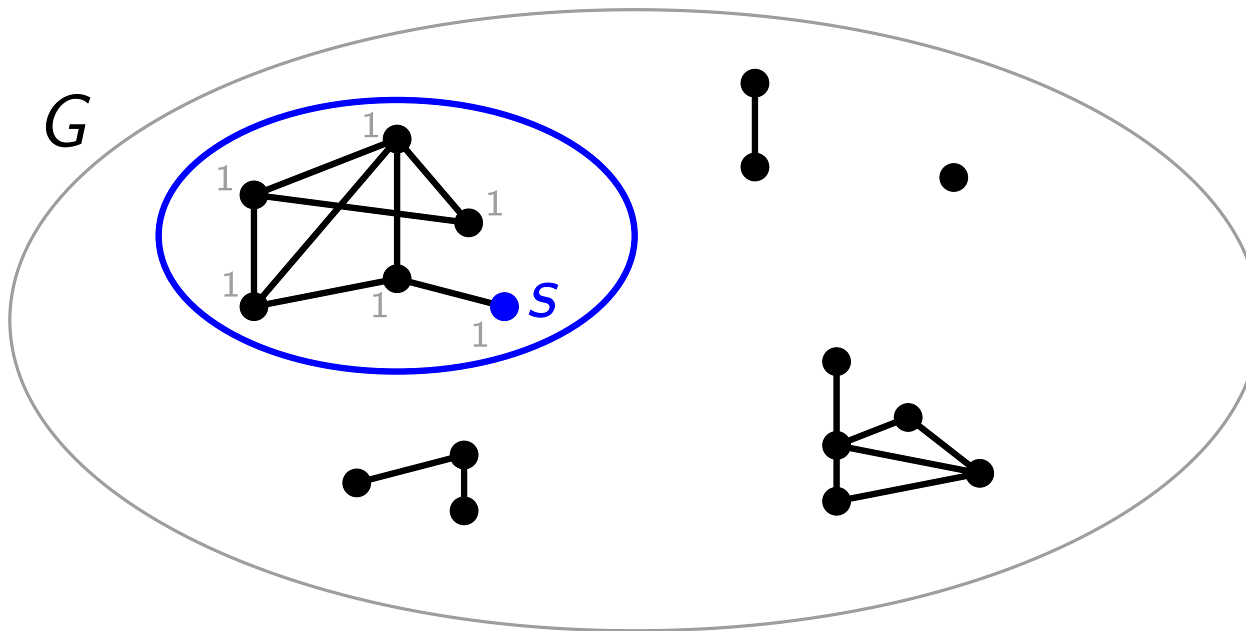
Breitensuche – Anwendung

Zusammenhangskomponente:
Maximale Teilmenge von Knoten, die über Wege miteinander verbunden sind.



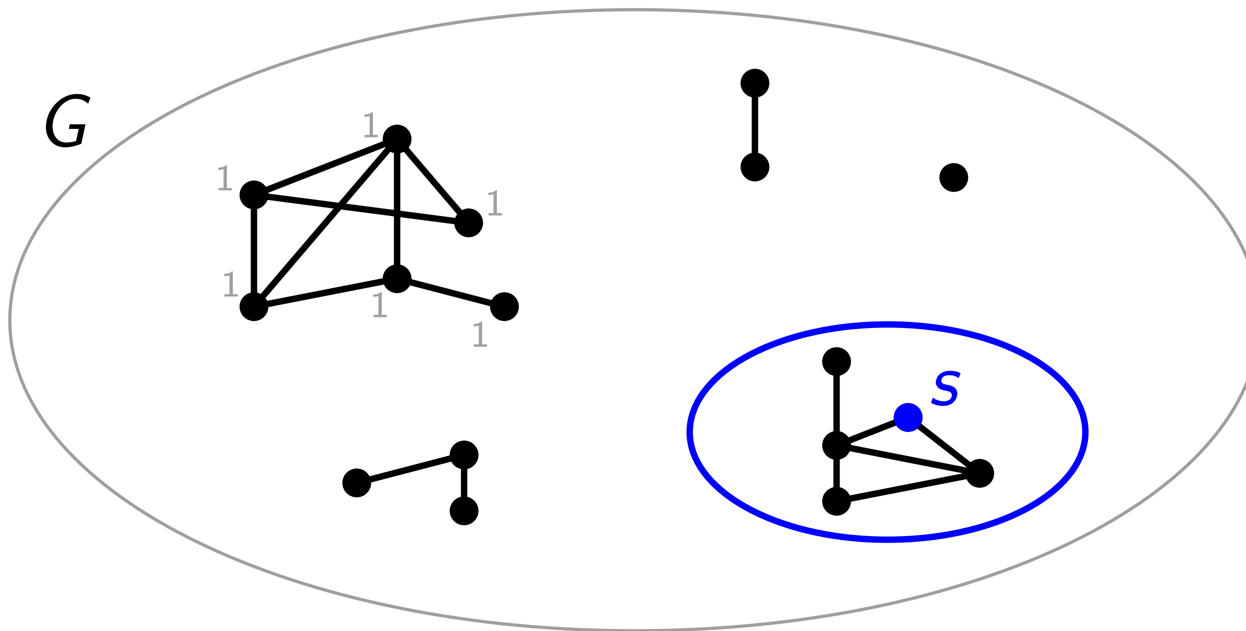
Breitensuche – Anwendung

Zusammenhangskomponente:
Maximale Teilmenge von Knoten, die über Wege miteinander verbunden sind.



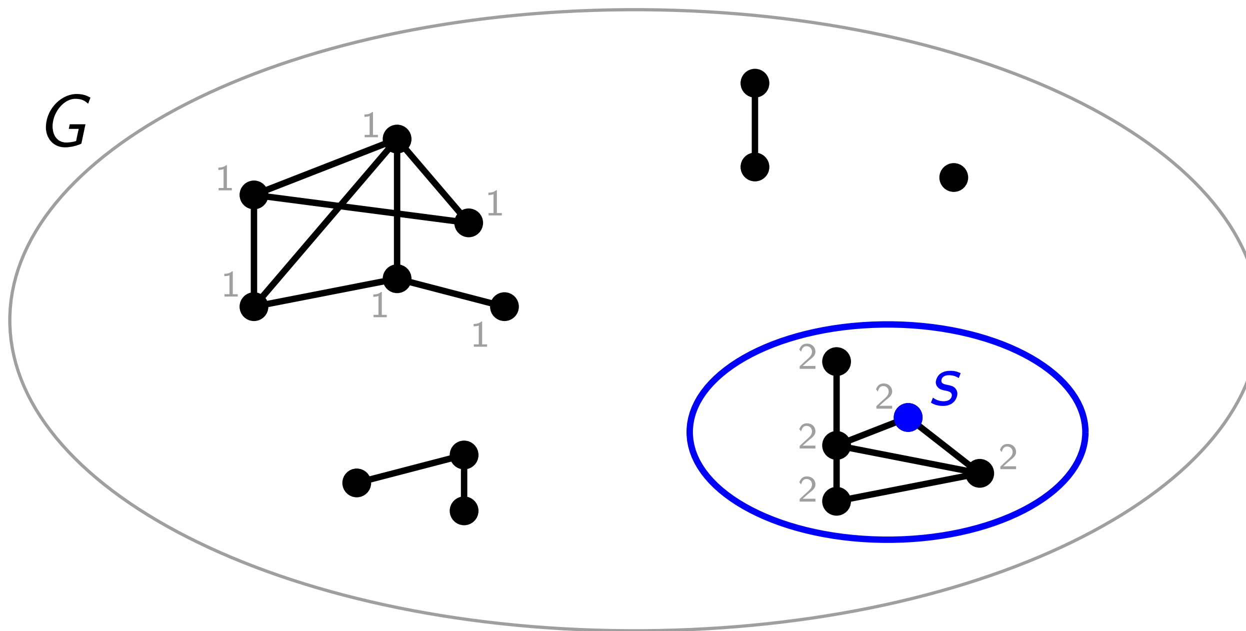
Breitensuche – Anwendung

Zusammenhangskomponente:
Maximale Teilmenge von Knoten, die über Wege miteinander verbunden sind.



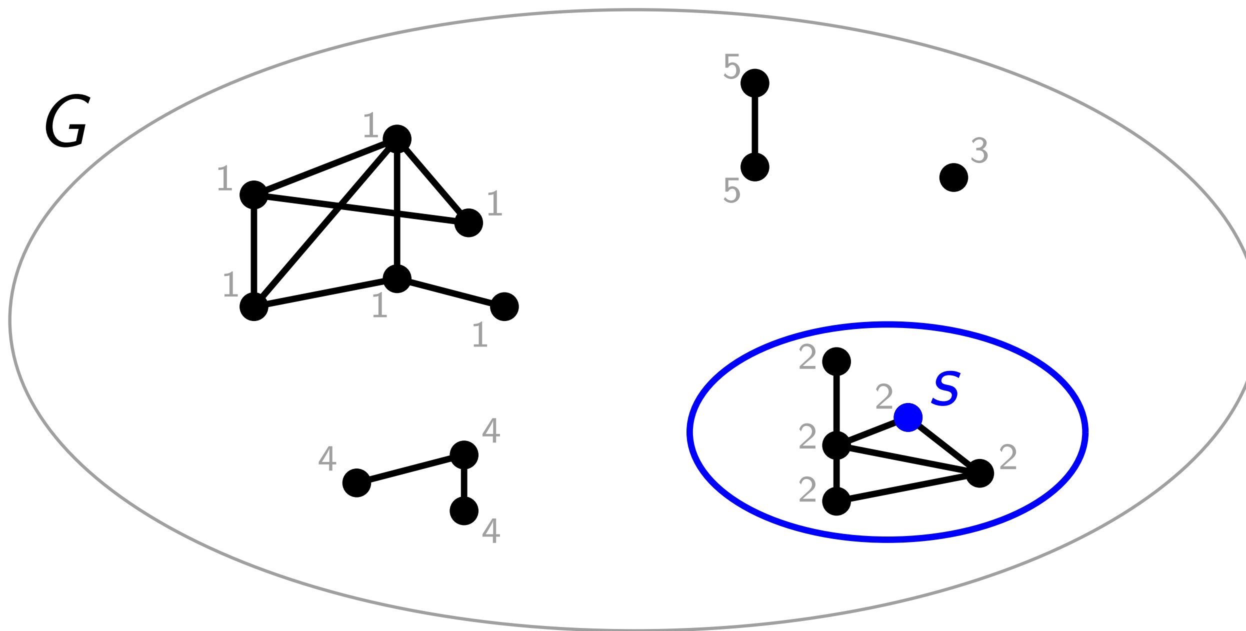
Breitensuche – Anwendung

Zusammenhangskomponente:
Maximale Teilmenge von Knoten, die über Wege miteinander verbunden sind.



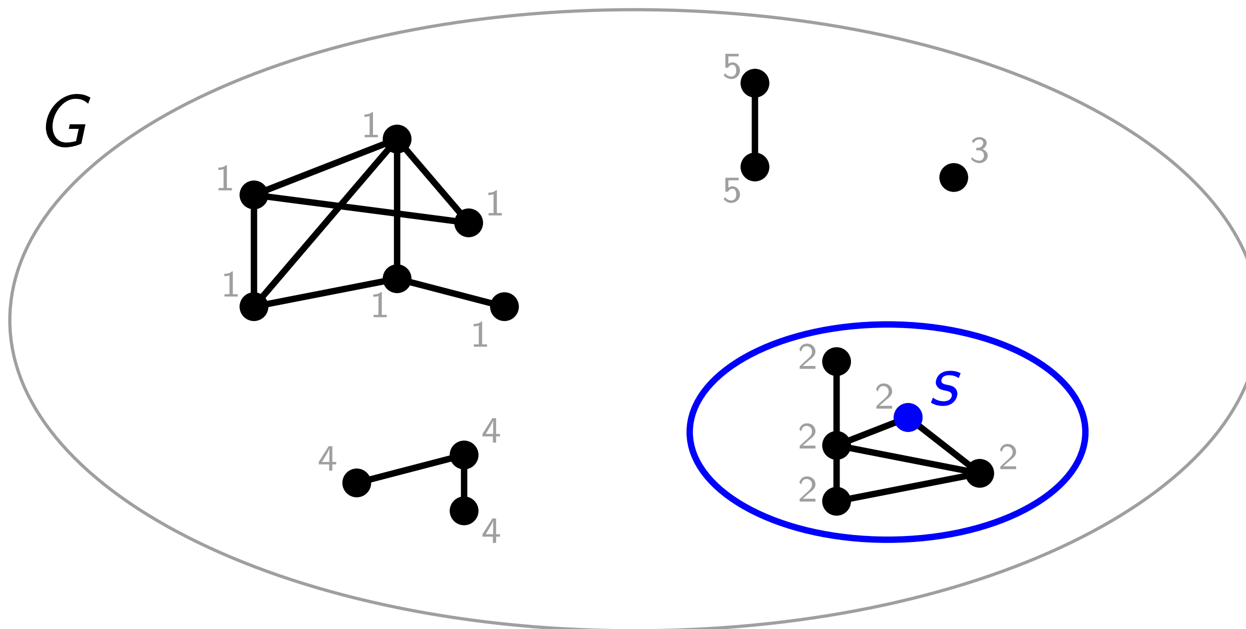
Breitensuche – Anwendung

Zusammenhangskomponente:
Maximale Teilmenge von Knoten, die über Wege miteinander verbunden sind.



Breitensuche – Anwendung

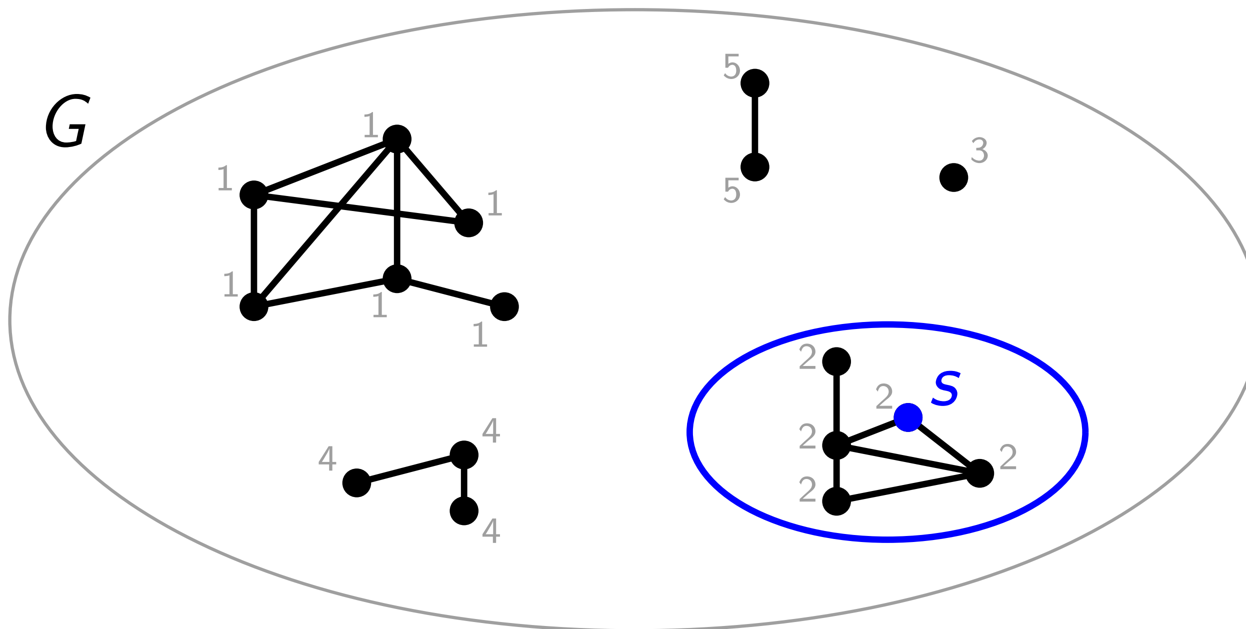
Zusammenhangskomponente: Maximale Teilmenge von Knoten, die über Wege miteinander verbunden sind.



G hat also fünf Zusammenhangskomponenten.

Breitensuche – Anwendung

Zusammenhangskomponente: Maximale Teilmenge von Knoten, die über Wege miteinander verbunden sind.

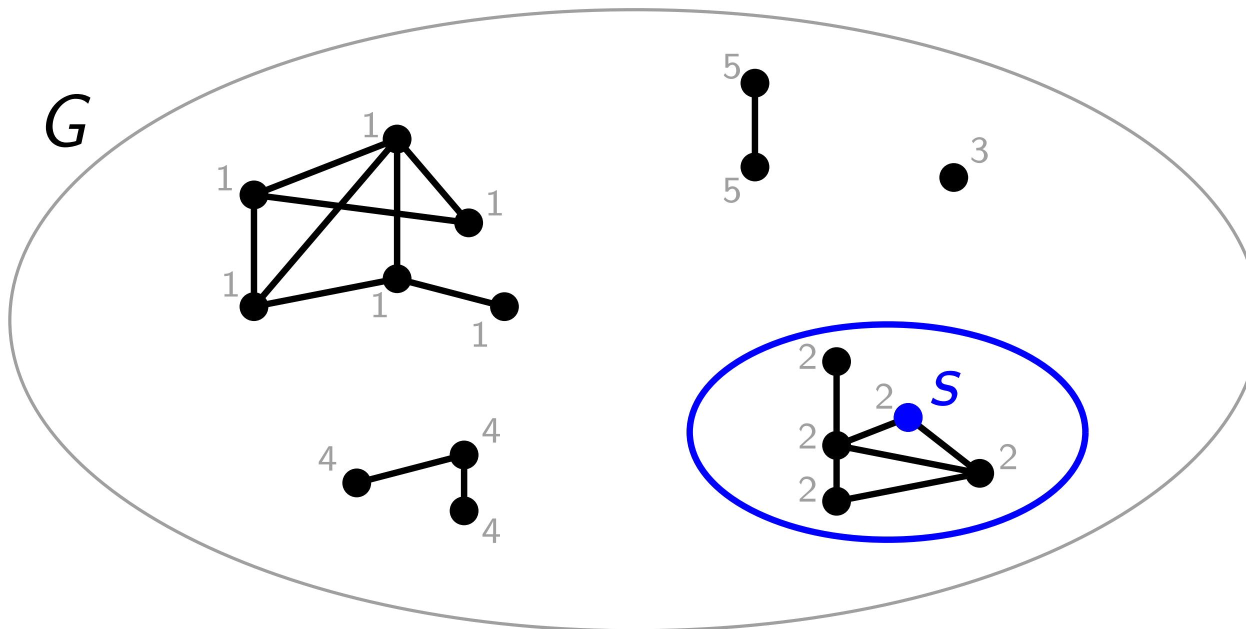


G hat also fünf Zusammenhangskomponenten.

Laufzeit fürs Zählen aller Zusammenhangskomponenten:

Breitensuche – Anwendung

Zusammenhangskomponente:
Maximale Teilmenge von Knoten, die über Wege miteinander verbunden sind.



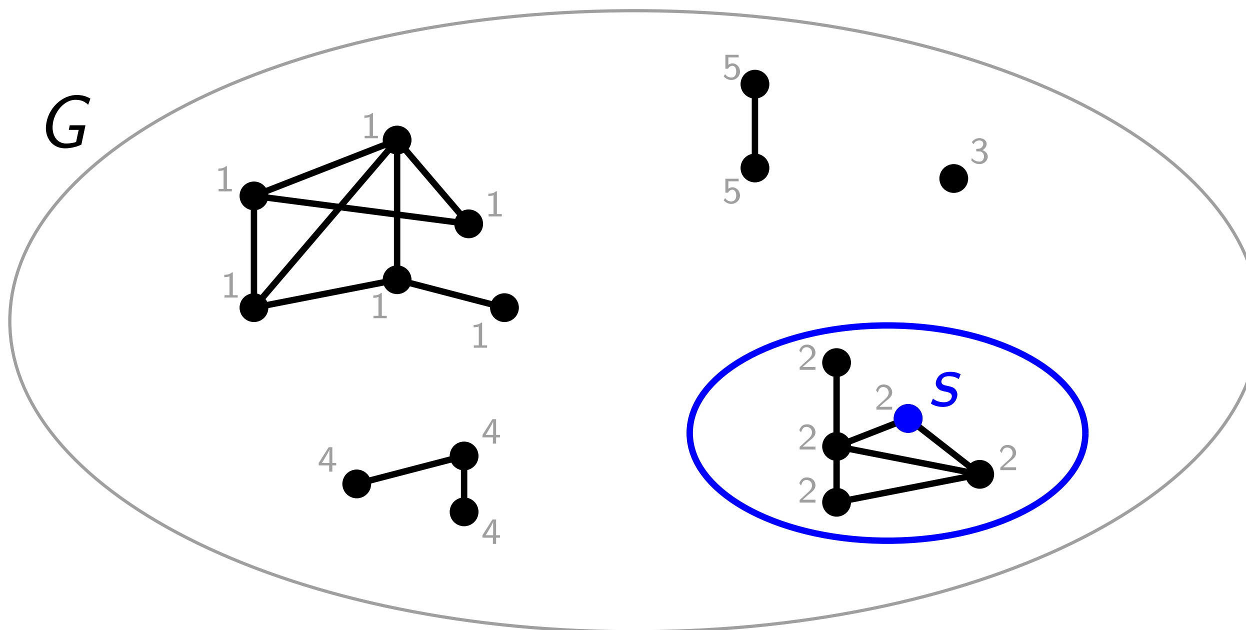
G hat also fünf Zusammenhangskomponenten.

Laufzeit fürs Zählen aller Zusammenhangskomponenten:

Auch (nur) $O(V + E)$!

Breitensuche – Anwendung

Zusammenhangskomponente:
Maximale Teilmenge von Knoten, die über Wege miteinander verbunden sind.



G hat also fünf Zusammenhangskomponenten.

Laufzeit fürs Zählen aller Zusammenhangskomponenten:

Auch (nur) $O(V + E)$!

Eine weitere wichtige Anwendung der Breitensuche ist die schnelle Berechnung von kürzesten Wegen in ungewichteten Graphen.

