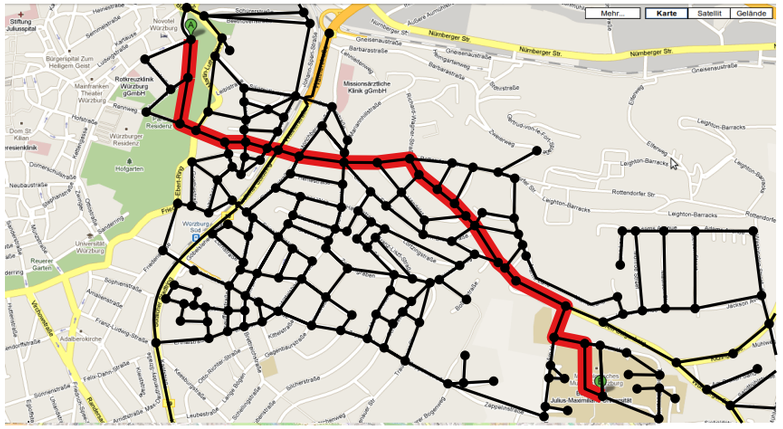
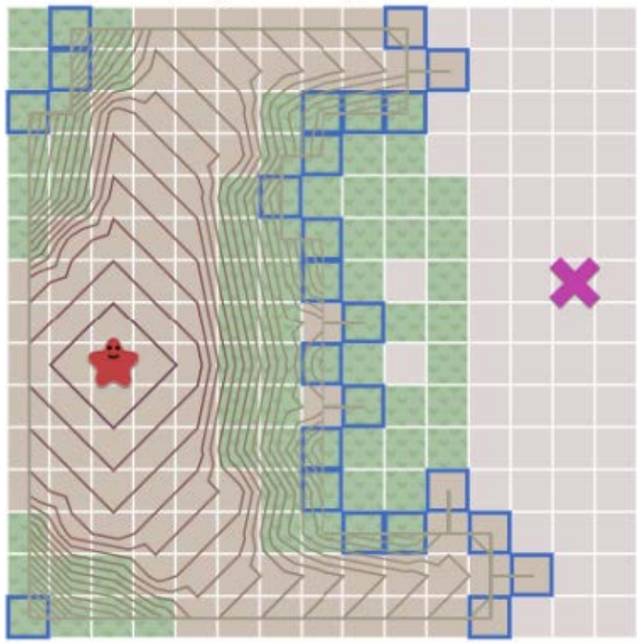
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Algorithmen und Datenstrukturen



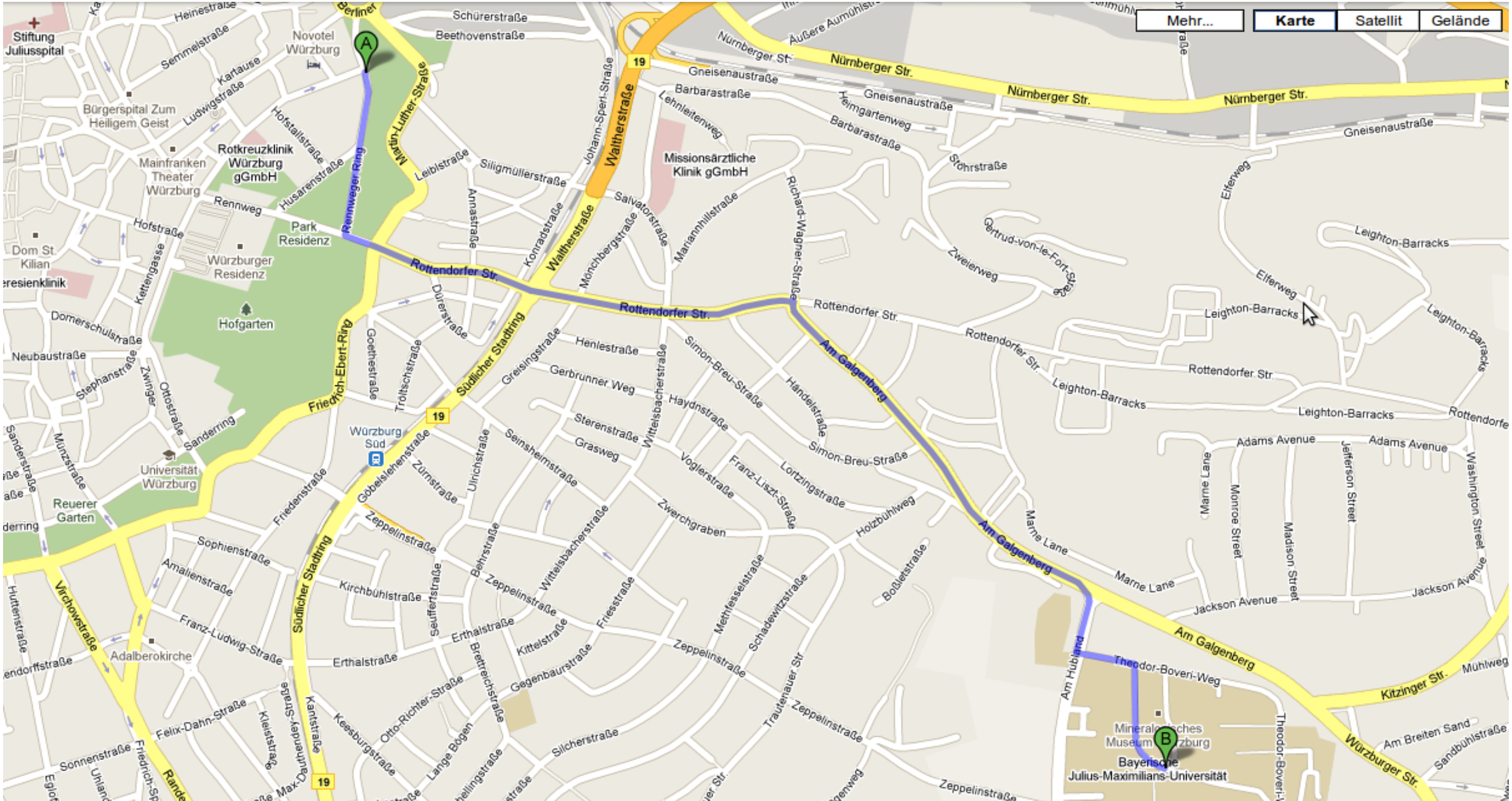
Vorlesung 19:
Kürzeste Wege und Dijkstras Algorithmus



Alexander Wolff

Wintersemester 2024

Routenplanung



Modellierung des Problems Routenplanung

Straßenkreuzung ➡

Straßenabschnitt ➡

Einbahnstraßenabschnitt ➡

Fahrtzeit für Abschnitt e ➡

Straßennetz ➡

Start ➡

Ziel ➡

Start-Ziel-Route ➡

Modellierung des Problems Routenplanung

Straßenkreuzung	➡	Knoten
Straßenabschnitt	➡	
Einbahnstraßenabschnitt	➡	
Fahrtzeit für Abschnitt e	➡	
Straßennetz	➡	
Start	➡	
Ziel	➡	
Start-Ziel-Route	➡	

Modellierung des Problems Routenplanung

Straßenkreuzung	➡	Knoten
Straßenabschnitt	➡	zwei entgegengerichtete Kanten
Einbahnstraßenabschnitt	➡	
Fahrtzeit für Abschnitt e	➡	
Straßennetz	➡	
Start	➡	
Ziel	➡	
Start-Ziel-Route	➡	

Modellierung des Problems Routenplanung

Straßenkreuzung	➡	Knoten
Straßenabschnitt	➡	zwei entgegengerichtete Kanten
Einbahnstraßenabschnitt	➡	in Fahrtrichtung gerichtete Kante
Fahrtzeit für Abschnitt e	➡	
Straßennetz	➡	
Start	➡	
Ziel	➡	
Start-Ziel-Route	➡	

Modellierung des Problems Routenplanung

Straßenkreuzung	➡	Knoten
Straßenabschnitt	➡	zwei entgegengerichtete Kanten
Einbahnstraßenabschnitt	➡	in Fahrtrichtung gerichtete Kante
Fahrtzeit für Abschnitt e	➡	Kantengewicht $w(e) \geq 0$
Straßennetz	➡	
Start	➡	
Ziel	➡	
Start-Ziel-Route	➡	

Modellierung des Problems Routenplanung

Straßenkreuzung	➡	Knoten
Straßenabschnitt	➡	zwei entgegengerichtete Kanten
Einbahnstraßenabschnitt	➡	in Fahrtrichtung gerichtete Kante
Fahrtzeit für Abschnitt e	➡	Kantengewicht $w(e) \geq 0$
Straßennetz	➡	gerichteter, gewichteter und zusammenhängender Graph G
Start	➡	
Ziel	➡	
Start-Ziel-Route	➡	

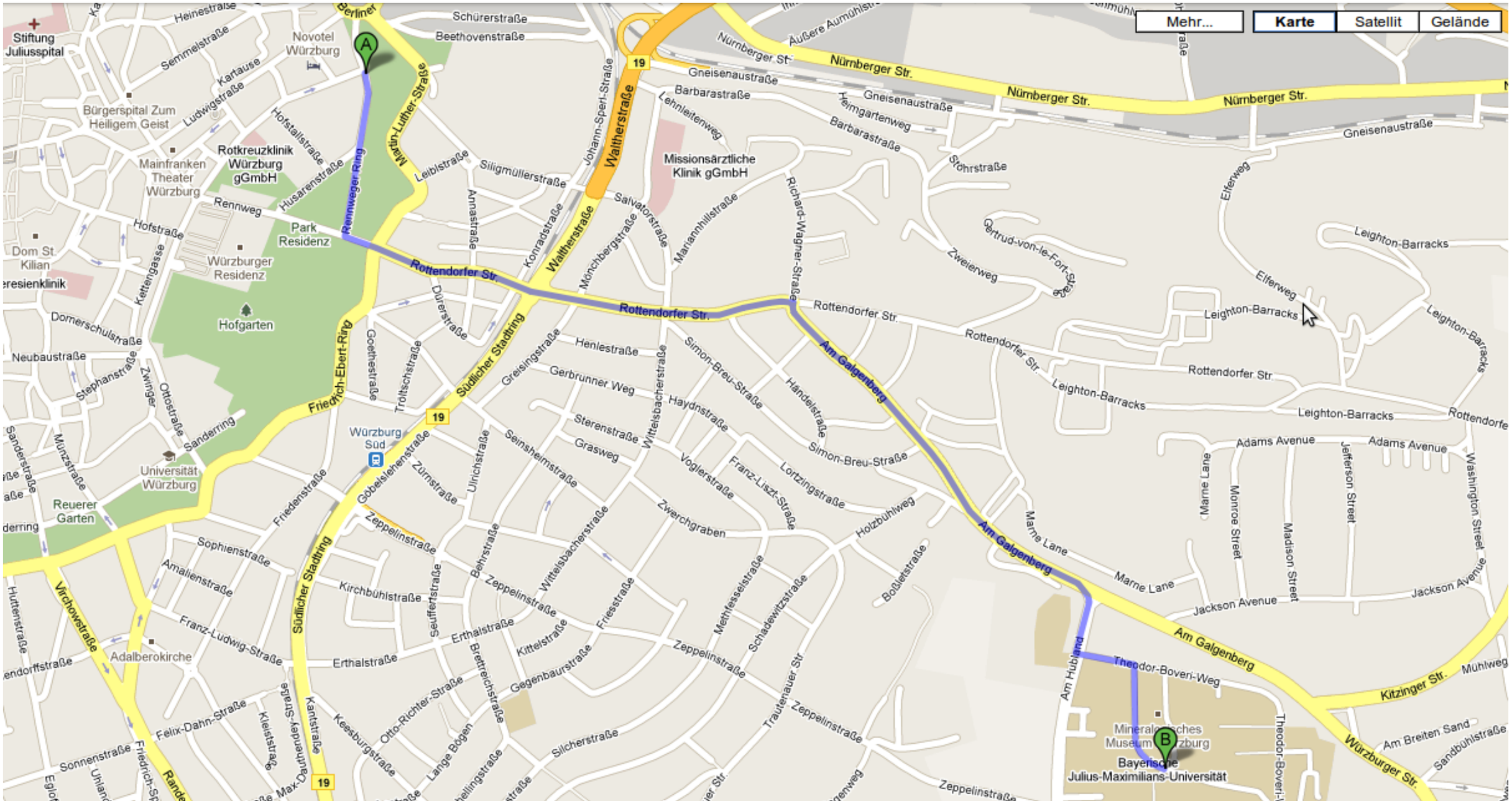
Modellierung des Problems Routenplanung

Straßenkreuzung	➡	Knoten
Straßenabschnitt	➡	zwei entgegengerichtete Kanten
Einbahnstraßenabschnitt	➡	in Fahrtrichtung gerichtete Kante
Fahrtzeit für Abschnitt e	➡	Kantengewicht $w(e) \geq 0$
Straßennetz	➡	gerichteter, gewichteter und zusammenhängender Graph G
Start	➡	Knoten $s \in V(G)$
Ziel	➡	Knoten $t \in V(G)$
Start-Ziel-Route	➡	

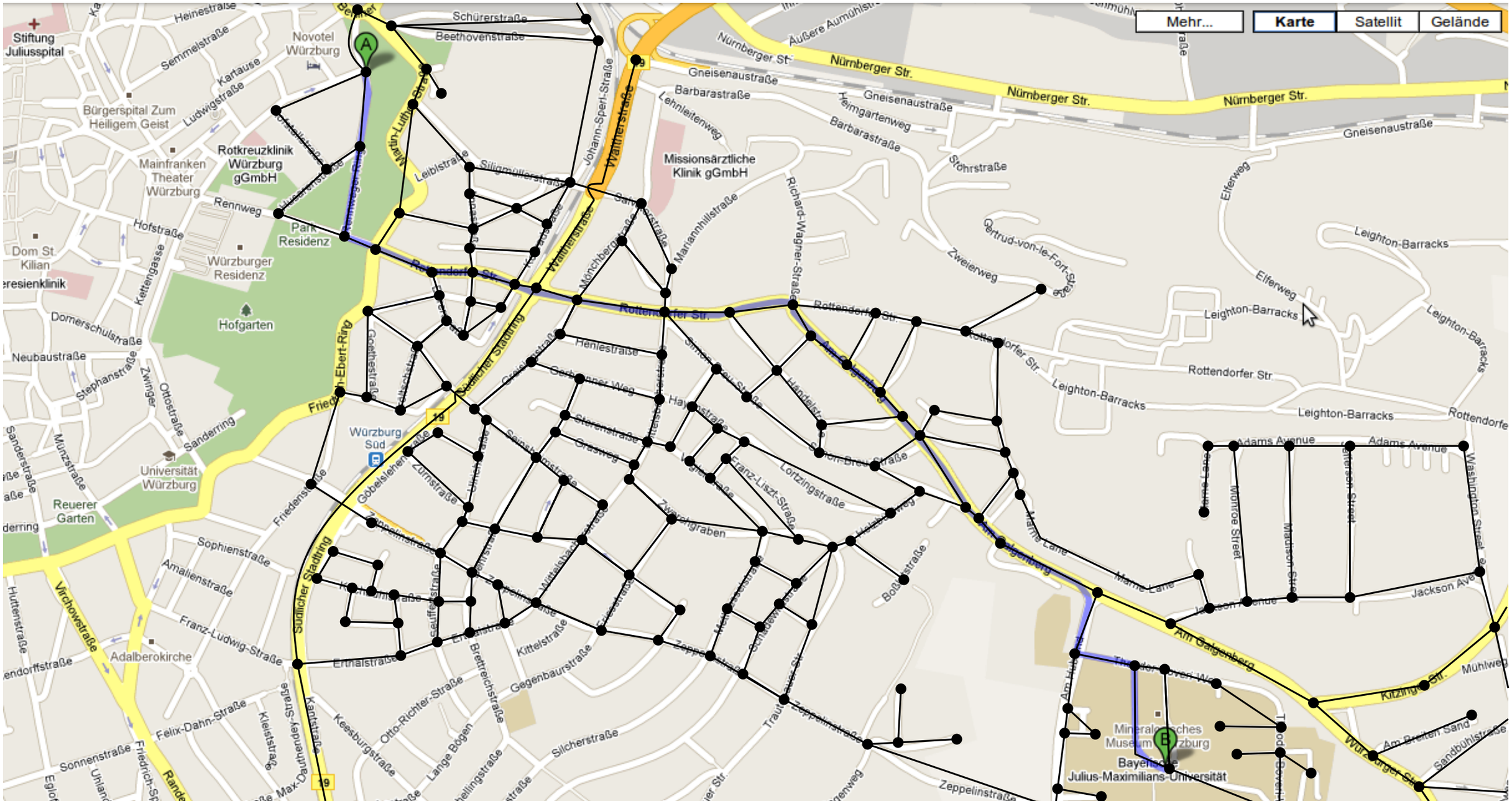
Modellierung des Problems Routenplanung

Straßenkreuzung	➡	Knoten
Straßenabschnitt	➡	zwei entgegengerichtete Kanten
Einbahnstraßenabschnitt	➡	in Fahrtrichtung gerichtete Kante
Fahrtzeit für Abschnitt e	➡	Kantengewicht $w(e) \geq 0$
Straßennetz	➡	gerichteter, gewichteter und zusammenhängender Graph G
Start	➡	Knoten $s \in V(G)$
Ziel	➡	Knoten $t \in V(G)$
Start-Ziel-Route	➡	s-t-Weg: Folge von Kanten $(s, v_1), (v_1, v_2), \ldots, (v_k, t)$ in G

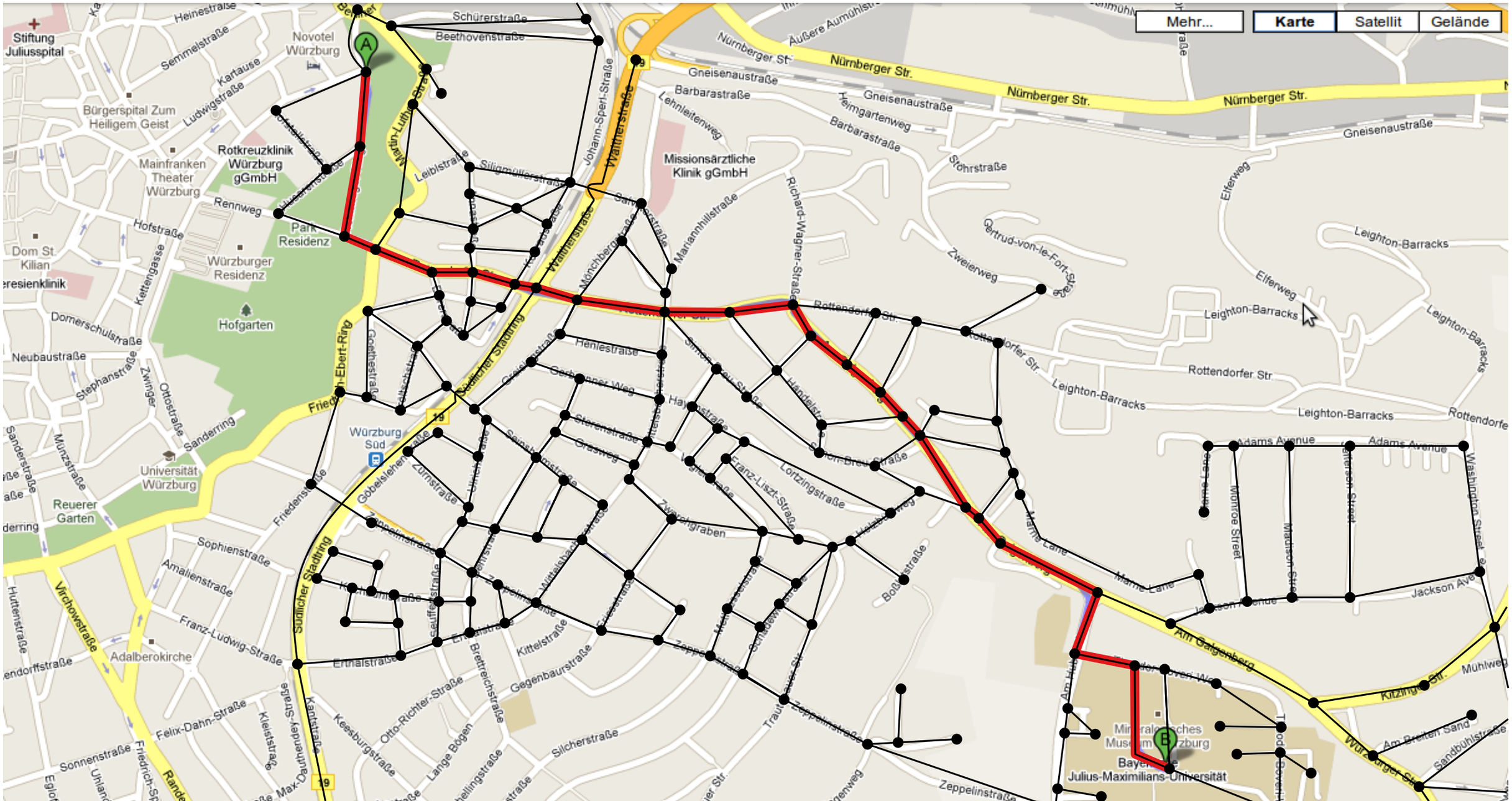
Routenplanung



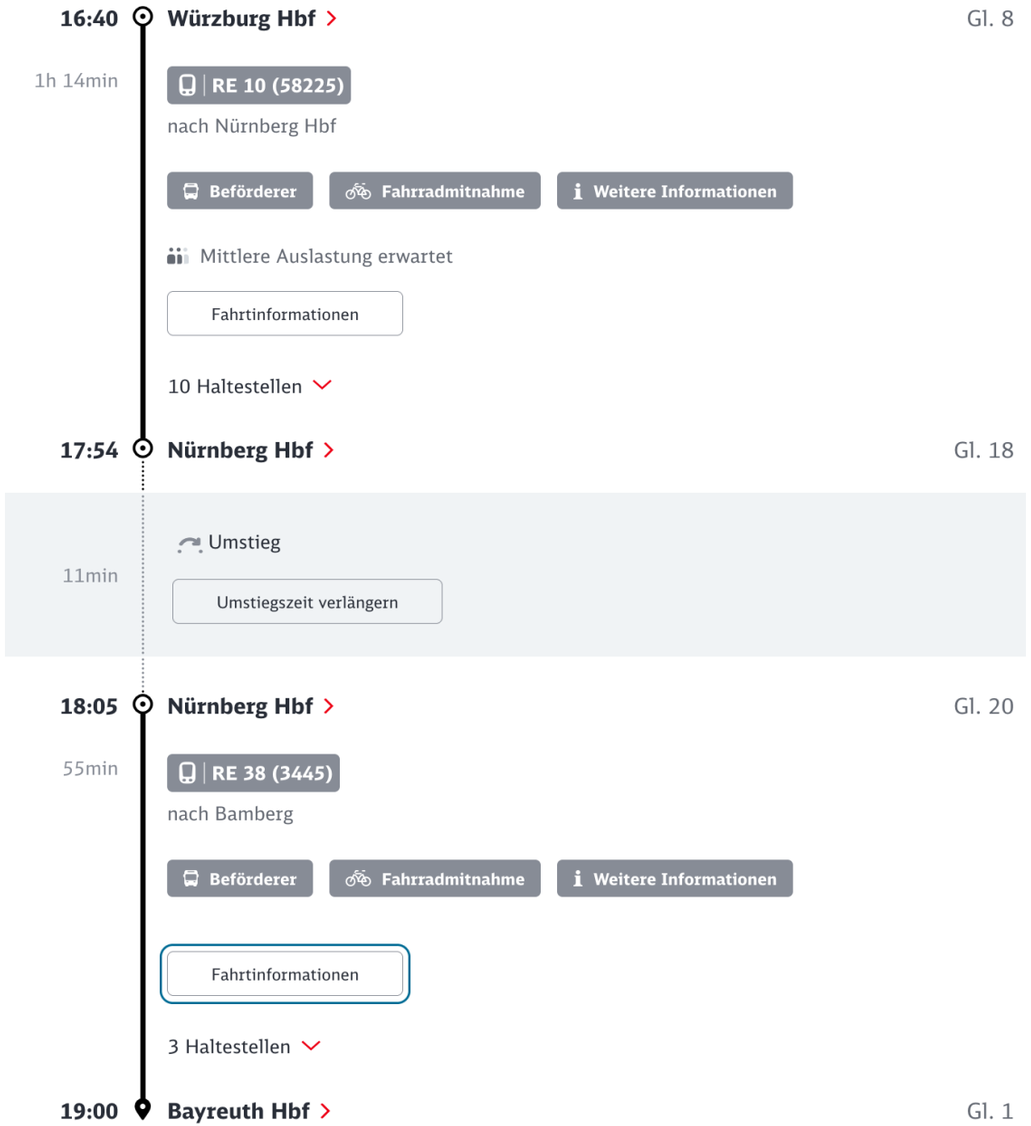
Routenplanung



Routenplanung



Routenplanung mit Zeitkomponente



Was ist das Problem?

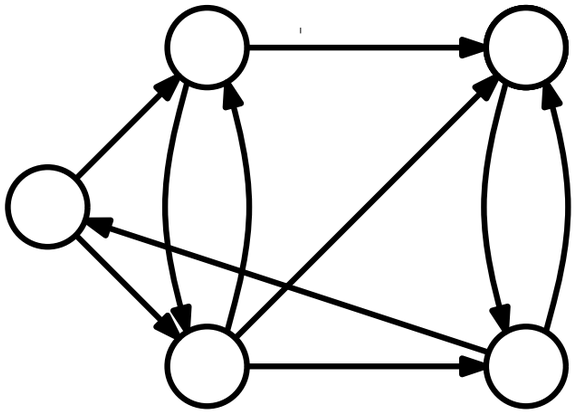
Eingabe:

- gerichteter, zusammenhängender Graph G
  mit nicht-negativen **Kantengewichten** $w\colon E(G) \to \mathbb{Q}_0^+$,

Was ist das Problem?

Eingabe:

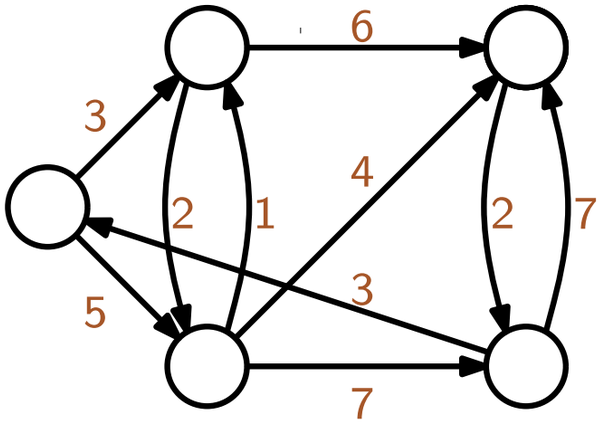
- gerichteter, zusammenhängender Graph G
  mit nicht-negativen **Kantengewichten** $w\colon E(G) \to \mathbb{Q}_0^+$,



Was ist das Problem?