Übersicht

1. Graphdurchlaufstrategien
2. Kürzeste Wege
   2.1 Breitensuche

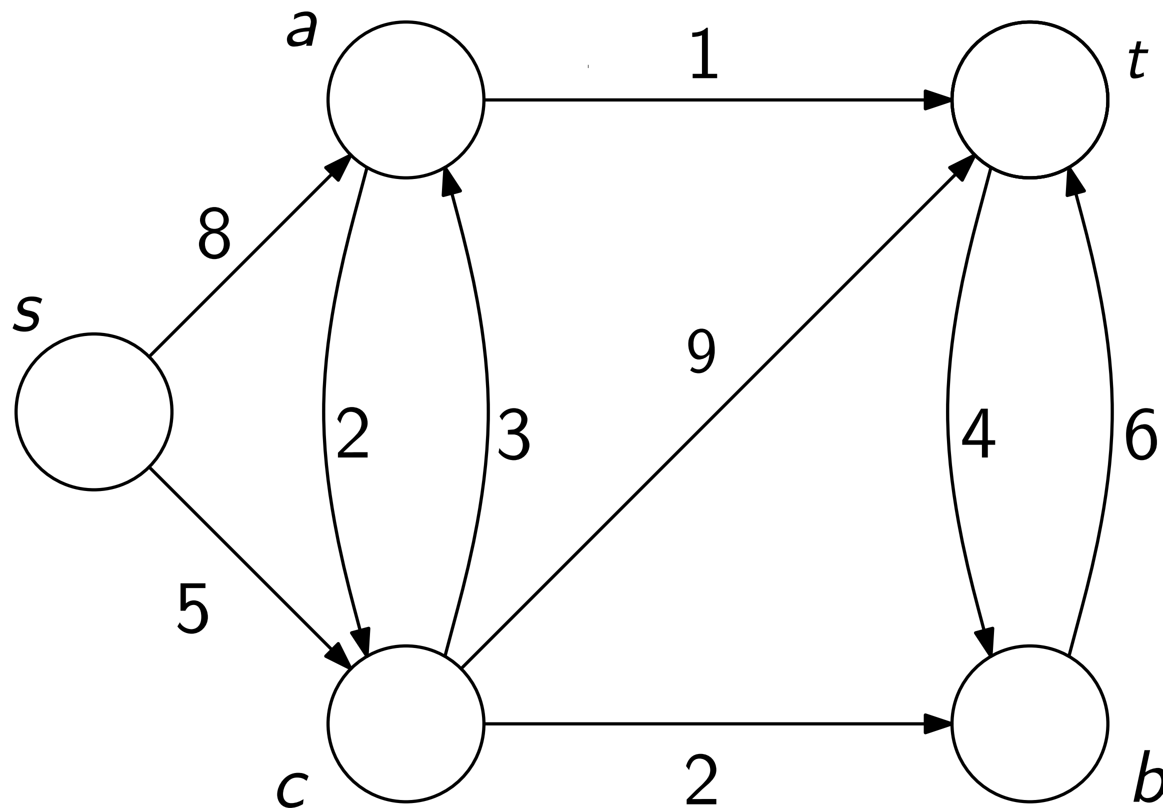
Übersicht

1. Graphdurchlaufstrategien
2. Kürzeste Wege
   - 2.1 Breitensuche
   - 2.2 Dijkstra
     - Beispiel
     - Pseudocode
3. Minimale Spannbäume

Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

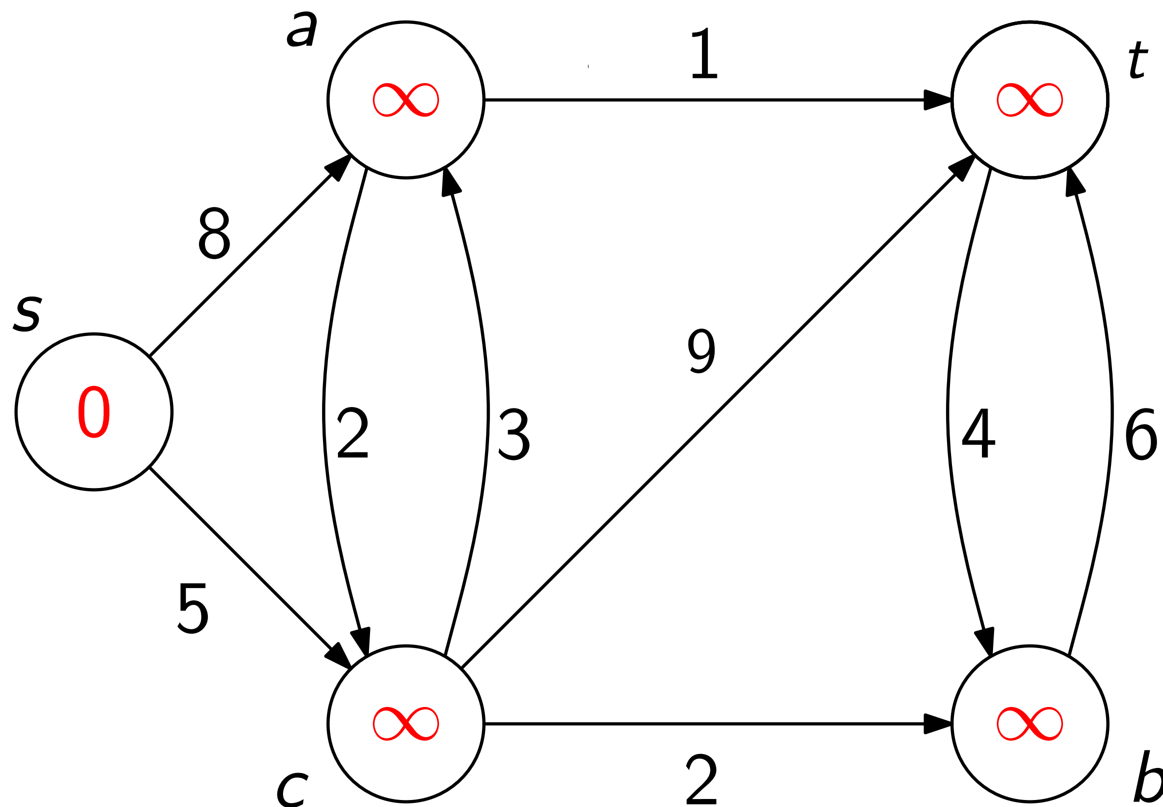
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

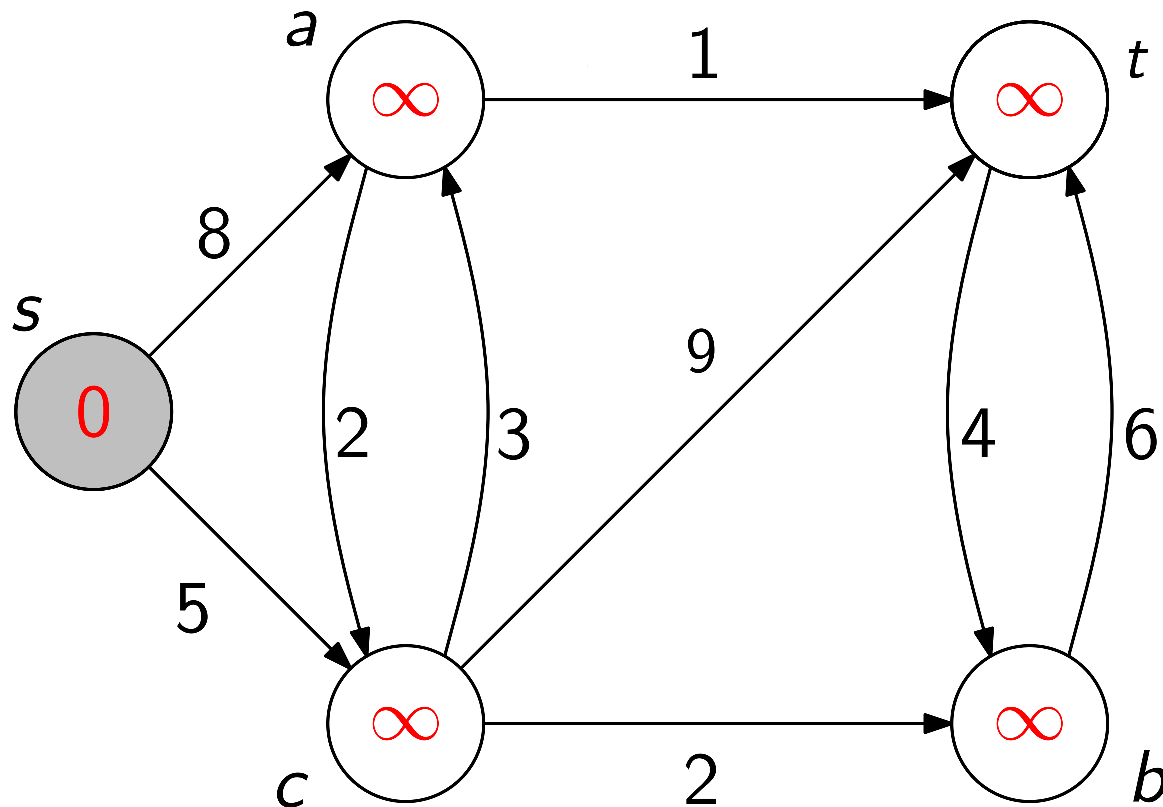
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

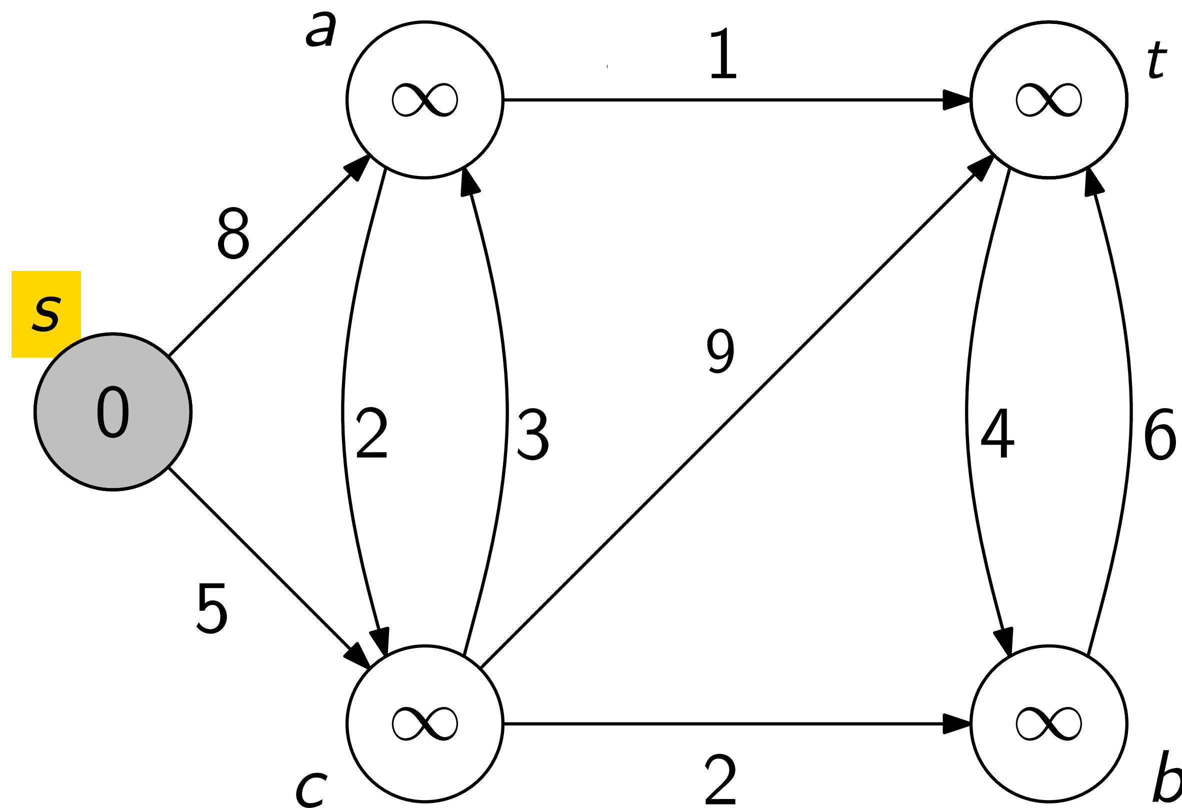
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

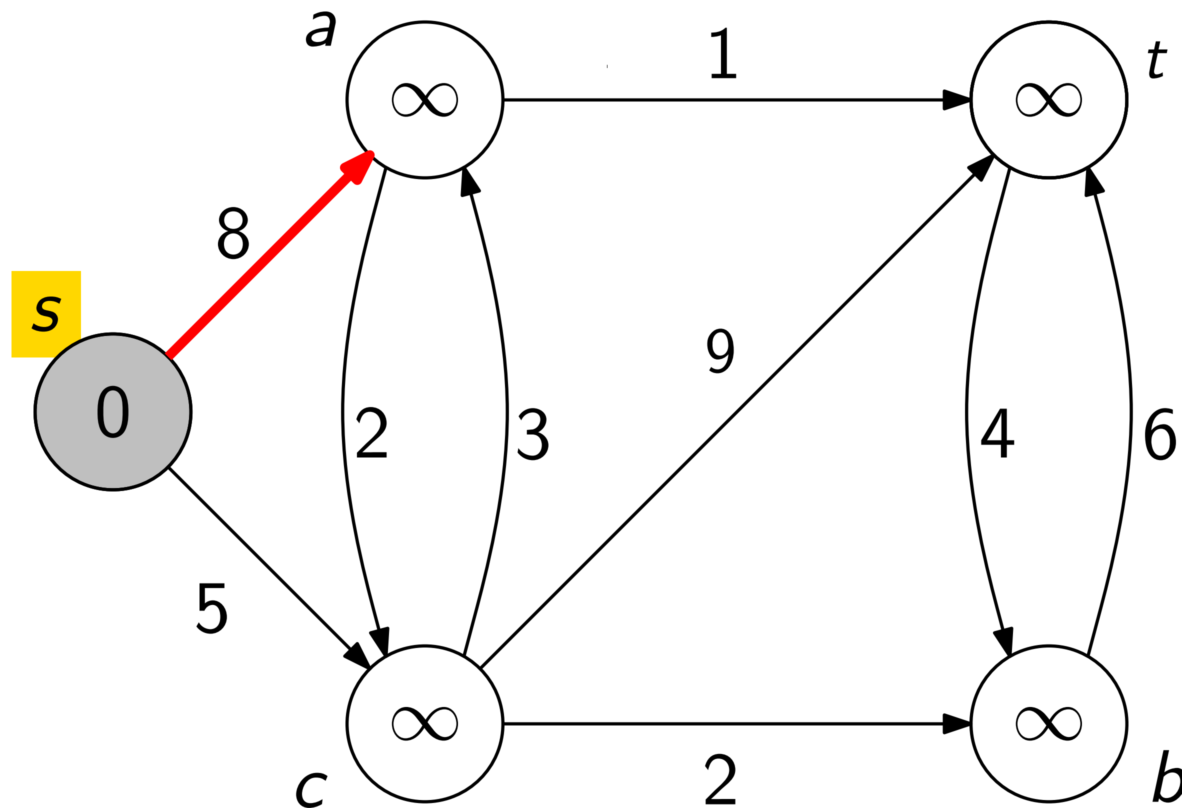
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

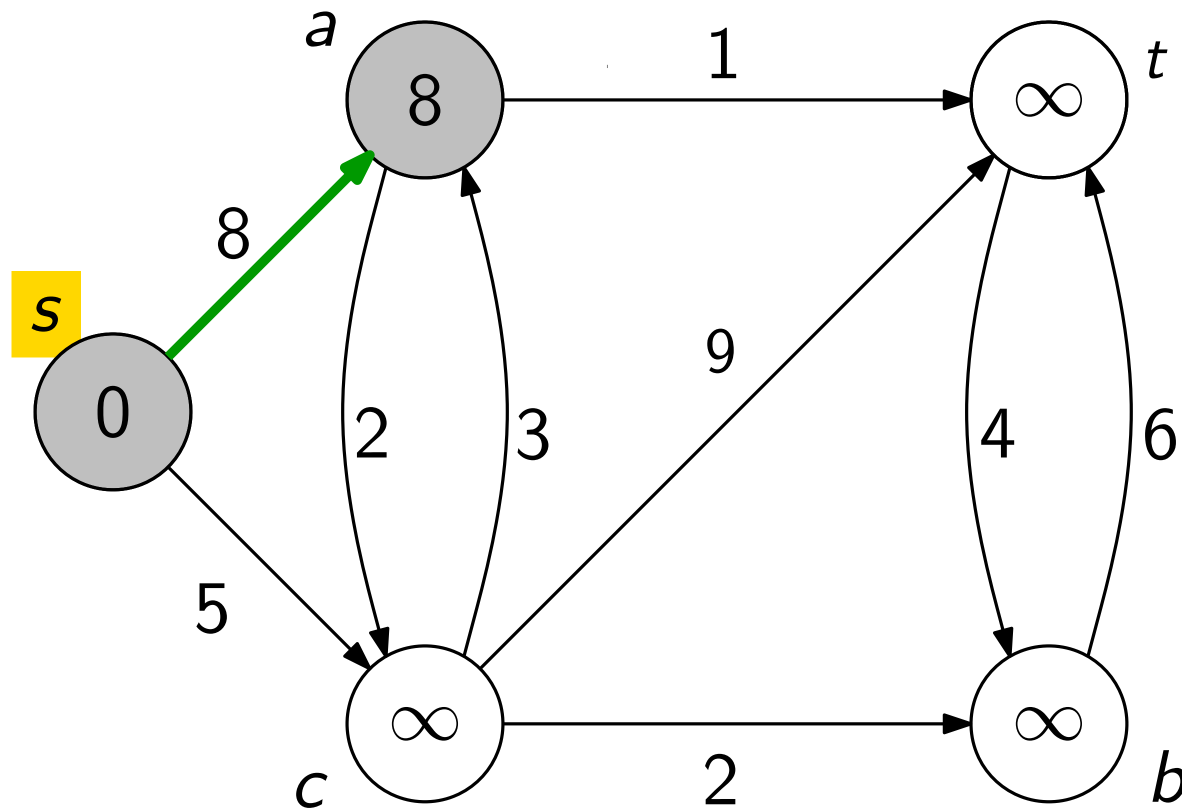
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

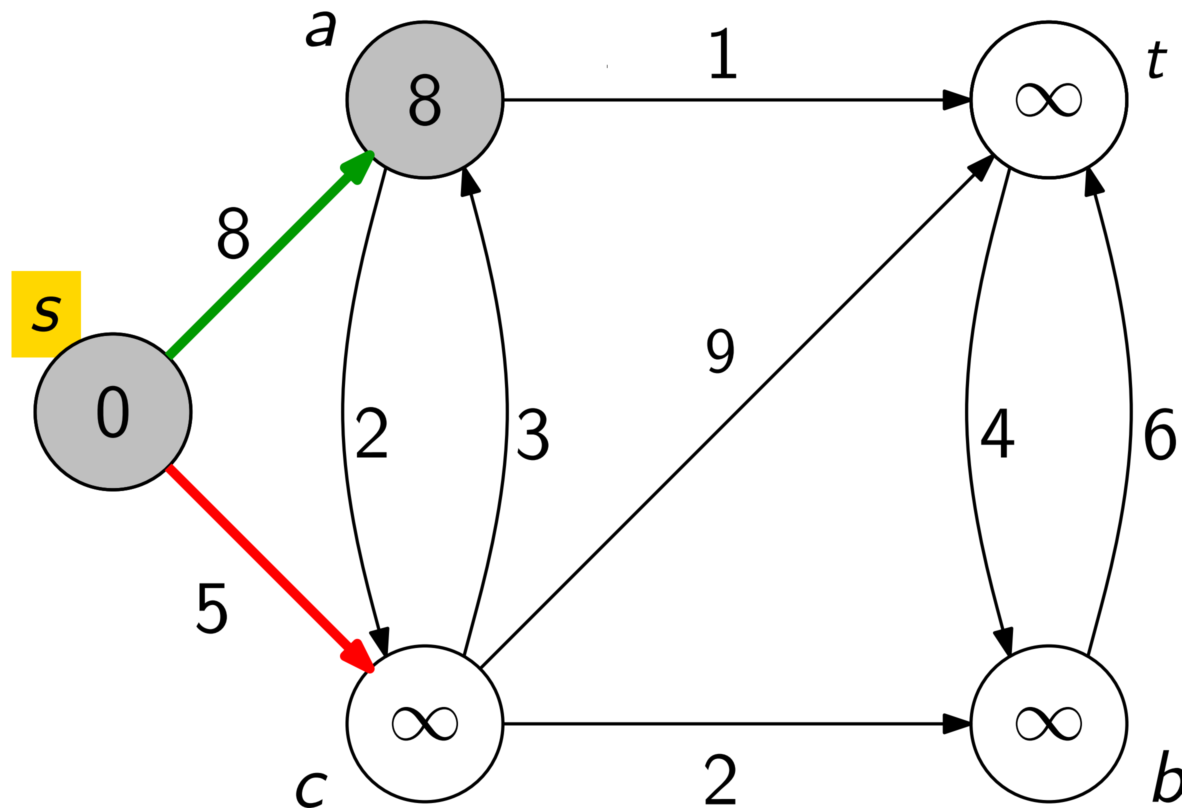
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

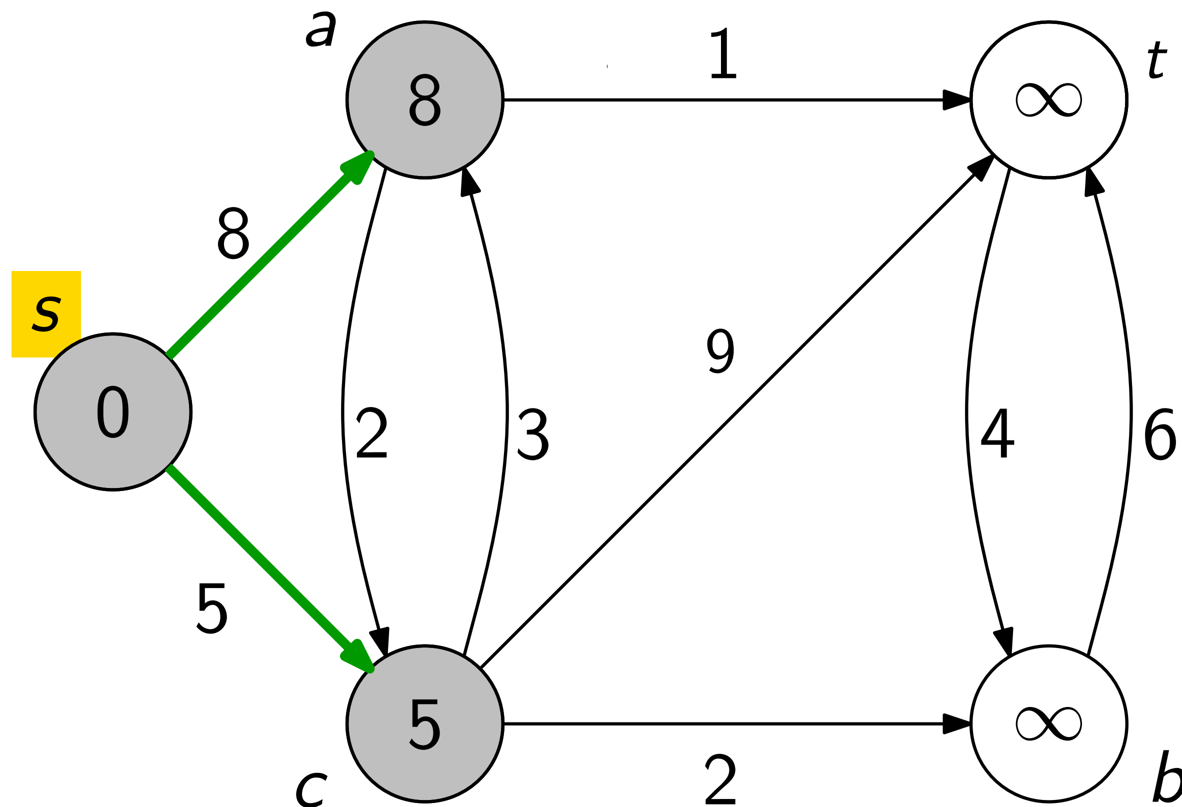
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

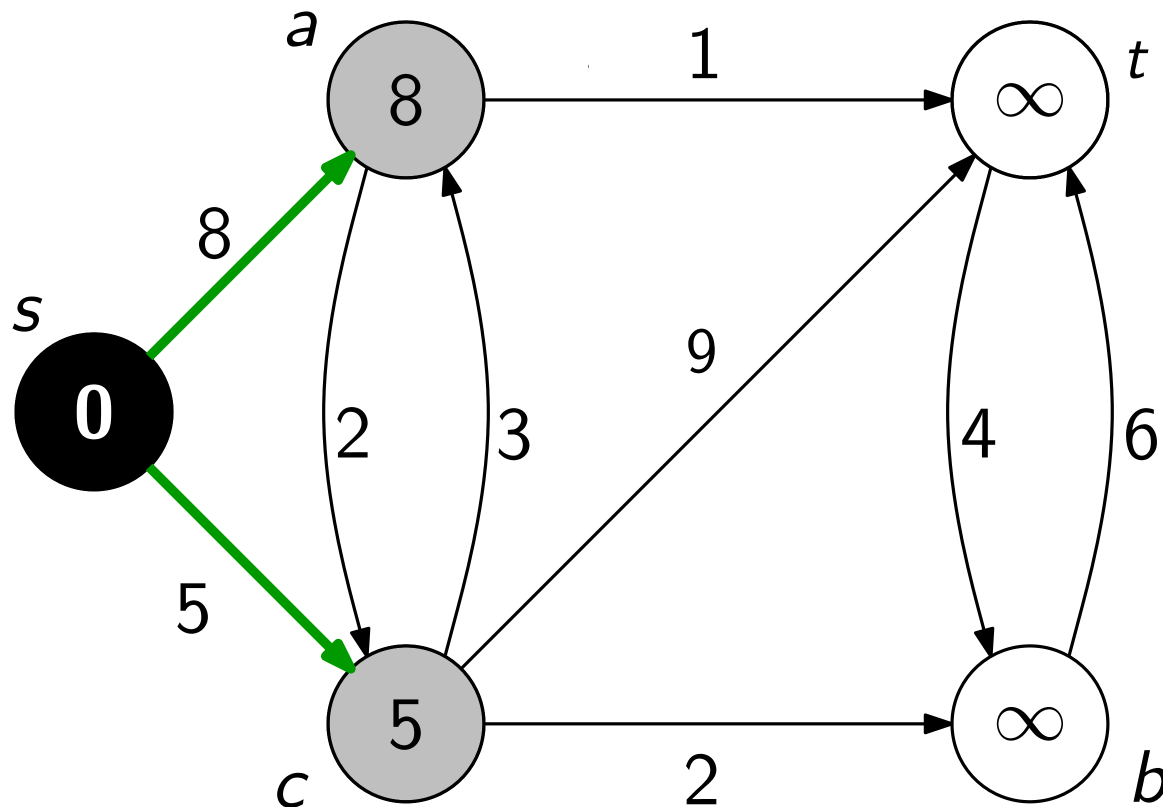
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

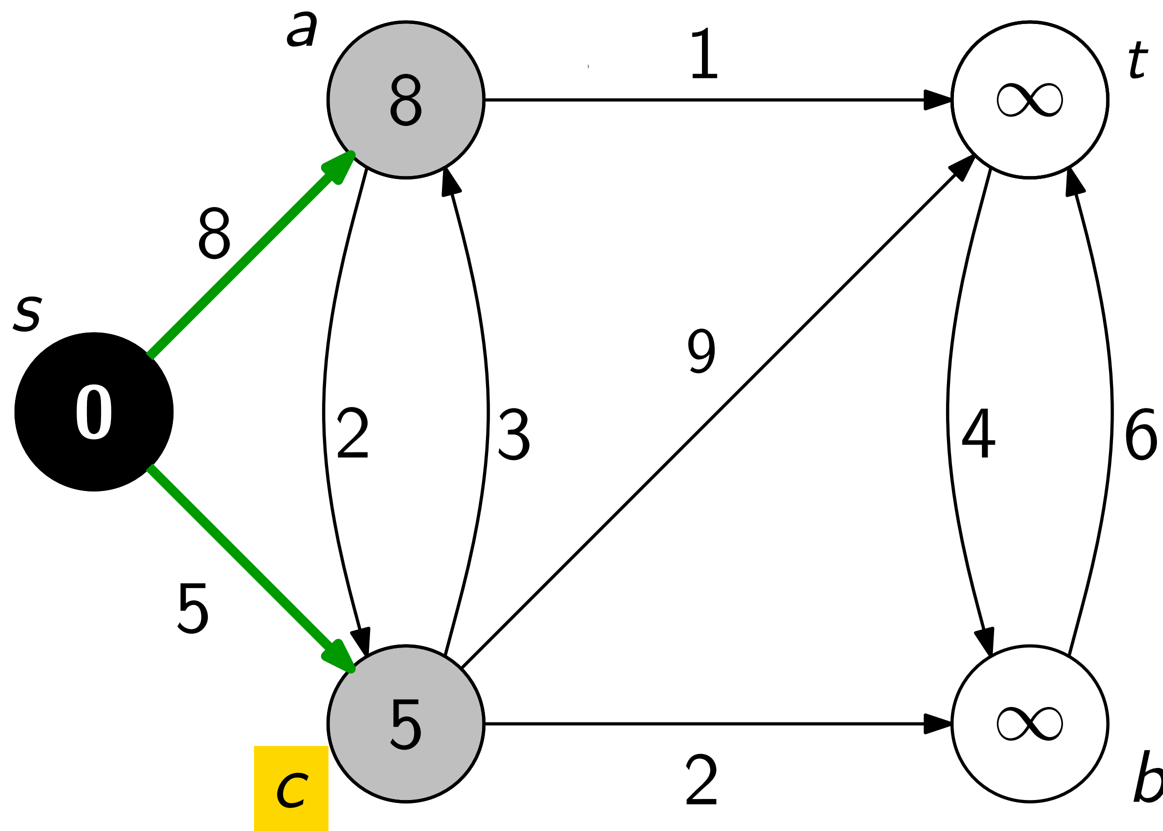
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