Eingabe:

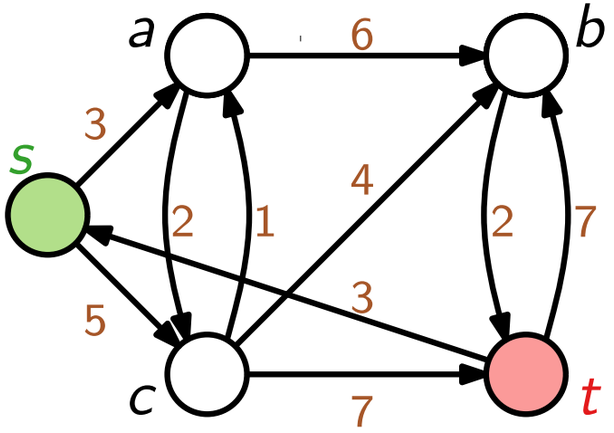
- gerichteter, zusammenhängender Graph G
  mit nicht-negativen **Kantengewichten** $w\colon E(G) \to \mathbb{Q}_0^+$,



Was ist das Problem?

Eingabe:

- gerichteter, zusammenhängender Graph G
  mit nicht-negativen **Kantengewichten** $w\colon E(G) \to \mathbb{Q}_0^+$,
- Knoten s und t



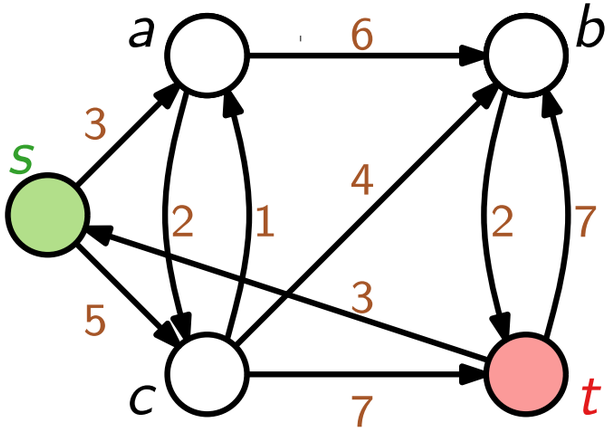
Was ist das Problem?

Eingabe:

- gerichteter, zusammenhängender Graph G
  mit nicht-negativen **Kantengewichten** $w\colon E(G) \to \mathbb{Q}_0^+$,
- Knoten s und t

Ausgabe:

- **kürzester s-t-Weg** W in G, d.h. $\sum_{e \in W} w(e)$ minimal.



Was ist das Problem?

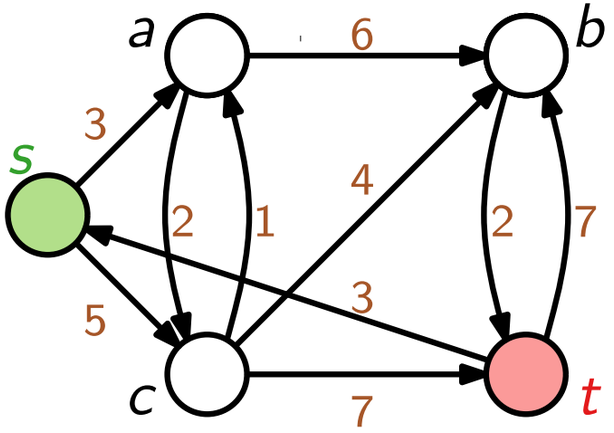
Eingabe:

- gerichteter, zusammenhängender Graph G
  mit nicht-negativen **Kantengewichten** $w\colon E(G) \to \mathbb{Q}_0^+$,
- Knoten s und t

Ausgabe:

- **kürzester s-t-Weg** W in G, d.h. $\sum_{e \in W} w(e)$ minimal.

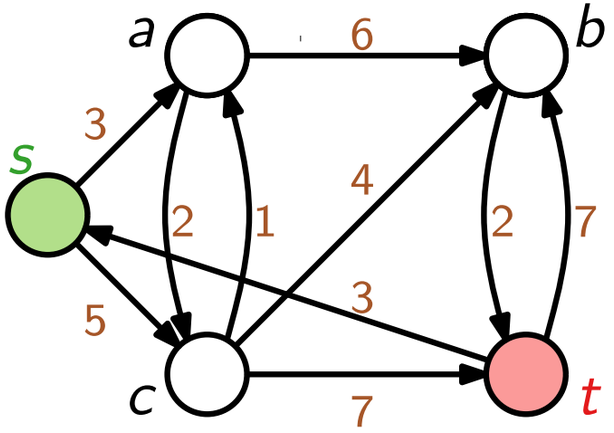
  Darstellung durch Vorgänger-Zeiger π:



Was ist das Problem?

Eingabe:

- gerichteter, zusammenhängender Graph G
  mit nicht-negativen **Kantengewichten** $w\colon E(G) \to \mathbb{Q}_0^+$,
- Knoten s und t



Ausgabe:

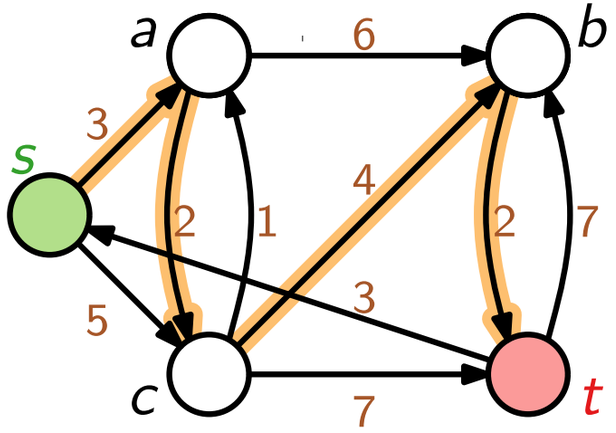
- **kürzester s-t-Weg** W in G, d.h. $\sum_{e \in W} w(e)$ minimal.

  Darstellung durch Vorgänger-Zeiger π:
  für jeden Knoten v sei $\pi(v) \in V(G) \cup \{nil\}$ Vorgänger von v
  auf kürzestem s-v-Weg.

Was ist das Problem?

Eingabe:

- gerichteter, zusammenhängender Graph G
  mit nicht-negativen **Kantengewichten** $w\colon E(G) \to \mathbb{Q}_0^+$,
- Knoten s und t



Ausgabe:

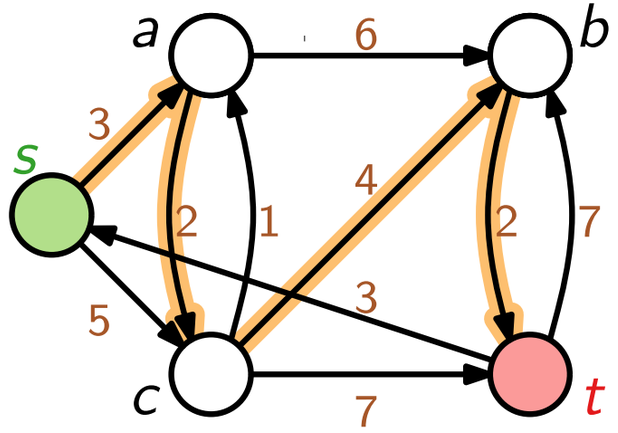
- **kürzester s-t-Weg** W in G, d.h. $\sum_{e \in W} w(e)$ minimal.

  Darstellung durch Vorgänger-Zeiger π:
  für jeden Knoten v sei $\pi(v) \in V(G) \cup \{nil\}$ Vorgänger von v
  auf kürzestem s-v-Weg.

Was ist das Problem?

Eingabe:

- gerichteter, zusammenhängender Graph G
  mit nicht-negativen **Kantengewichten** $w\colon E(G) \to \mathbb{Q}_0^+$,
- Knoten s ~~und t~~



Ausgabe:

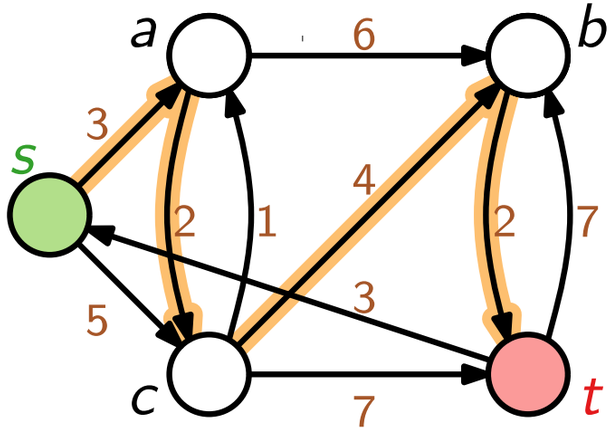
- **kürzester** s-t-**Wege** W_t in G, d.h. $\sum_{e \in W} w(e)$ minimal. (für jedes $t \in V(G)$)

  Darstellung durch Vorgänger-Zeiger π:
  für jeden Knoten v sei $\pi(v) \in V(G) \cup \{nil\}$ Vorgänger von v
  auf kürzestem s-v-Weg.

Was ist das Problem?

Eingabe:

- gerichteter, zusammenhängender Graph G
  mit nicht-negativen **Kantengewichten** $w\colon E(G) \to \mathbb{Q}_0^+$,
- Knoten s ~~und t~~



Ausgabe:

- **kürzester** s-t-**Wege** W_t in G, d.h. $\sum_{e \in W} w(e)$ minimal. (für jedes $t \in V(G)$)

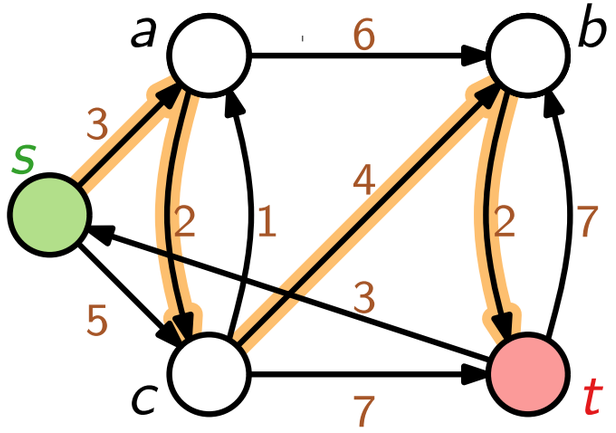
  Darstellung durch Vorgänger-Zeiger π:
  für jeden Knoten v sei $\pi(v) \in V(G) \cup \{nil\}$ Vorgänger von v
  auf kürzestem s-v-Weg.

Nebenbemerkung: **Analoge** Berechnungsverfahren?

Was ist das Problem?

Eingabe:

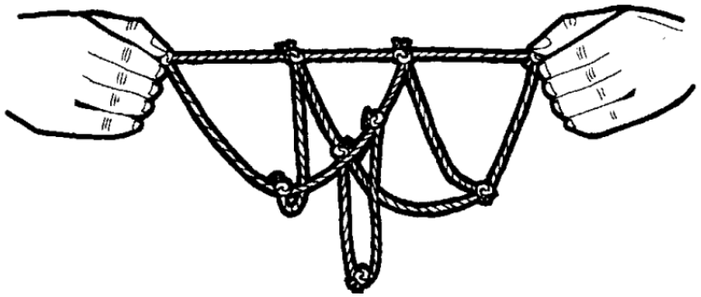
- gerichteter, zusammenhängender Graph G
  mit nicht-negativen **Kantengewichten** $w\colon E(G) \to \mathbb{Q}_0^+$,
- Knoten s ~~und t~~



Ausgabe:

- **kürzester** s-t-**Wege** W_t in G, d.h. $\sum_{e \in W} w(e)$ minimal. (für jedes $t \in V(G)$)

  Darstellung durch Vorgänger-Zeiger π:
  für jeden Knoten v sei $\pi(v) \in V(G) \cup \{nil\}$ Vorgänger v
  auf kürzestem s-v-Weg.

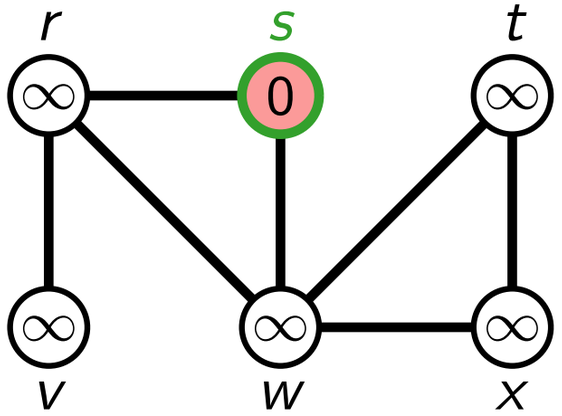


Nebenbemerkung: **Analoge** Berechnungsverfahren?
(nur für ungerichtete Graphen!)

Abbildung aus:
Solving the Shortest Path Problem Using an Analog Network
Linkai Bu & Tzi-Dar Chiueh

Wiederholung Breitensuche

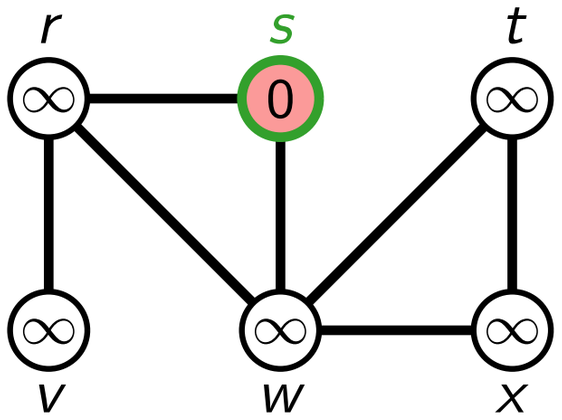
```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```




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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```

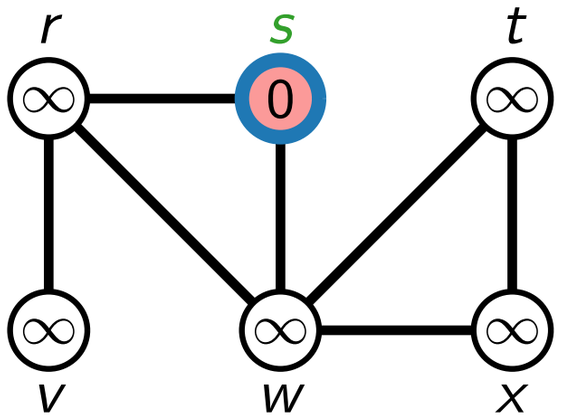


Q:

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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

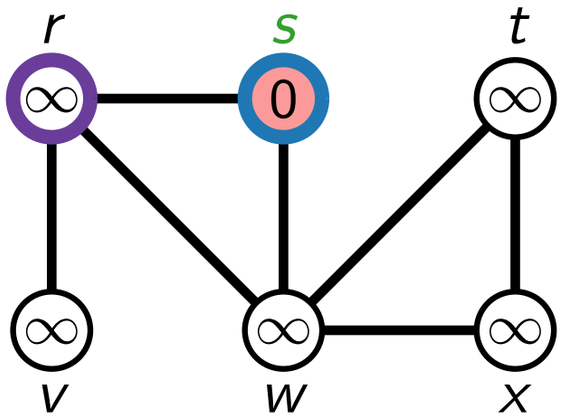


Q:

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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```

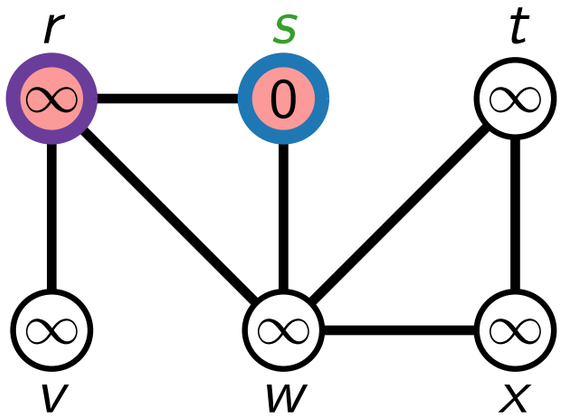


Q:

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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

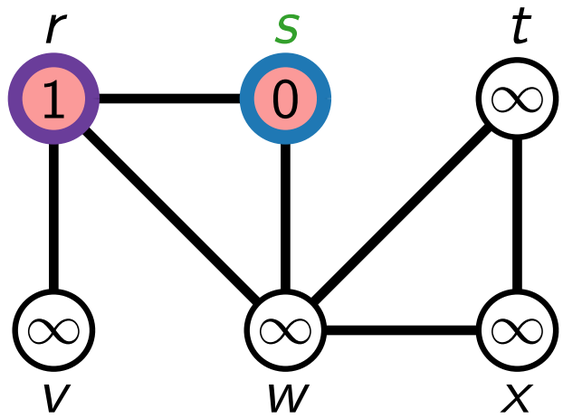


Q:

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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

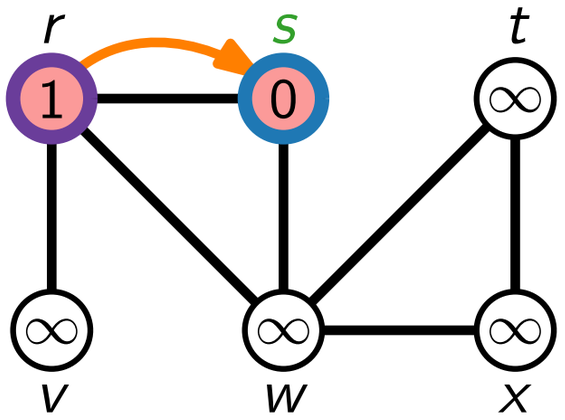


Q:

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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```

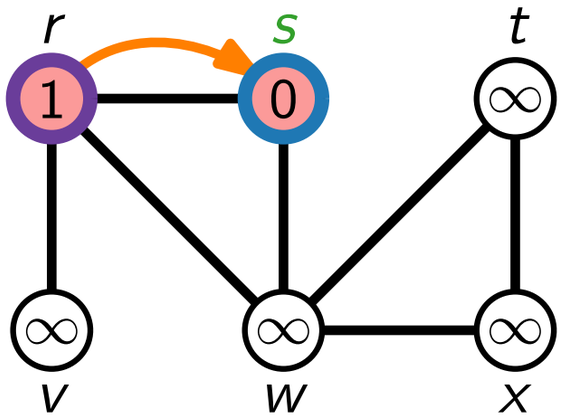


Q:

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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

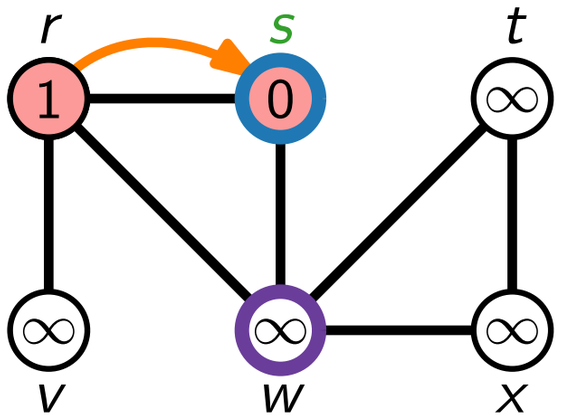


Q:

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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

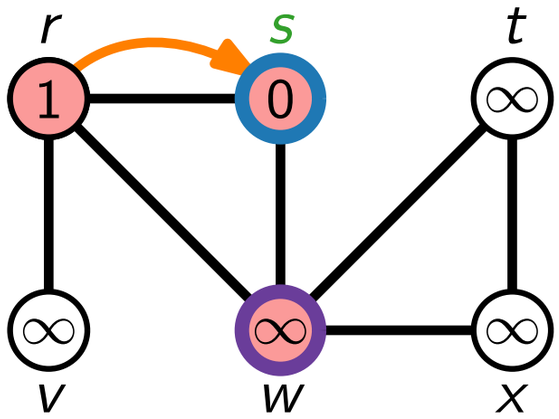


Q:

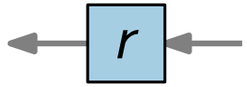
```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```



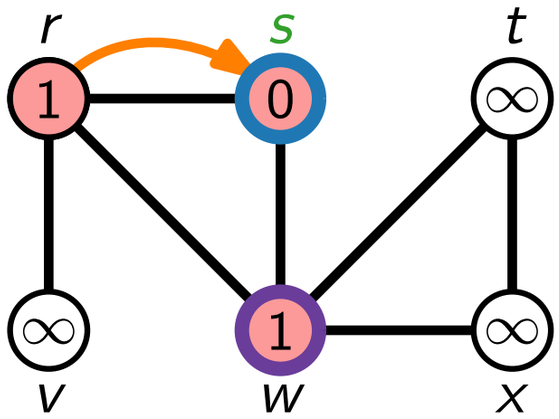
Q:




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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

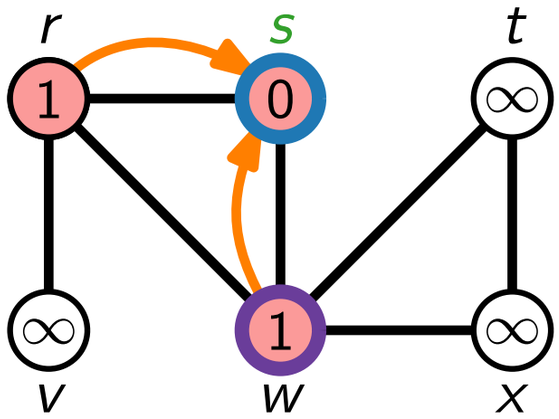


Q:

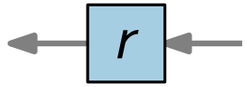
```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```



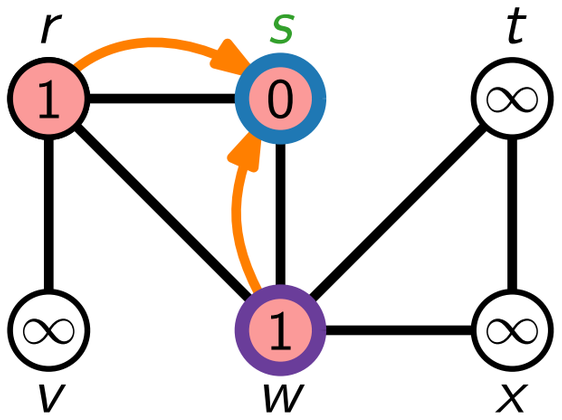
Q:

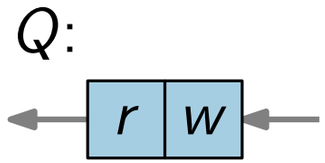



```
INITIALIZE(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```

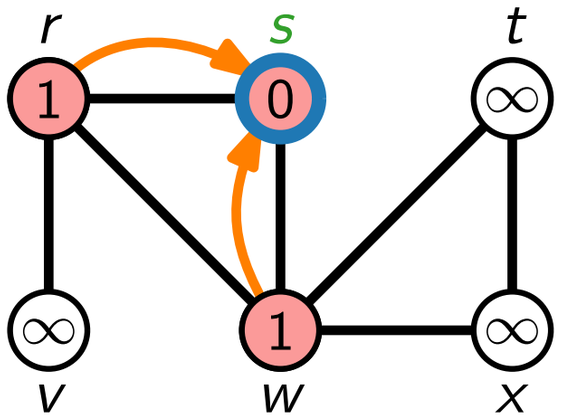


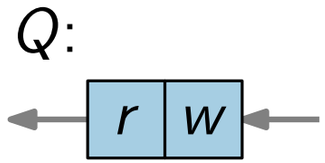



```
INITIALIZE(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

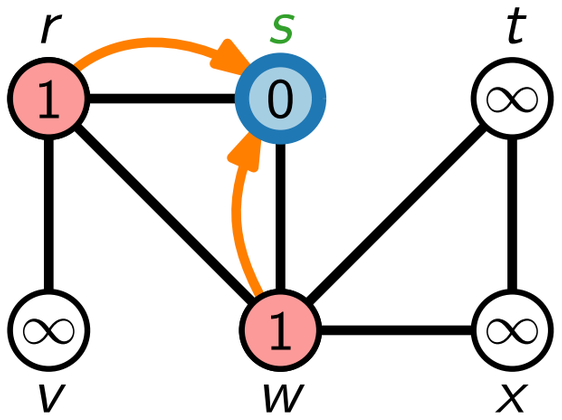


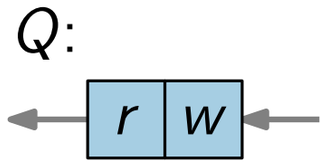



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

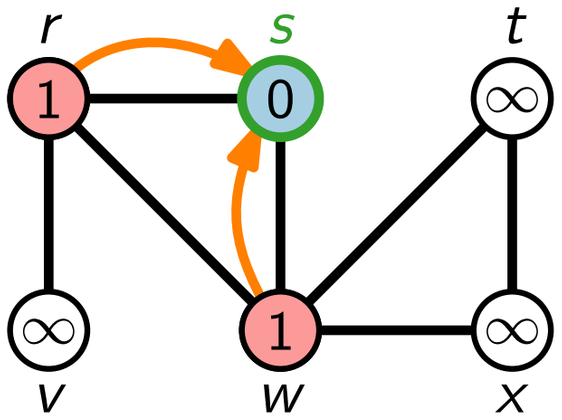


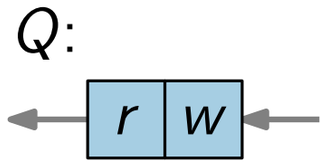



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

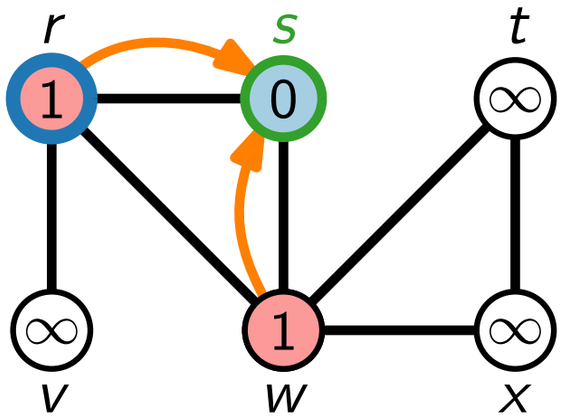