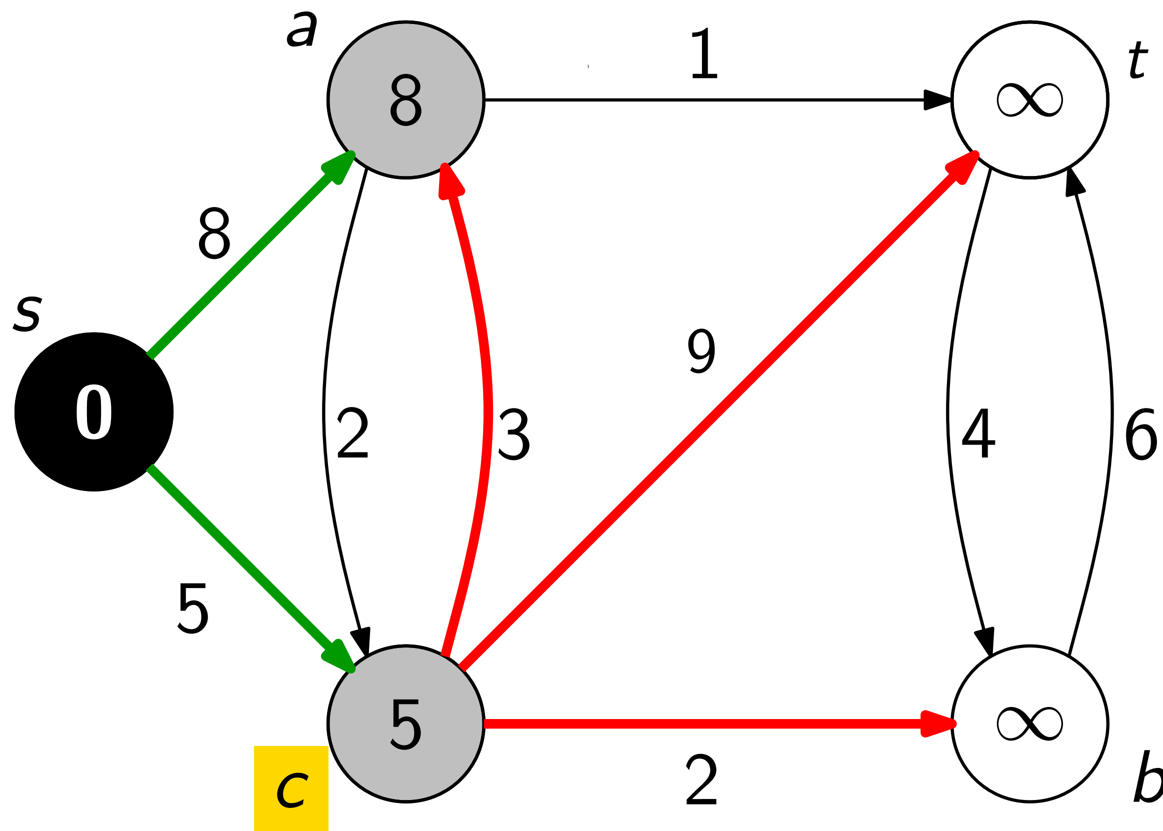
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

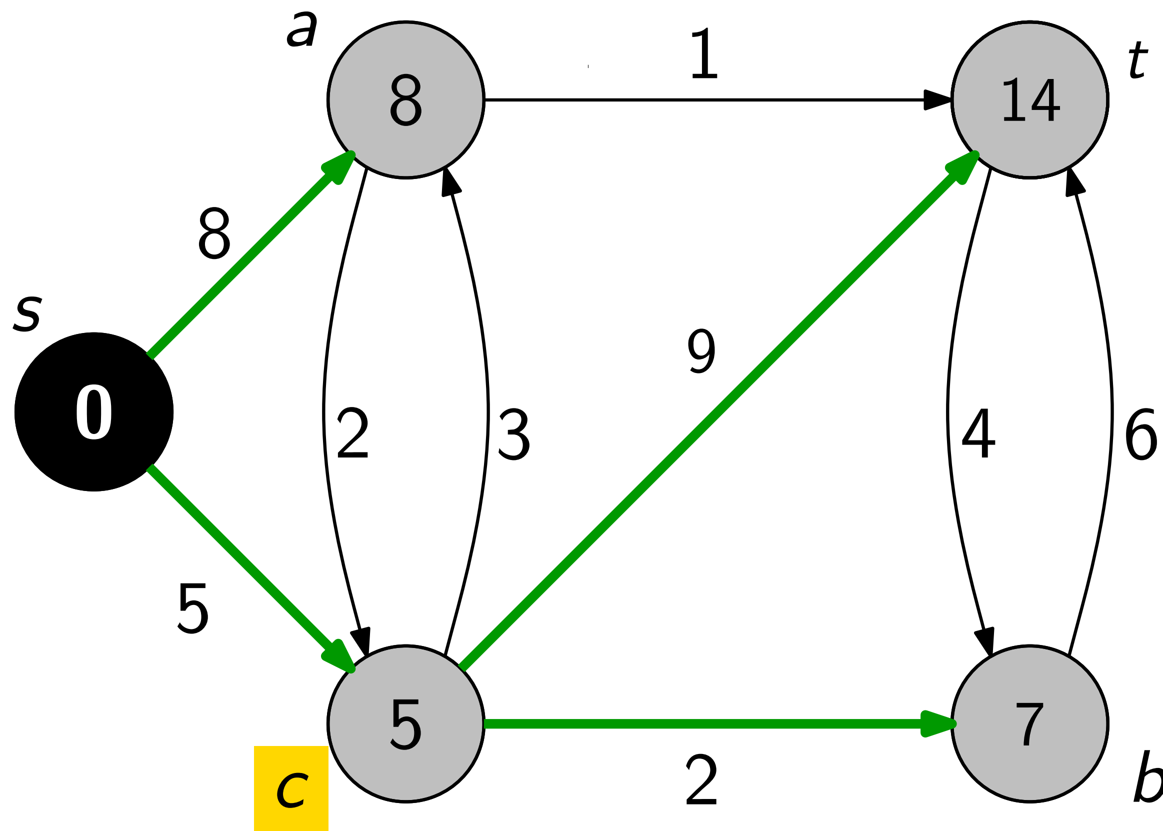
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

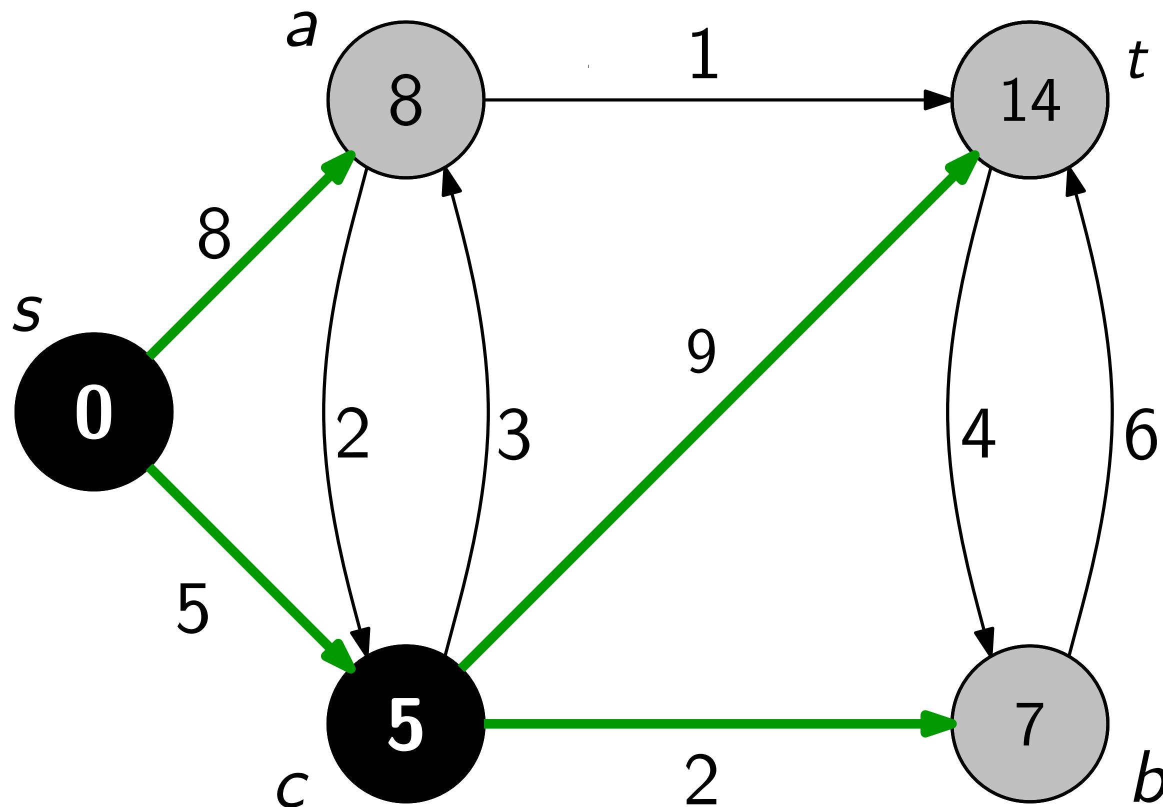
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

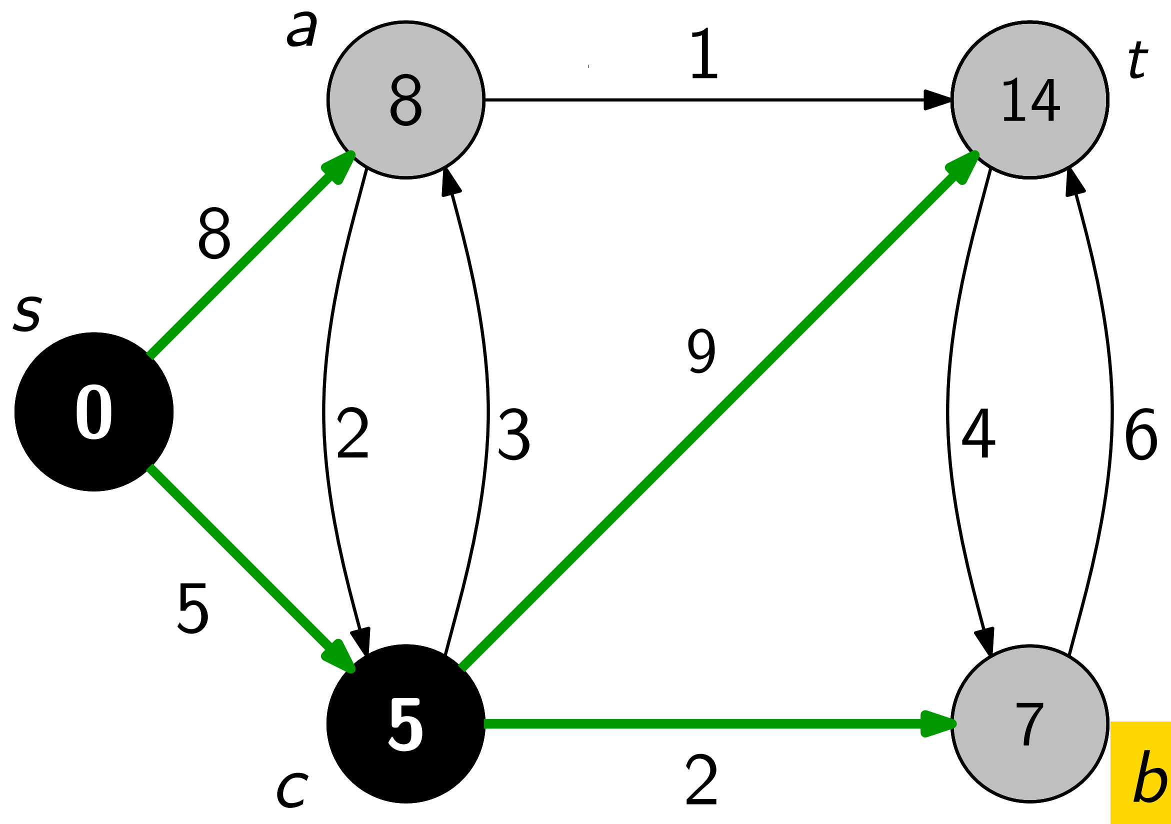
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

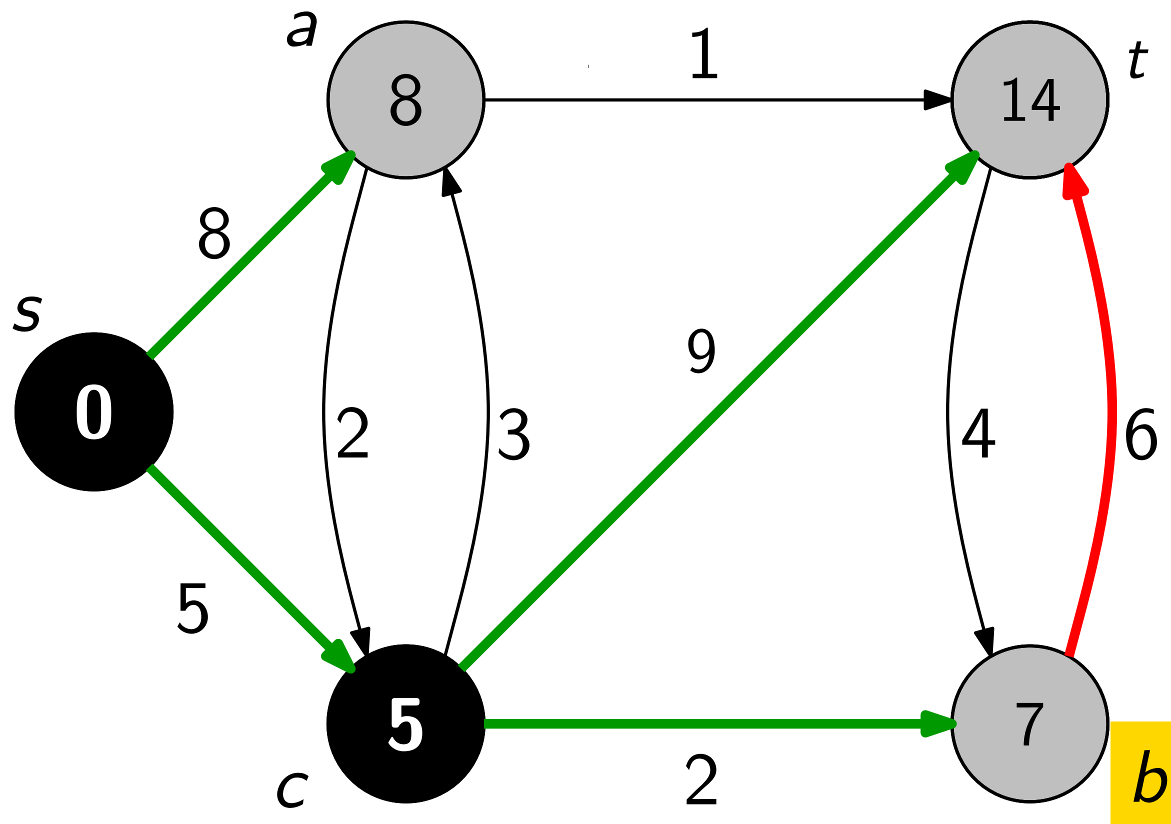
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

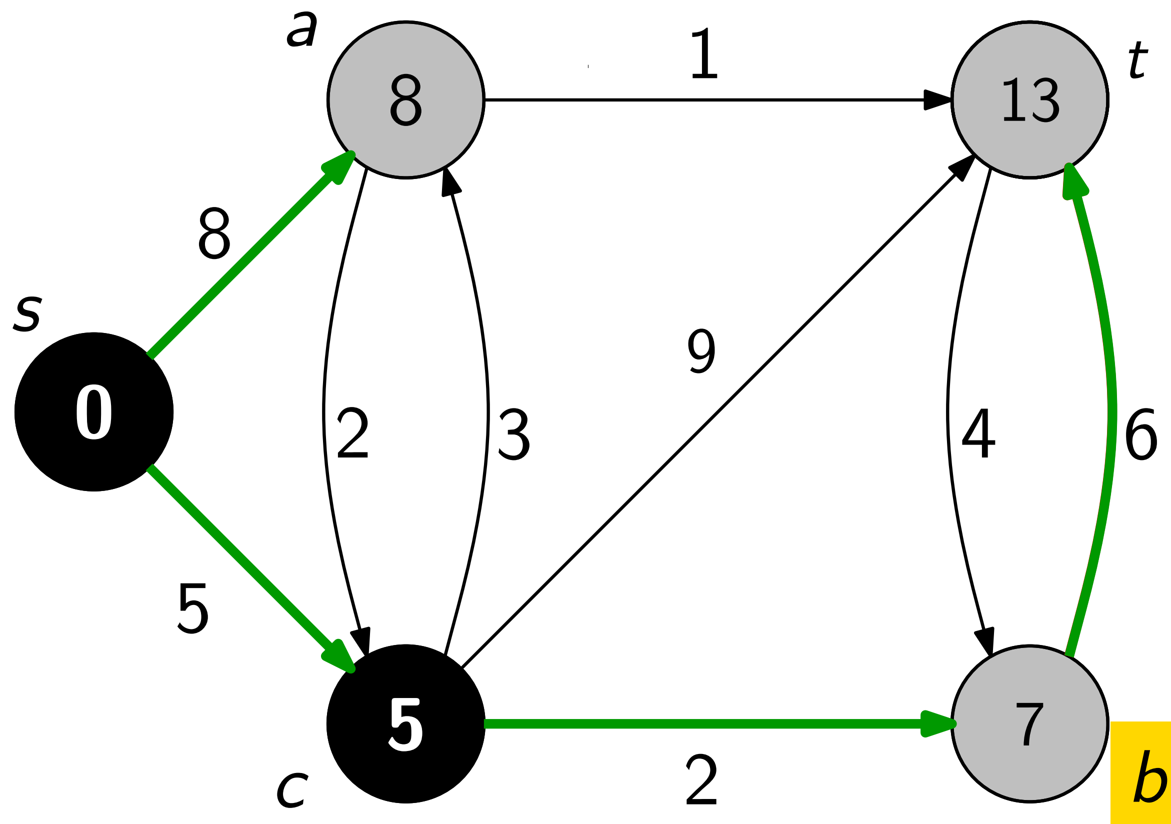
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

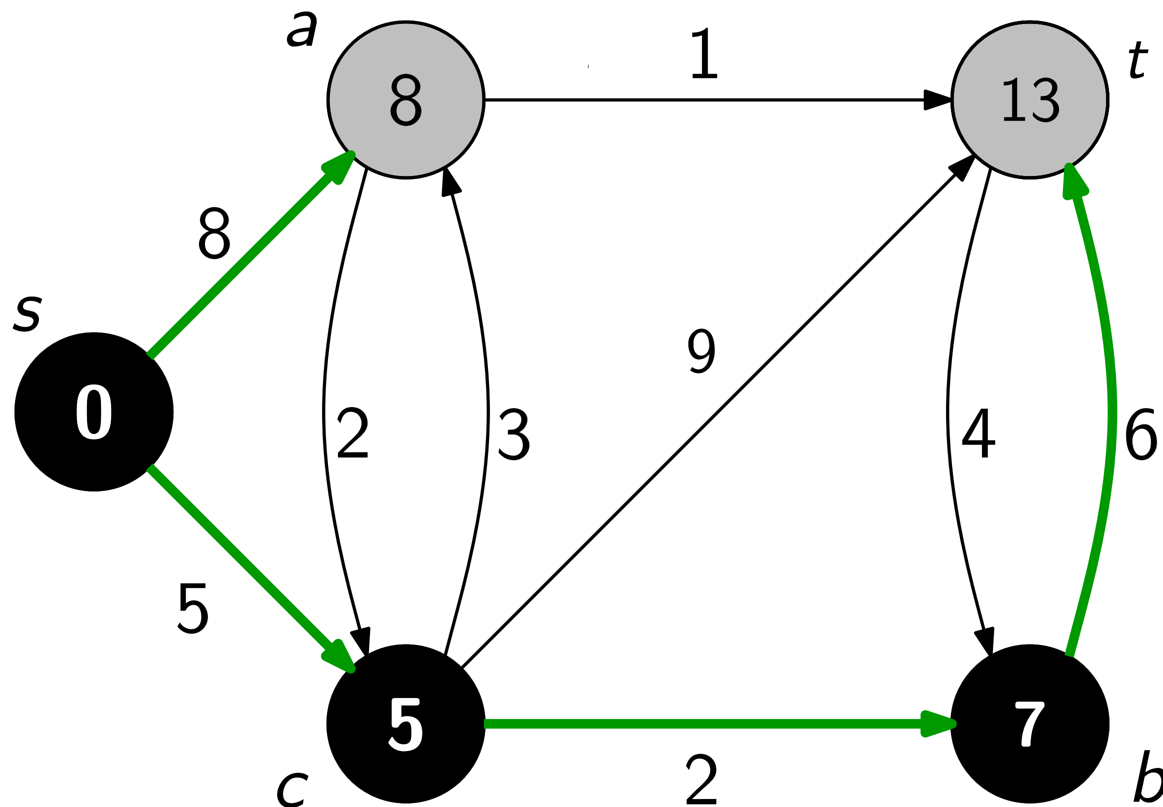
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

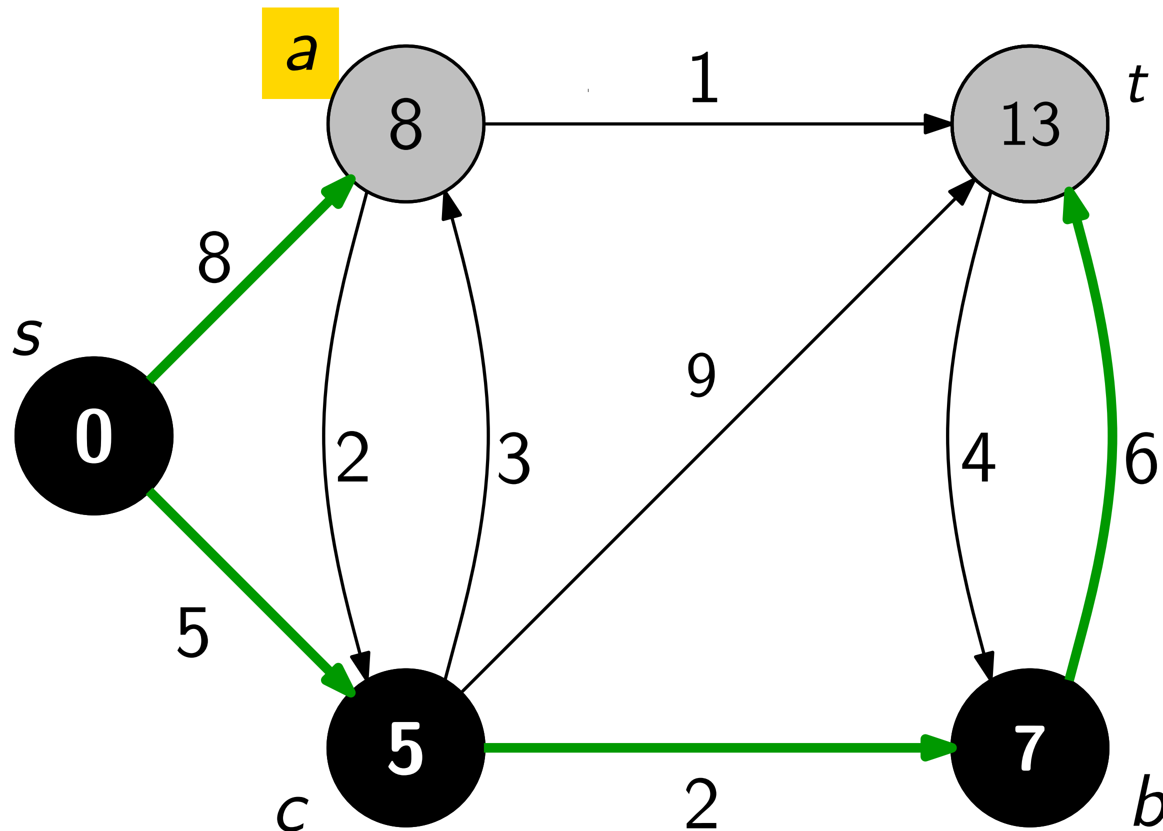
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