```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

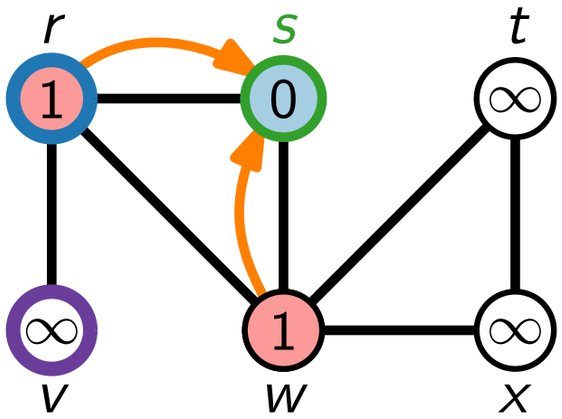


Q:

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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

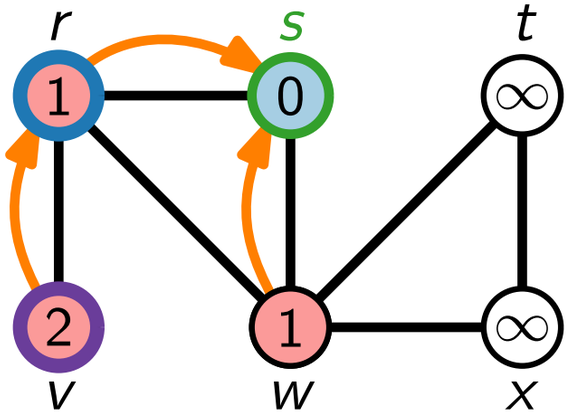


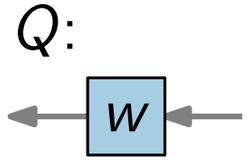
Q:

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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

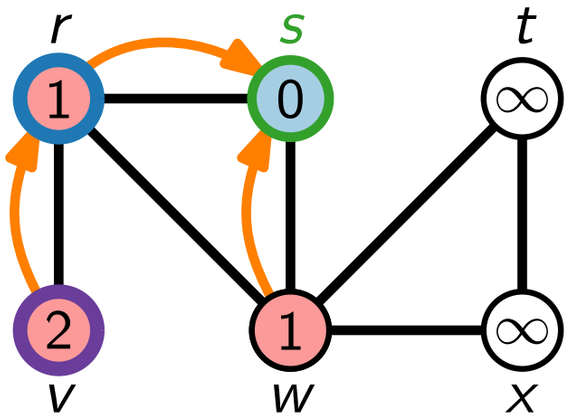


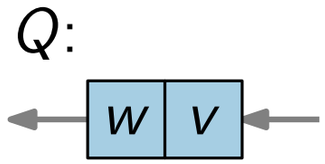



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```

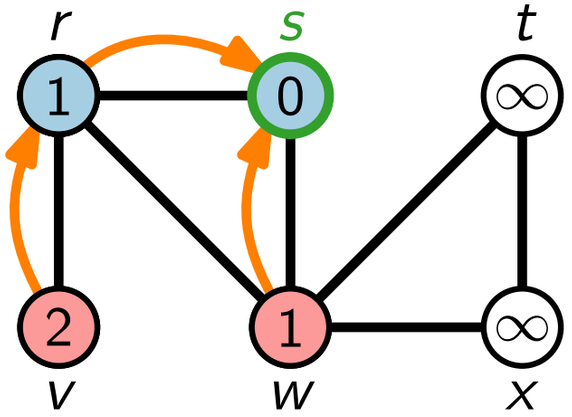


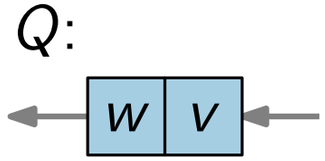



```
INITIALIZE(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```

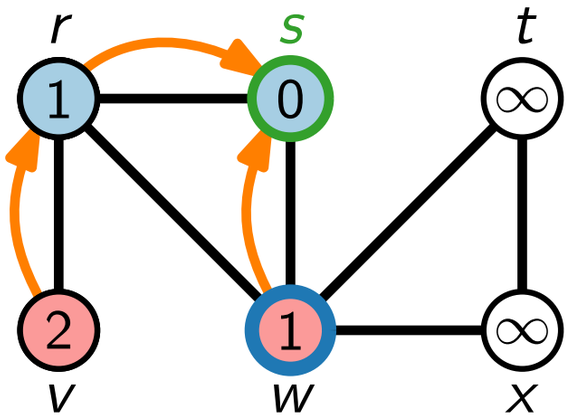




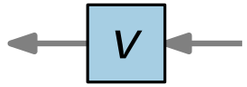

```
INITIALIZE(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```



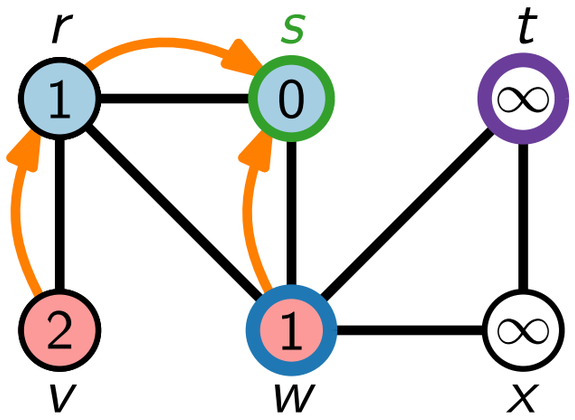
Q:



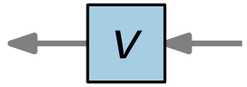

```
INITIALIZE(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```



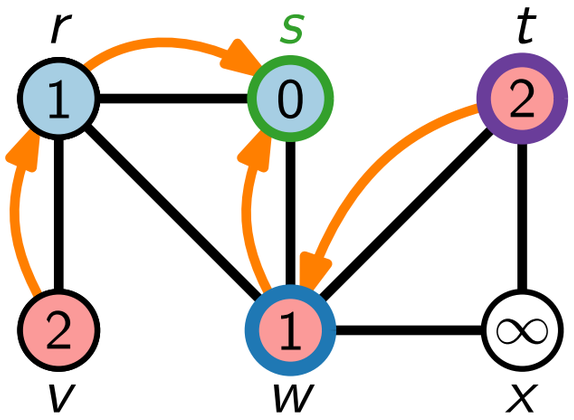
Q:




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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```

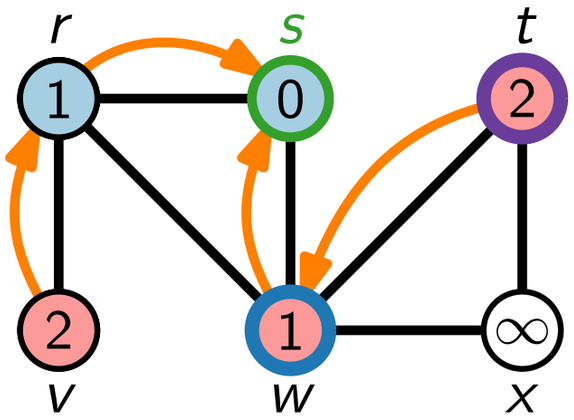


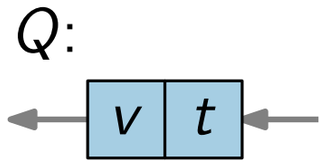
Q:

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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```

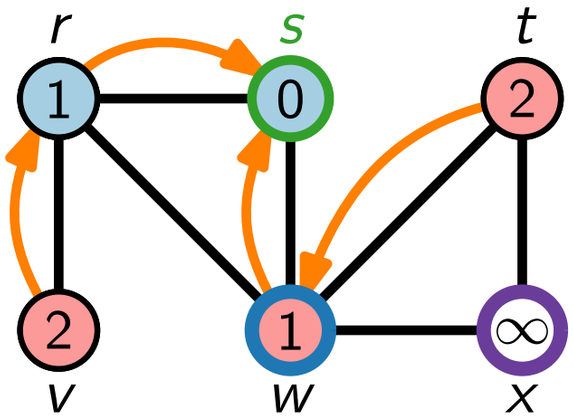


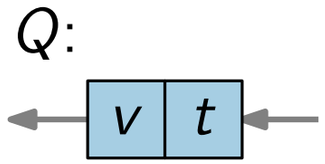



```
INITIALIZE(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```

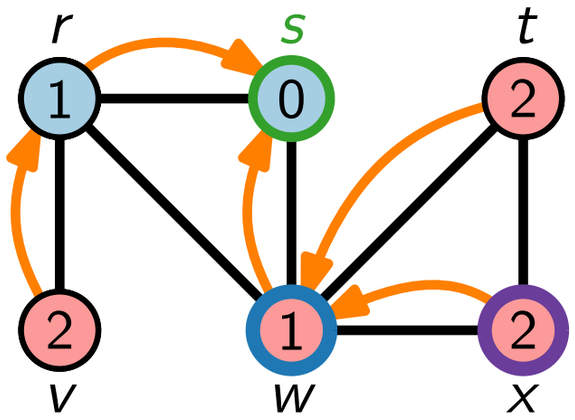




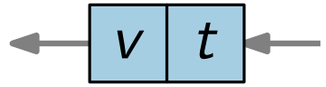

```
INITIALIZE(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```



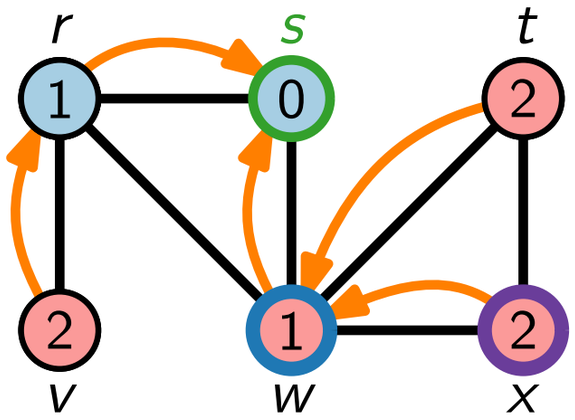
Q:

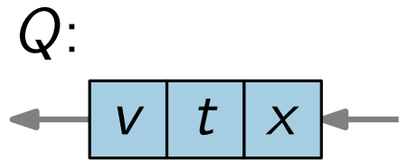



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```

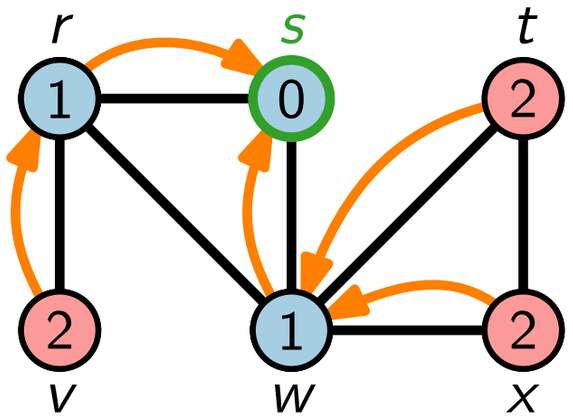


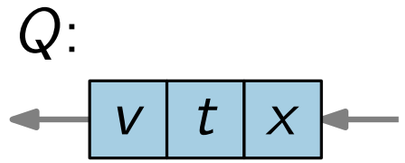



```
INITIALIZE(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```

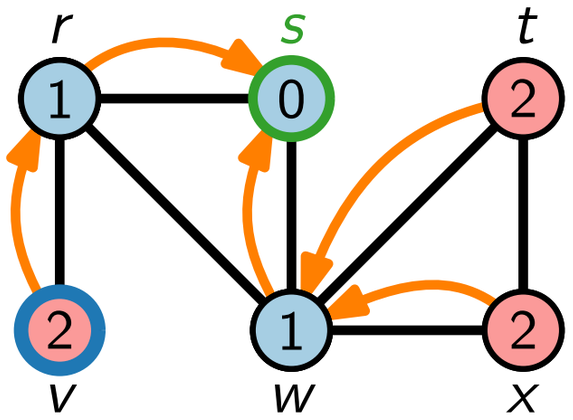


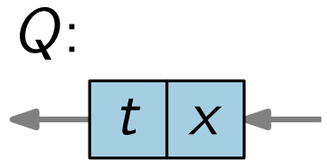



```
INITIALIZE(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```

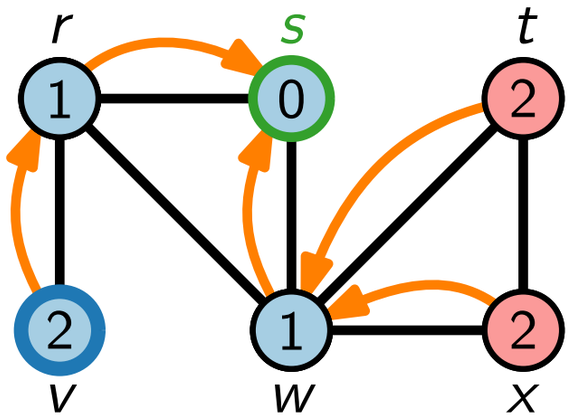




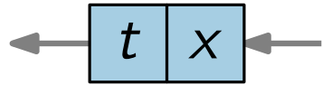

```
INITIALIZE(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```



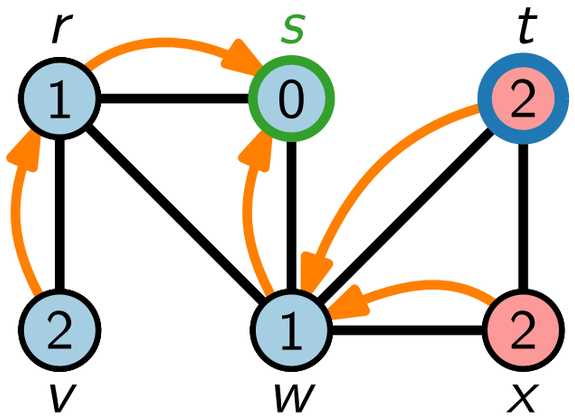
Q:



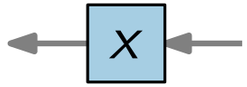

```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```



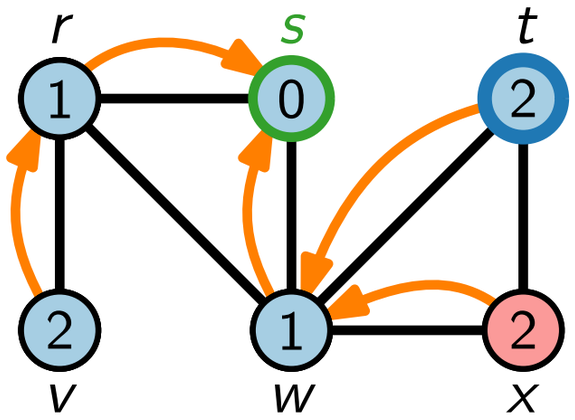
Q:



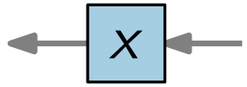

```
INITIALIZE(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```



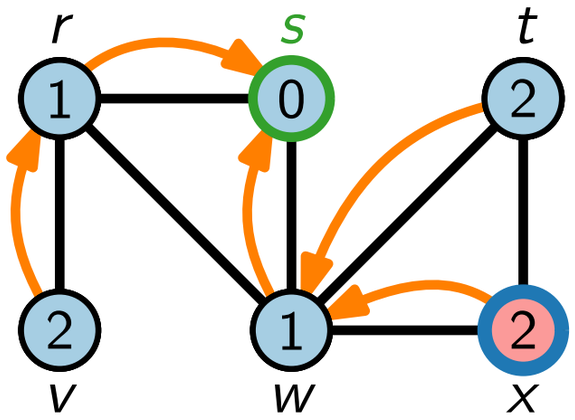
Q:




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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```

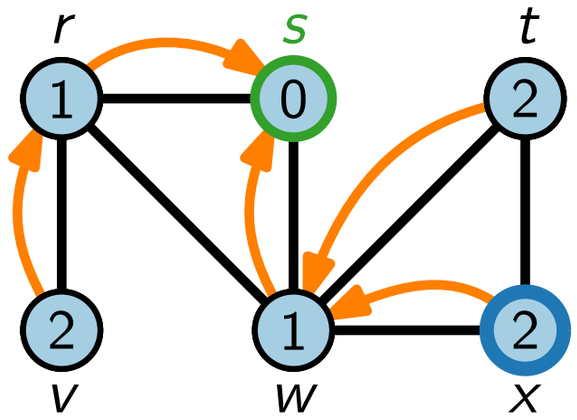


Q:

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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```

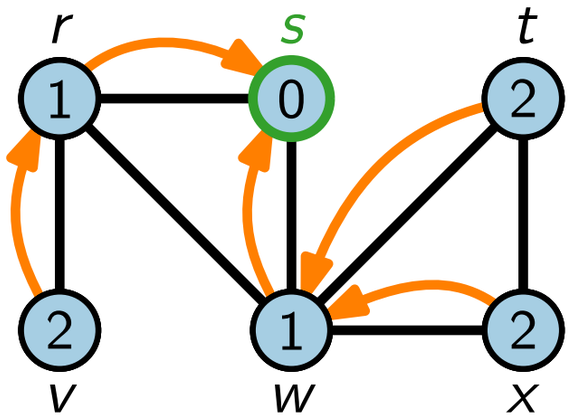


Q:

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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```

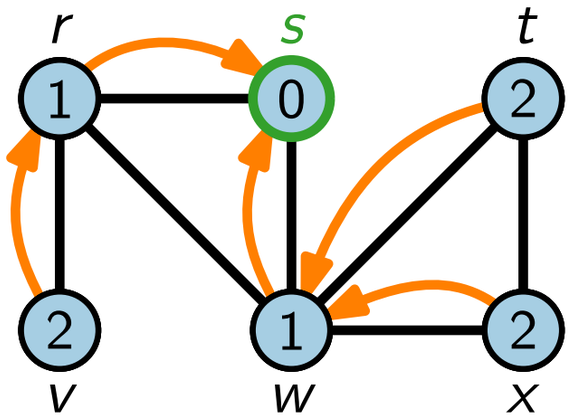


Q:

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

Wiederholung Breitensuche

```
BFS(Graph G, Vertex s)
    INITIALIZE(G, s)
    Q = new Queue()
    Q.ENQUEUE(s)
    while not Q.EMPTY() do
        u = Q.DEQUEUE()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.ENQUEUE(v)
        u.color = blue
```



Q:

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

Laufzeit?

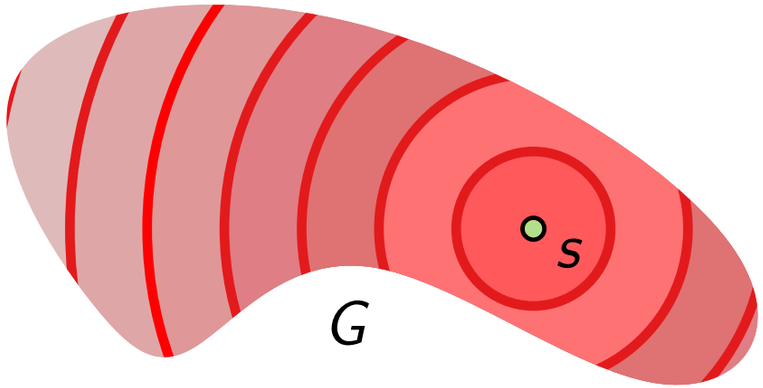
$$\underbrace{\mathcal{O}(|V|)}_{\text{INITIALIZE}} + \underbrace{\mathcal{O}(|V|)}_{\text{EN-/DEQUEUES}} + \underbrace{\mathcal{O}(|E|)}_{\text{Adjazenzlisten (foreach-Schleifen)}} = \mathcal{O}(|V| + |E|)$$

Beob. über Knotengrade!

Ausbreitung

Breitensuche (breadth-first search, BFS)

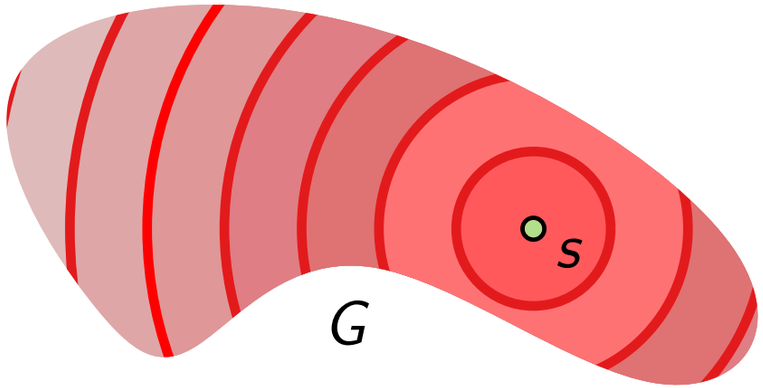
Alle Kanten gleich schwer

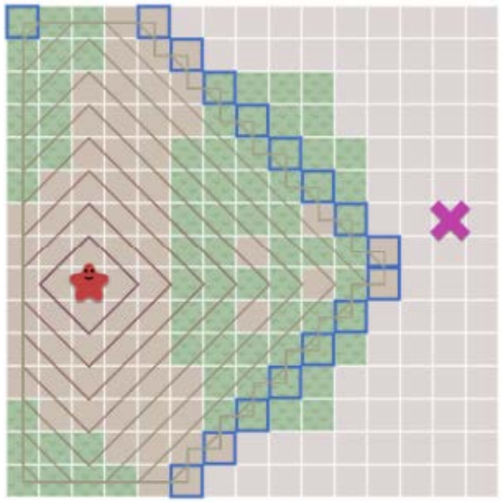


Ausbreitung

Breitensuche (breadth-first search, BFS)

Alle Kanten gleich schwer



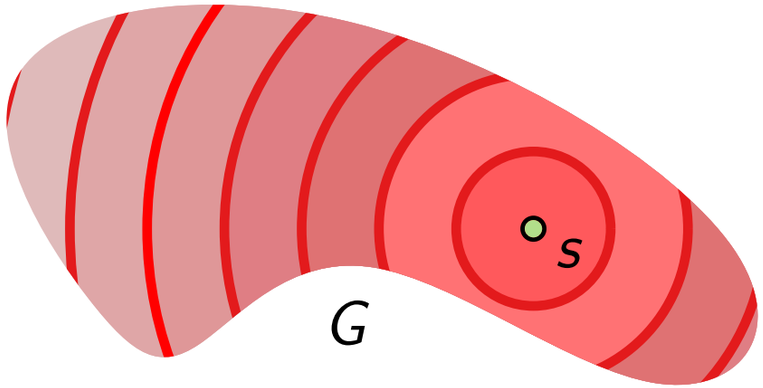


Amit Patel, "Introduction to the A^* Algorithm", Red Blob Games, 2014, https://www.redblobgames.com/pathfinding/a-star/introduction.html

Ausbreitung

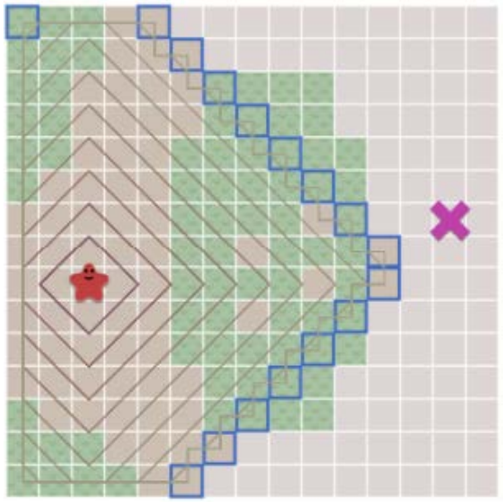
Breitensuche (breadth-first search, BFS)

Alle Kanten gleich schwer



Dijkstra

Kanten unterschiedlich schwer

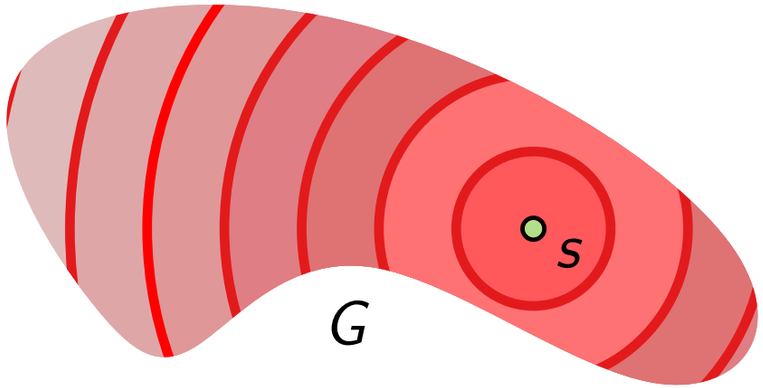


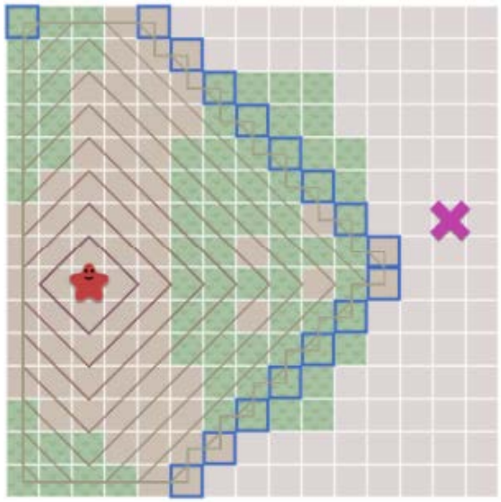
Amit Patel, "Introduction to the A^* Algorithm", Red Blob Games, 2014, https://www.redblobgames.com/pathfinding/a-star/introduction.html

Ausbreitung

Breitensuche (breadth-first search, BFS)

Alle Kanten gleich schwer

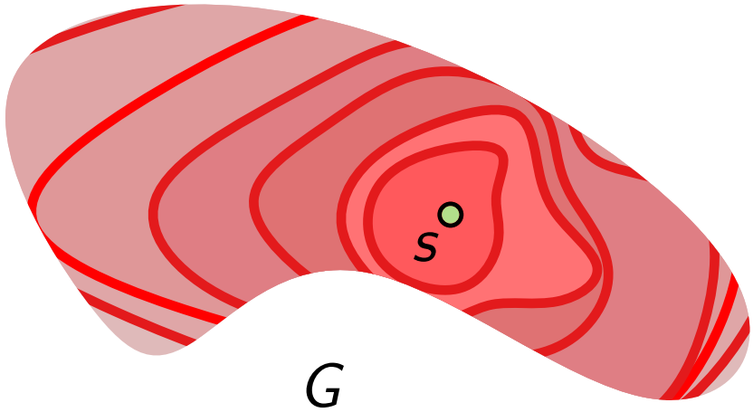




Amit Patel, "Introduction to the A^* Algorithm", Red Blob Games, 2014, https://www.redblobgames.com/pathfinding/a-star/introduction.html