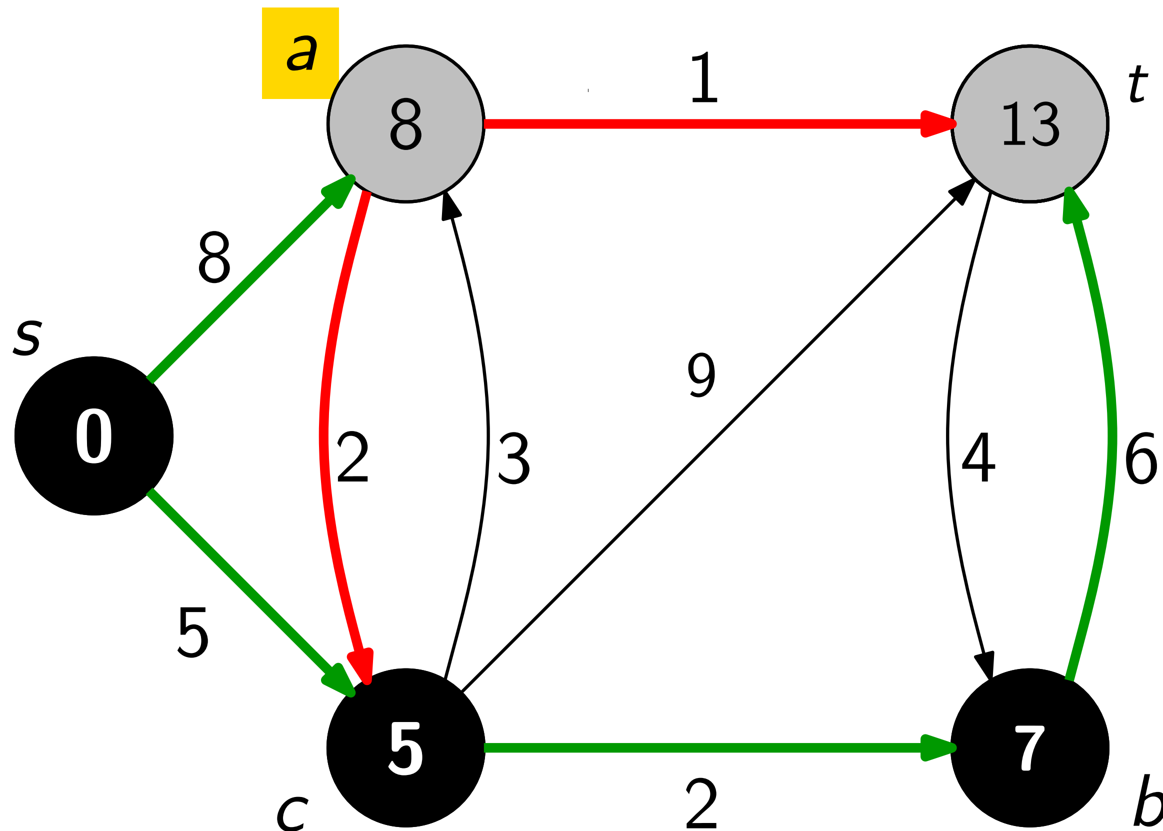
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

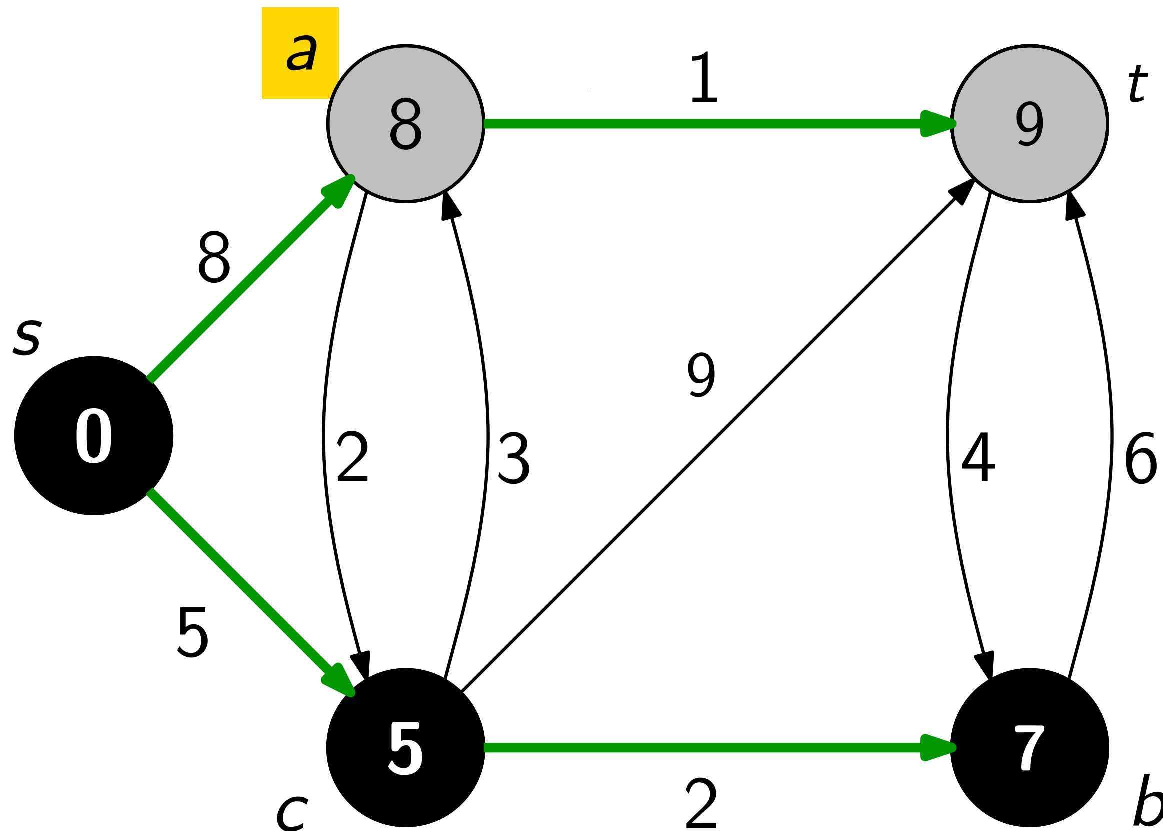
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

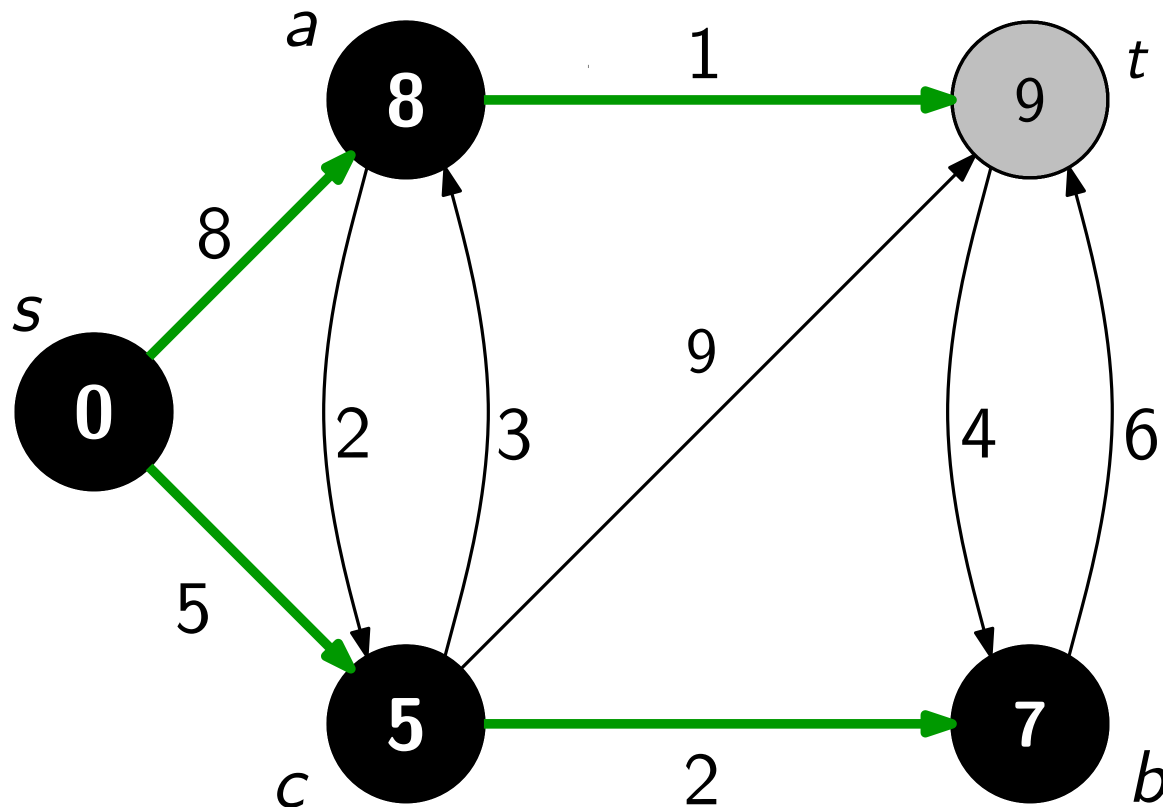
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

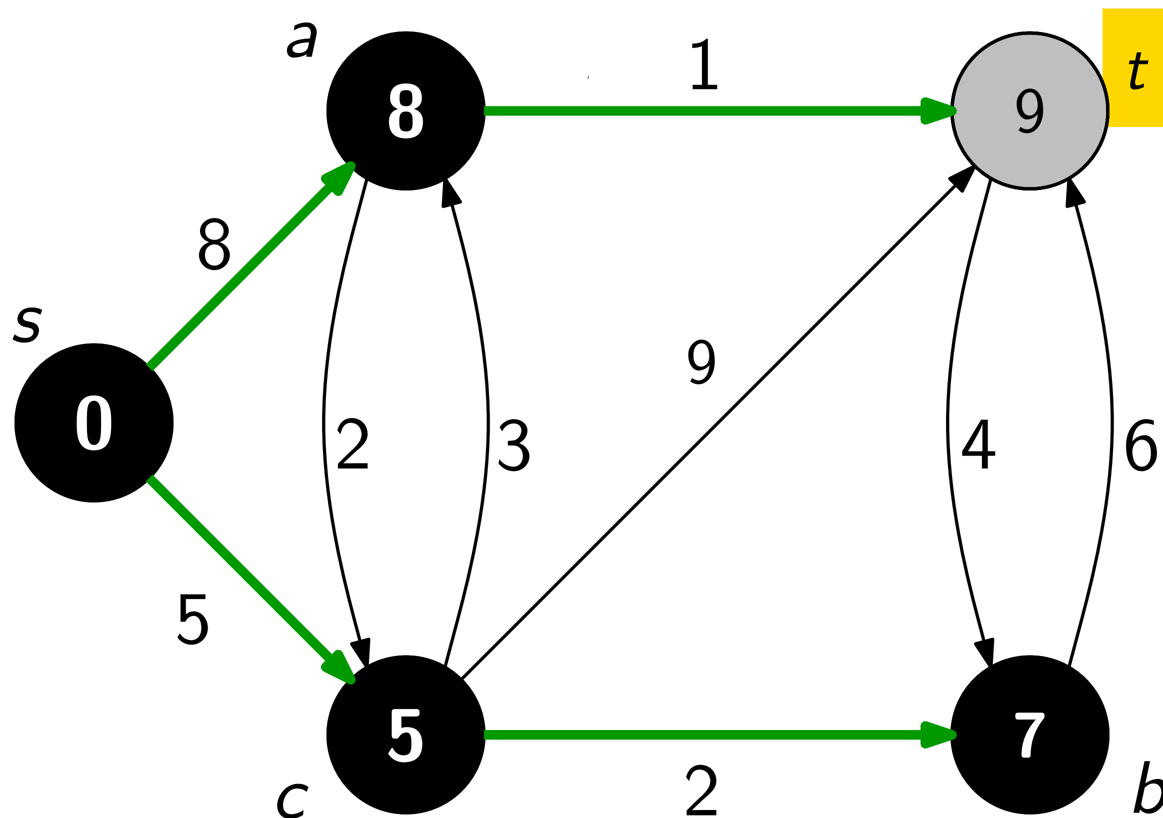
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

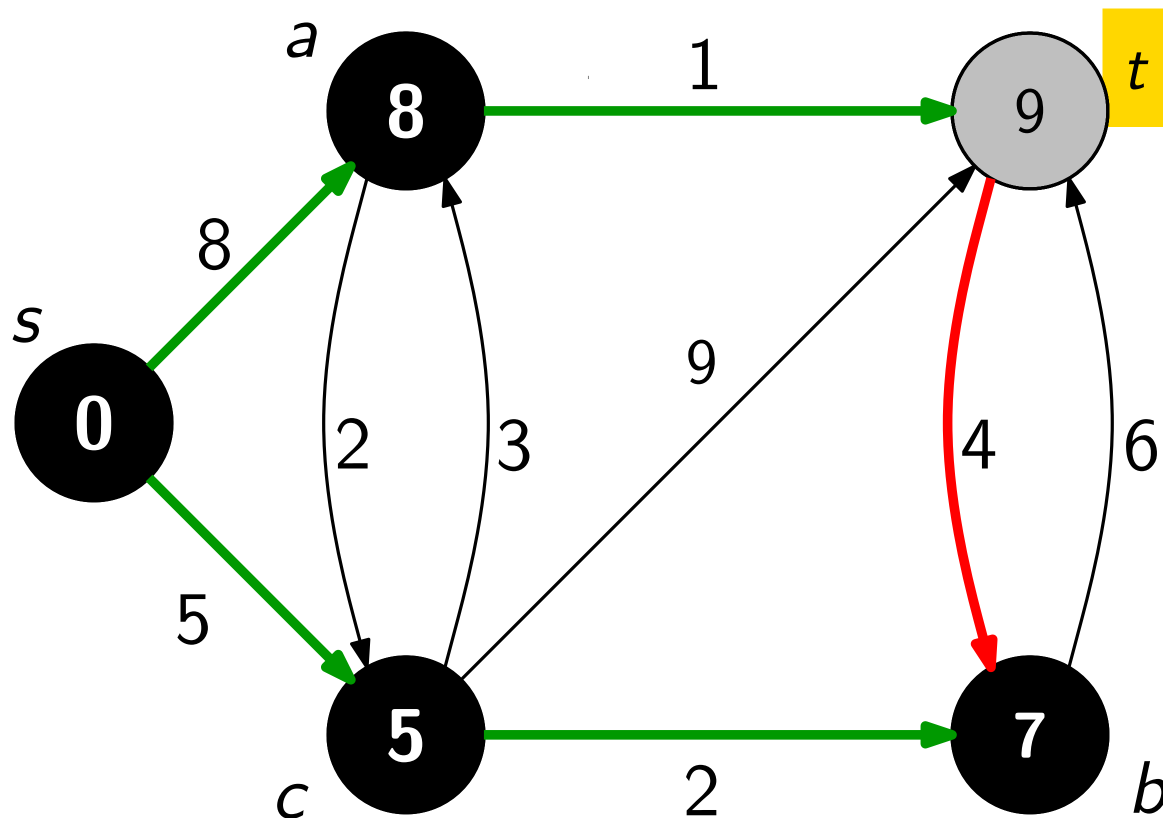
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

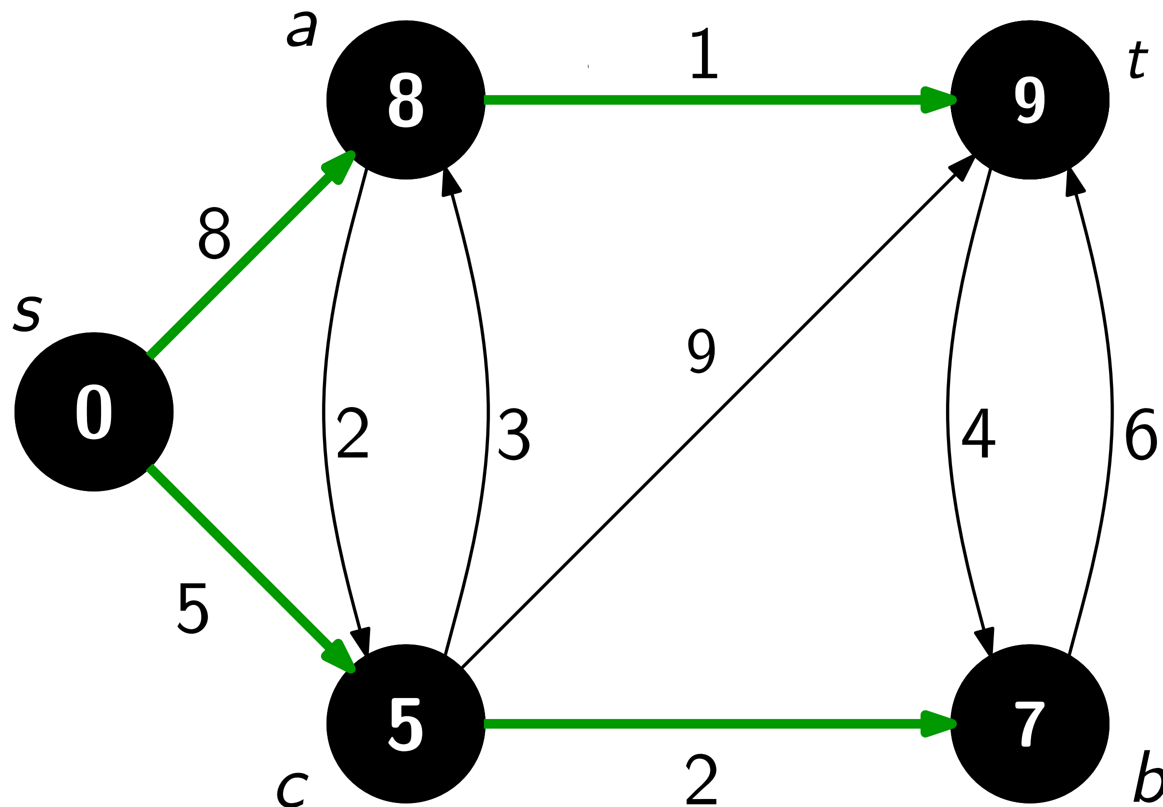
Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



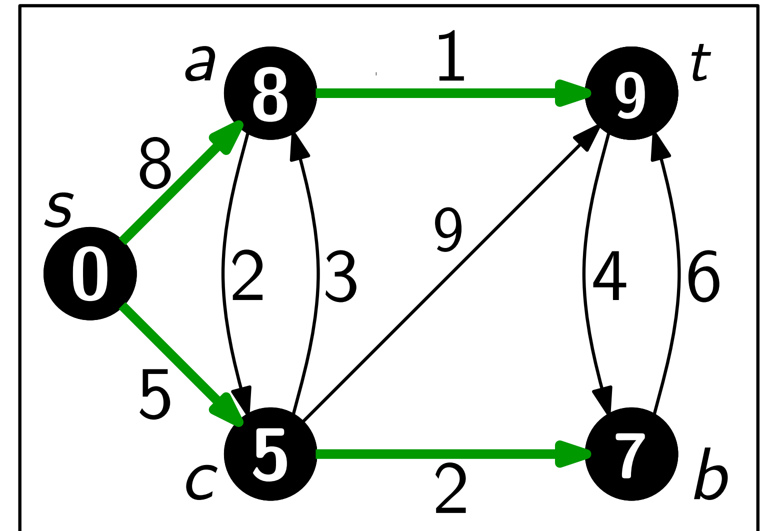
Dijkstra – Beispiel

Eingabe: gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen Kantengewichten und Knoten s

Ausgabe: kürzeste s-t-Wege in G mit Vorgänger-Zeiger π



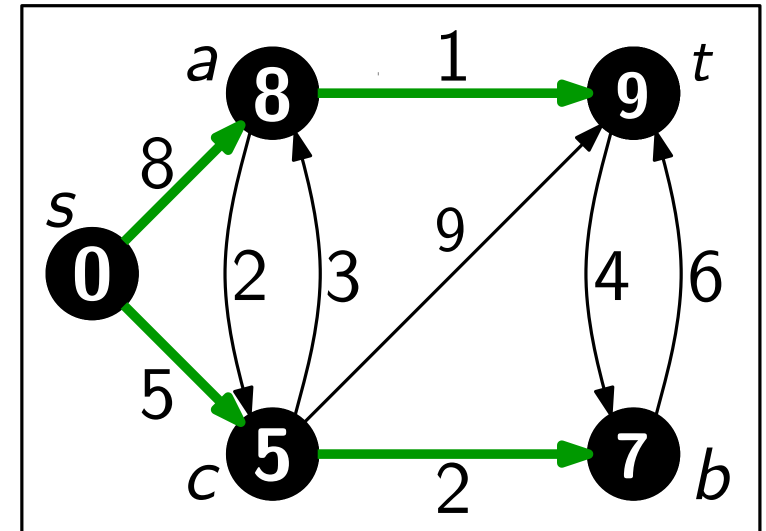
Dijkstra – Pseudocode



Initialize(Graph G, Vertex s)
foreach $u \in V$ **do**
$u.color = white$
$u.d = \infty$
$u.\pi = nil$
$s.color = gray$
$s.d = 0$

Dijkstra – Pseudocode

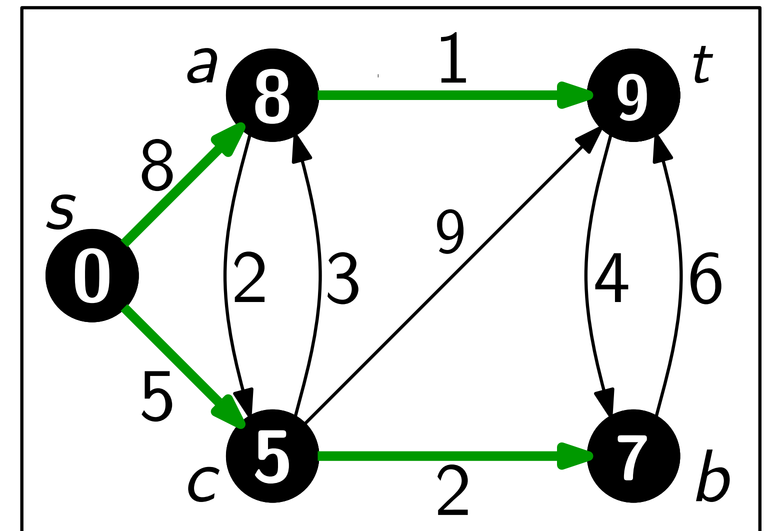
```
Dijkstra(WeightedGraph G, Vertex s)
  Initialize(G, s)
  Q = new PriorityQueue(V, d)
  while not Q.Empty() do
      u = Q.ExtractMin()
      foreach v ∈ Adj[u] do
          Relax(u, v; w)
      u.color = black
```




```
Initialize(Graph G, Vertex s)
  foreach u ∈ V do
      u.color = white
      u.d = ∞
      u.π = nil
  s.color = gray
  s.d = 0
```

Dijkstra – Pseudocode

```
Dijkstra(WeightedGraph G, Vertex s)
  Initialize(G, s)
  Q = new PriorityQueue(V, d)
  while not Q.Empty() do
      u = Q.ExtractMin()
      foreach v ∈ Adj[u] do
          Relax(u, v; w)
      u.color = black
```




```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

```
Initialize(Graph G, Vertex s)
  foreach u ∈ V do
      u.color = white
      u.d = ∞
      u.π = nil
  s.color = gray
  s.d = 0
```

Dijkstra – Pseudocode

Laufzeit?

```
Dijkstra(WeightedGraph G, Vertex s)
  Initialize(G, s)
  Q = new PriorityQueue(V, d)
  while not Q.Empty() do
      u = Q.ExtractMin()
      foreach v ∈ Adj[u] do
          Relax(u, v; w)
      u.color = black
```

```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

```
Initialize(Graph G, Vertex s)
  foreach u ∈ V do
      u.color = white
      u.d = ∞
      u.π = nil
  s.color = gray
  s.d = 0
```

Dijkstra – Pseudocode

Laufzeit?

```
Dijkstra(WeightedGraph G, Vertex s)
  Initialize(G, s)
  Q = new PriorityQueue(V, d)
  while not Q.Empty() do
      u = Q.ExtractMin()
      foreach v ∈ Adj[u] do
          Relax(u, v; w)
      u.color = black
```

$O(V)$ Zeit

```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

```
Initialize(Graph G, Vertex s)
  foreach u ∈ V do
      u.color = white
      u.d = ∞
      u.π = nil
  s.color = gray
  s.d = 0
```

Dijkstra – Pseudocode

Laufzeit?

```
Dijkstra(WeightedGraph G, Vertex s)
  Initialize(G, s)
  Q = new PriorityQueue(V, d)
  while not Q.Empty() do
    u = Q.ExtractMin()
    foreach v ∈ Adj[u] do
      Relax(u, v; w)
    u.color = black
```

$O(V)$ Zeit

```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
    v.color = gray
    v.d = u.d + w(u, v)
    v.π = u
    Q.DecreaseKey(v, v.d)
```

Dijkstra – Pseudocode

Laufzeit?

```
Dijkstra(WeightedGraph G, Vertex s)
  Initialize(G, s)
  Q = new PriorityQueue(V, d)
  while not Q.Empty() do
      u = Q.ExtractMin()
      foreach v ∈ Adj[u] do
          Relax(u, v; w)
      u.color = black
```

$O(V)$ Zeit

Genau $|V|$ mal.

```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

Dijkstra – Pseudocode

Laufzeit?

```
Dijkstra(WeightedGraph G, Vertex s)
  Initialize(G, s)
  Q = new PriorityQueue(V, d)
  while not Q.Empty() do
      u = Q.ExtractMin()
      foreach v ∈ Adj[u] do
          Relax(u, v; w)
      u.color = black
```

$O(V)$ Zeit

Genau $|V|$ mal.

```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

Dijkstra – Pseudocode

Laufzeit?

```
Dijkstra(WeightedGraph G, Vertex s)
  Initialize(G, s)
  Q = new PriorityQueue(V, d)
  while not Q.Empty() do
      u = Q.ExtractMin()
      foreach v ∈ Adj[u] do
          Relax(u, v; w)
      u.color = black
```

```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

$O(V)$ Zeit

Genau $|V|$ mal.

Für jeden Knoten $u \in V$ genau $|\text{Adj}[u]|$ ($= \deg u$) mal,

Dijkstra – Pseudocode

Laufzeit?

```
Dijkstra(WeightedGraph G, Vertex s)
  Initialize(G, s)
  Q = new PriorityQueue(V, d)
  while not Q.Empty() do
      u = Q.ExtractMin()
      foreach v ∈ Adj[u] do
          Relax(u, v; w)
      u.color = black
```

```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

$O(V)$ Zeit

Genau $|V|$ mal.

Für jeden Knoten $u \in V$ genau $|\mathrm{Adj}[u]|$ ($= \deg u$) mal, also insg. $2|E|$ mal.

Dijkstra – Pseudocode

Laufzeit?

```
Dijkstra(WeightedGraph G, Vertex s)
  Initialize(G, s)
  Q = new PriorityQueue(V, d)
  while not Q.Empty() do
      u = Q.ExtractMin()
      foreach v ∈ Adj[u] do
          Relax(u, v; w)
      u.color = black
```

```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

$O(V)$ Zeit

Genau $|V|$ mal.

Für jeden Knoten $u \in V$ genau $|\mathrm{Adj}[u]|$ $(= \deg u)$ mal, also insg. $2|E|$ mal.

Also wird DecreaseKey $\leq 2|E|$ mal aufgerufen.

Dijkstra – Pseudocode

Laufzeit?

Dijkstra(WeightedGraph G, Vertex s)
  Initialize(G, s)
  $Q =$ **new** PriorityQueue(V, d)
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in \text{Adj}[u]$ **do**
      Relax($u, v; w$)
    $u.color = black$

Relax($u, v; w$)
  if $v.d > u.d + w(u, v)$ **then**
    $v.color = gray$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
    Q.DecreaseKey($v, v.d$)

$O(V)$ Zeit

Genau $|V|$ mal.

Für jeden Knoten $u \in V$ genau $|\text{Adj}[u]|$ ($= \deg u$) mal, also insg. $2|E|$ mal.

Also wird DecreaseKey $\leq 2|E|$ mal aufgerufen.

Dijkstra – Pseudocode

Laufzeit?

```
Dijkstra(WeightedGraph G, Vertex s)
  Initialize(G, s)
  Q = new PriorityQueue(V, d)
  while not Q.Empty() do
      u = Q.ExtractMin()
      foreach v ∈ Adj[u] do
          Relax(u, v; w)
      u.color = black
```

```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

$O(V)$ Zeit

Genau $|V|$ mal.

Für jeden Knoten $u \in V$ genau $|\mathrm{Adj}[u]|$ $(= \deg u)$ mal, also insg. $2|E|$ mal.

Also wird DecreaseKey $\leq 2|E|$ mal aufgerufen.

Gesamt: Implementierungsabh.!
beste: $O(E + V \log V)$

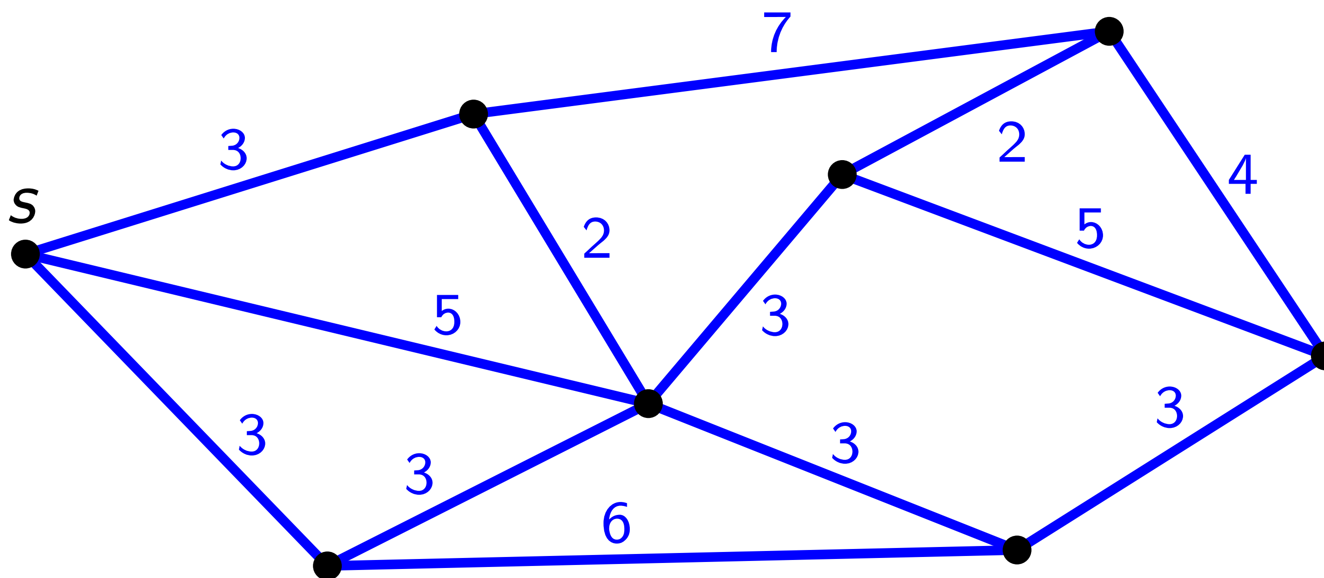
Übersicht

1. Graphdurchlaufstrategien
2. Kürzeste Wege
3. Minimale Spannbäume
   3.1 Jarník-Prim
       Beispiel
       Pseudocode
   3.2 Kruskal

Jarník-Prim – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten und Knoten s

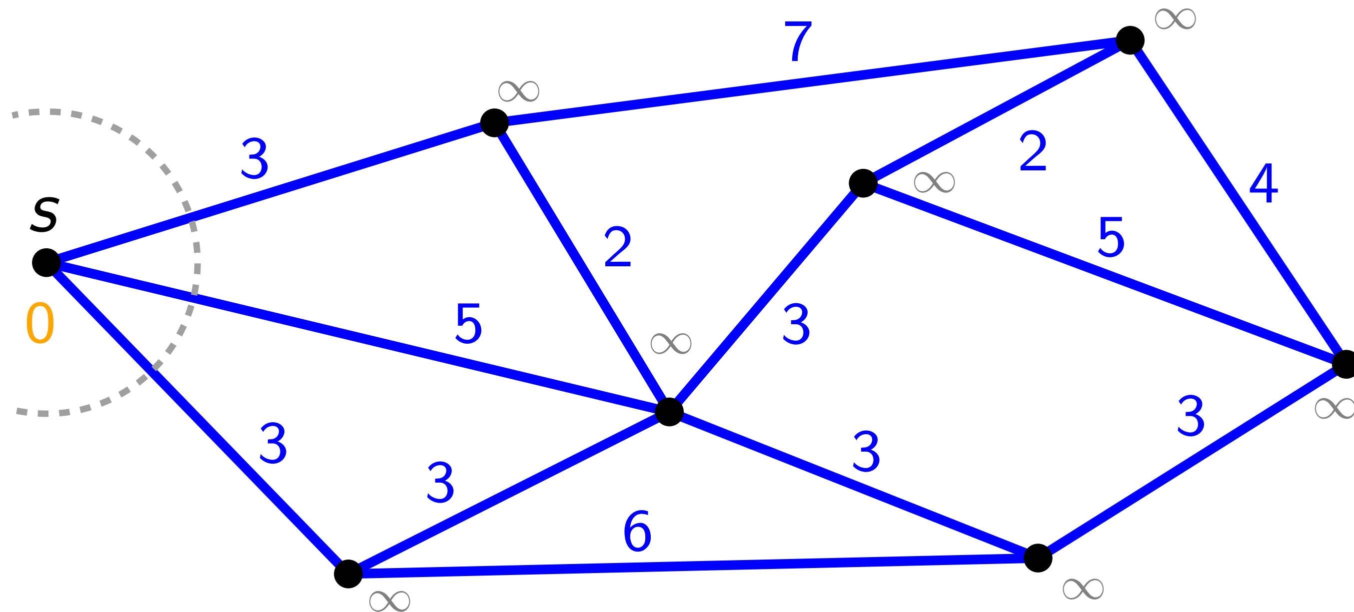
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Jarník-Prim – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten und Knoten s

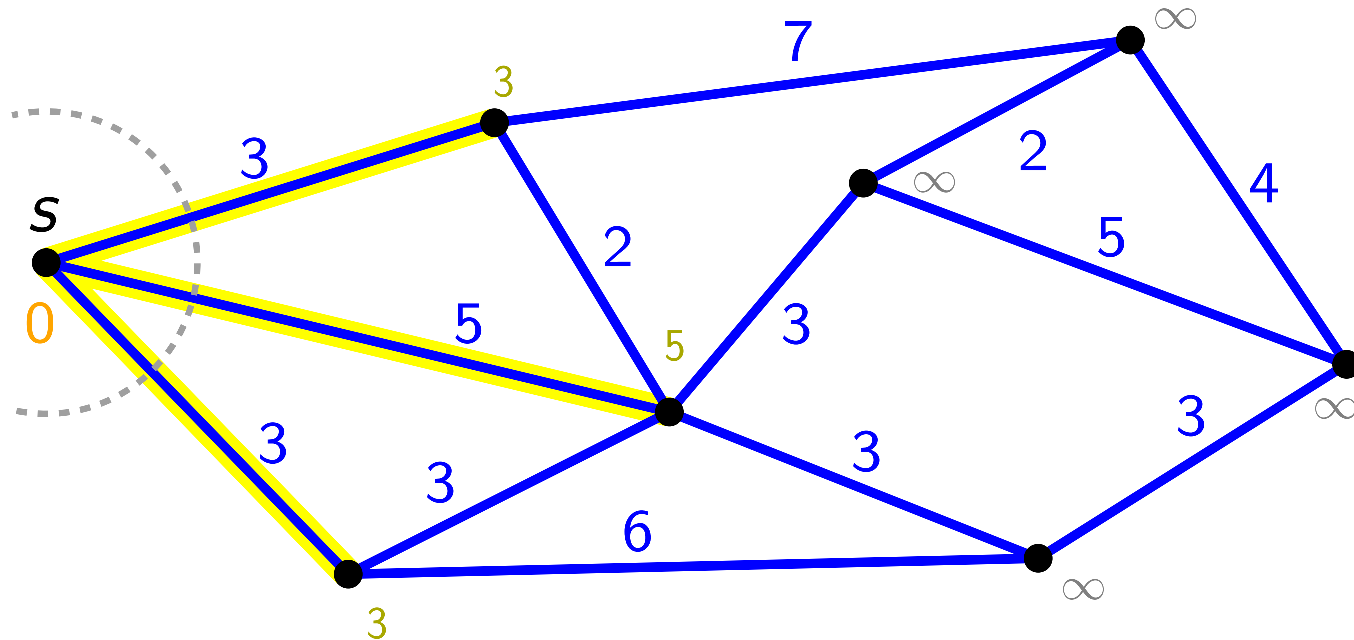
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Jarník-Prim – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten und Knoten s

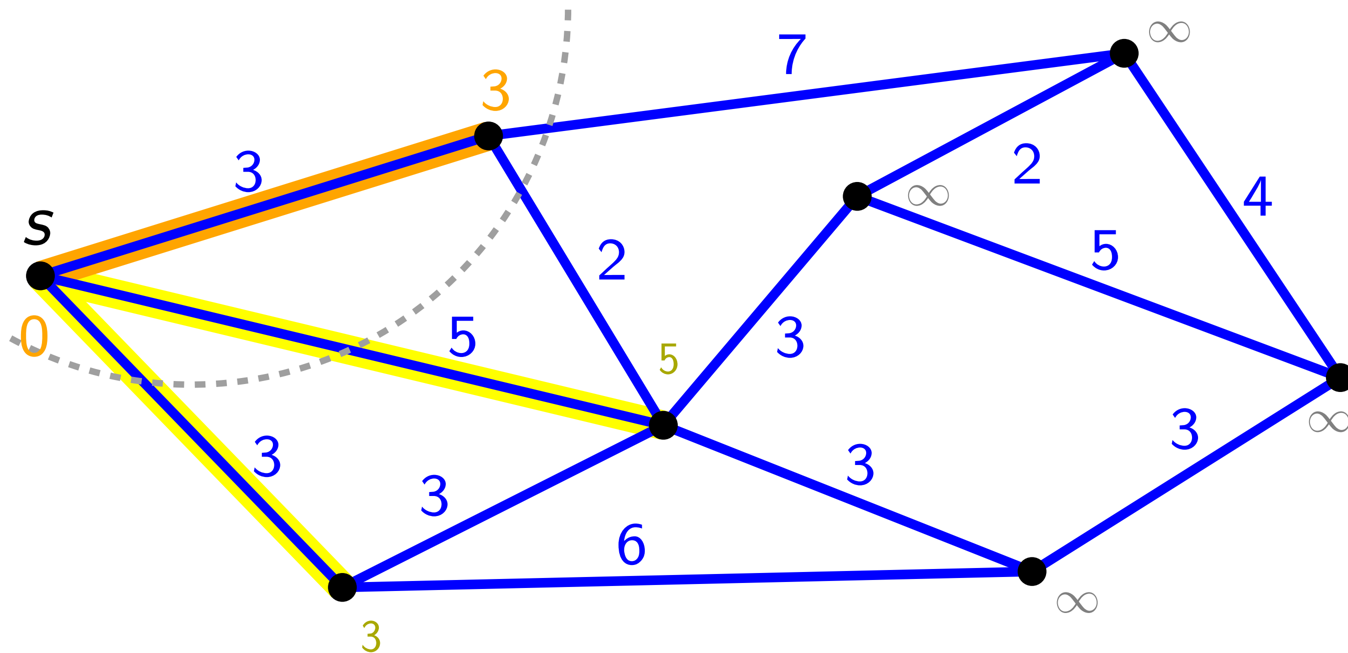
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Jarník-Prim – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten und Knoten s

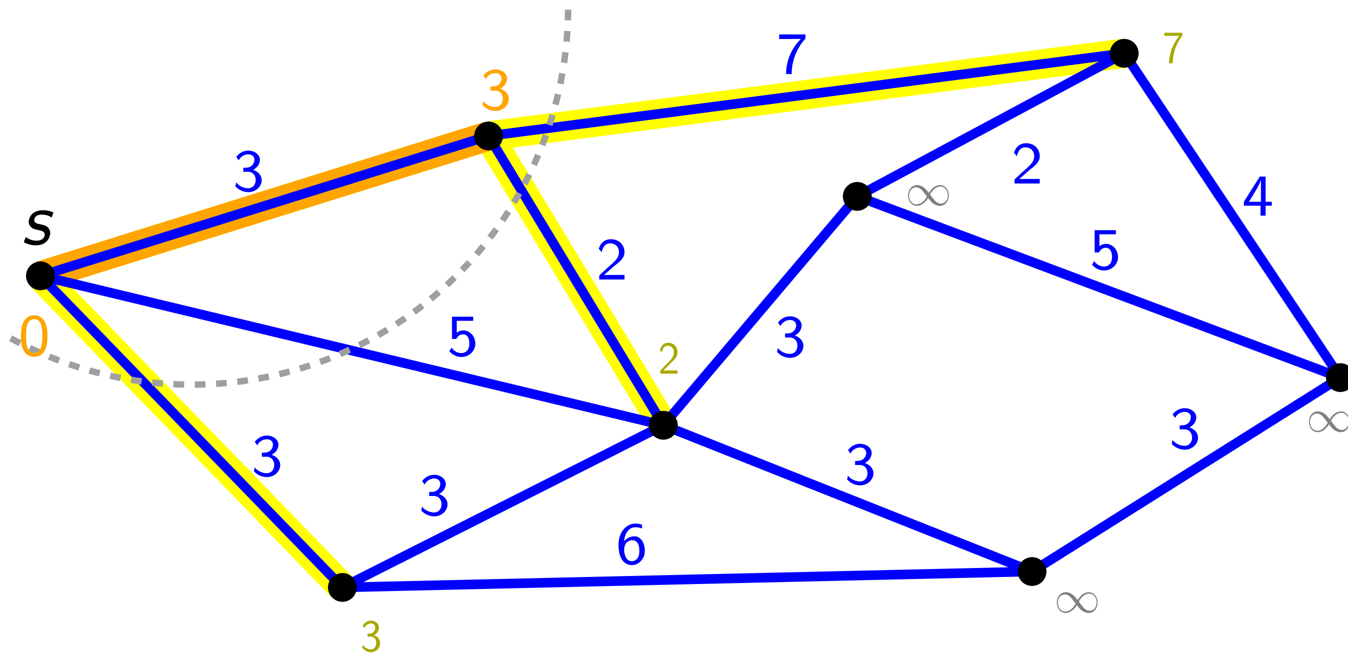
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Jarník-Prim – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten und Knoten s

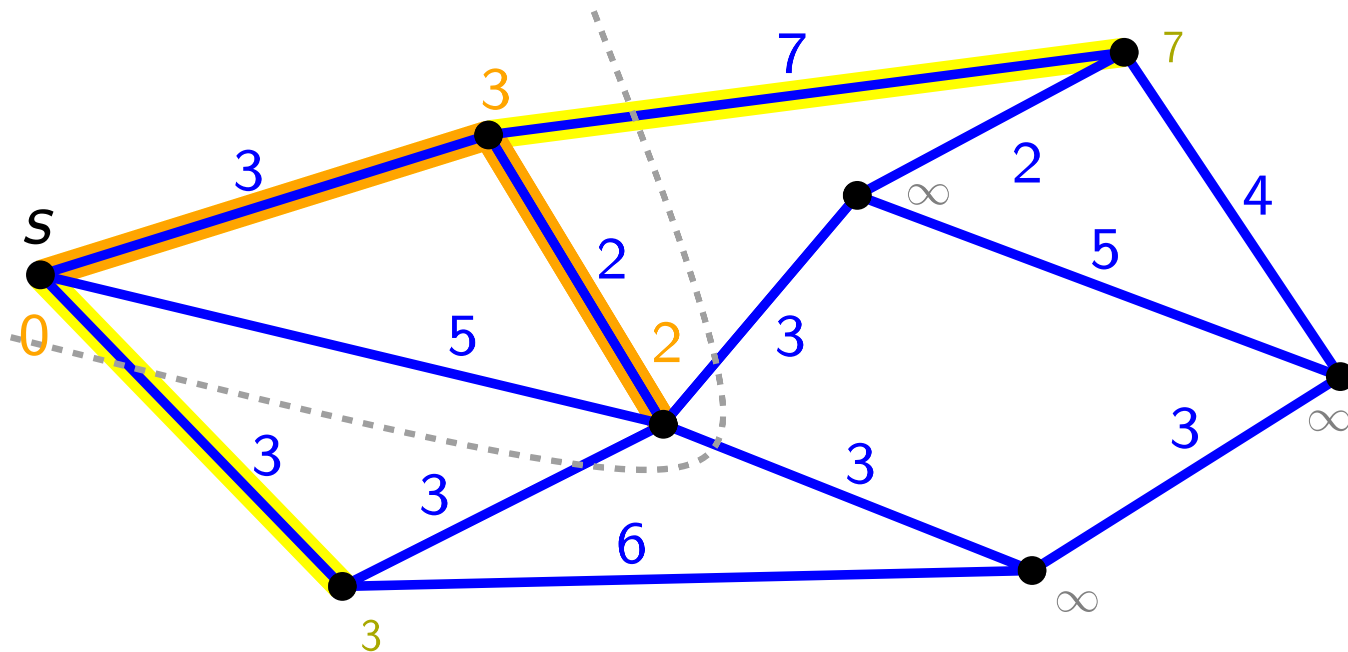
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Jarník-Prim – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V,E)$ mit Kantengewichten und Knoten s

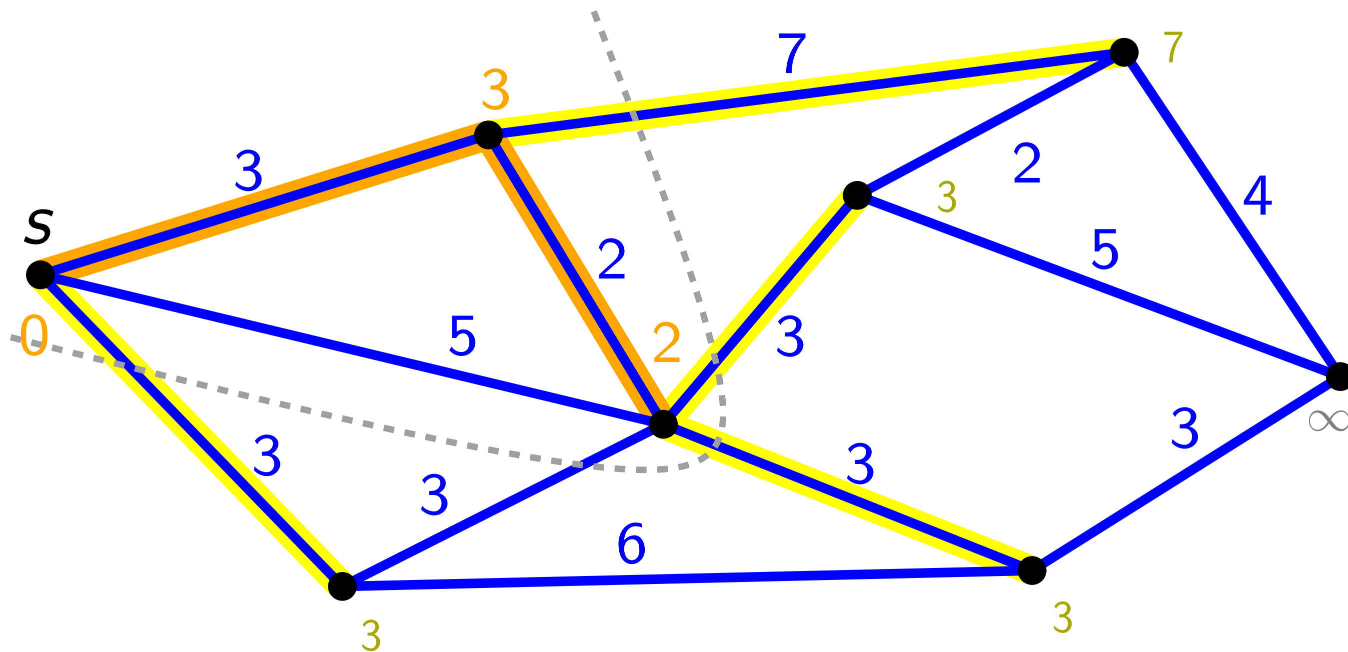
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Jarník-Prim – Beispiel

Eingabe: ungerichteteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten und Knoten s

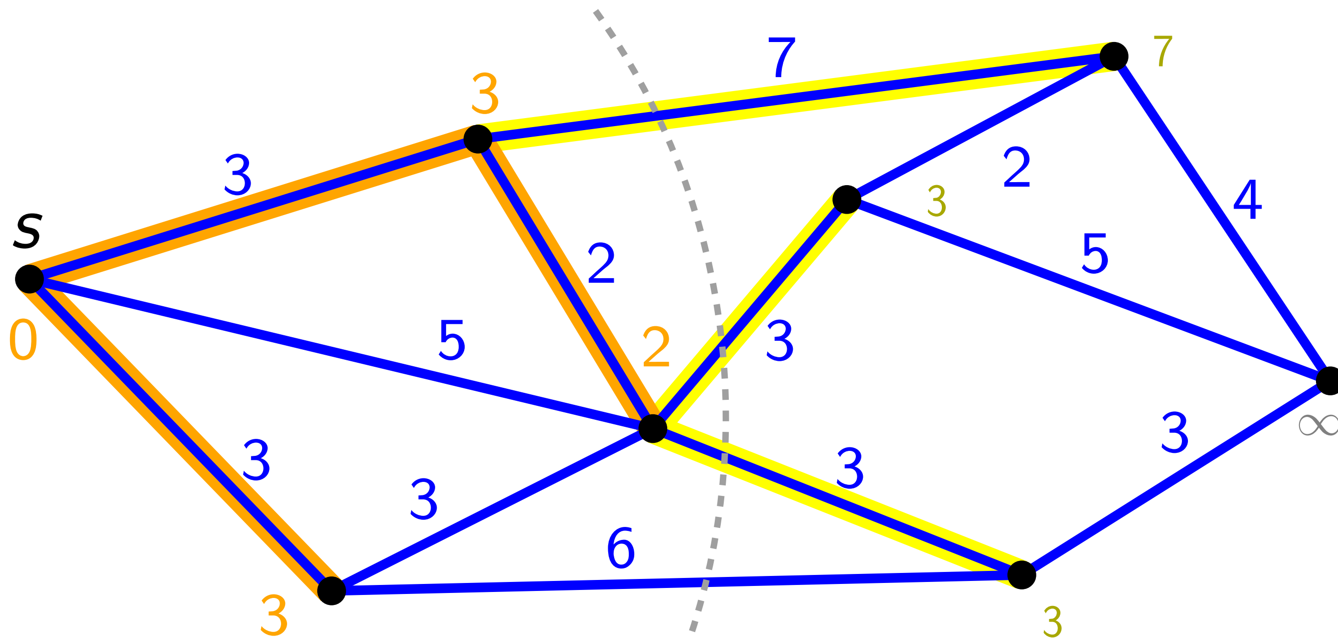
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Jarník-Prim – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten und Knoten s

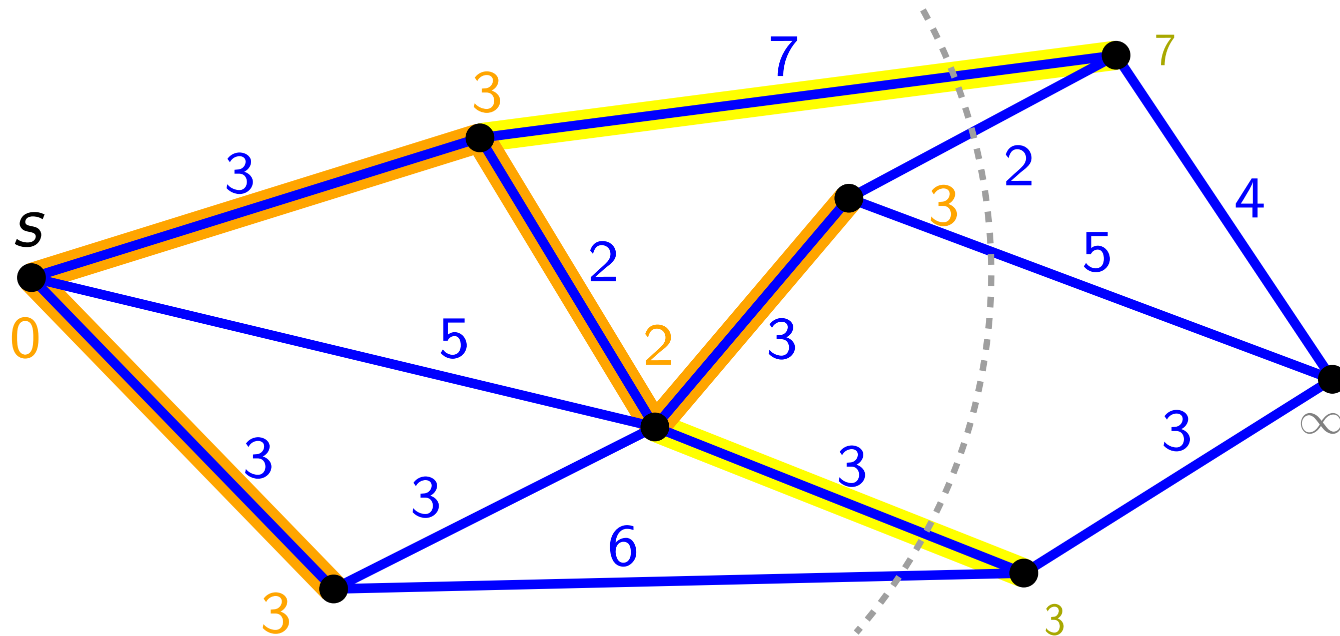
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Jarník-Prim – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten und Knoten s

Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Jarník-Prim – Beispiel

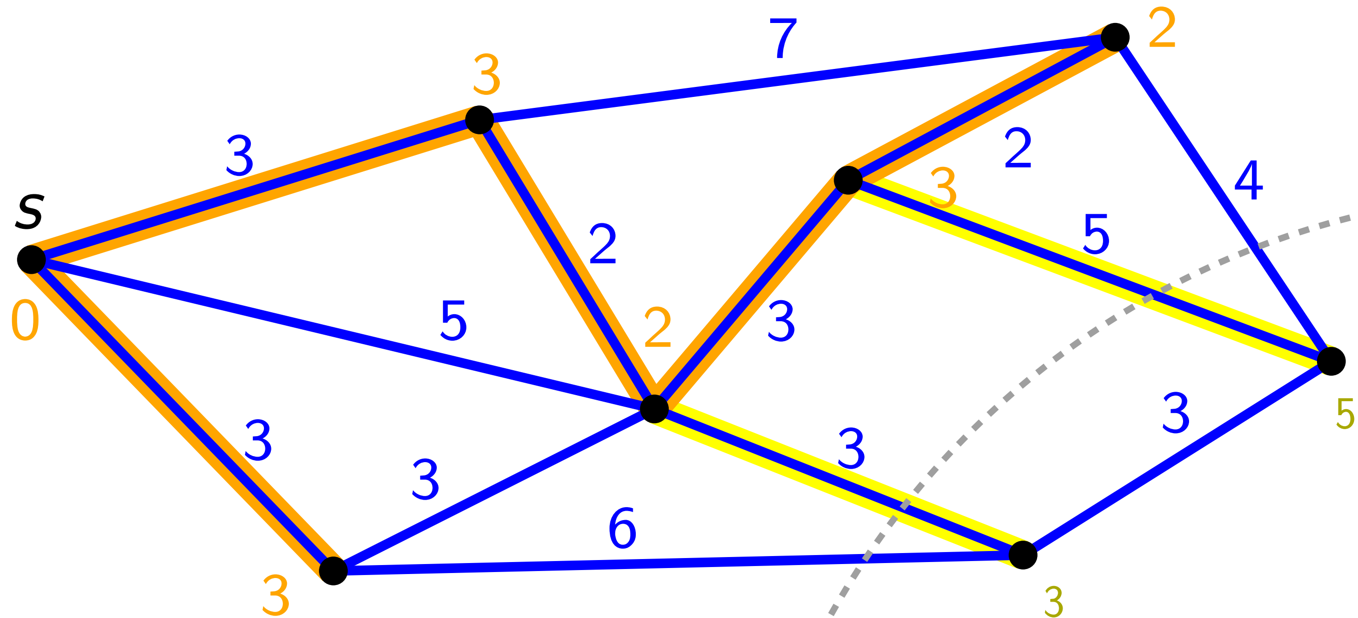
Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten und Knoten s

Ausgabe: gewichtsminimaler Baum T, der G aufspannt

Jarník-Prim – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten und Knoten s

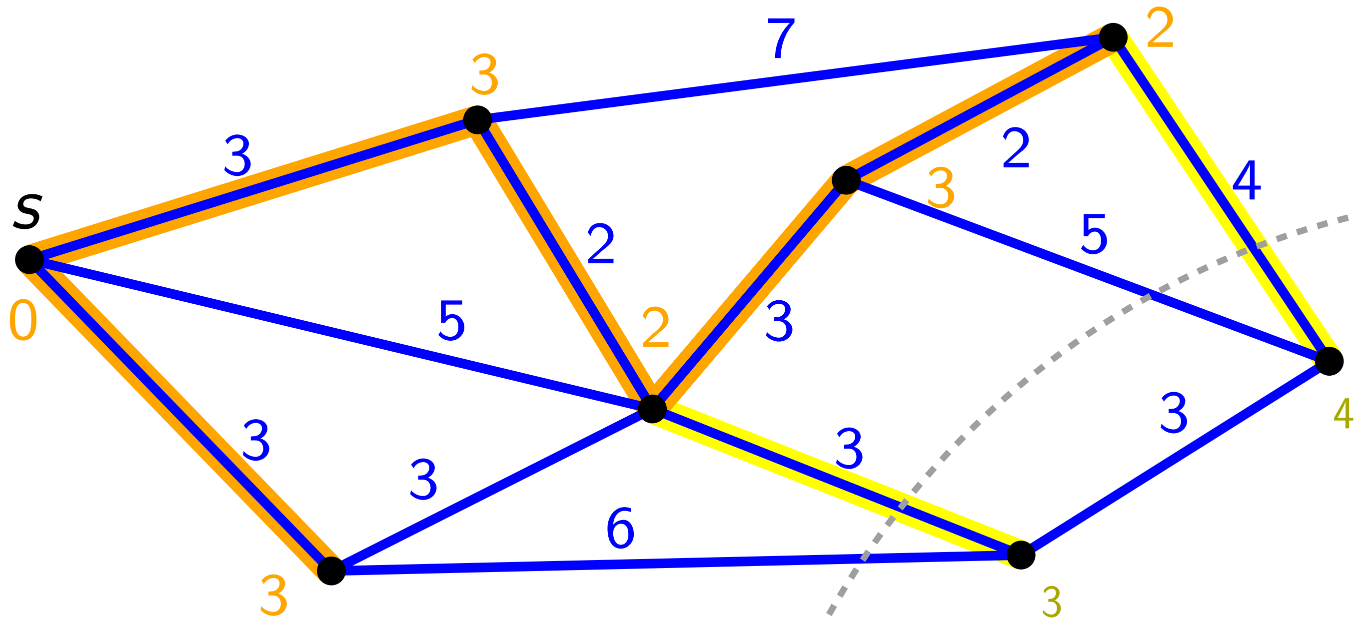
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Jarník-Prim – Beispiel

Eingabe: ungerichteteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten und Knoten s

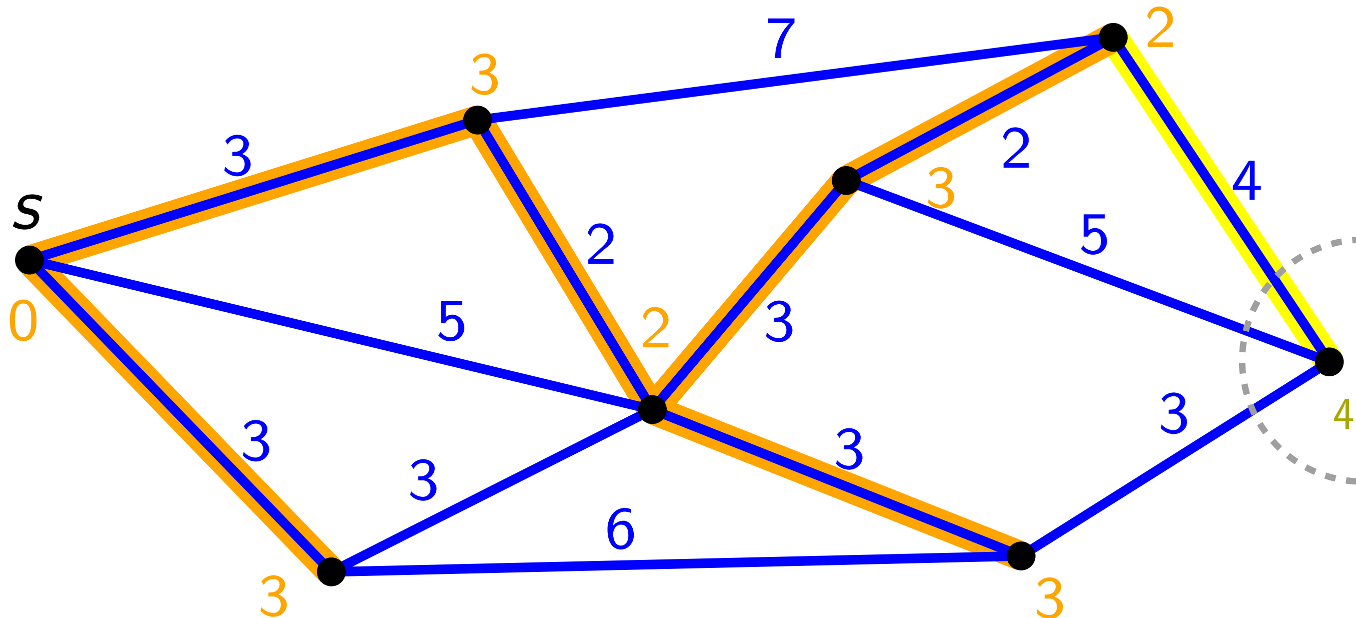
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Jarník-Prim – Beispiel

Eingabe: ungerichteteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten und Knoten s

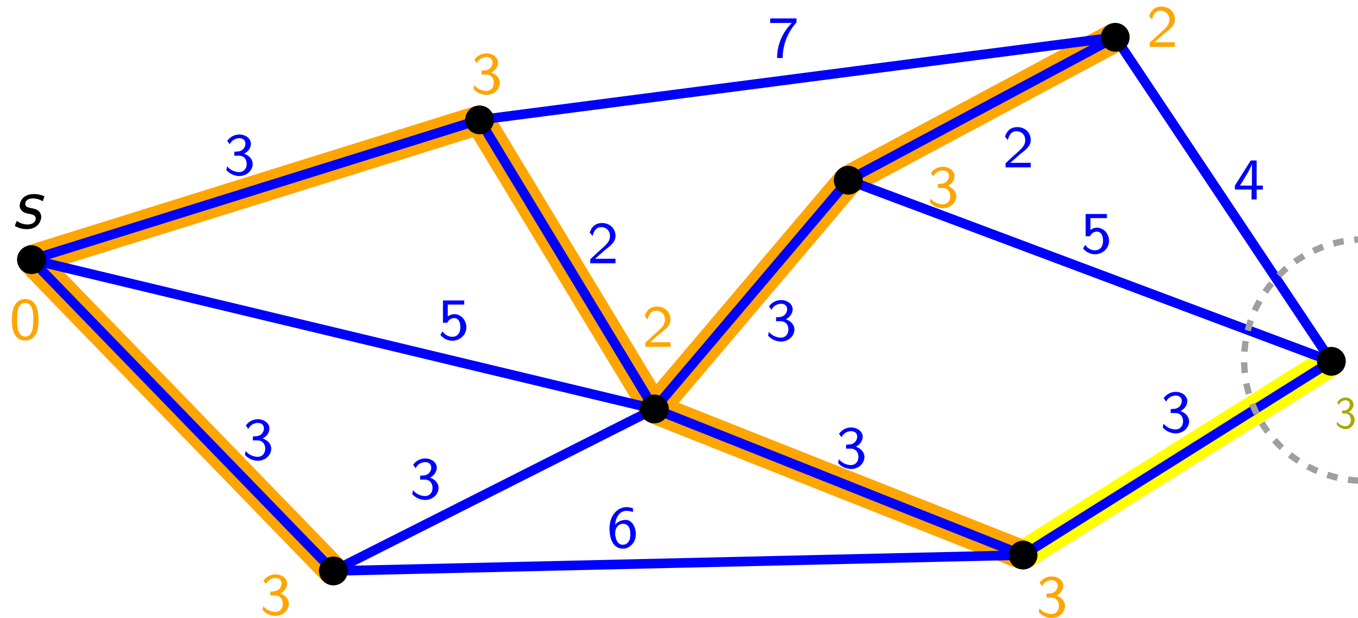
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Jarník-Prim – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V,E)$ mit Kantengewichten und Knoten s

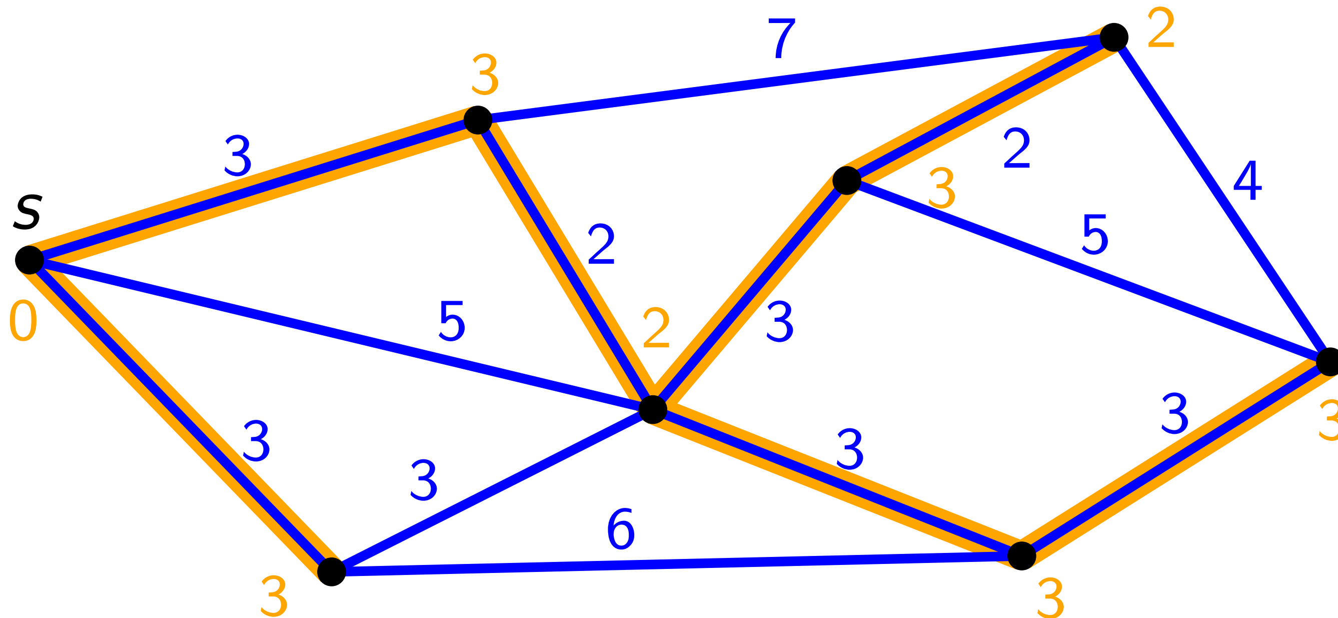
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Jarník-Prim – Beispiel

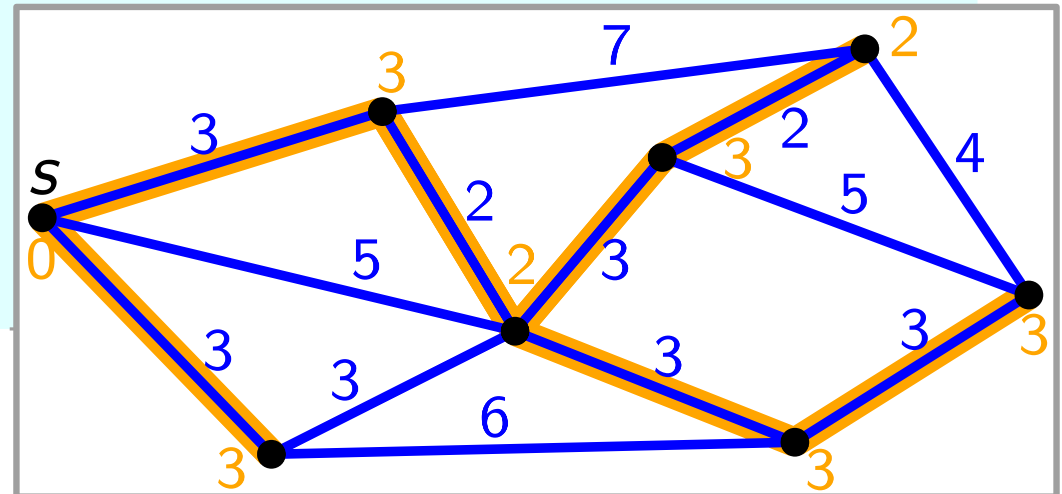
Eingabe: ungerichteteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten und Knoten s

Ausgabe: gewichtsminimaler Baum T, der G aufspannt



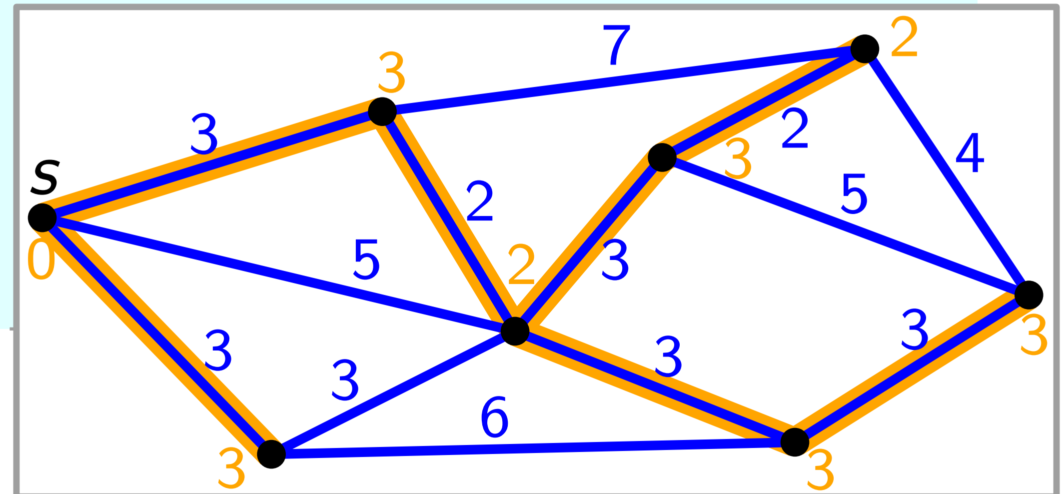
Jarník-Prim – Pseudocode

Dijkstra(WeightedUndirectedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  $Q =$ **new** PriorityQueue(V, d)
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in \mathrm{Adj}[u]$ **do**
      Relax $(u, v; w)$



Jarník-Prim – Pseudocode