Dijkstra

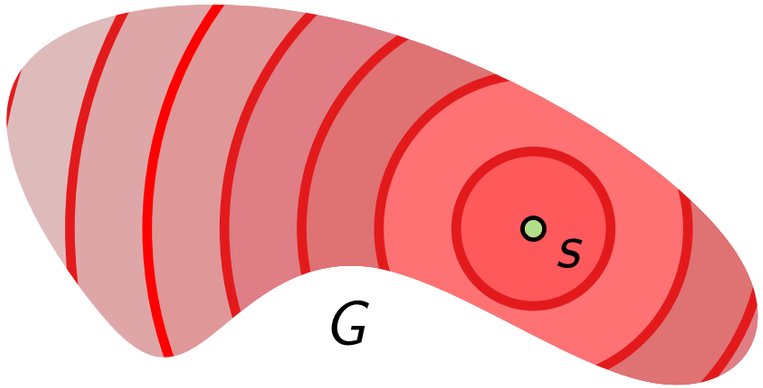
Kanten unterschiedlich schwer

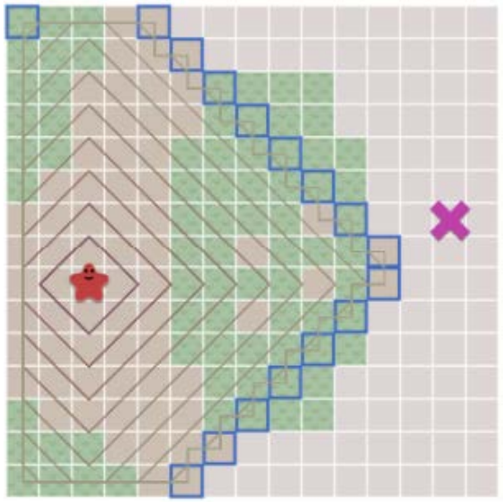


Ausbreitung

Breitensuche (breadth-first search, BFS)

Alle Kanten gleich schwer

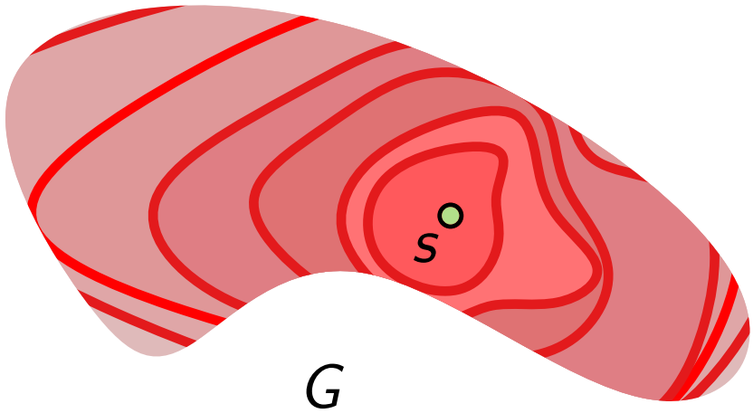


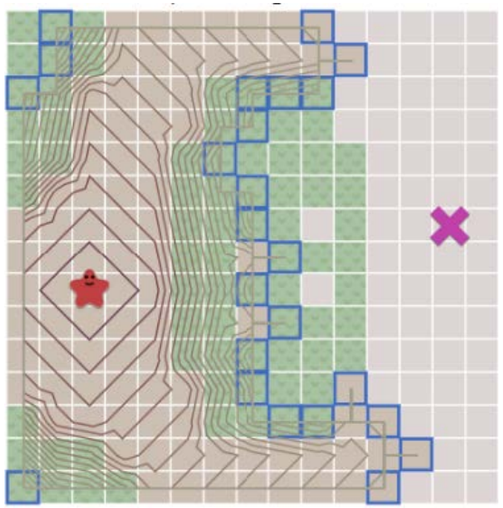


Amit Patel, "Introduction to the A^* Algorithm", Red Blob Games, 2014, https://www.redblobgames.com/pathfinding/a-star/introduction.html

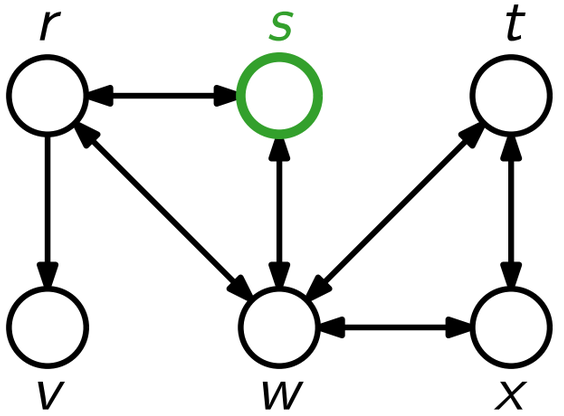
Dijkstra

Kanten unterschiedlich schwer

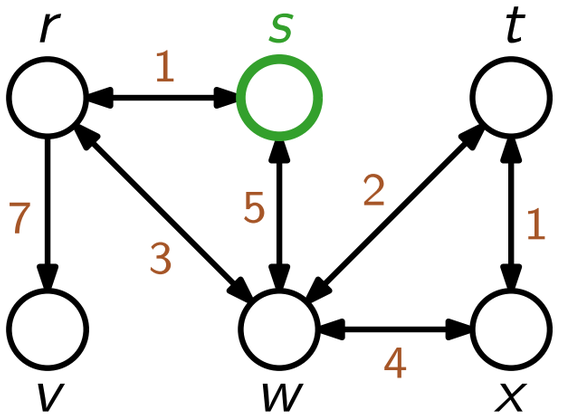




Dijkstra

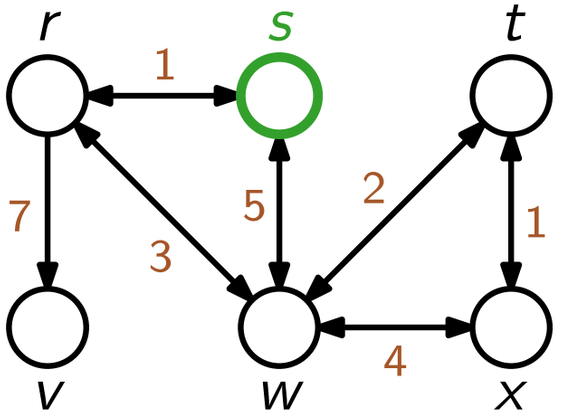


Dijkstra



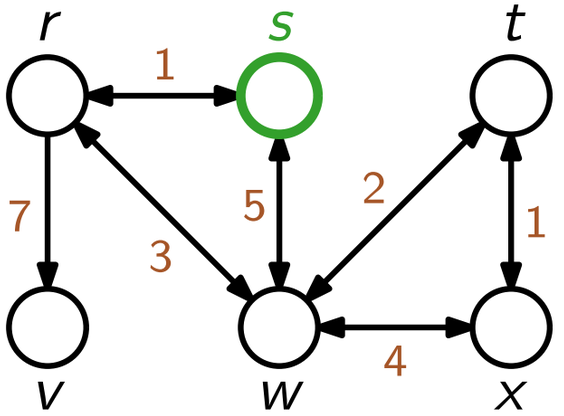
Dijkstra

Edsger W. Dijkstra
(Rotterdam 1930–2002 Nuenen)



Dijkstra

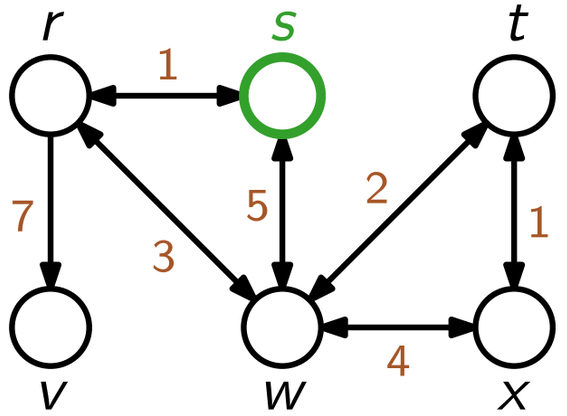
Edsger W. Dijkstra
(Rotterdam 1930–2002 Nuenen)



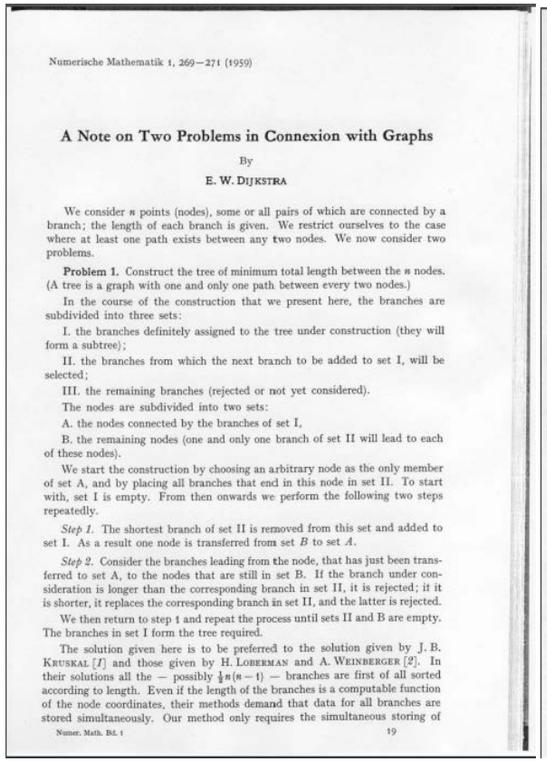
A note on two problems in connexion with graphs.
In: Numerische Mathematik, Band 1, 1959

Dijkstra

Edsger W. Dijkstra
(Rotterdam 1930–2002 Nuenen)



A note on two problems in connexion with graphs.
In: Numerische Mathematik, Band 1, 1959



Numerische Mathematik 1, 269—271 (1959)

A Note on Two Problems in Connexion with Graphs

By

E. W. Dijkstra

We consider n points (nodes), some or all pairs of which are connected by a branch; the length of each branch is given. We restrict ourselves to the case where at least one path exists between any two nodes. We now consider two problems.

Problem 1. Construct the tree of minimum total length between the n nodes. (A tree is a graph with one and only one path between every two nodes.)

In the course of the construction that we present here, the branches are subdivided into three sets:

I. the branches definitely assigned to the tree under construction (they will form a subtree);

II. the branches from which the next branch to be added to set I, will be selected;

III. the remaining branches (rejected or not yet considered).

The nodes are subdivided into two sets:

A. the nodes connected by the branches of set I,

B. the remaining nodes (one and only one branch of set II will lead to each of these nodes).

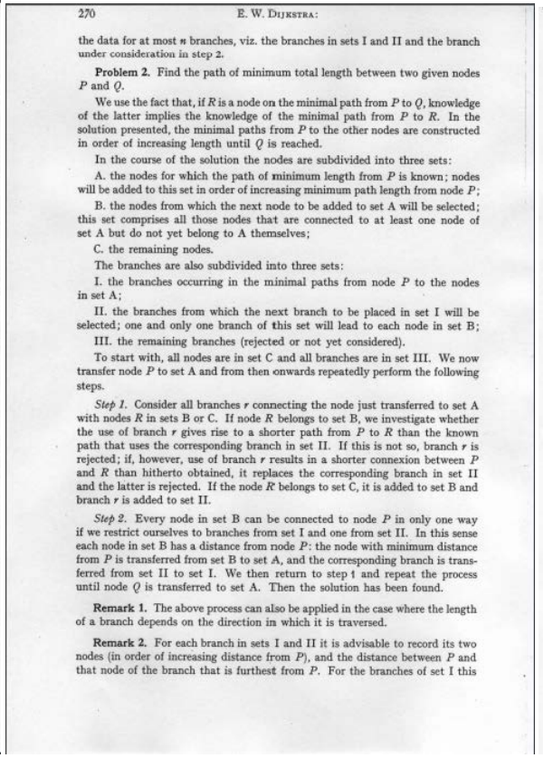
We start the construction by choosing an arbitrary node as the only member of set A, and by placing all branches that end in this node in set II. To start with, set I is empty. From then onwards we perform the following two steps repeatedly.

Step 1. The shortest branch of set II is removed from this set and added to set I. As a result one node is transferred from set B to set A.

Step 2. Consider the branches leading from the node, that has just been transferred to set A, to the nodes that are still in set B. If the branch under consideration is longer than the corresponding branch in set II, it is rejected; if it is shorter, it replaces the corresponding branch in set II, and the latter is rejected.

We then return to step 1 and repeat the process until sets II and B are empty. The branches in set I form the tree required.

The solution given here is to be preferred to the solution given by J. B. Kruskal [1] and those given by H. Loberman and A. Weinberger [2]. In their solutions all the — possibly $\frac{1}{2}n(n-1)$ — branches are first of all sorted according to length. Even if the length of the branches is a computable function of the node coordinates, their methods demand that data for all branches are stored simultaneously. Our method only requires the simultaneous storing of



the data for at most n branches, viz. the branches in sets I and II and the branch under consideration in step 2.

Problem 2. Find the path of minimum total length between two given nodes P and Q.

We use the fact that, if R is a node on the minimal path from P to Q, knowledge of the latter implies the knowledge of the minimal path from P to R. In the solution presented, the minimal paths from P to the other nodes are constructed in order of increasing length until Q is reached.

In the course of the solution the nodes are subdivided into three sets:

A. the nodes for which the path of minimum length from P is known; nodes will be added to this set in order of increasing minimum path length from node P;

B. the nodes from which the next node to be added to set A will be selected; this set comprises all those nodes that are connected to at least one node of set A but do not yet belong to A themselves;

C. the remaining nodes.

The branches are also subdivided into three sets:

I. the branches occurring in the minimal paths from node P to the nodes in set A;

II. the branches from which the next branch to be placed in set I will be selected; one and only one branch of this set will lead to each node in set B;

III. the remaining branches (rejected or not yet considered).

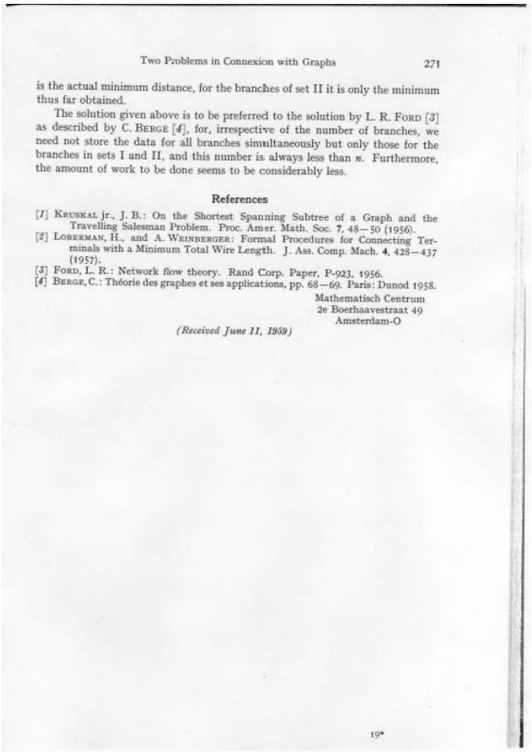
To start with, all nodes are in set C and all branches are in set III. We now transfer node P to set A and from then onwards repeatedly perform the following steps.

Step 1. Consider all branches r connecting the node just transferred to set A with nodes R in sets B or C. If node R belongs to set B, we investigate whether the use of branch r gives rise to a shorter path from P to R than the known path that uses the corresponding branch in set II. If this is not so, branch r is rejected; if, however, use of branch r results in a shorter connexion between P and R than hitherto obtained, it replaces the corresponding branch in set II and the latter is rejected. If the node R belongs to set C, it is added to set B and branch r is added to set II.

Step 2. Every node in set B can be connected to node P in only one way if we restrict ourselves to branches from set I and one from set II. In this sense each node in set B has a distance from node P: the node with minimum distance from P is transferred from set B to set A, and the corresponding branch is transferred from set II to set I. We then return to step 1 and repeat the process until node Q is transferred to set A. Then the solution has been found.

Remark 1. The above process can also be applied in the case where the length of a branch depends on the direction in which it is traversed.

Remark 2. For each branch in sets I and II it is advisable to record its two nodes (in order of increasing distance from P), and the distance between P and that node of the branch that is furthest from P. For the branches of set I this



is the actual minimum distance, for the branches of set II it is only the minimum thus far obtained.

The solution given above is to be preferred to the solution by L. R. Ford [3] as described by C. Berge [4], for, irrespective of the number of branches, we need not store the data for all branches simultaneously but only those for the branches in sets I and II, and this number is always less than n. Furthermore, the amount of work to be done seems to be considerably less.

References

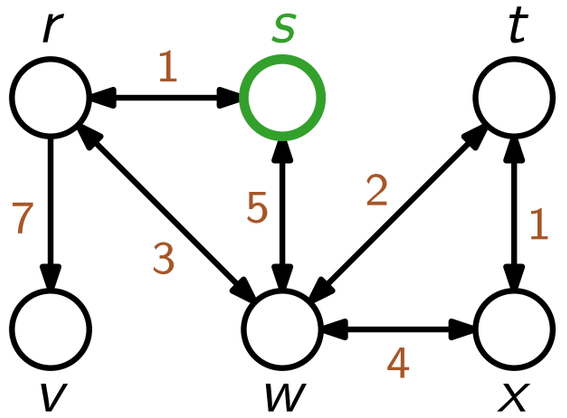
[1] Kruskal jr., J. B.: On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem. Proc. Amer. Math. Soc. 7, 48—50 (1956).
[2] Loberman, H., and A. Weinberger: Formal Procedures for Connecting Terminals with a Minimum Total Wire Length. J. Ass. Comp. Mach. 4, 428—437 (1957).
[3] Ford, L. R.: Network flow theory. Rand Corp. Paper, P-923, 1956.
[4] Berge, C.: Théorie des graphes et ses applications, pp. 68—69. Paris: Dunod 1958.

Mathematisch Centrum
2e Boerhaavestraat 49
Amsterdam-O

(Received June 11, 1959)

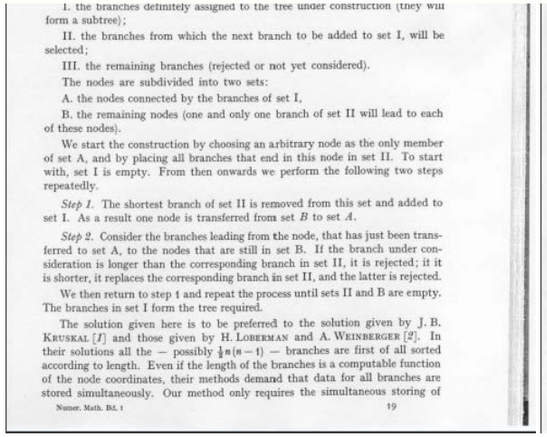
Dijkstra

Edsger W. Dijkstra
(Rotterdam 1930–2002 Nuenen)



A note on two problems in connexion with graphs.
In: Numerische Mathematik, Band 1, 1959

The solution given above is to be preferred to the solution by L. R. Ford [3] as described by C. Berge [4], for, irrespective of the number of branches, we need not store the data for all branches simultaneously but only those for the branches in sets I and II, and this number is always less than n. Furthermore, the amount of work to be done seems to be considerably less.



I. the branches definitely assigned to the tree under construction (they will form a subtree);

II. the branches from which the next branch to be added to set I, will be selected;

III. the remaining branches (rejected or not yet considered).

The nodes are subdivided into two sets:

A. the nodes connected by the branches of set I,

B. the remaining nodes (one and only one branch of set II will lead to each of these nodes).

We start the construction by choosing an arbitrary node as the only member of set A, and by placing all branches that end in this node in set II. To start with, set I is empty. From then onwards we perform the following two steps repeatedly.

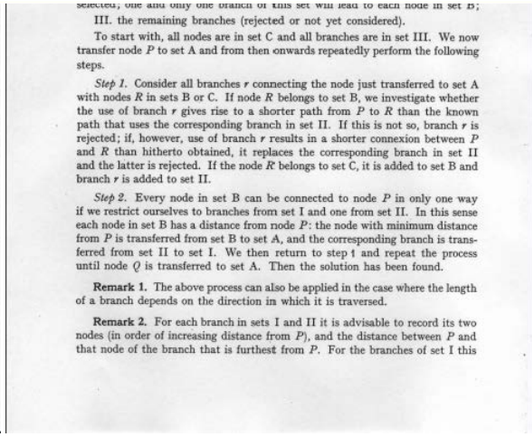
Step 1. The shortest branch of set II is removed from this set and added to set I. As a result one node is transferred from set B to set A.

Step 2. Consider the branches leading from the node, that has just been transferred to set A, to the nodes that are still in set B. If the branch under consideration is longer than the corresponding branch in set II, it is rejected; if it is shorter, it replaces the corresponding branch in set II, and the latter is rejected.

We then return to step 1 and repeat the process until sets II and B are empty. The branches in set I form the tree required.

The solution given here is to be preferred to the solution given by J. B. Kruskal [1] and those given by H. Loberman and A. Weinberger [2]. In their solutions all the — possibly $\frac{1}{2}n(n-1)$ — branches are first of all sorted according to length. Even if the length of the branches is a computable function of the node coordinates, their methods demand that data for all branches are stored simultaneously. Our method only requires the simultaneous storing of

Numer. Math. Bd. 1 19



selected; one and only one branch of this set will lead to each node in set B;

III. the remaining branches (rejected or not yet considered).

To start with, all nodes are in set C and all branches are in set III. We now transfer node P to set A and from then onwards repeatedly perform the following steps.

Step 1. Consider all branches r connecting the node just transferred to set A with nodes R in sets B or C. If node R belongs to set B, we investigate whether the use of branch r gives rise to a shorter path from P to R than the known path that uses the corresponding branch in set II. If this is not so, branch r is rejected; if, however, use of branch r results in a shorter connexion between P and R than hitherto obtained, it replaces the corresponding branch in set II and the latter is rejected. If the node R belongs to set C, it is added to set B and branch r is added to set II.

Step 2. Every node in set B can be connected to node P in only one way if we restrict ourselves to branches from set I and one from set II. In this sense each node in set B has a distance from node P: the node with minimum distance from P is transferred from set B to set A, and the corresponding branch is transferred from set II to set I. We then return to step 1 and repeat the process until node Q is transferred to set A. Then the solution has been found.

Remark 1. The above process can also be applied in the case where the length of a branch depends on the direction in which it is traversed.

Remark 2. For each branch in sets I and II it is advisable to record its two nodes (in order of increasing distance from P), and the distance between P and that node of the branch that is furthest from P. For the branches of set I this

(Received June 11, 1959)

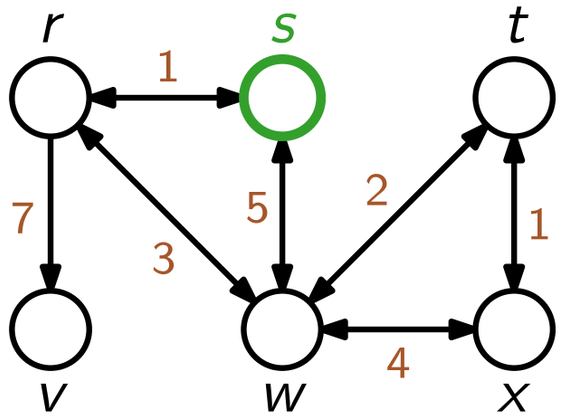
19*

Dijkstra

Edsger W. Dijkstra
(Rotterdam 1930–2002 Nuenen)

ARMAC, 1960

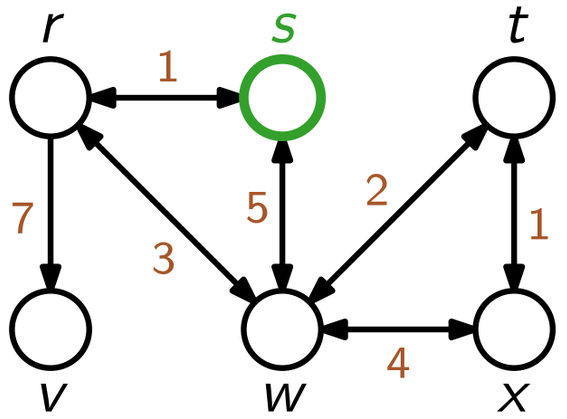
Dijkstra




```
BFS(Graph G, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

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

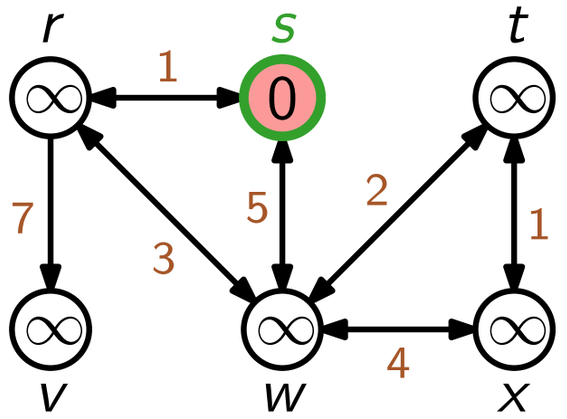
Dijkstra




```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

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

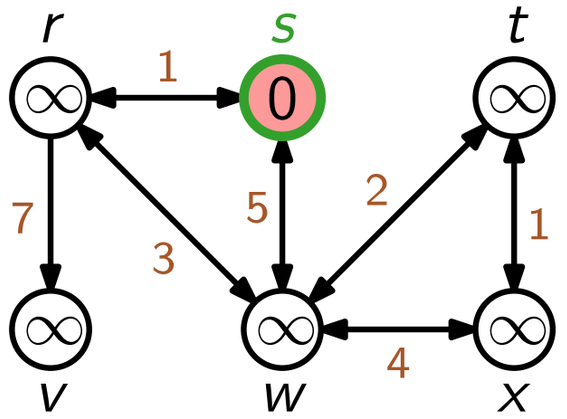
Dijkstra




```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new Queue()
    Q.Enqueue(s)
    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

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

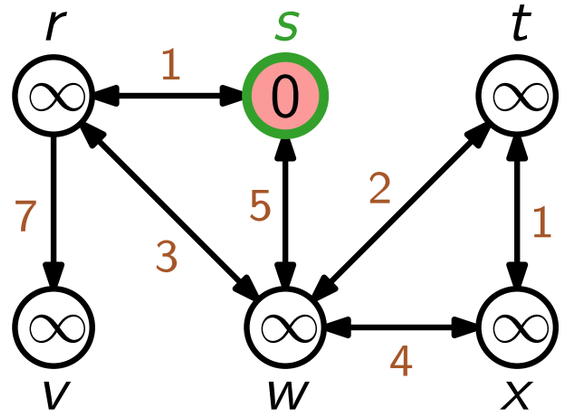
Dijkstra

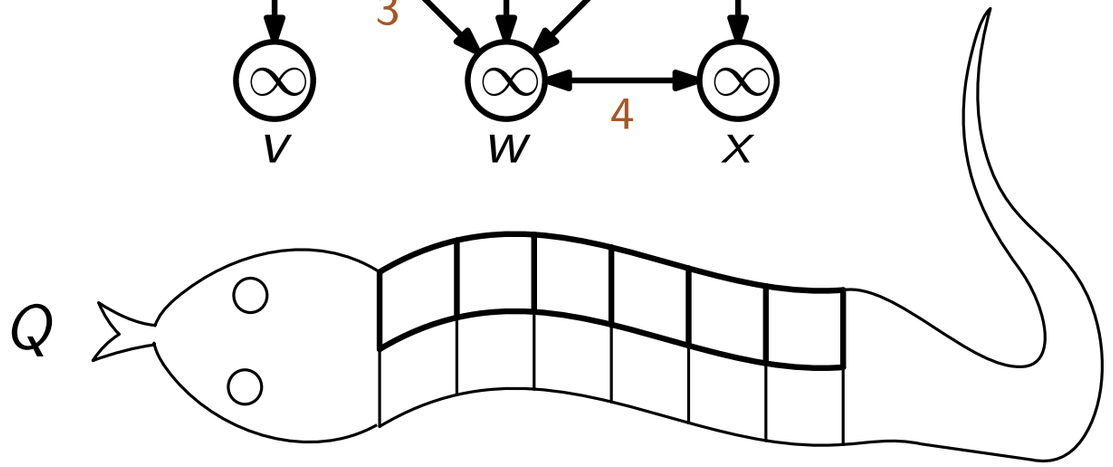



```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

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

Dijkstra

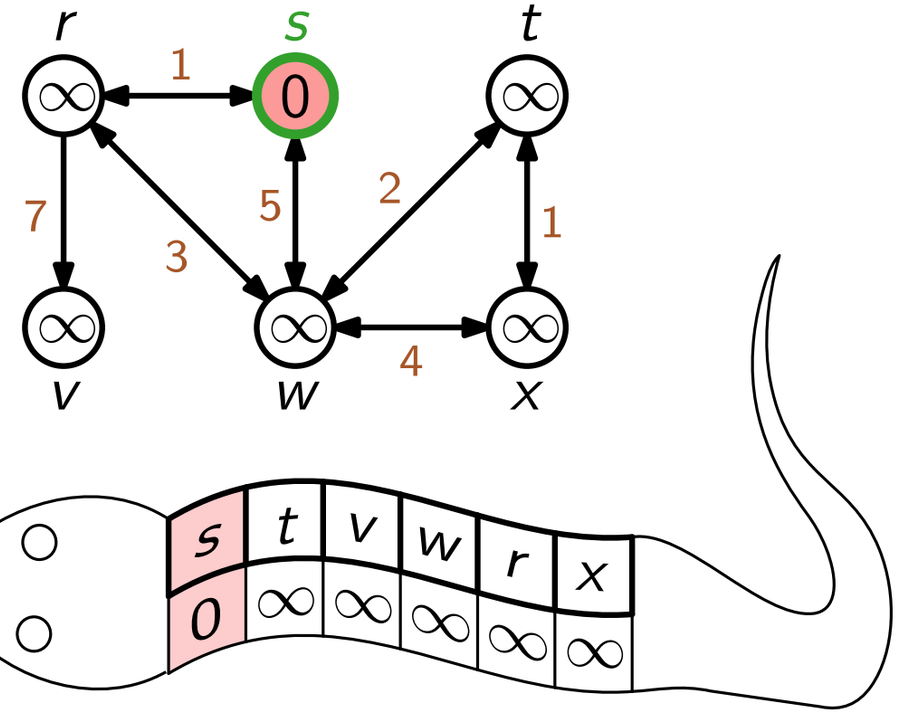





```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

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

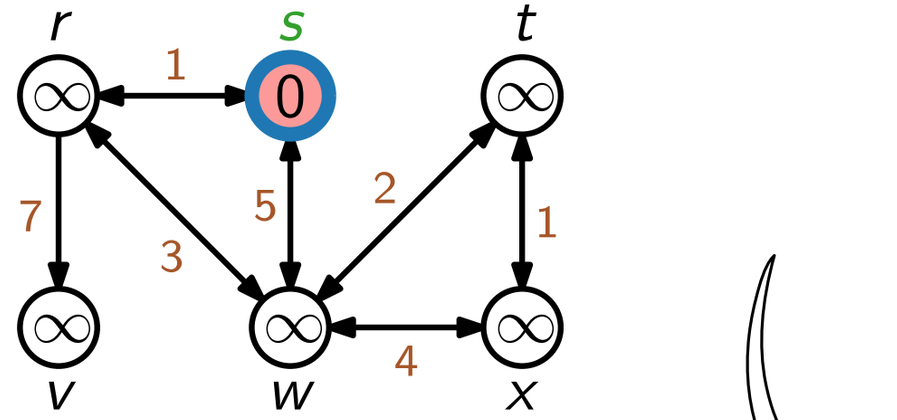
Dijkstra

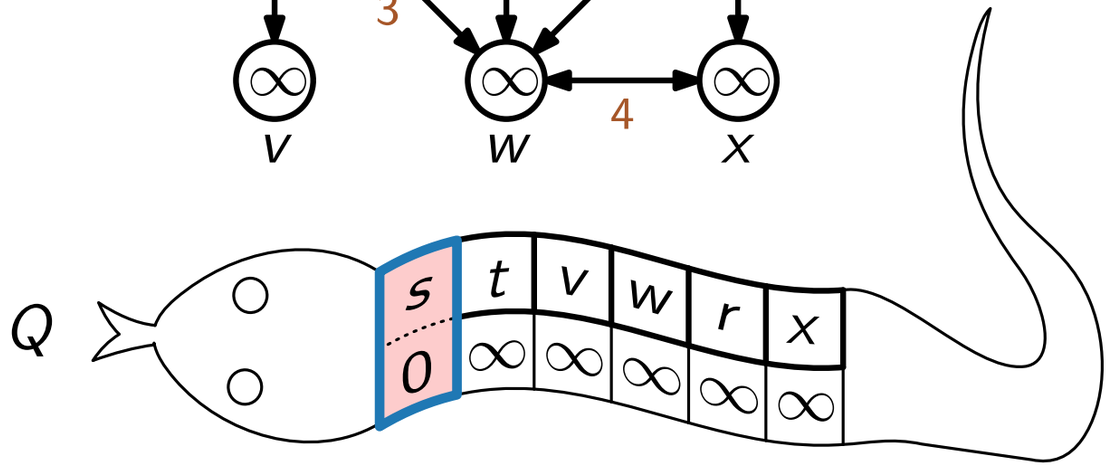



```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.Dequeue()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

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

Dijkstra



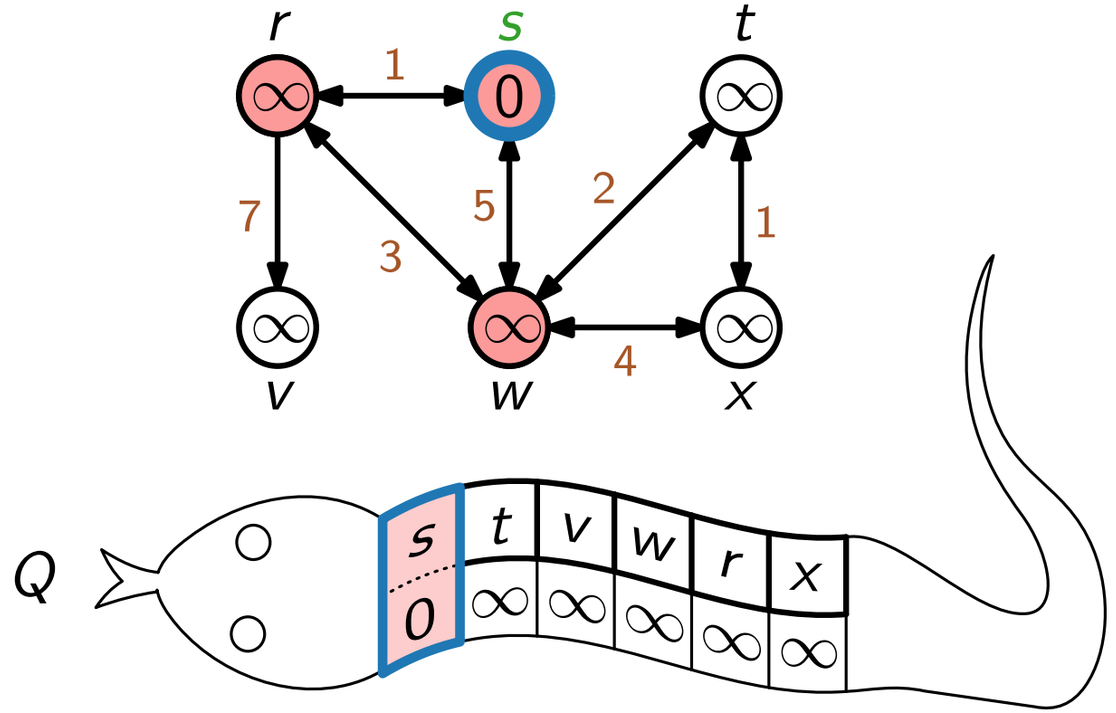



```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

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

Dijkstra

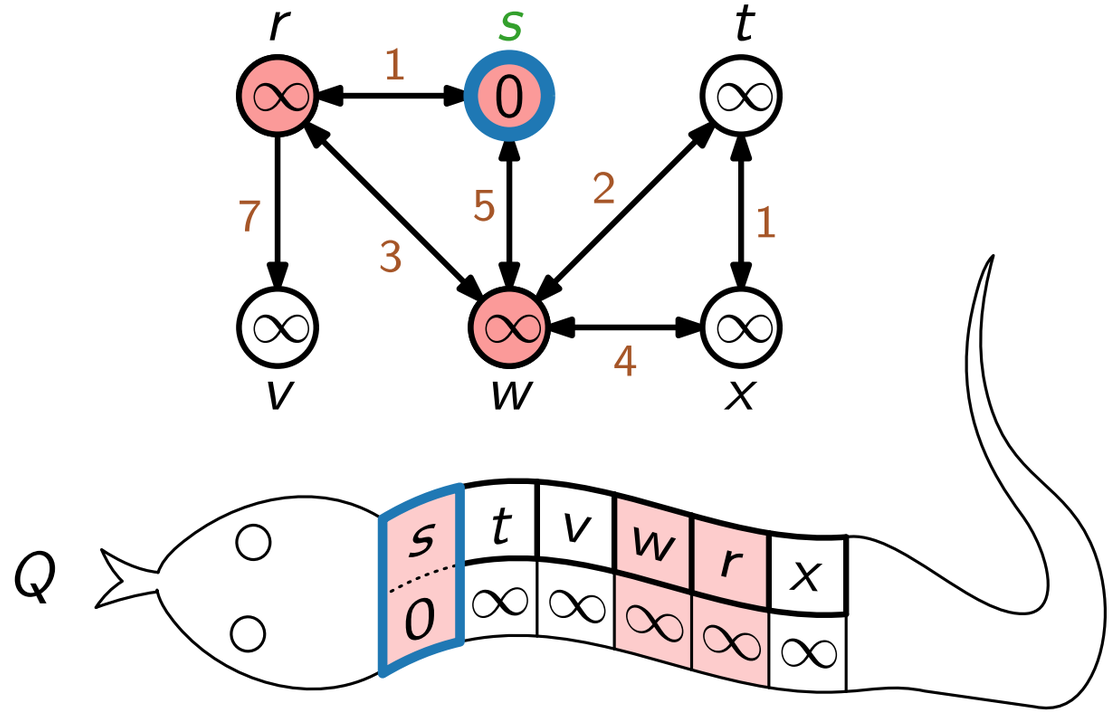
```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```




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

Dijkstra

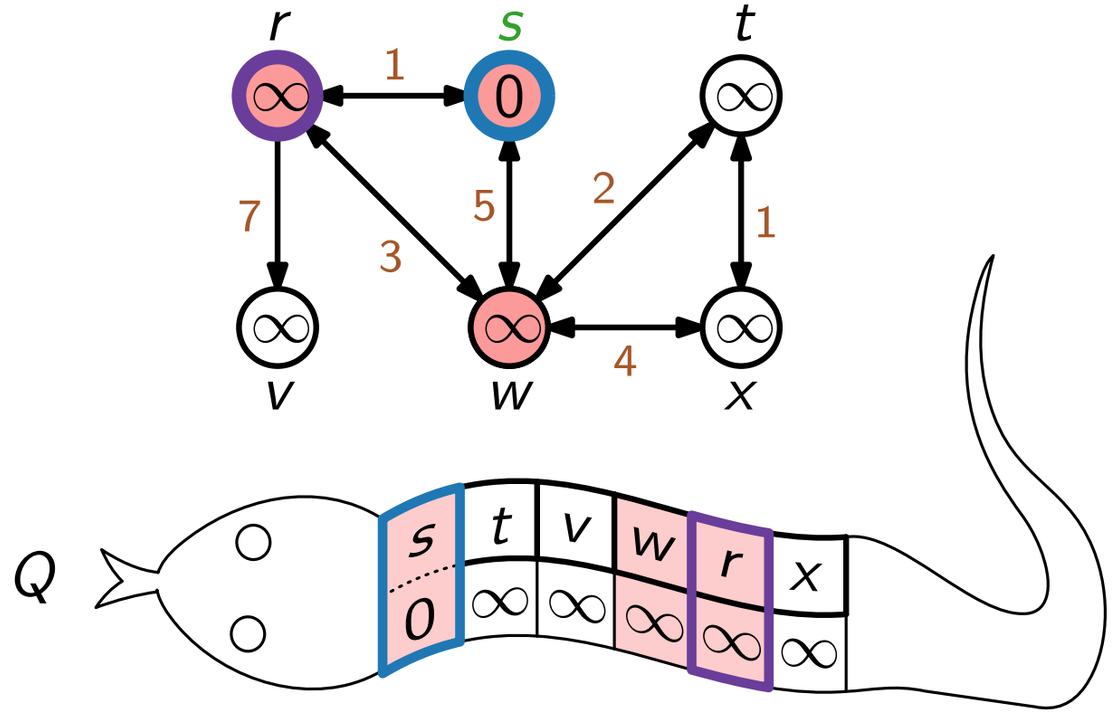
```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```




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

Dijkstra

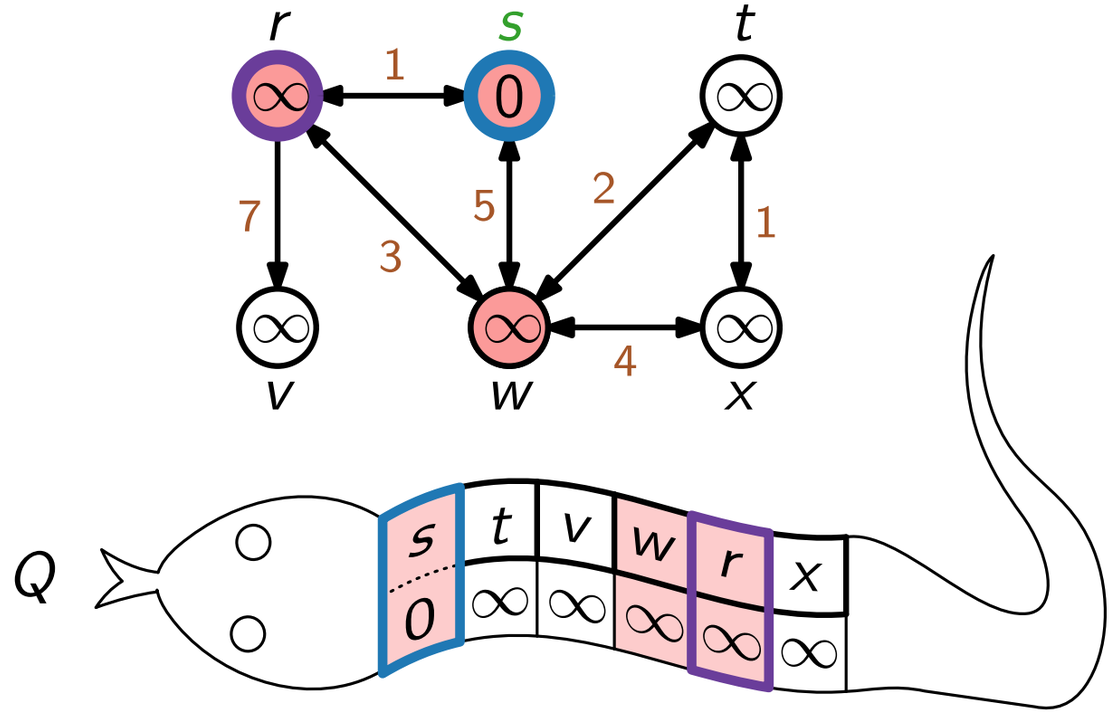
```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```




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

Dijkstra

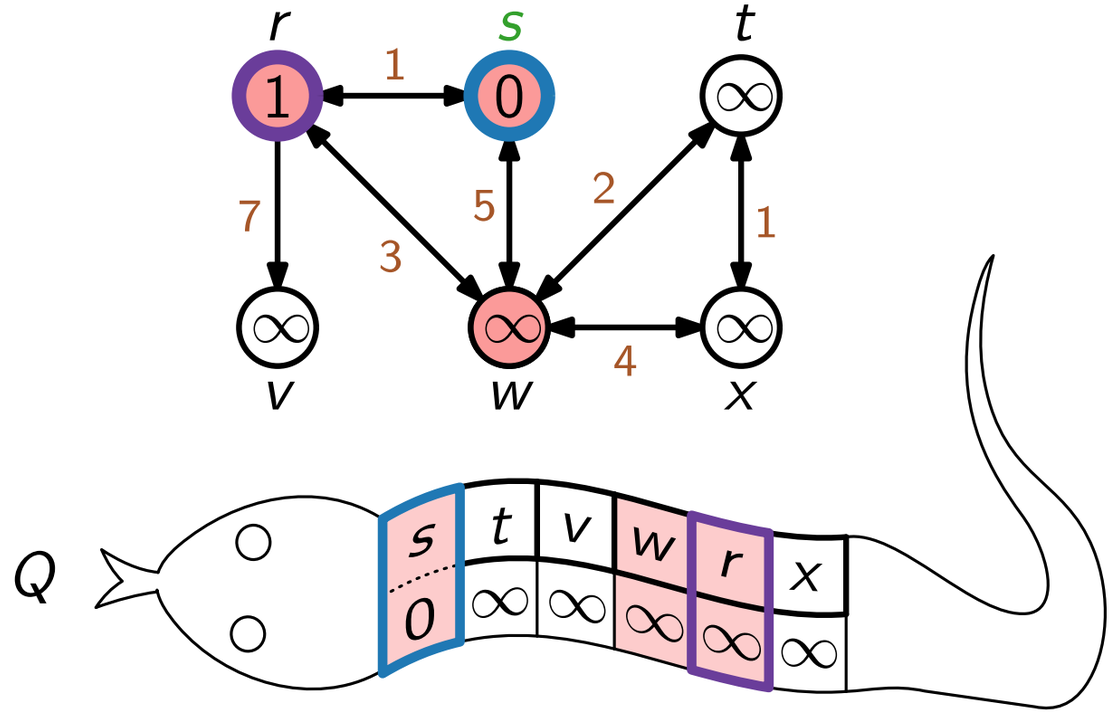
```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```




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

Dijkstra

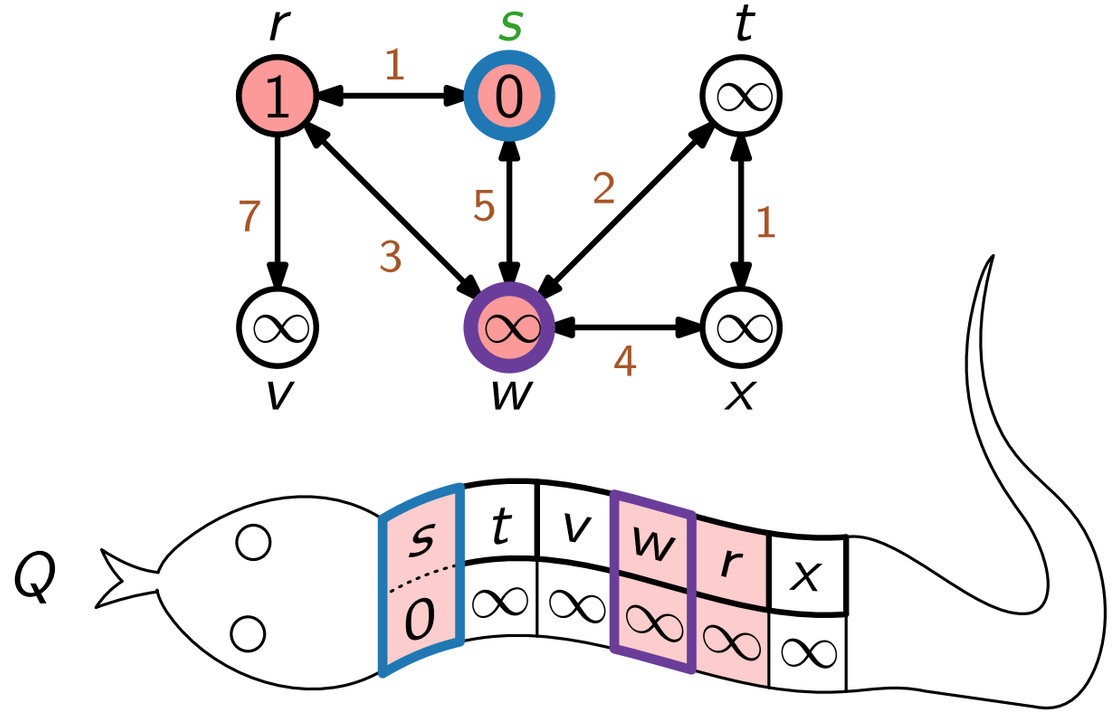
```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```




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

Dijkstra

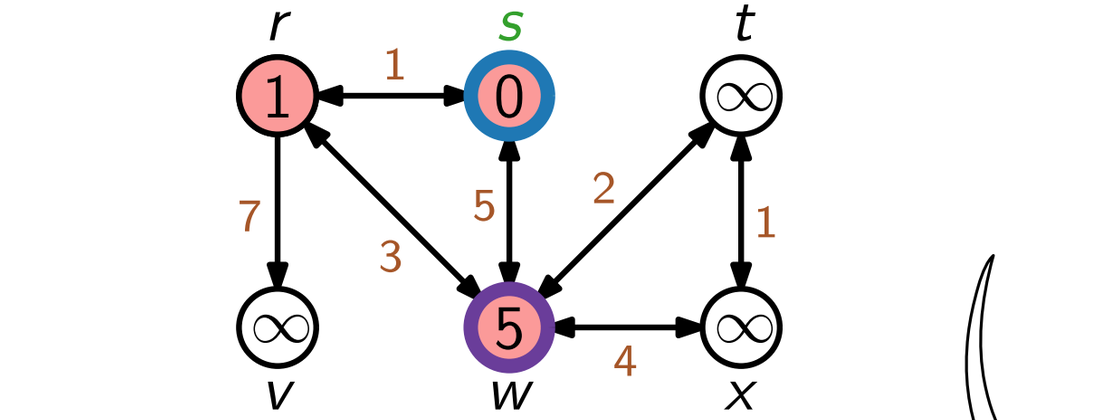
```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

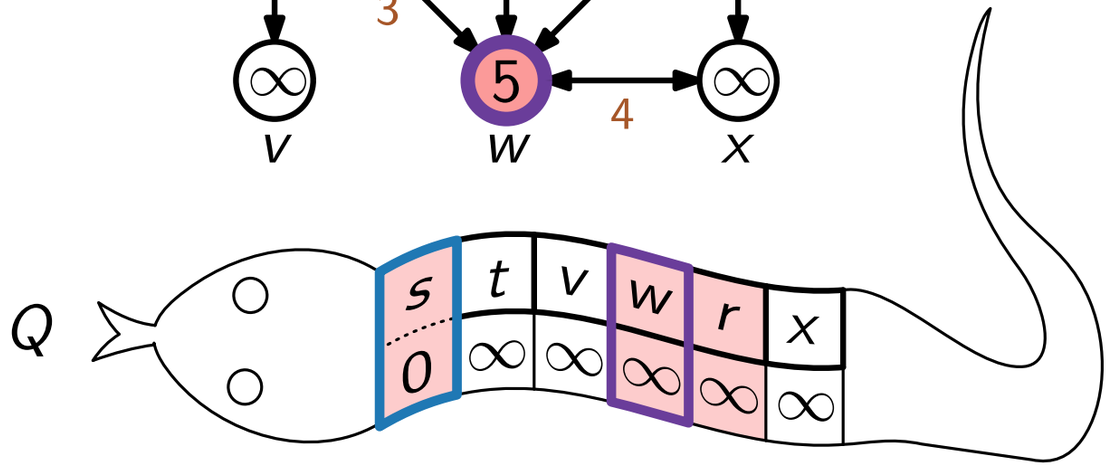



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

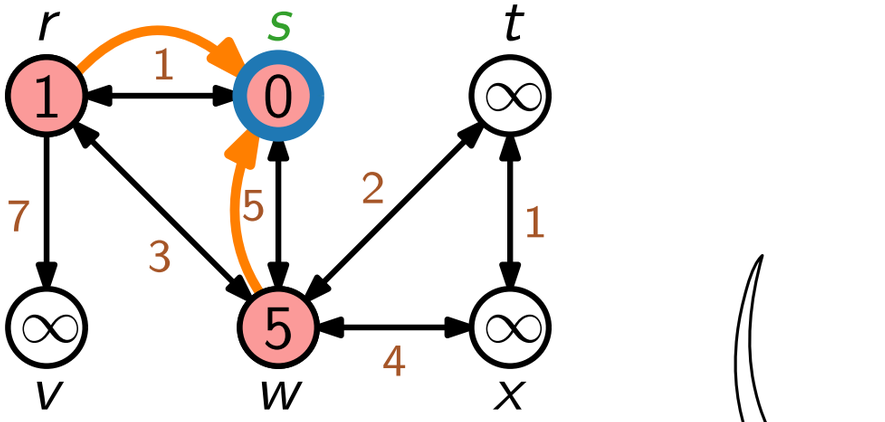


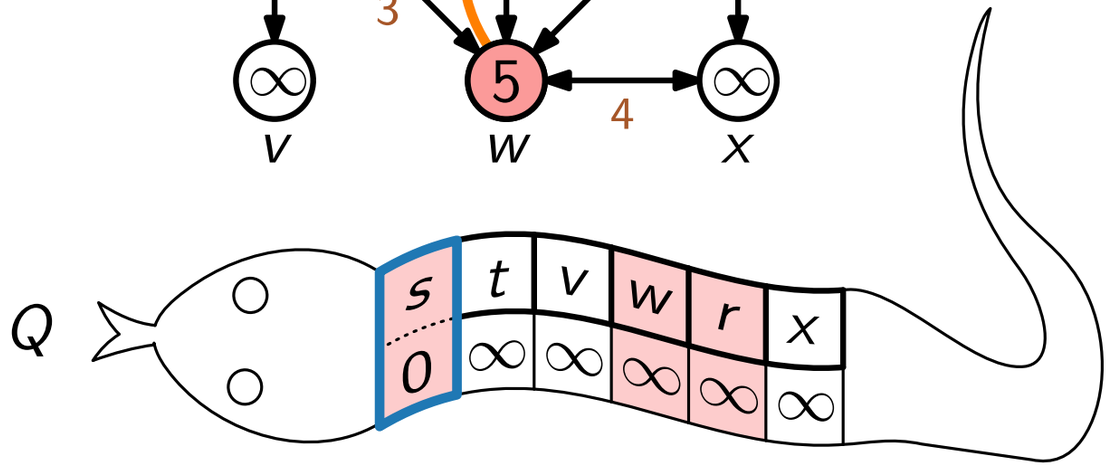



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.Enqueue(v)
        u.color = blue
```