```
JarníkPrimMST
~~Dijkstra~~(WeightedUndirectedGraph G = (V, E; w), Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V, d)
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            Relax(u, v; w)
```



Jarník-Prim – Pseudocode

JarníkPrimMST

~~Dijkstra~~(WeightedUndirectedGraph $G = (V, E; w)$, Vertex s)

Initialize(G, s)

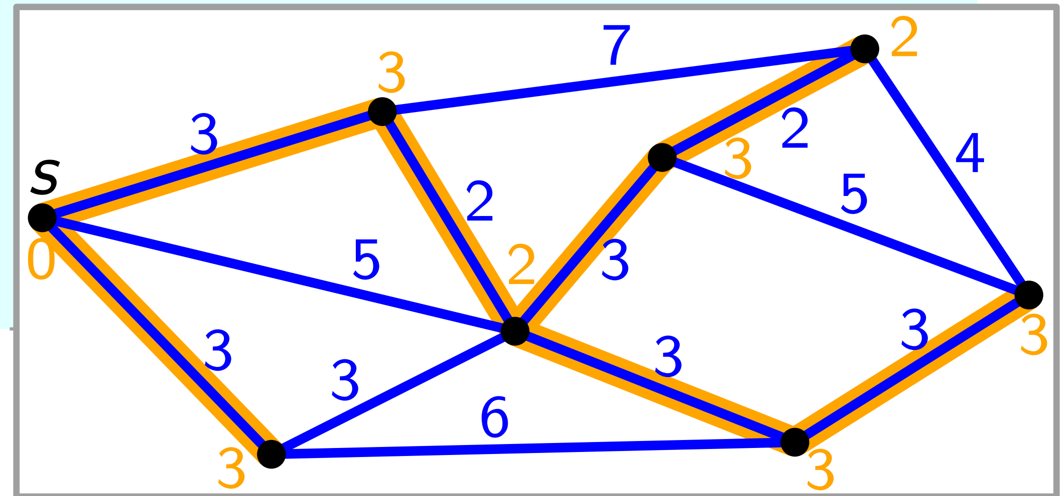
Q = **new** PriorityQueue(V, d)

while not Q.Empty() **do**

$u = Q$.ExtractMin()

foreach $v \in \text{Adj}[u]$ **do**

Relax′($u, v; w$)



Jarník-Prim – Pseudocode

JarníkPrimMST

~~Dijkstra~~(WeightedUndirectedGraph $G = (V, E; w)$, Vertex s)

Initialize(G, s)

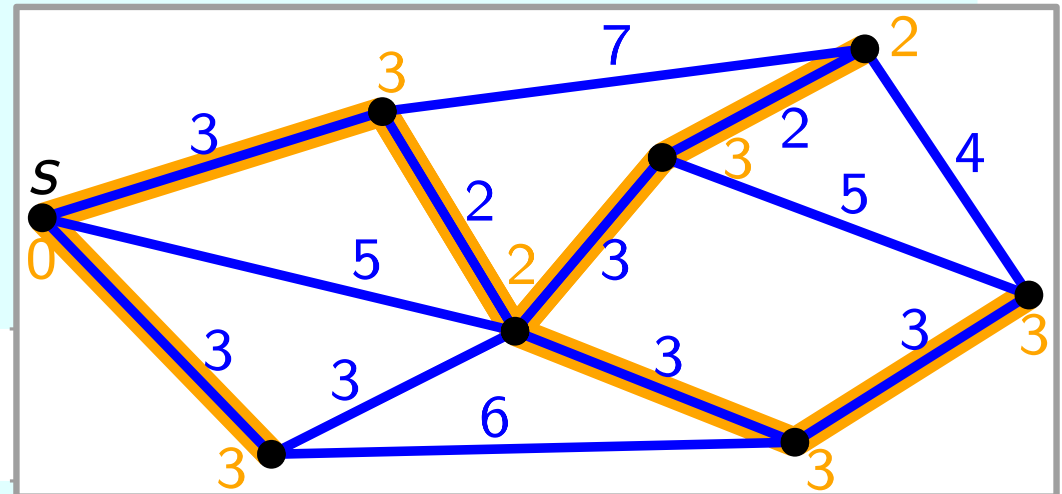
Q = **new** PriorityQueue(V, d)

while not Q.Empty() **do**

  $u = Q$.ExtractMin()

  foreach $v \in \text{Adj}[u]$ **do**

    Relax′($u, v; w$)



Relax ($u, v; w$)

if $v.d > u.d + w(u, v)$ **then**

  $v.d = u.d + w(u, v)$

  $v.\pi = u$

  Q.UpdateKey($v, v.d$)

Jarník-Prim – Pseudocode

JarníkPrimMST

~~Dijkstra~~(WeightedUndirectedGraph $G = (V, E; w)$, Vertex s)

Initialize(G, s)

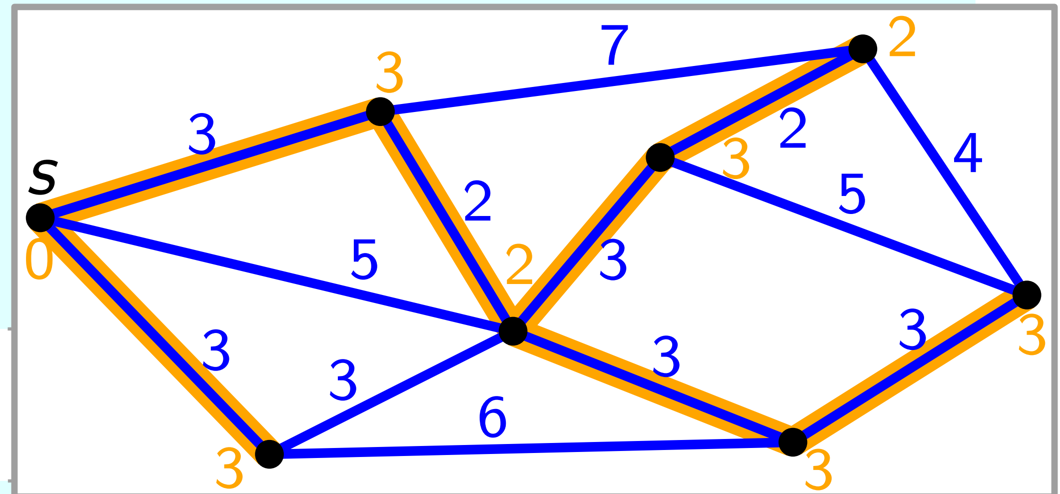
$Q =$ **new** PriorityQueue(V, d)

while not Q.Empty() **do**

$u = Q$.ExtractMin()

foreach $v \in \text{Adj}[u]$ **do**

Relax′($u, v; w$)



Relax′($u, v; w$)

if $v.d >$ ~~$u.d +$~~ $w(u, v)$ **then**

$v.d =$ ~~$u.d +$~~ $w(u, v)$

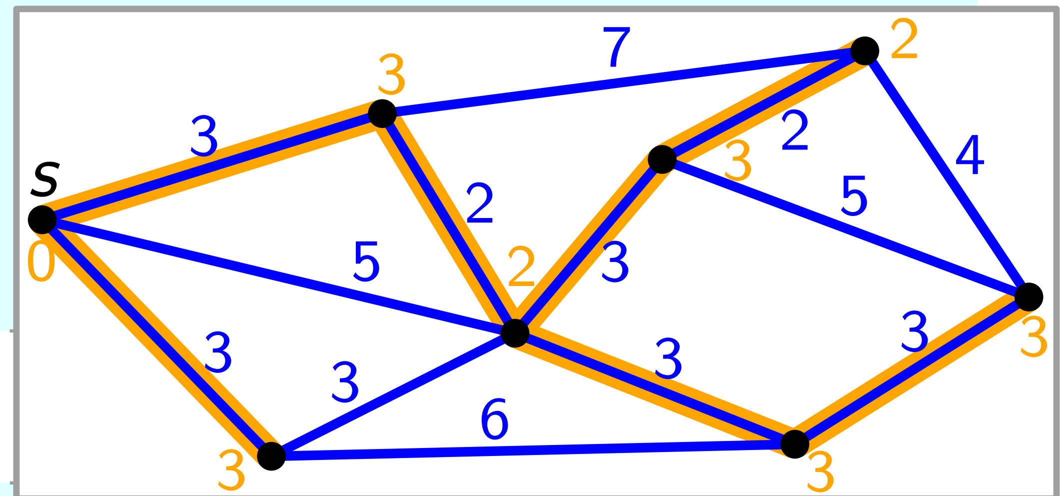
$v.\pi = u$

Q.UpdateKey($v, v.d$)

Jarník-Prim – Pseudocode

JarníkPrimMST

~~Dijkstra~~(WeightedUndirectedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  $Q =$ **new** PriorityQueue(V, d)
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in \text{Adj}[u]$ **do**
      Relax′($u, v; w$)



$v \in Q$ **and** ...

Relax′($u, v; w$)
  if $v.d >$ ~~$u.d +$~~ $w(u, v)$ **then**
    $v.d =$ ~~$u.d +$~~ $w(u, v)$
    $v.\pi = u$
    Q.UpdateKey($v, v.d$)

Jarník-Prim – Pseudocode

JarníkPrimMST

~~Dijkstra~~(WeightedUndirectedGraph $G = (V, E; w)$, Vertex s)

Initialize(G, s)

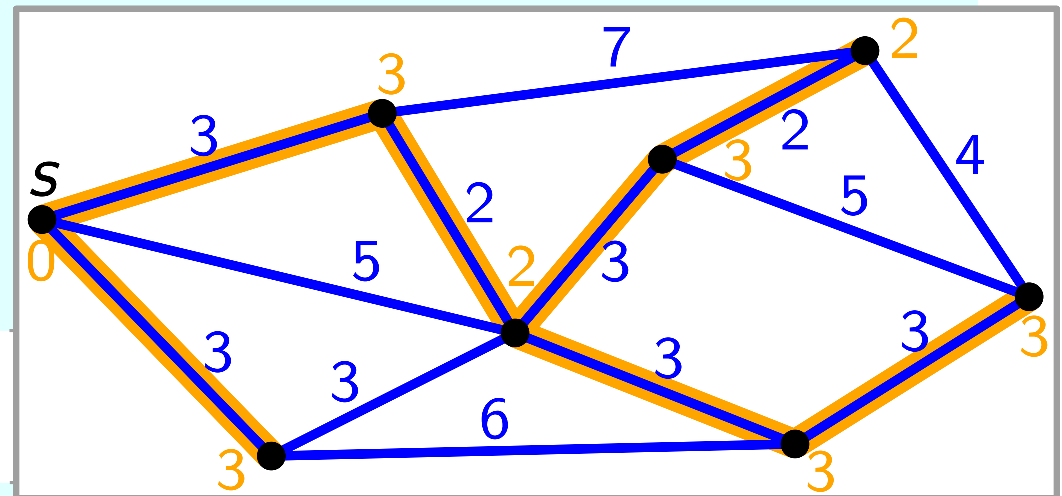
Q = **new** PriorityQueue(V, d)

while not Q.Empty() **do**

$u = Q$.ExtractMin()

foreach $v \in \text{Adj}[u]$ **do**

Relax′($u, v; w$)



$v \in Q$ **and** ...

Relax′($u, v; w$)

if $v.d >$ ~~$u.d +$~~ $w(u, v)$ **then**

$v.d =$ ~~$u.d +$~~ $w(u, v)$

$v.\pi = u$

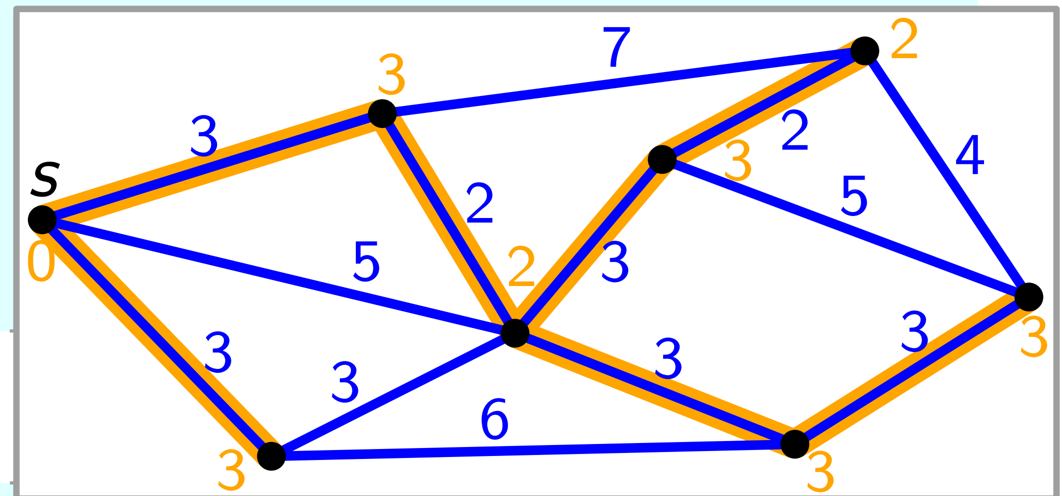
Q.UpdateKey($v, v.d$)

Laufzeit?

Jarník-Prim – Pseudocode

JarníkPrimMST

~~Dijkstra~~(WeightedUndirectedGraph $G = (V, E; w)$, Vertex s)
  Initialize(G, s)
  $Q =$ **new** PriorityQueue(V, d)
  while not Q.Empty() **do**
    $u = Q$.ExtractMin()
    foreach $v \in \text{Adj}[u]$ **do**
      Relax′($u, v; w$)



$v \in Q$ **and** ...

Relax′($u, v; w$)
  if $v.d >$ ~~$u.d +$~~ $w(u, v)$ **then**
    $v.d =$ ~~$u.d +$~~ $w(u, v)$
    $v.\pi = u$
    Q.UpdateKey($v, v.d$)

Laufzeit?

→ siehe Dijkstra

→ $O(E + V \log V)$

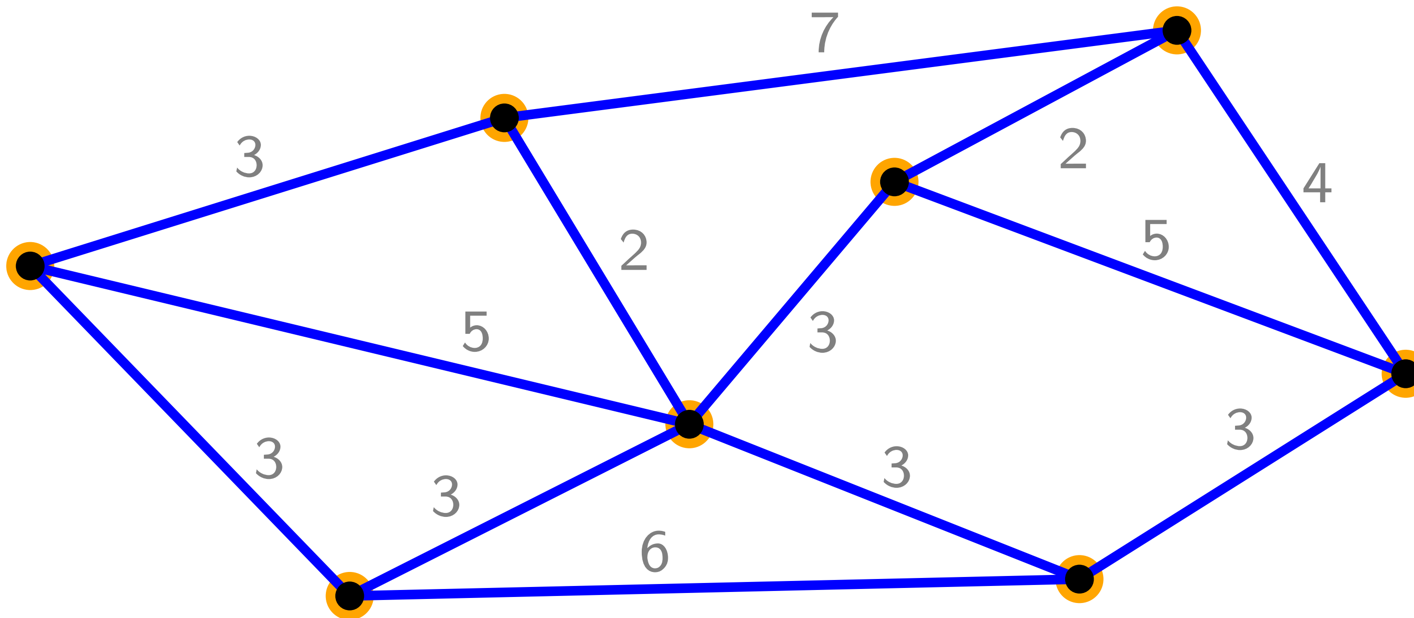
Übersicht

1. Graphdurchlaufstrategien
2. Kürzeste Wege
3. Minimale Spannbäume
    3.1 Prim
    3.2 Kruskal
        Beispiel
        Pseudocode

Kruskal – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten

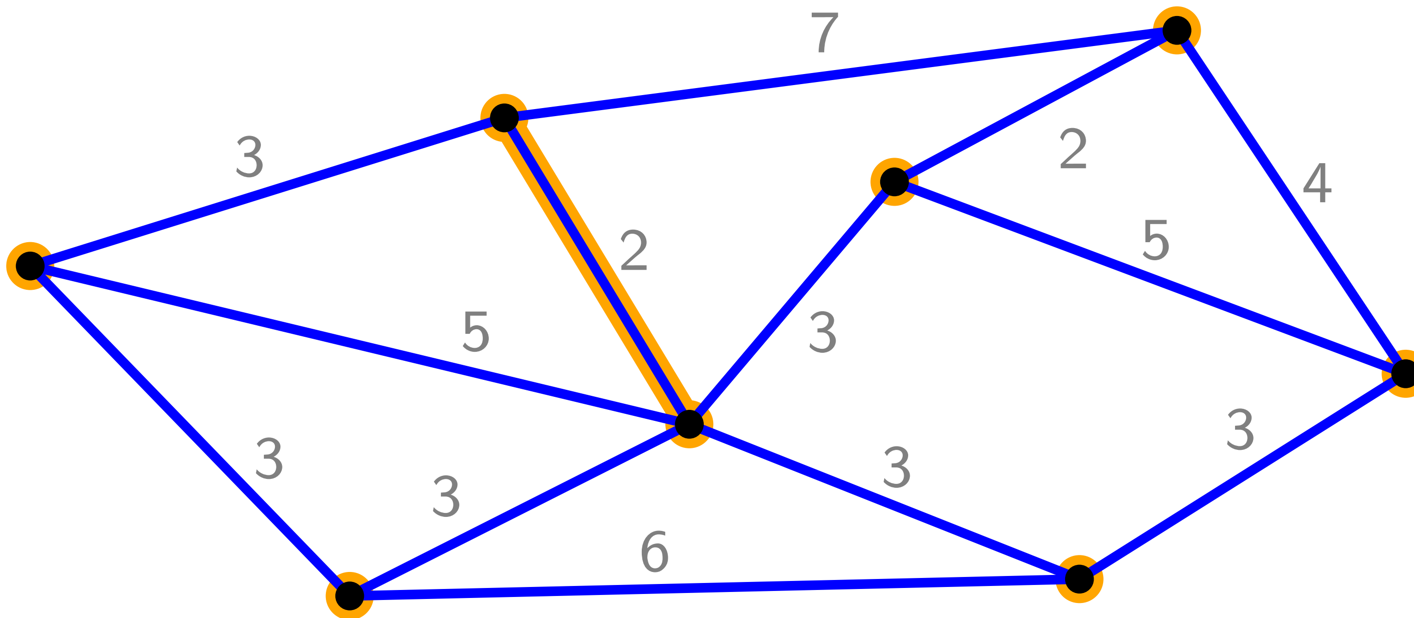
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Kruskal – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten

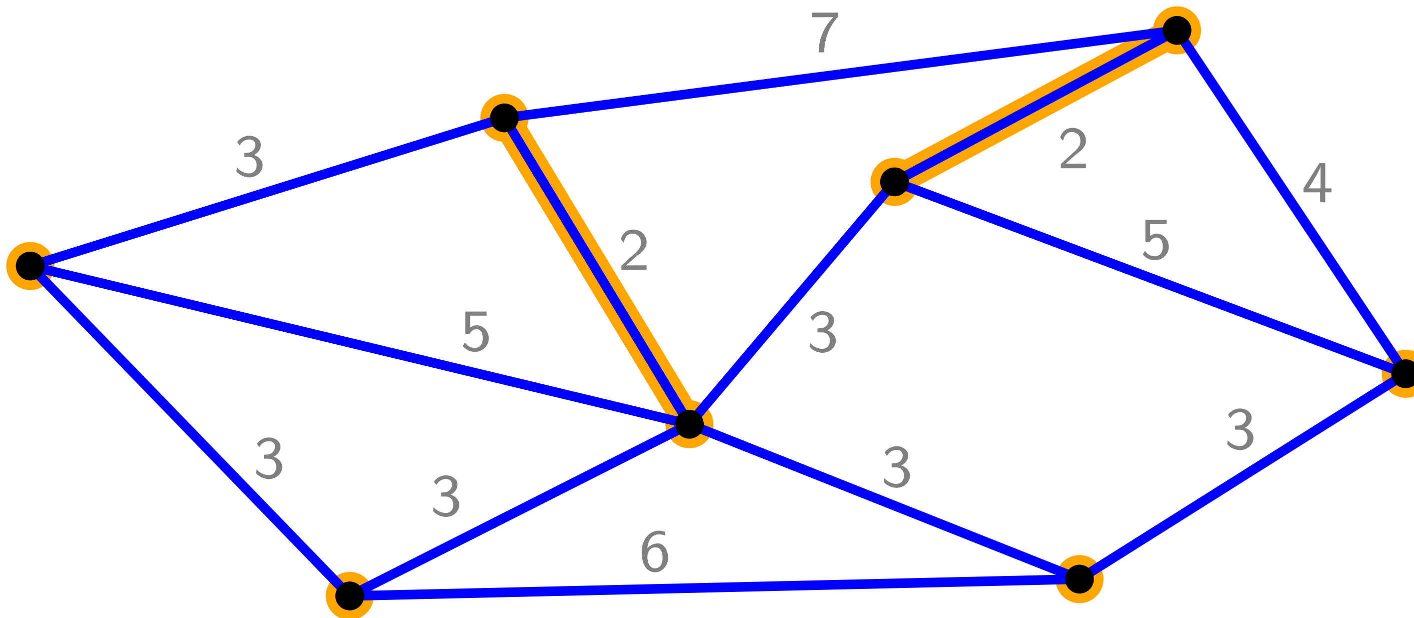
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Kruskal – Beispiel

Eingabe: ungerichteteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten

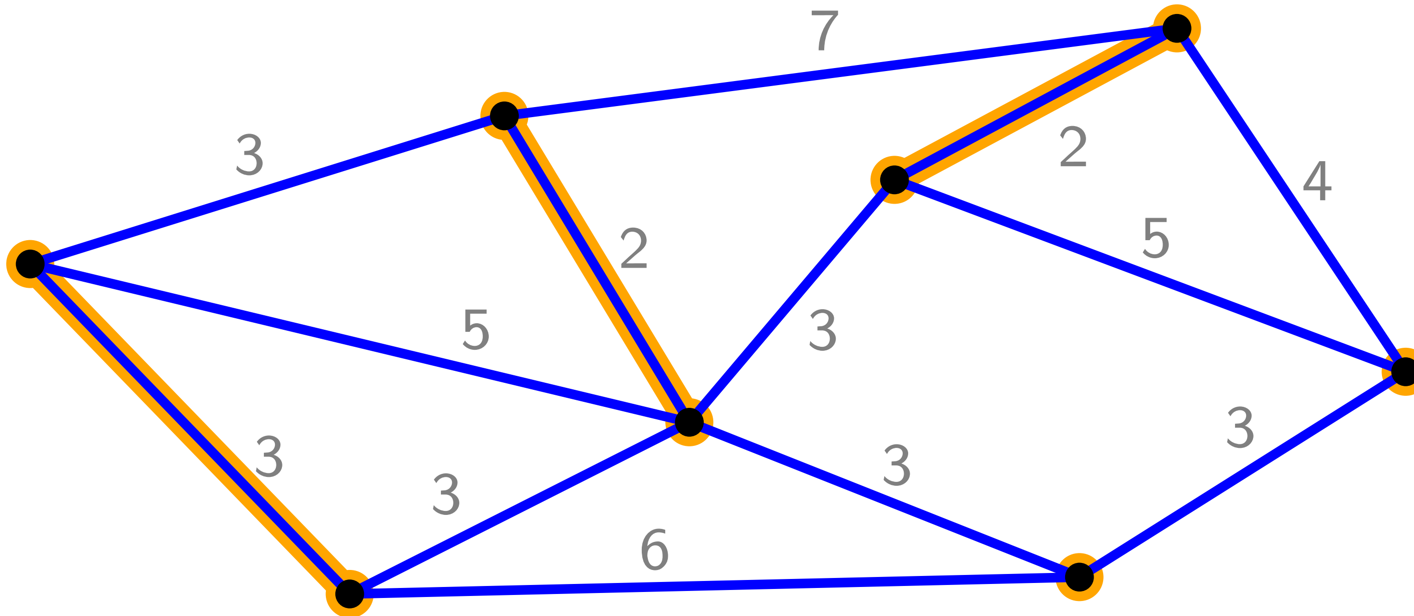
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Kruskal – Beispiel

Eingabe: ungerichteteter, zusammenhängender Graph $G(V,E)$ mit Kantengewichten

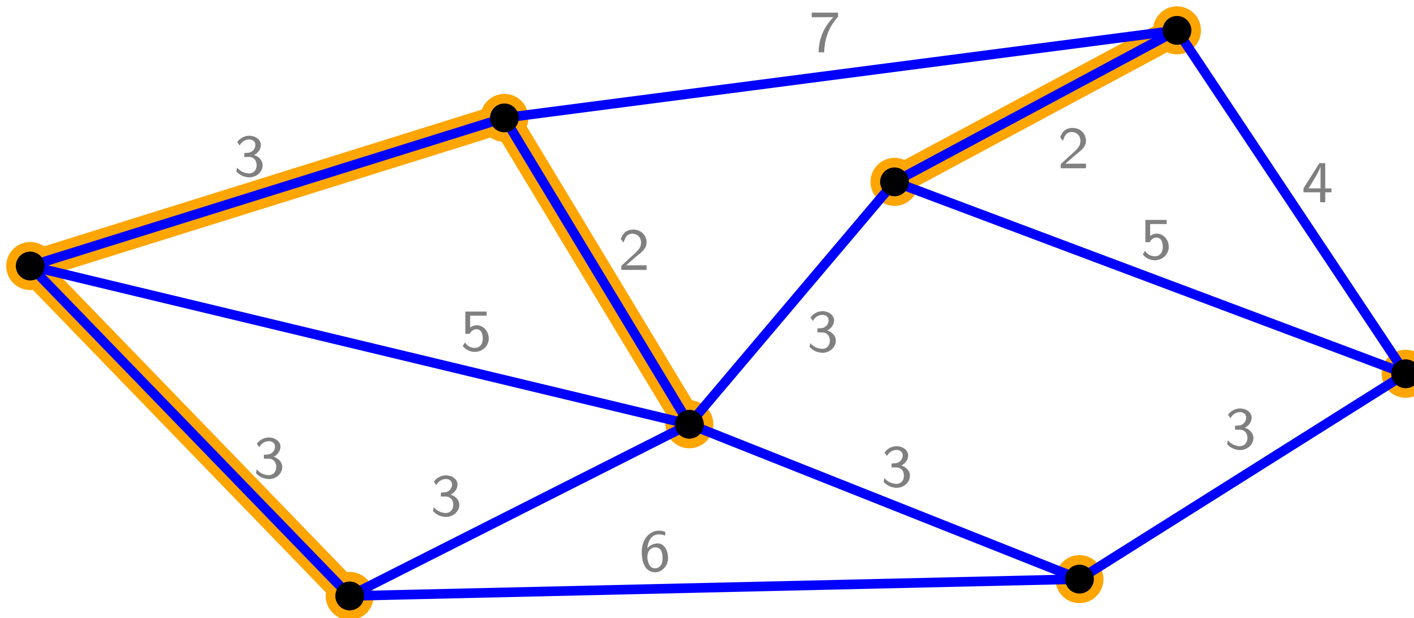
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Kruskal – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten

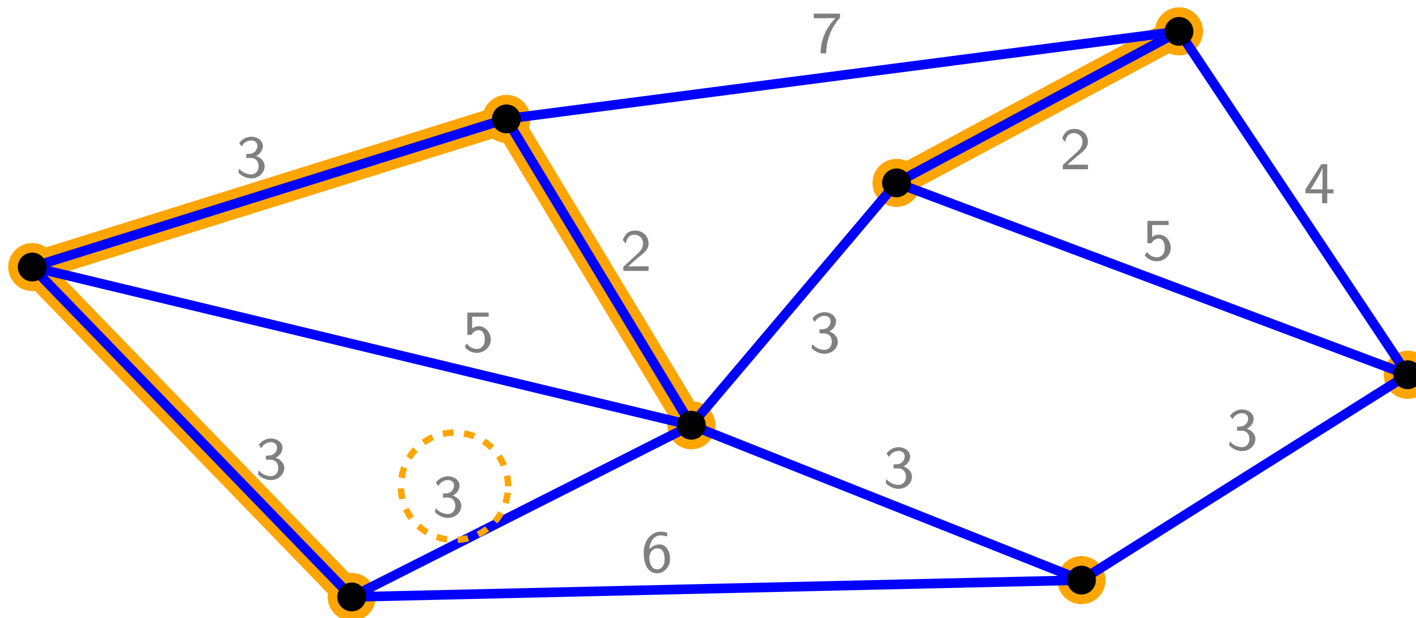
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Kruskal – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten

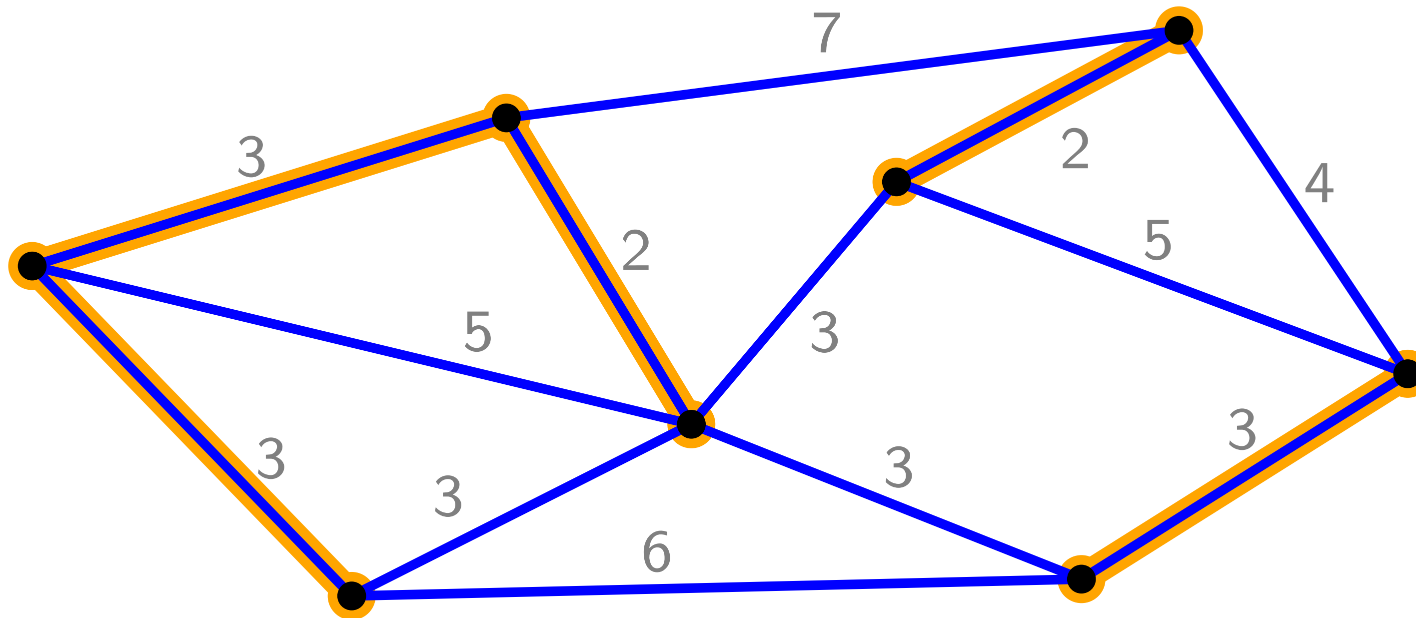
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Kruskal – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten

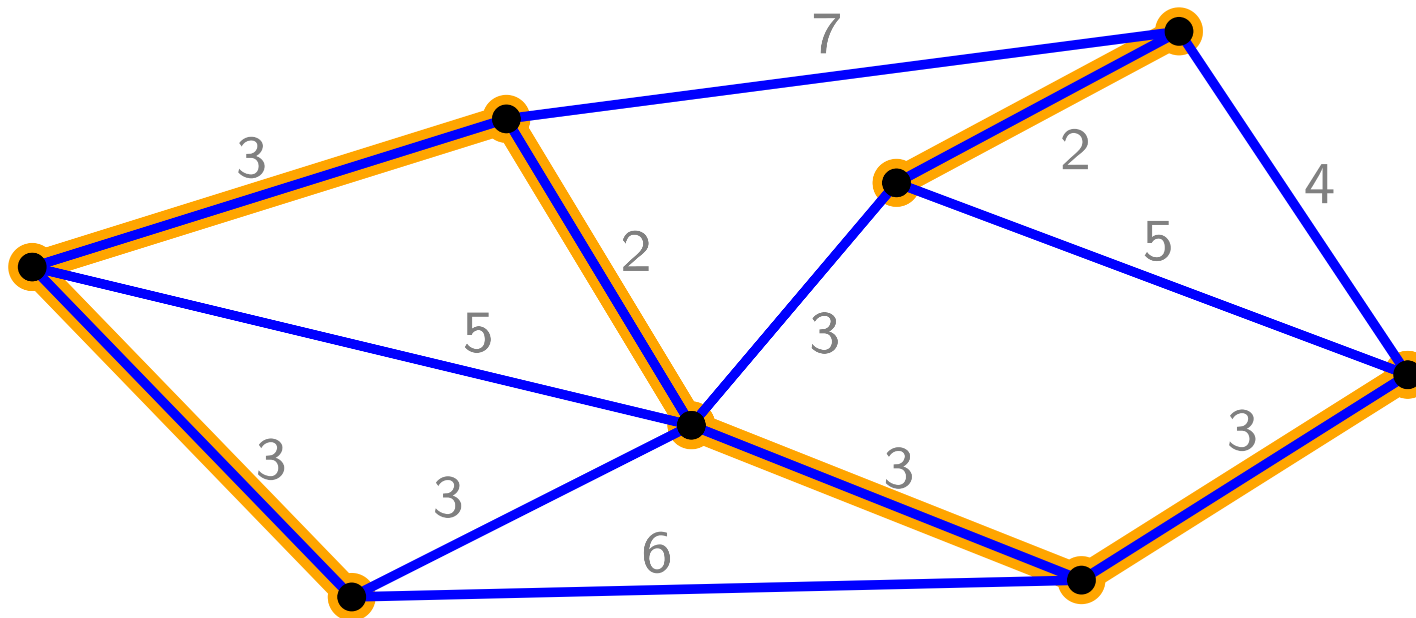
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Kruskal – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten

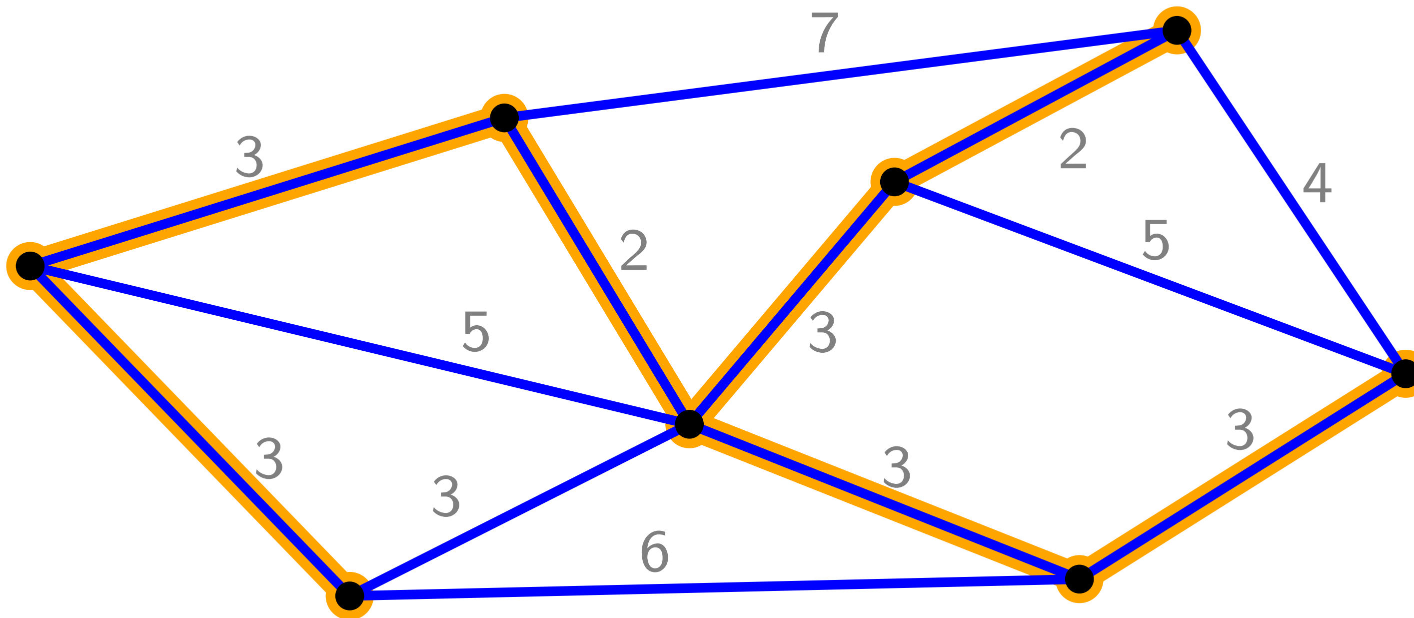
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Kruskal – Beispiel

Eingabe: ungerichteter, zusammenhängender Graph $G(V, E)$ mit Kantengewichten

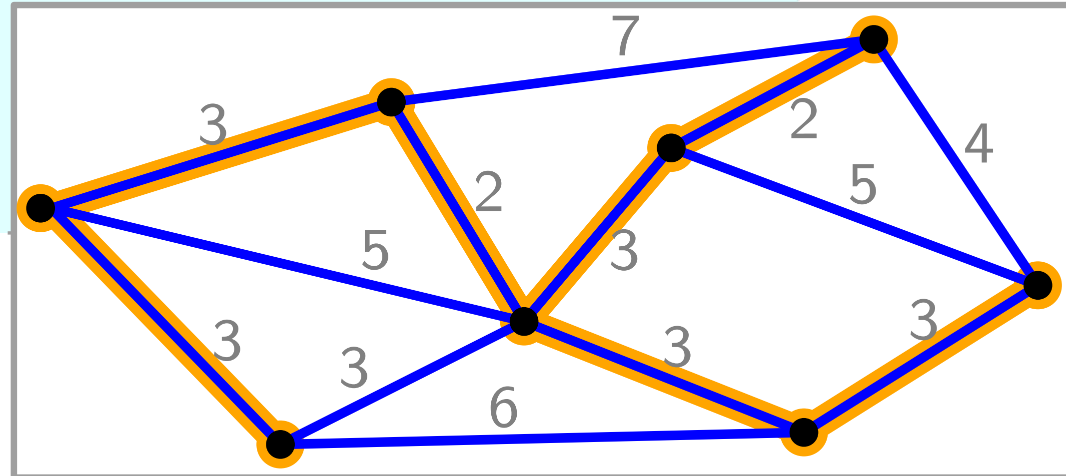
Ausgabe: gewichtsminimaler Baum T, der G aufspannt



Kruskal – Pseudocode

KruskalMST(WeightedUndirectedGraph $G = (V, E; w)$)
  $A = \emptyset$
  foreach $v \in V$ **do**
    MakeSet(v)

  return A



Kruskal – Pseudocode

KruskalMST(WeightedUndirectedGraph $G = (V, E; w)$)

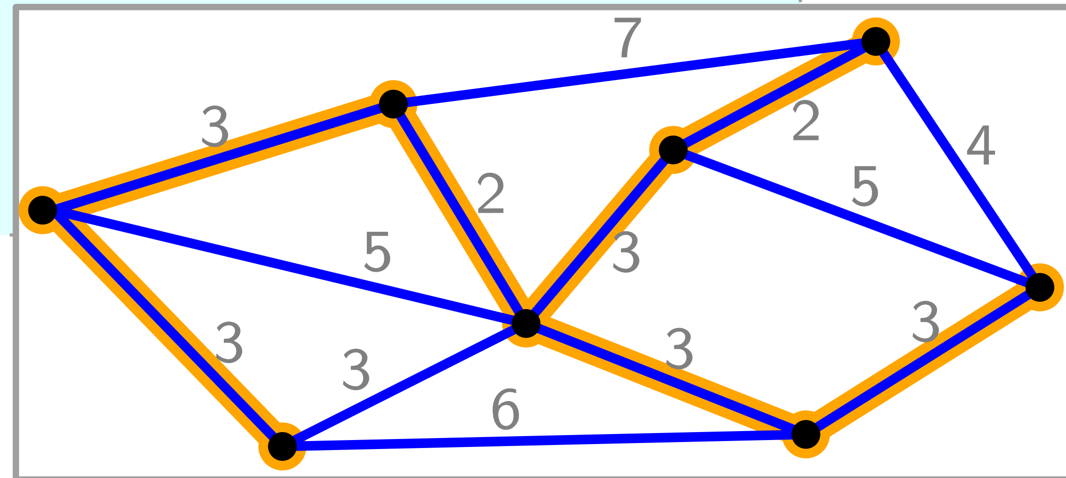
$A = \emptyset$

foreach $v \in V$ **do**

  MakeSet(v)

Sortiere E nicht-absteigend nach Gewicht w

return A



Kruskal – Pseudocode

KruskalMST(WeightedUndirectedGraph $G = (V, E; w)$)

$A = \emptyset$

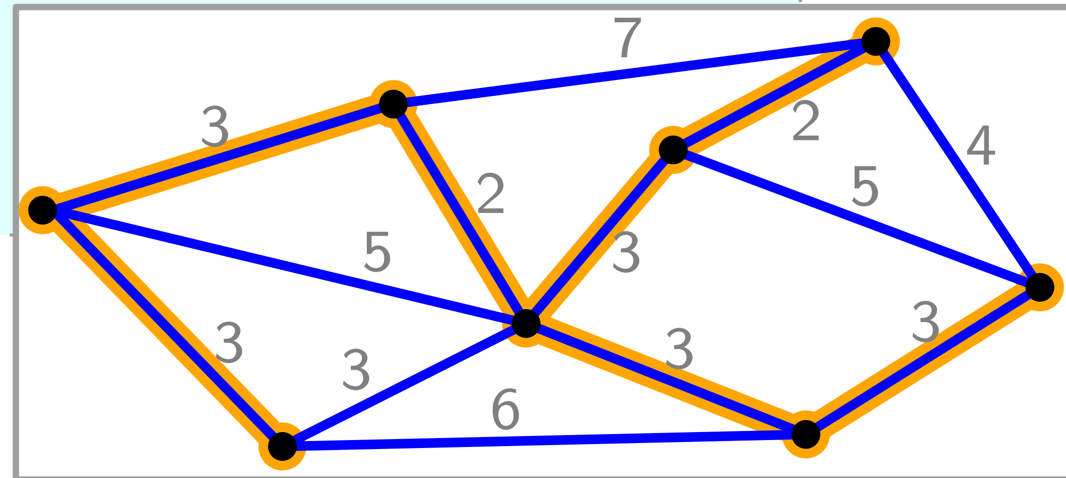
foreach $v \in V$ **do**

MakeSet(v)

Sortiere E nicht-absteigend nach Gewicht w

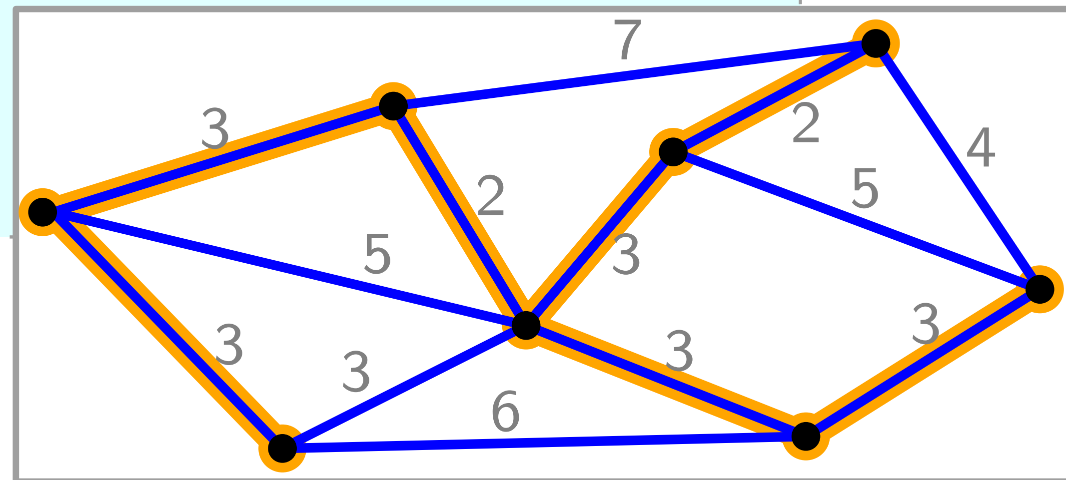
foreach $uv \in E$ **do**

return A



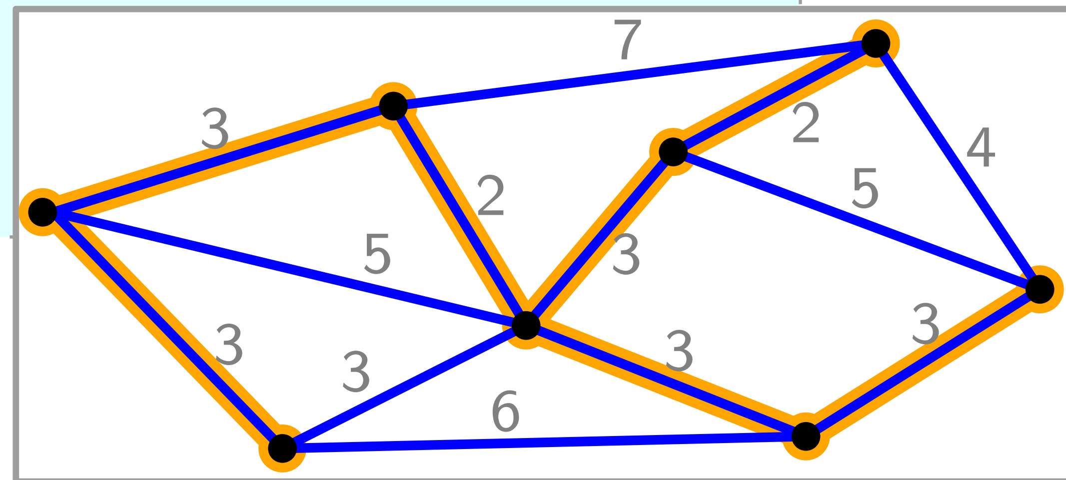
Kruskal – Pseudocode

KruskalMST(WeightedUndirectedGraph $G = (V, E; w)$)
  $A = \emptyset$
  foreach $v \in V$ **do**
    MakeSet(v)
  Sortiere E nicht-absteigend nach Gewicht w
  foreach $uv \in E$ **do**
    if FindSet(u) $\neq$ FindSet(v) **then**
      $A = A \cup \{uv\}$
      Union(u, v)
  return A



Kruskal – Pseudocode

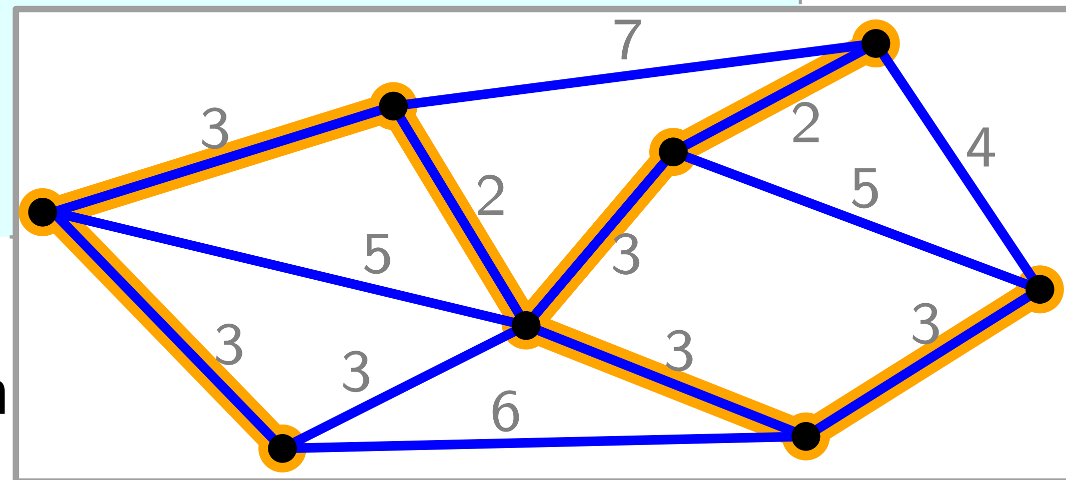
KruskalMST(WeightedUndirectedGraph $G = (V, E; w)$)
  $A = \emptyset$
  foreach $v \in V$ **do**
    MakeSet(v)
  Sortiere E nicht-absteigend nach Gewicht w
  foreach $uv \in E$ **do**
    if FindSet(u) $\neq$ FindSet(v) **then**
      $A = A \cup \{uv\}$
      Union(u, v)
  return A



Laufzeit?

Kruskal – Pseudocode