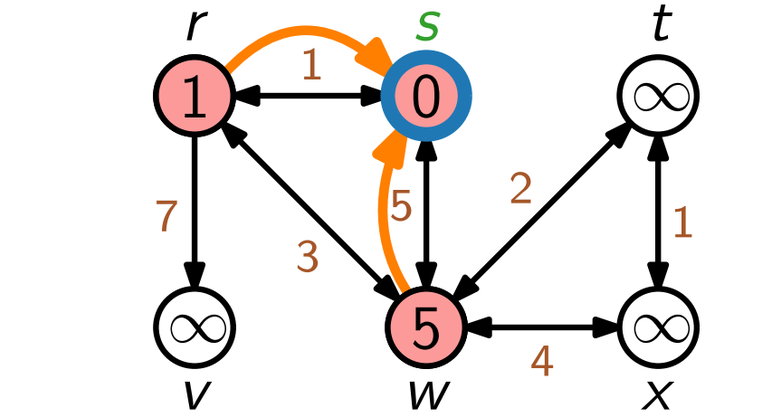


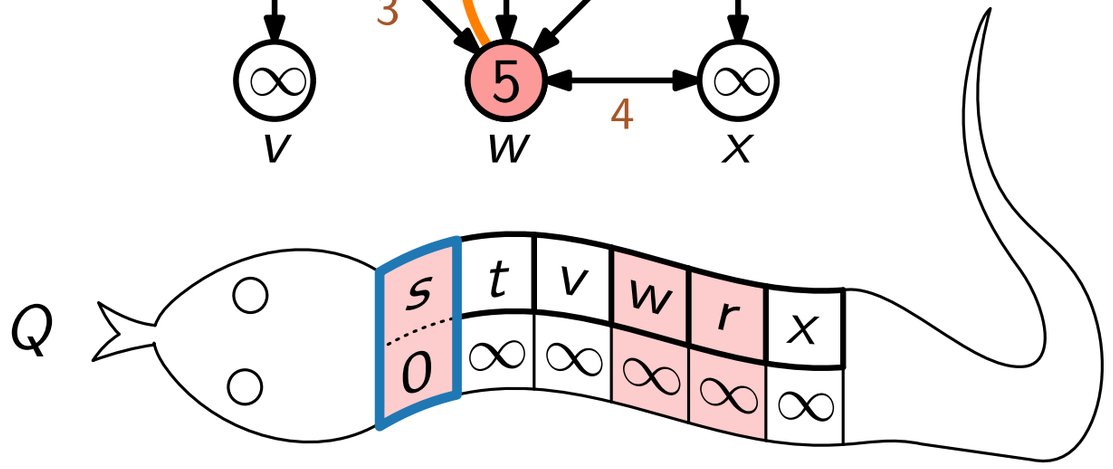



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

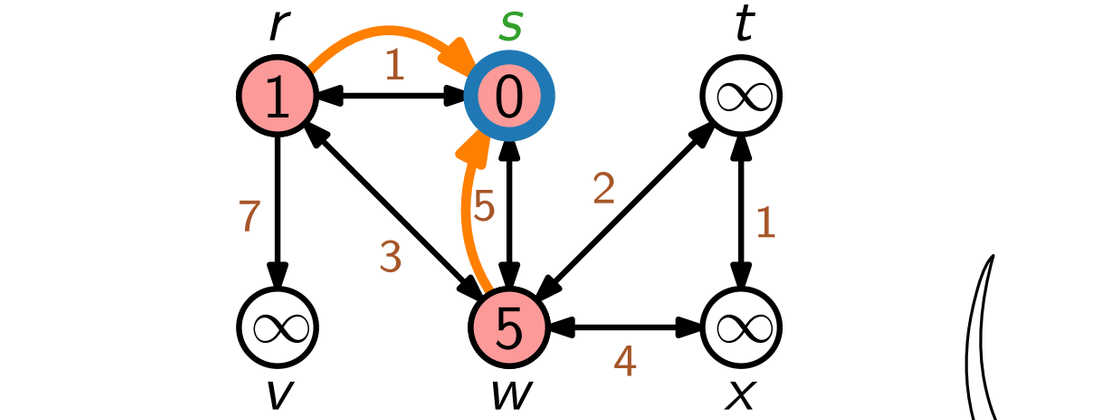


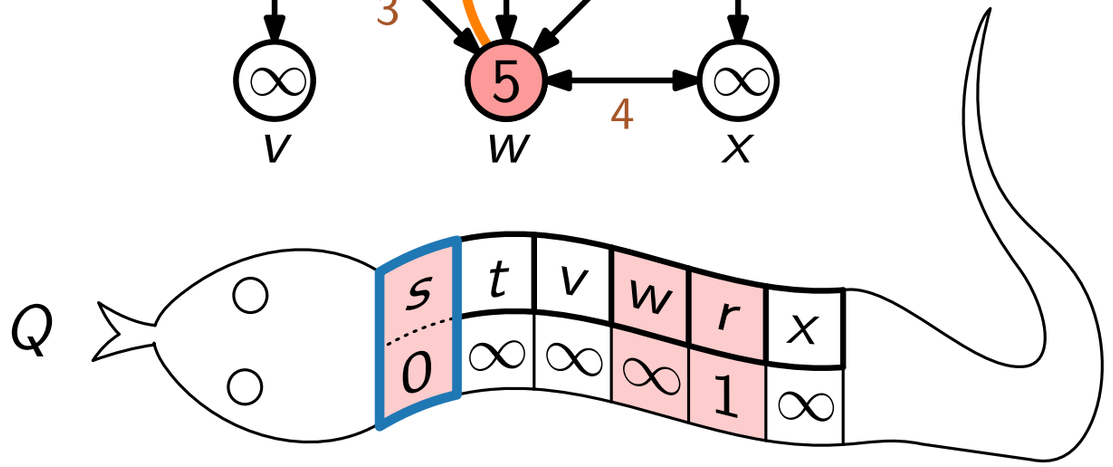



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

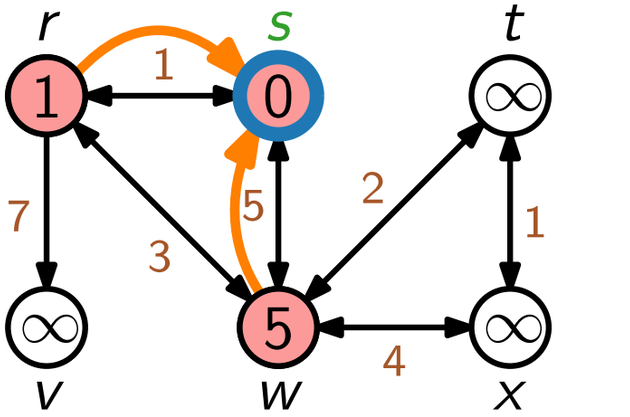


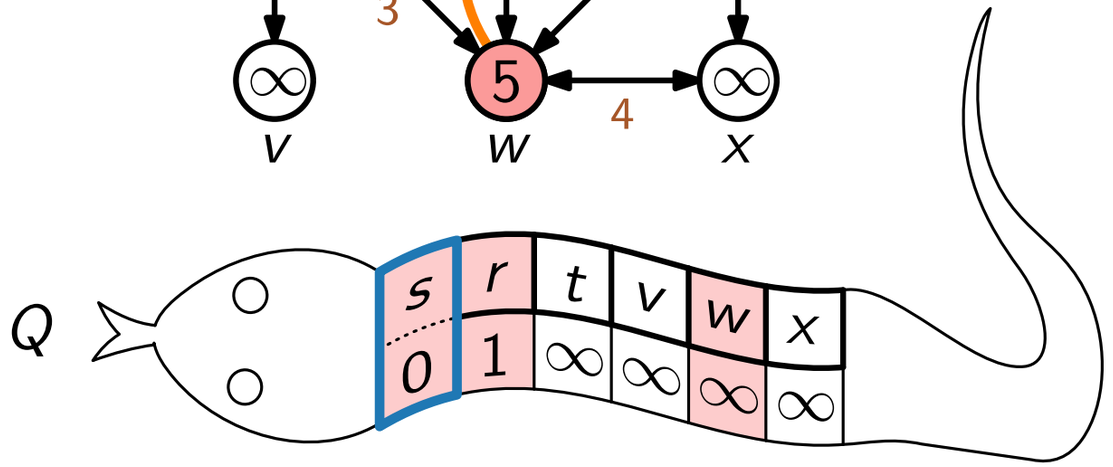



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

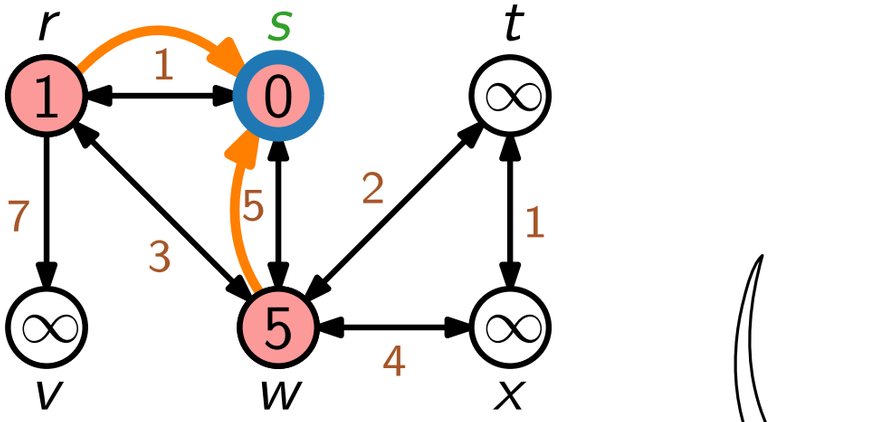


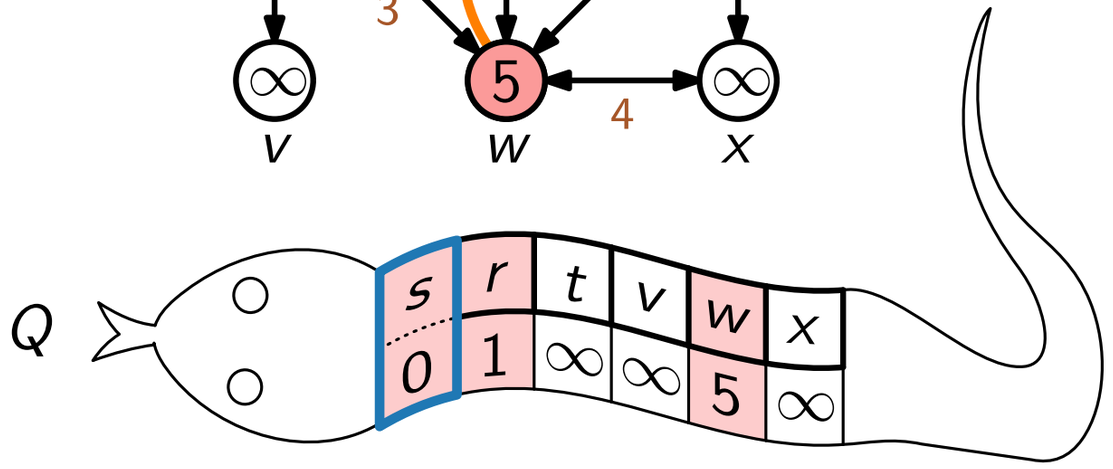



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

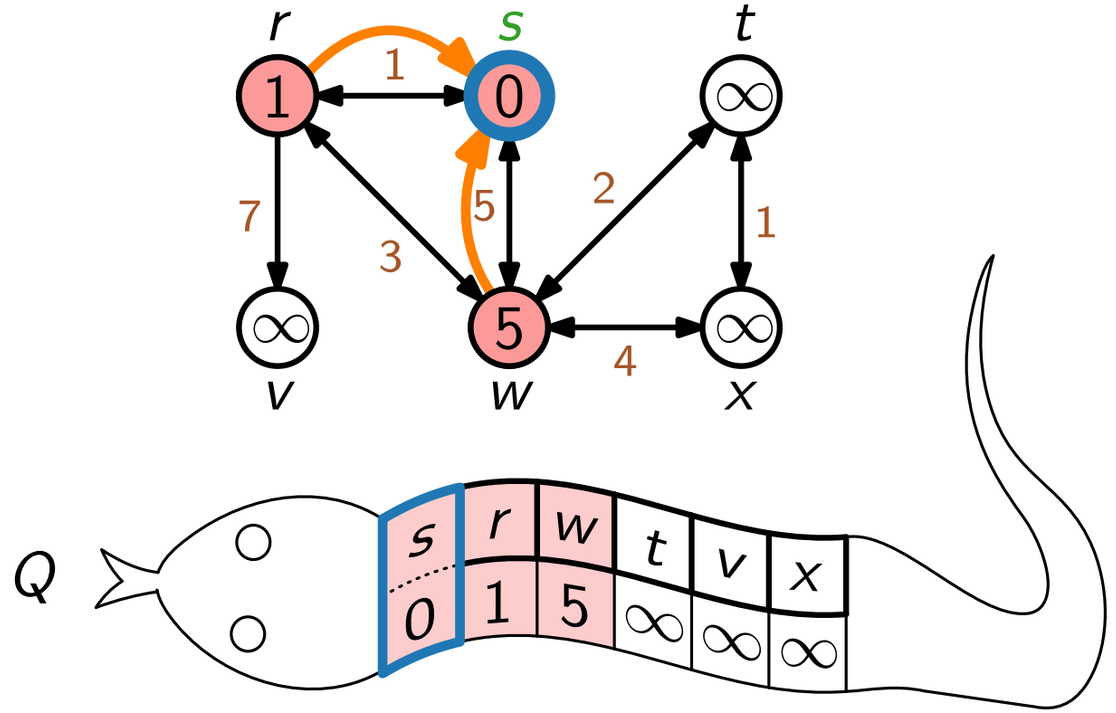





```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

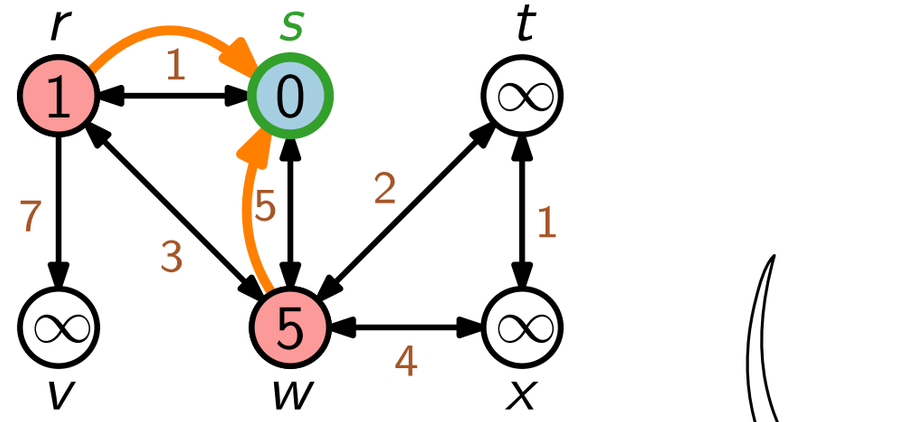
```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

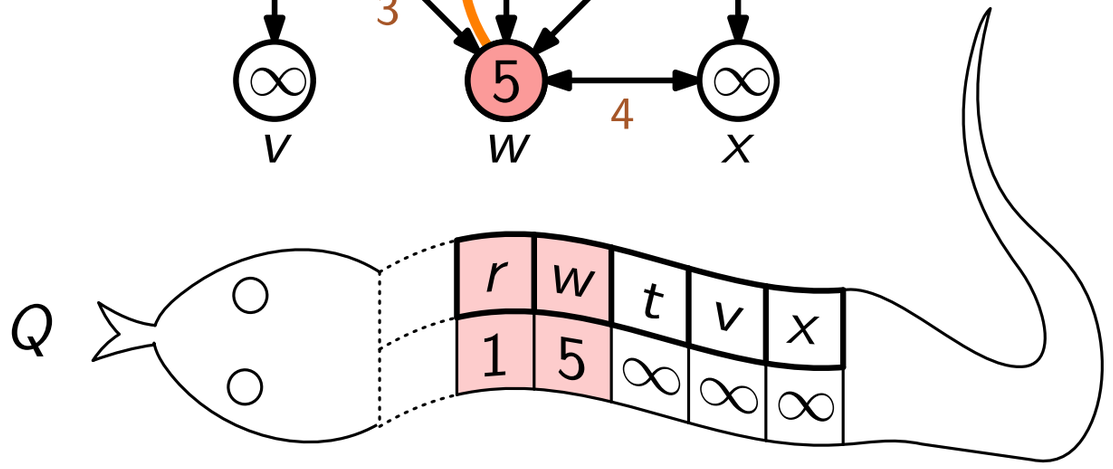



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

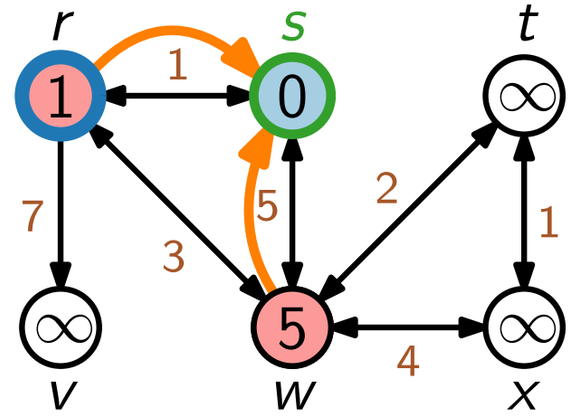


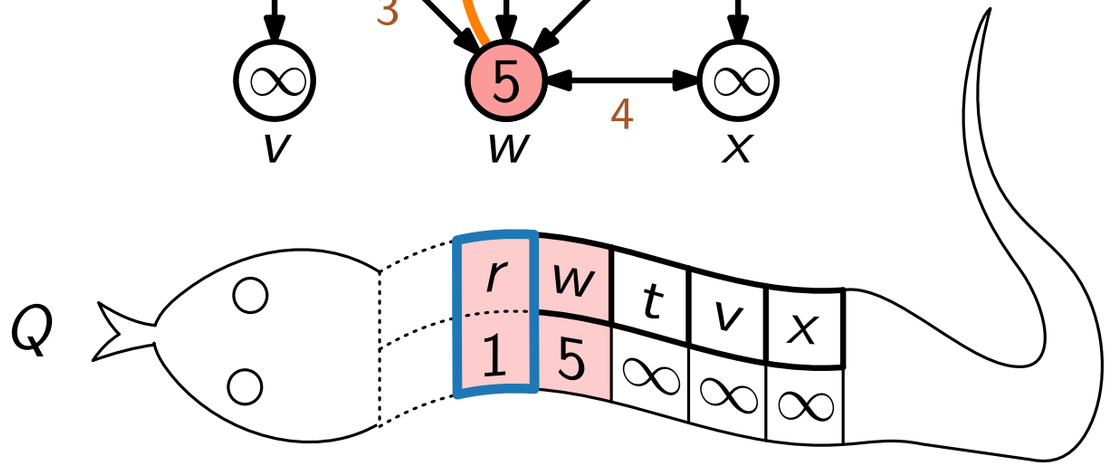



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

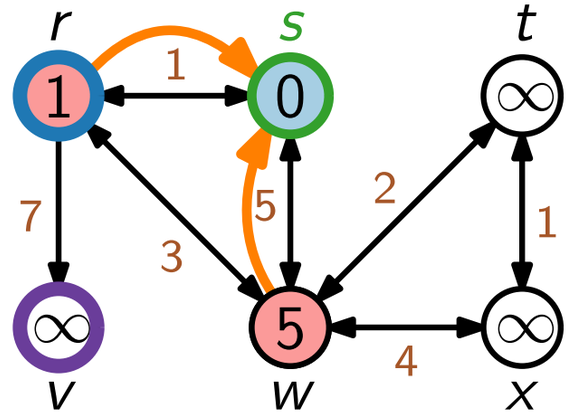


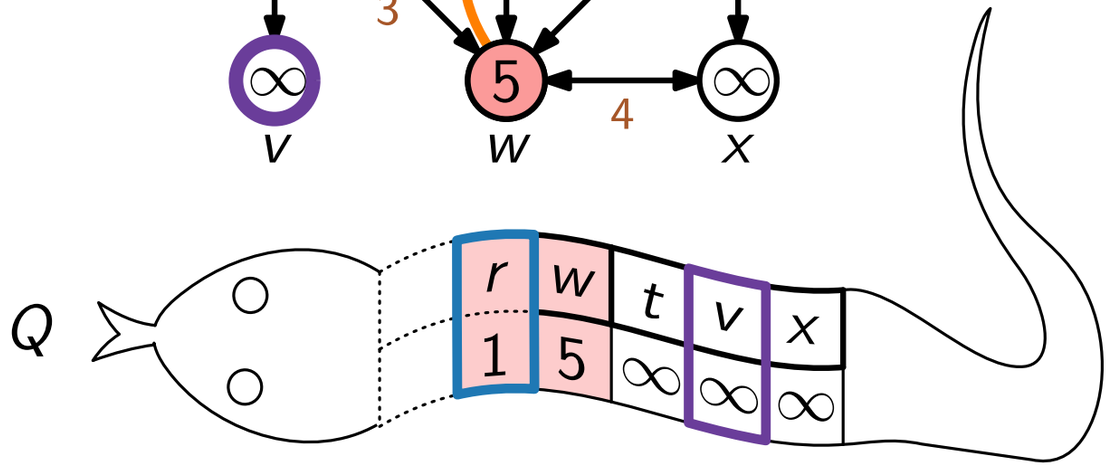



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

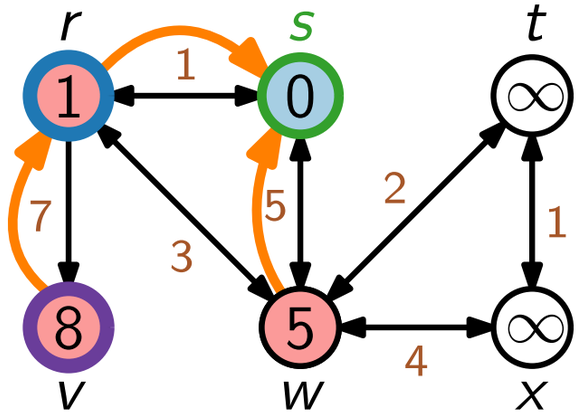


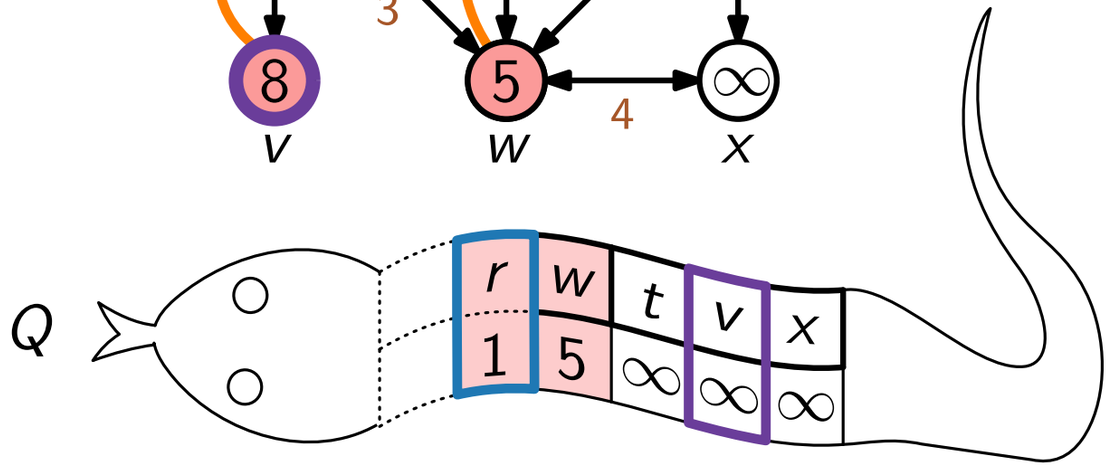



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

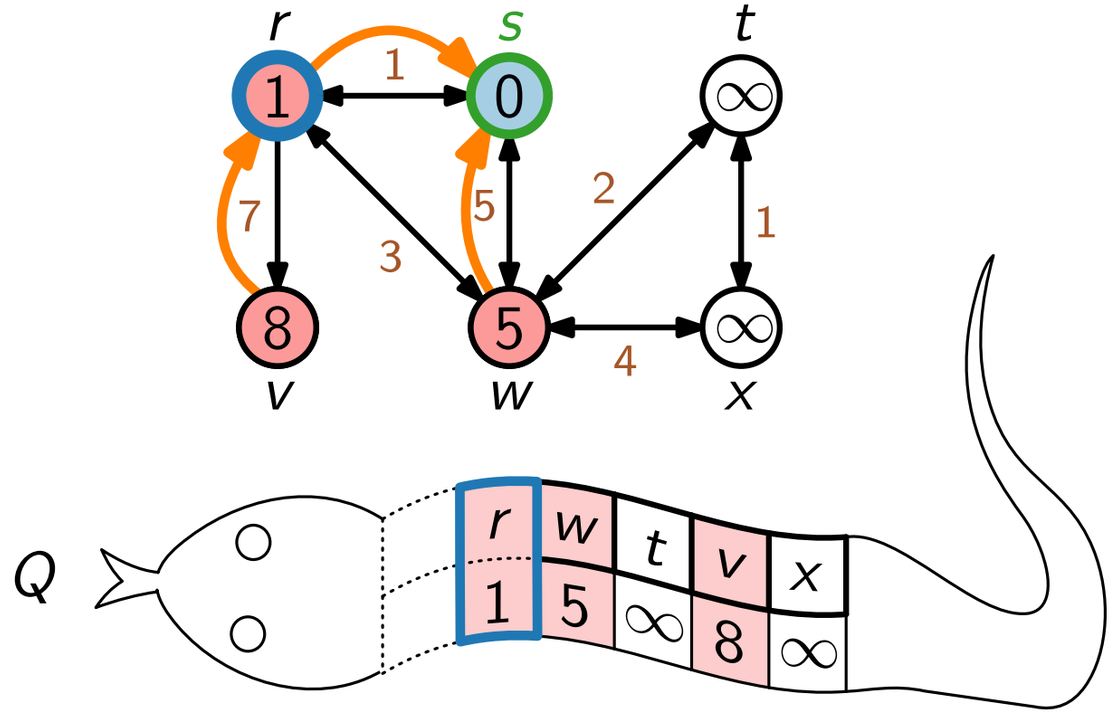





```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

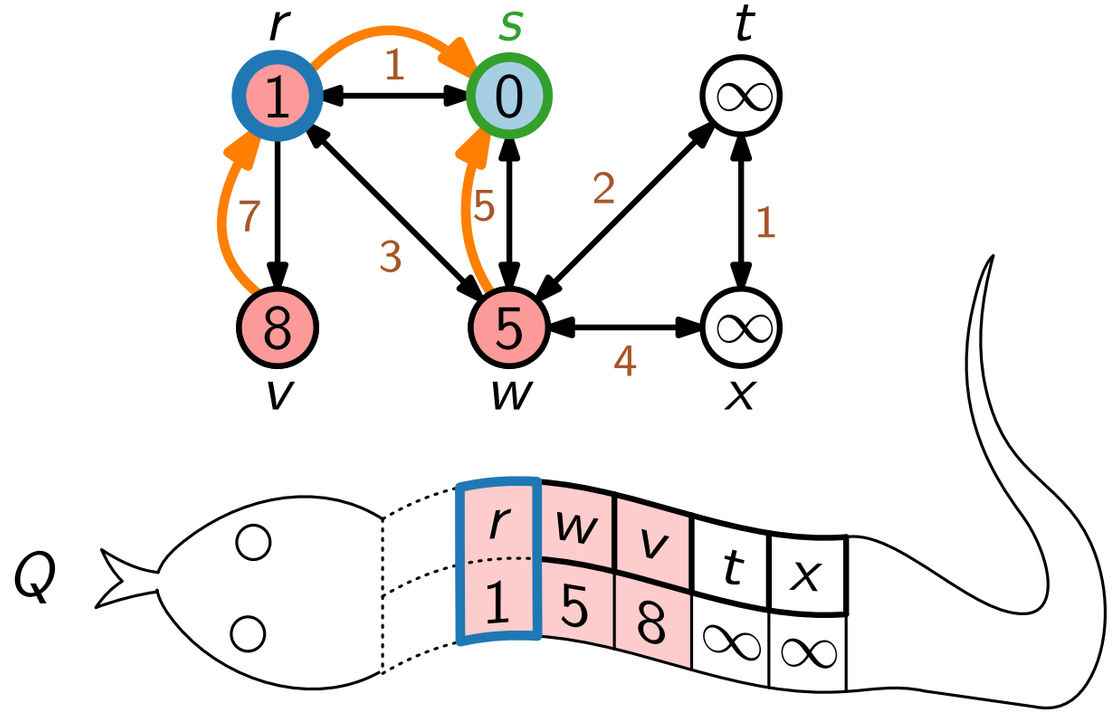
```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```




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

Dijkstra

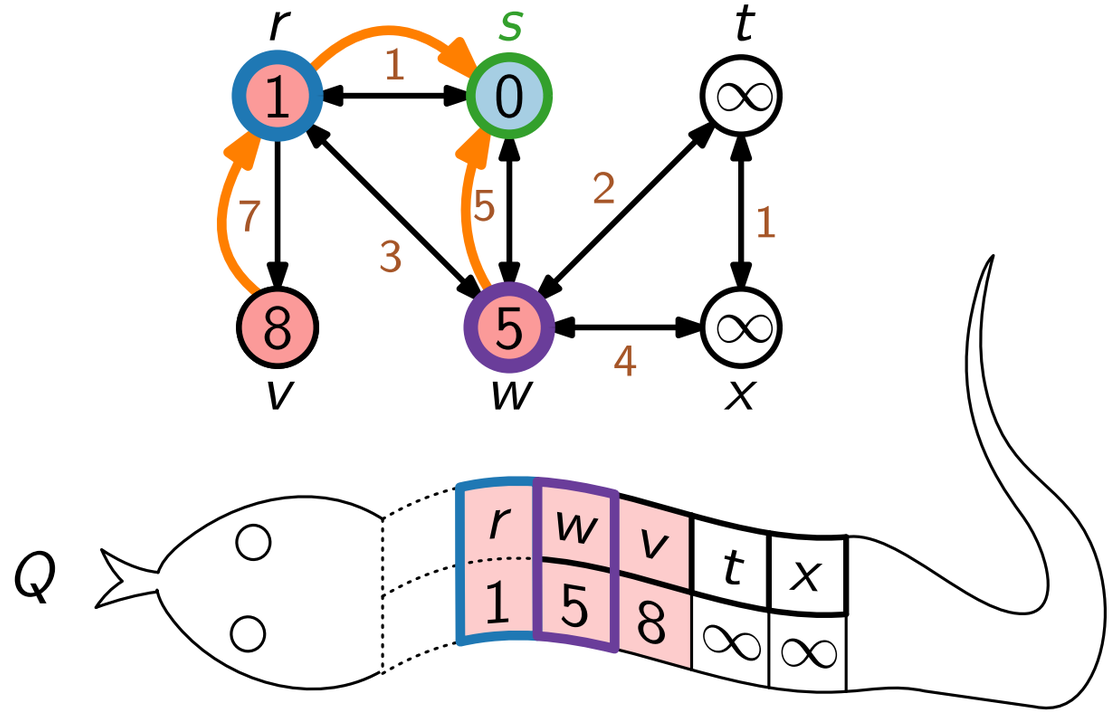
```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```




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

Dijkstra

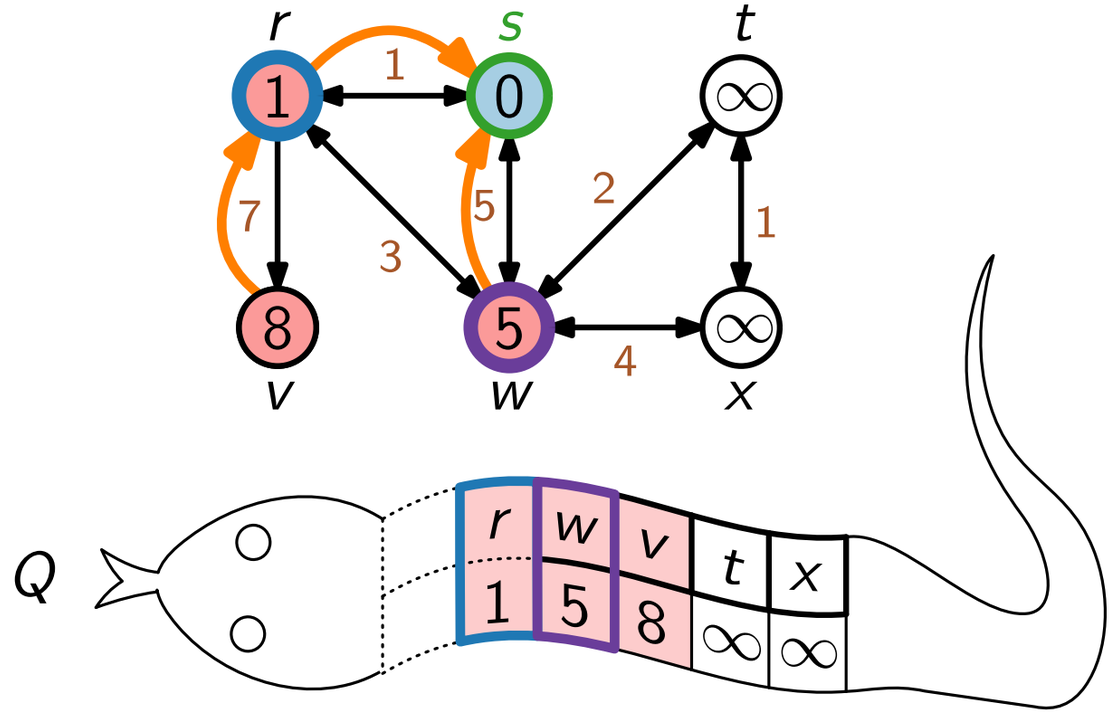
```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.color == white then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```




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

Dijkstra

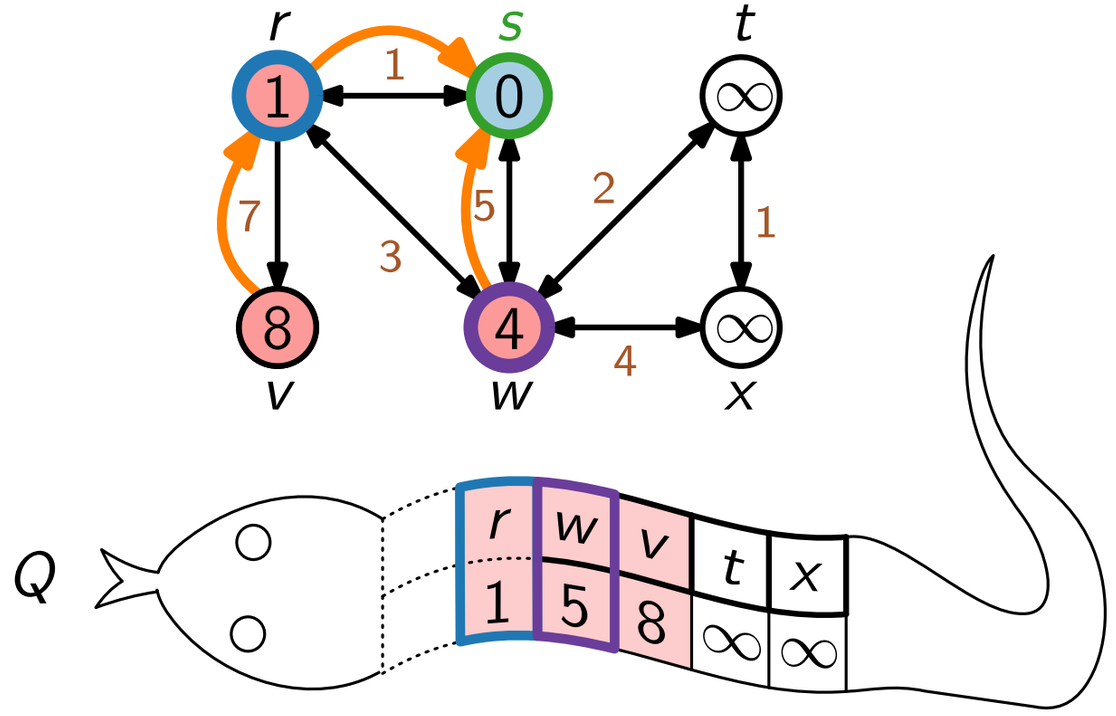
```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```




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

Dijkstra

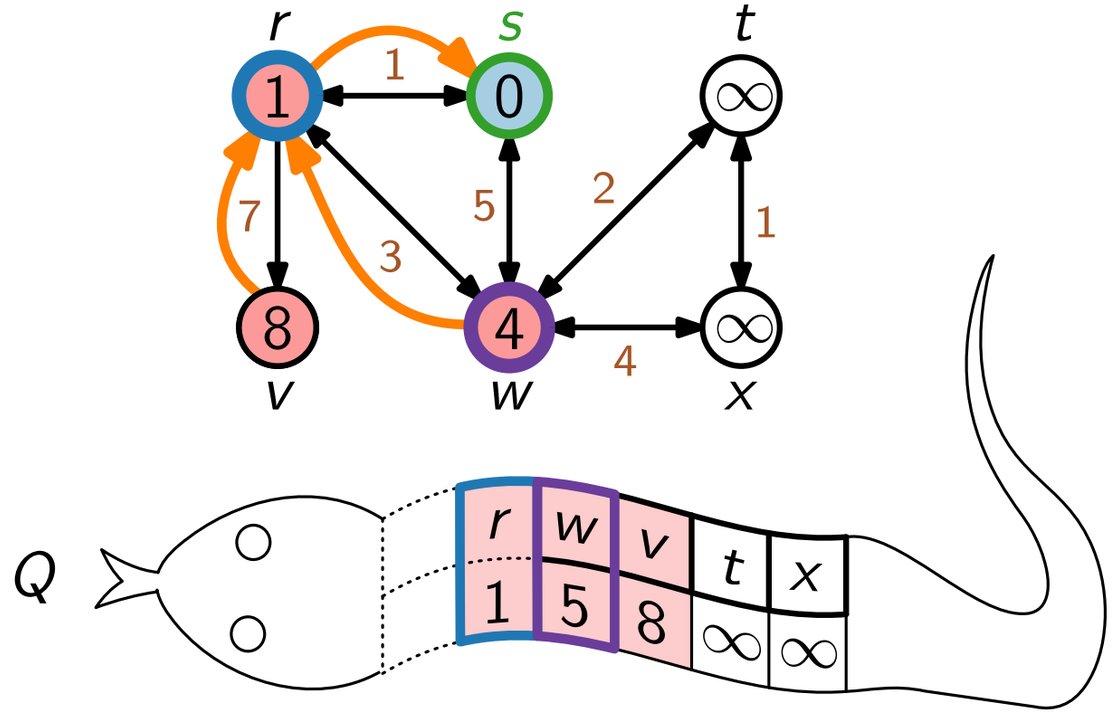
```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```




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

Dijkstra

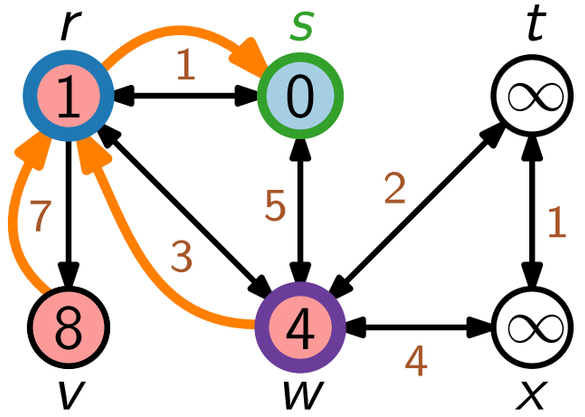
```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

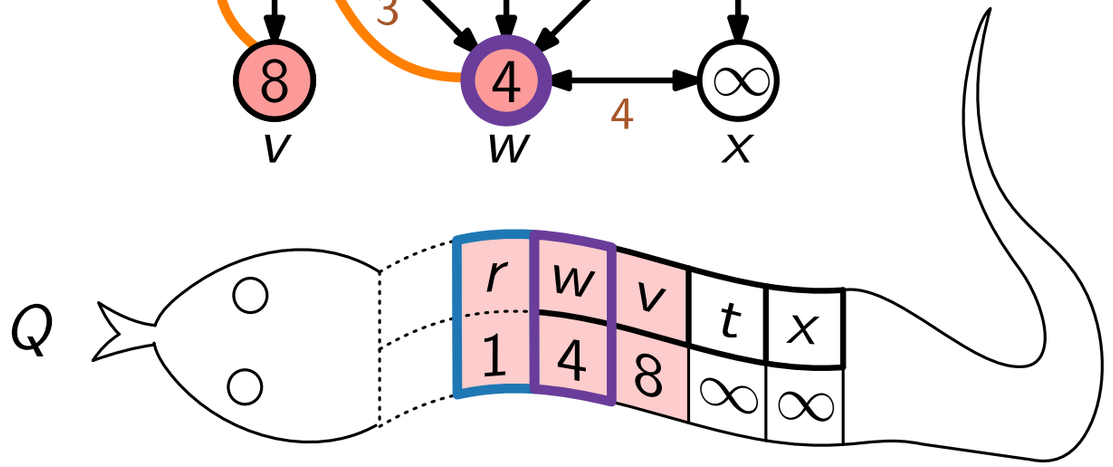



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

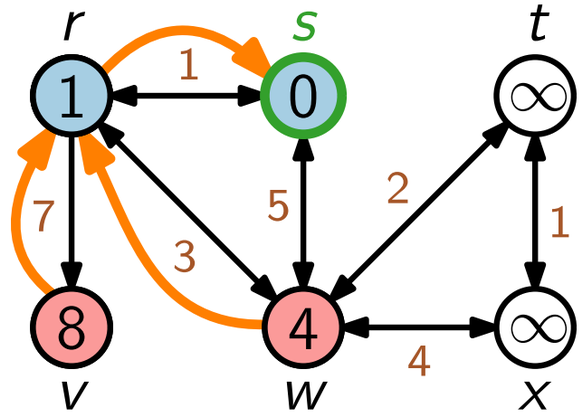


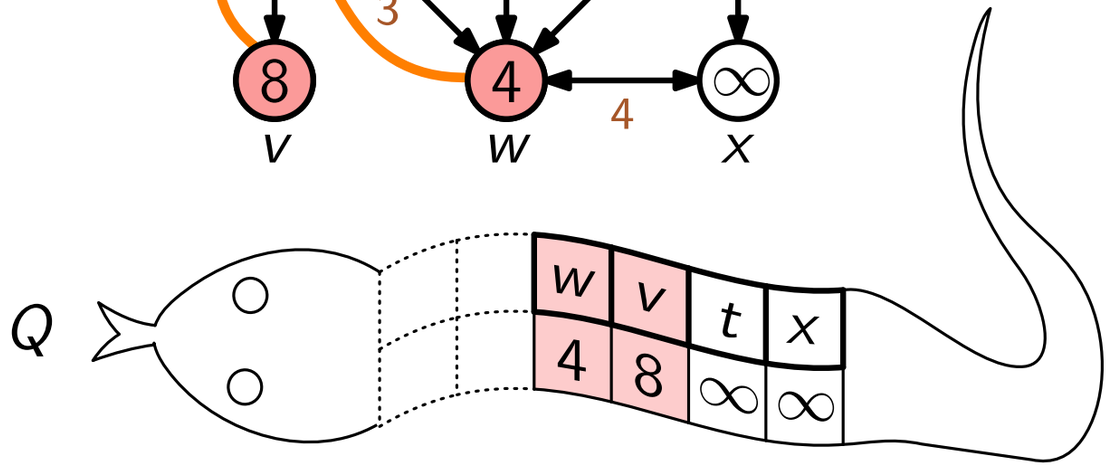



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

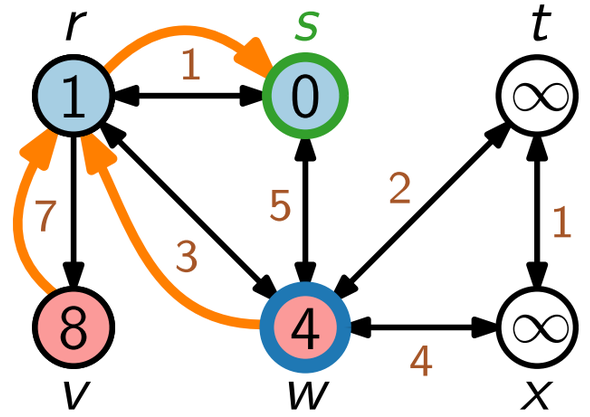


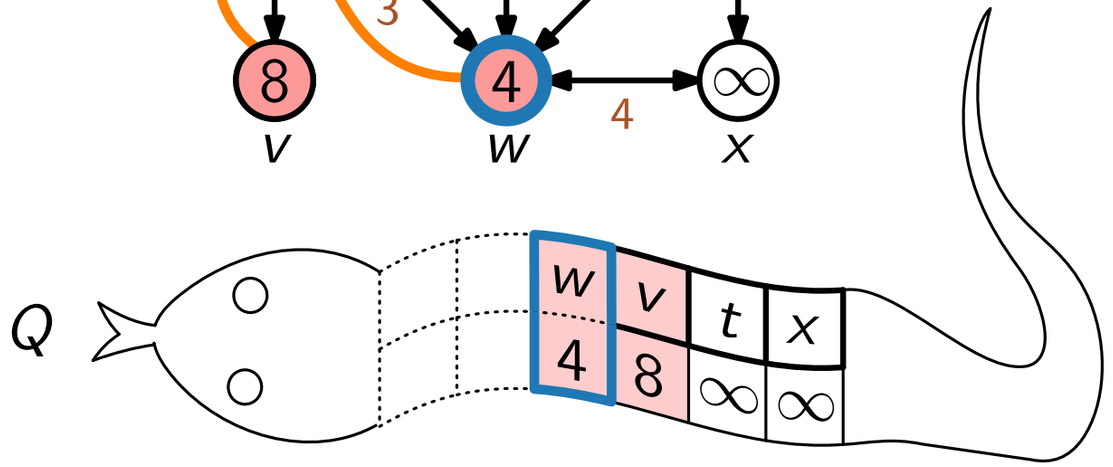



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

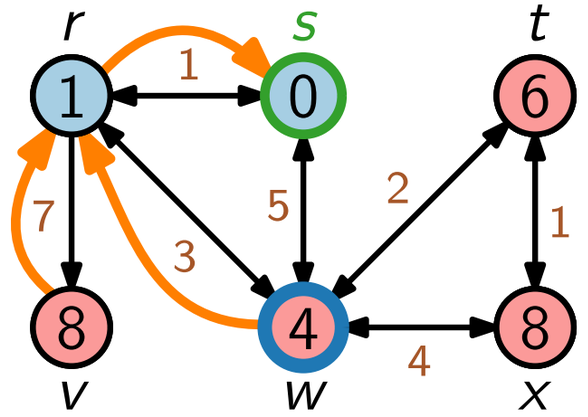


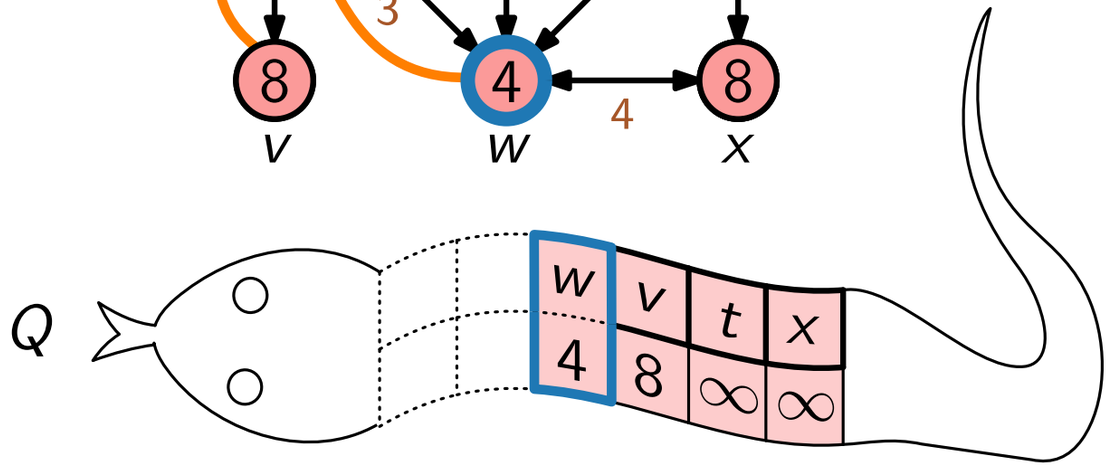



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

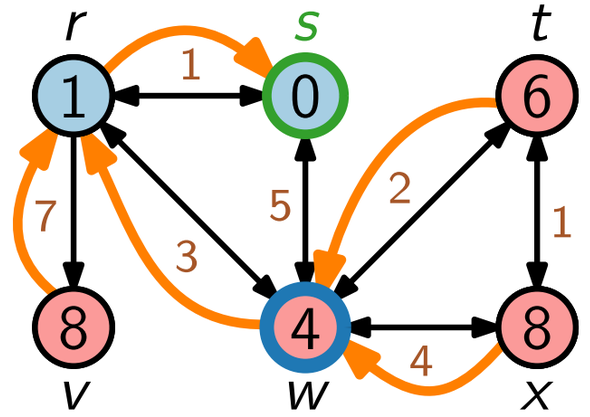


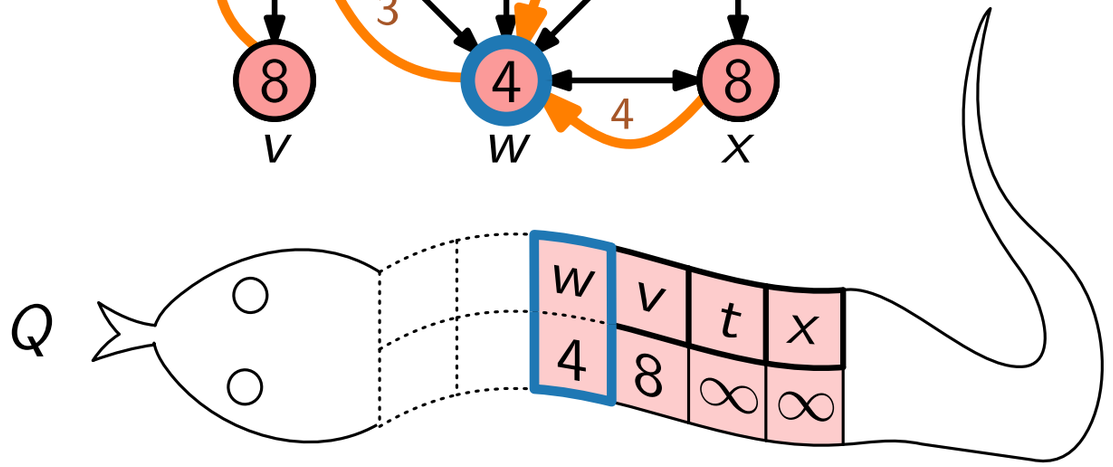



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

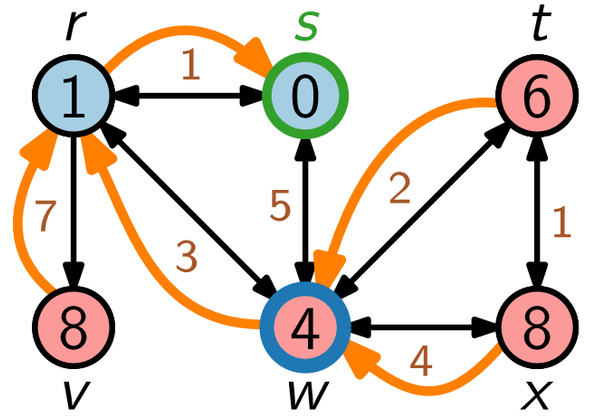


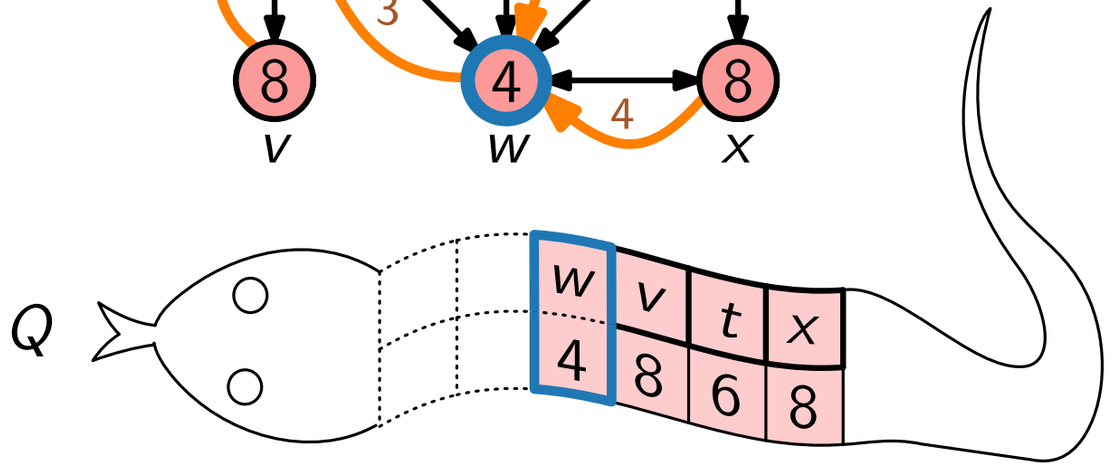



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

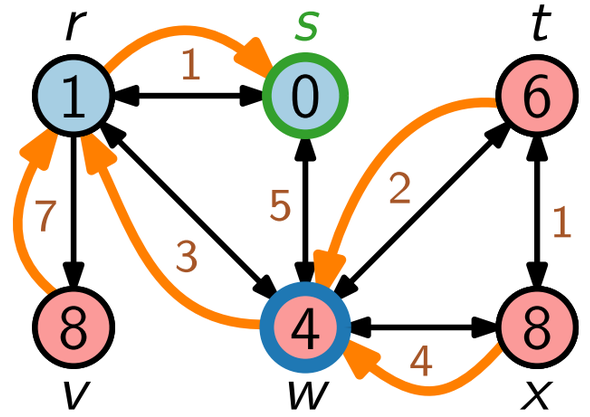


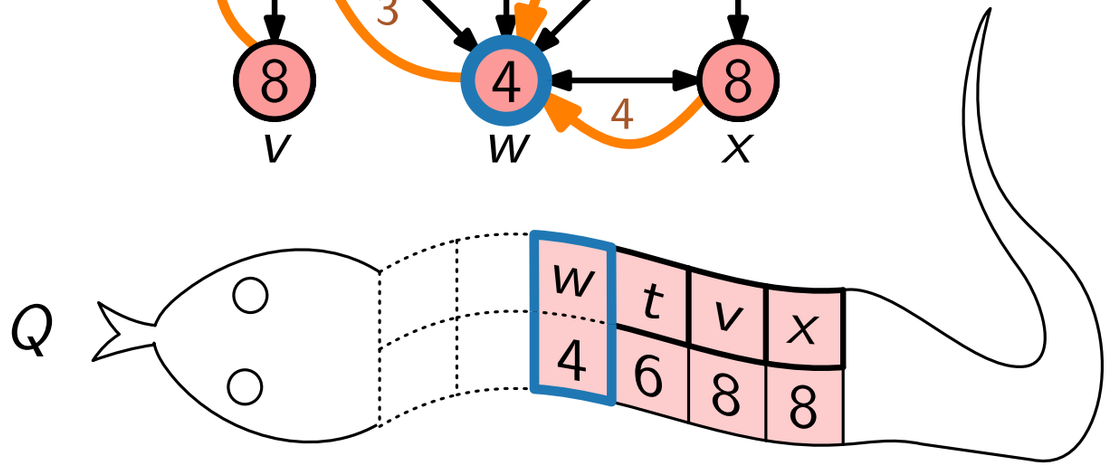



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

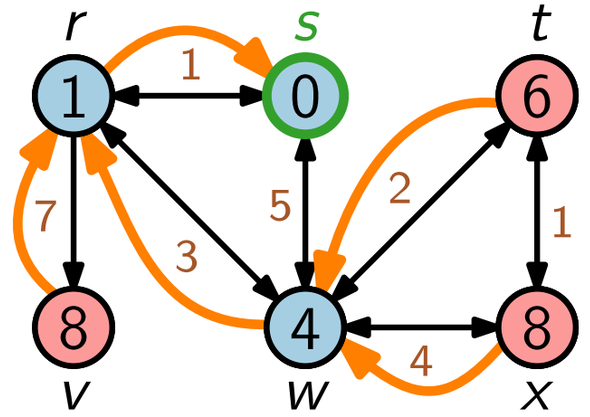


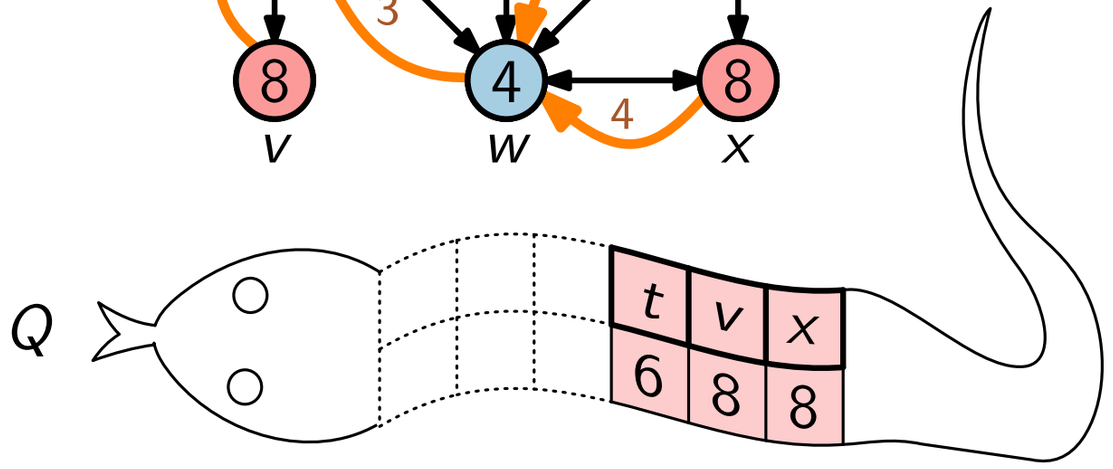



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

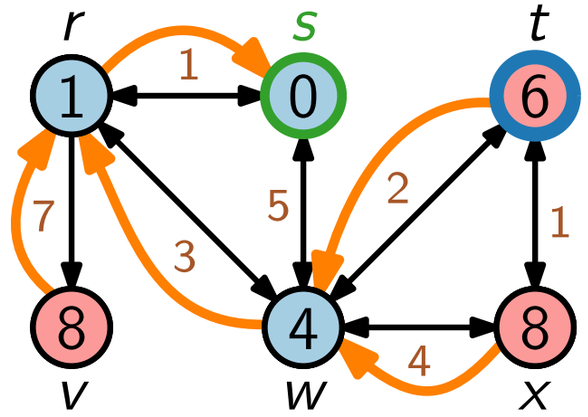


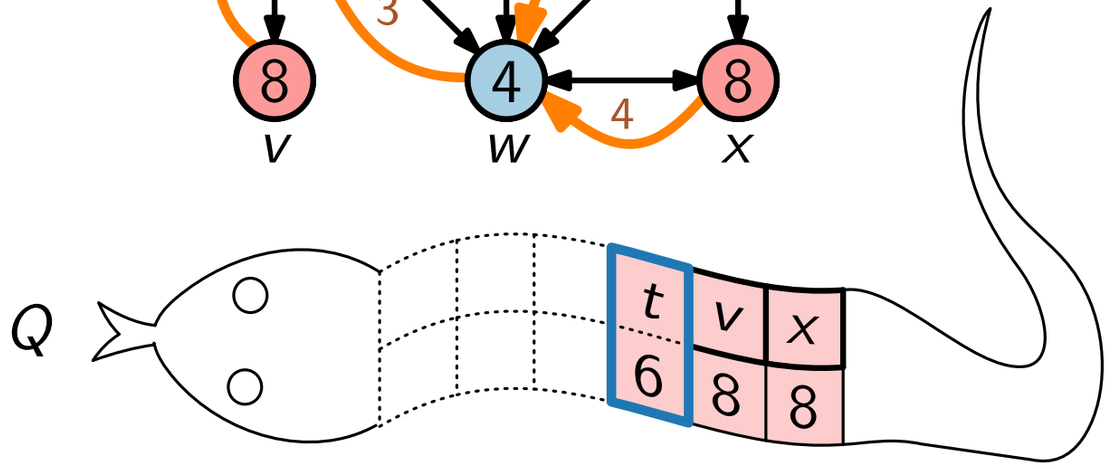



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

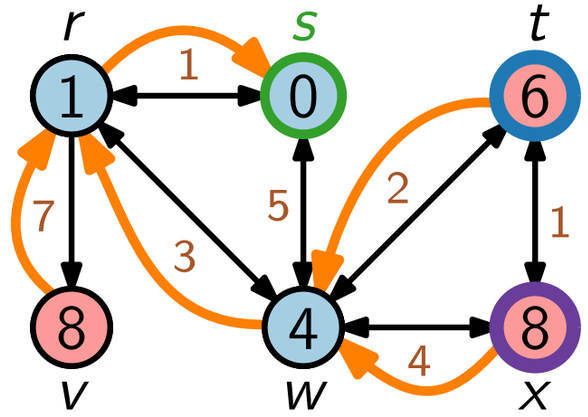


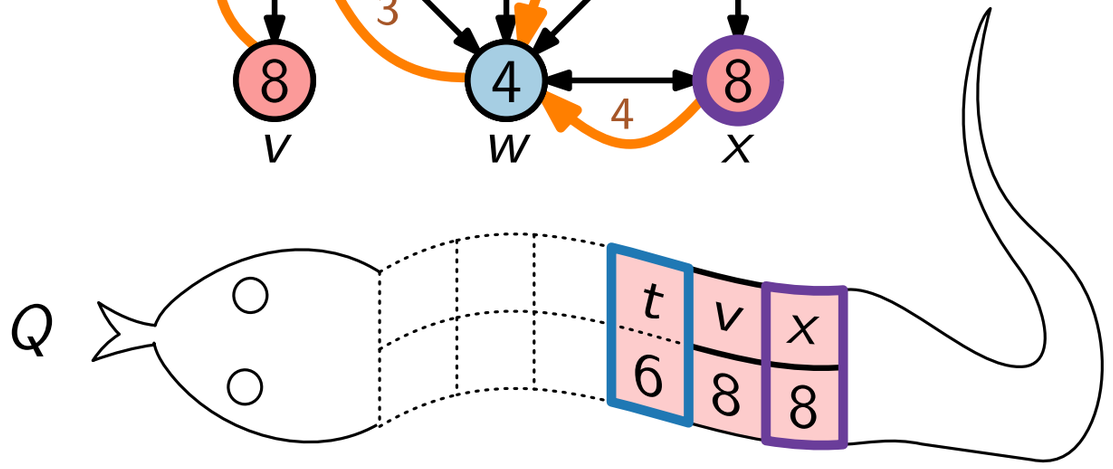



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

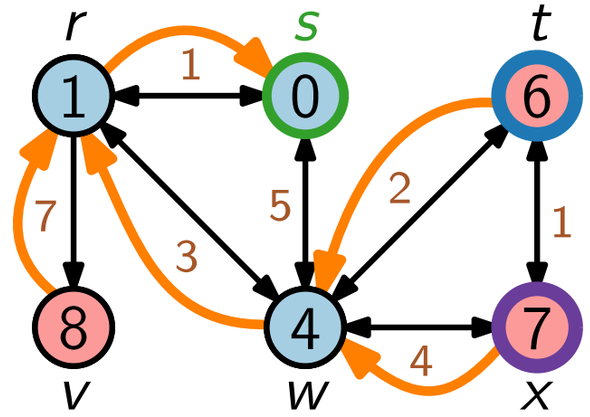
```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