```
KruskalMST(WeightedUndirectedGraph G = (V, E; w))
  A = ∅
  foreach v ∈ V do
    MakeSet(v)
  Sortiere E nicht-absteigend nach Gewicht w
  foreach uv ∈ E do
    if FindSet(u) ≠ FindSet(v) then
      A = A ∪ {uv}
      Union(u, v)
  return A
```

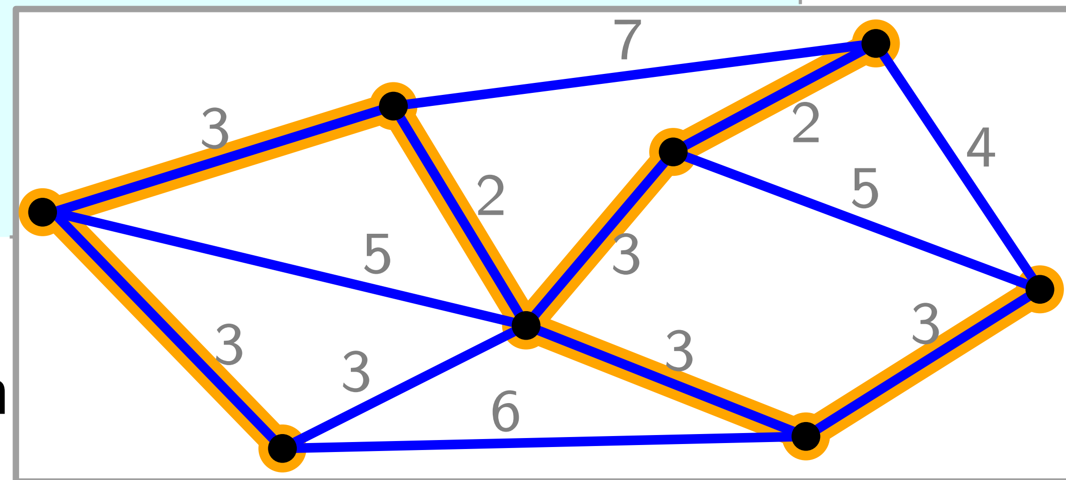


Laufzeit?

$|V| \cdot \text{MakeSet} + (|V| - 1) \cdot \text{Union}$
$+ 2|E| \cdot \text{FindSet} + \text{Sort}(E)$

Kruskal – Pseudocode

```
KruskalMST(WeightedUndirectedGraph G = (V, E; w))
  A = ∅
  foreach v ∈ V do
    MakeSet(v)
  Sortiere E nicht-absteigend nach Gewicht w
  foreach uv ∈ E do
    if FindSet(u) ≠ FindSet(v) then
      A = A ∪ {uv}
      Union(u, v)
  return A
```

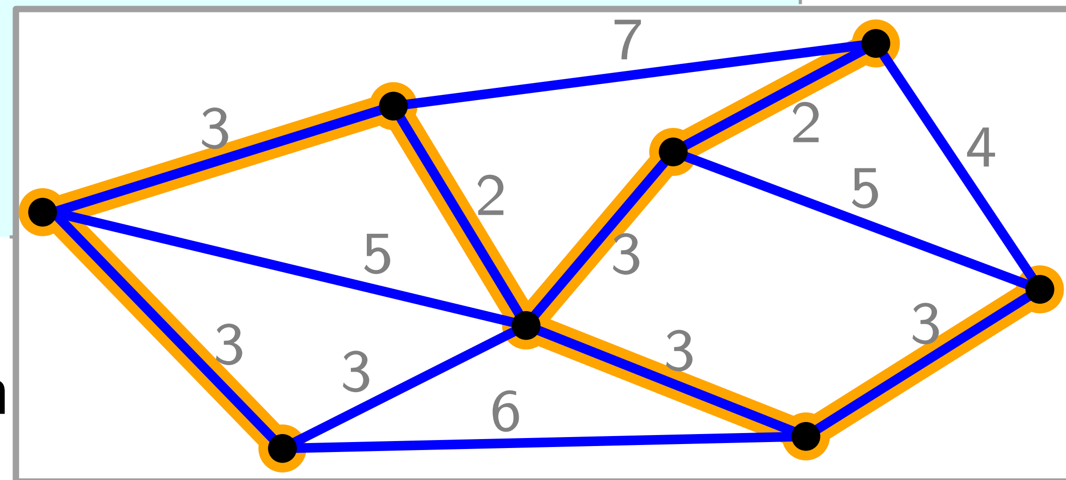


Laufzeit?

$|V| \cdot \text{MakeSet} + (|V| - 1) \cdot \text{Union}$
$+ 2|E| \cdot \text{FindSet} + \text{Sort}(E)$
$= O(E \log V)$

Kruskal – Pseudocode

KruskalMST(WeightedUndirectedGraph $G = (V, E; w)$)
  $A = \emptyset$
  foreach $v \in V$ **do**
    MakeSet(v)
  Sortiere E nicht-absteigend nach Gewicht w
  foreach $uv \in E$ **do**
    if FindSet(u) $\neq$ FindSet(v) **then**
      $A = A \cup \{uv\}$
      Union(u, v)
  return A



Laufzeit?

$|V| \cdot$MakeSet $+ (|V| - 1) \cdot$Union
$+ 2|E| \cdot$FindSet $+$ Sort(E)
$= O(E \log V)$ ← Warum? Vergleiche ADS-Skript!