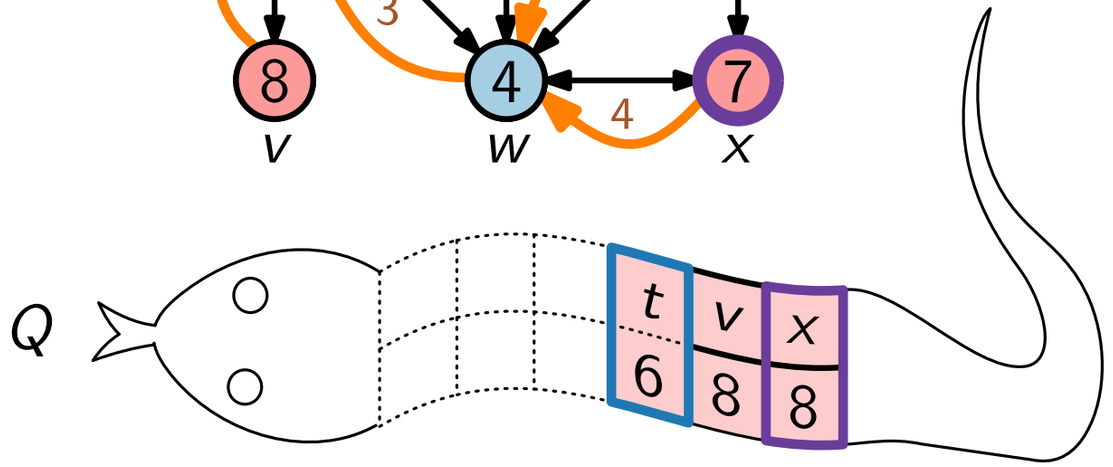





```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra



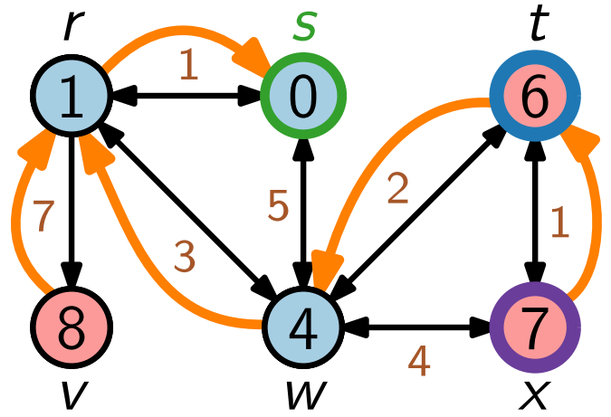


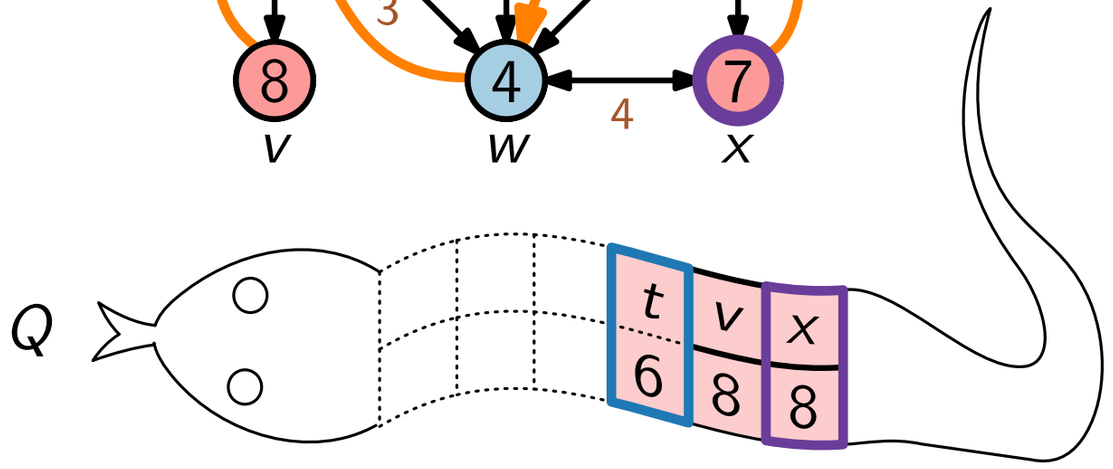

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

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

Dijkstra

```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

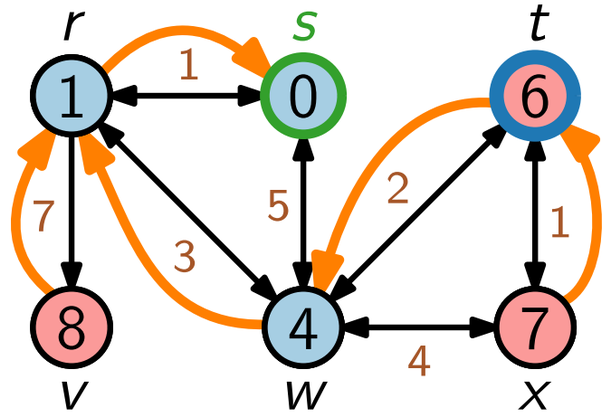


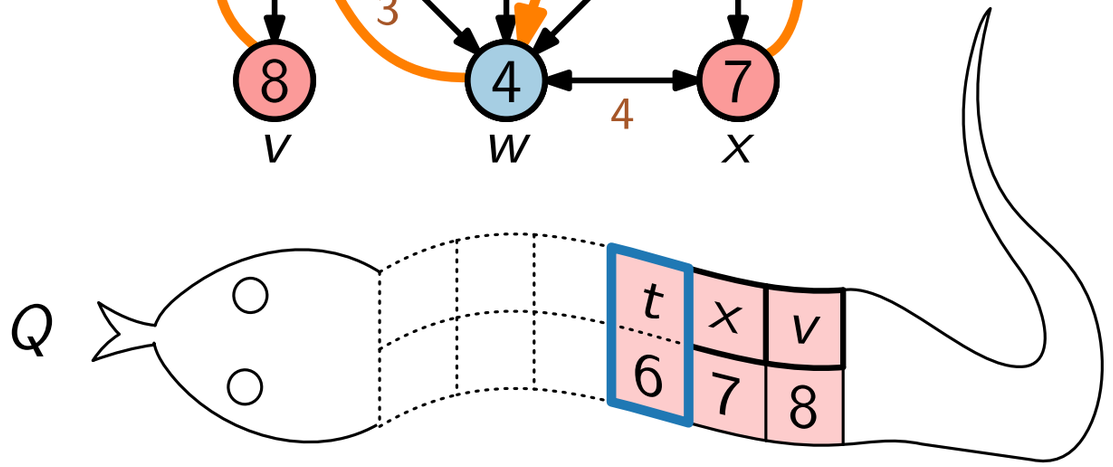



```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

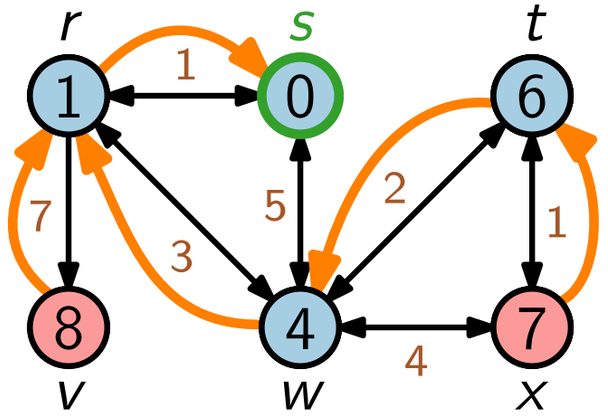
```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

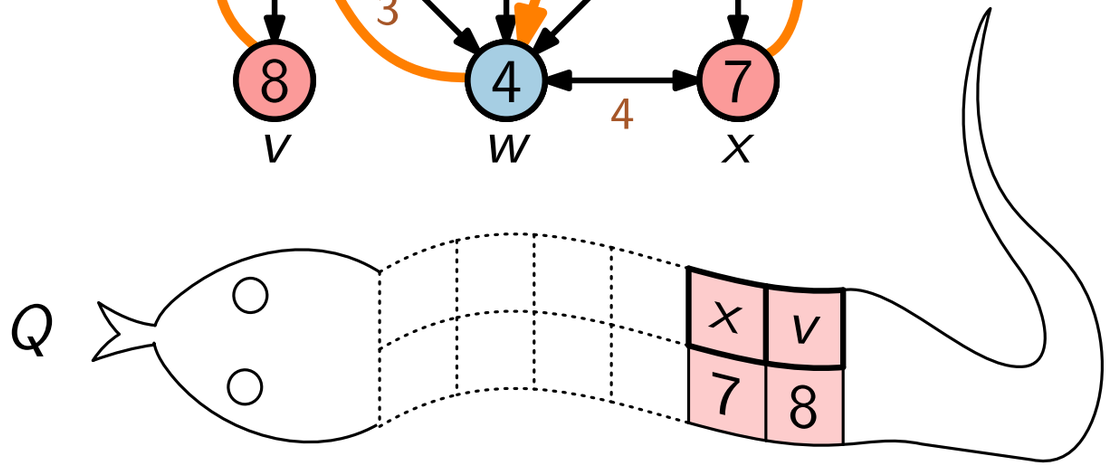





```
Initialize(Graph G, Vertex s)
    foreach u ∈ V do
        u.color = white
        u.d = ∞
        u.π = nil
    s.color = red
    s.d = 0
```

Dijkstra

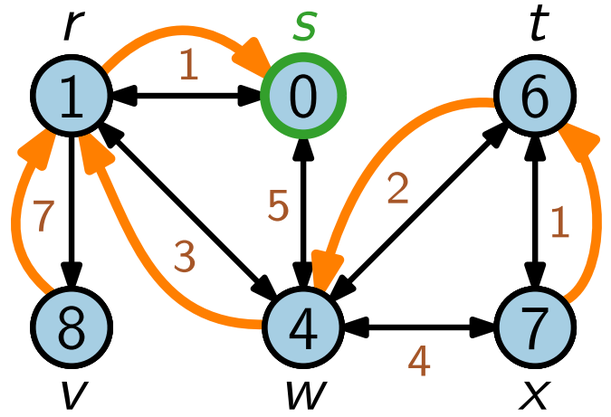


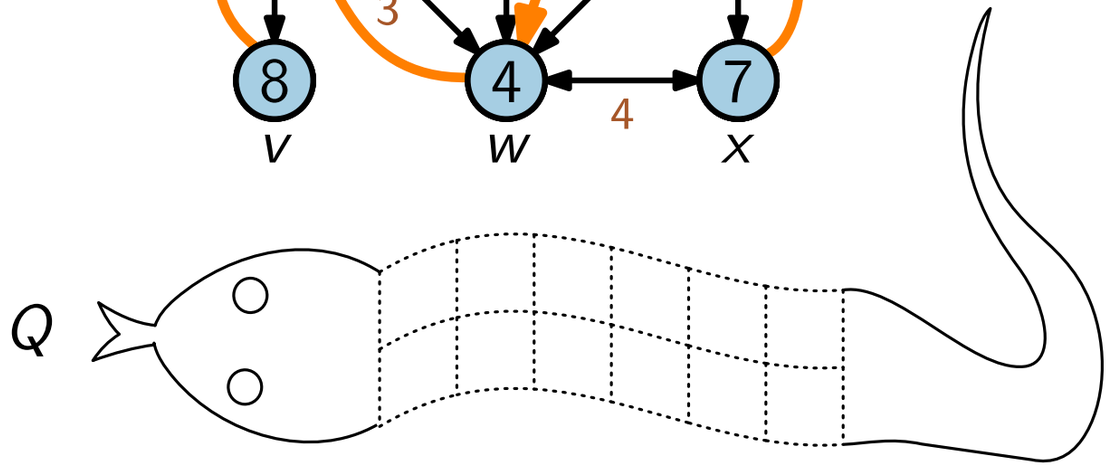



```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

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

Dijkstra

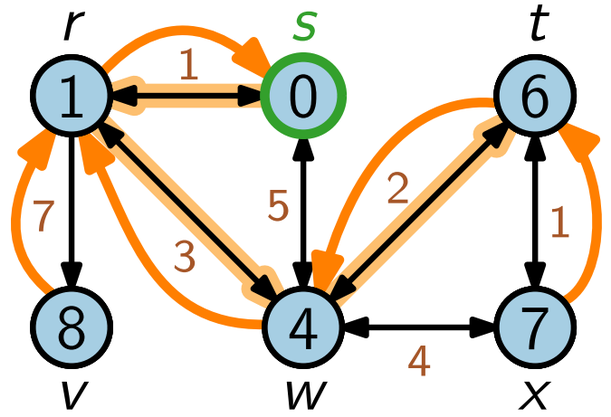


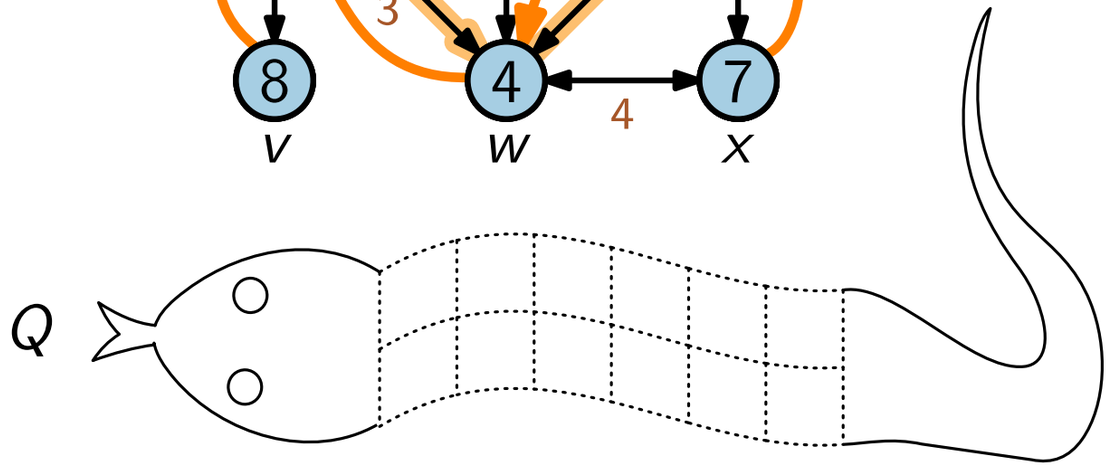



```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

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

Dijkstra

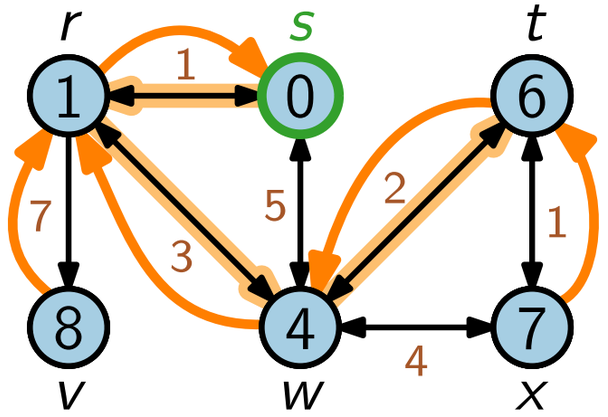


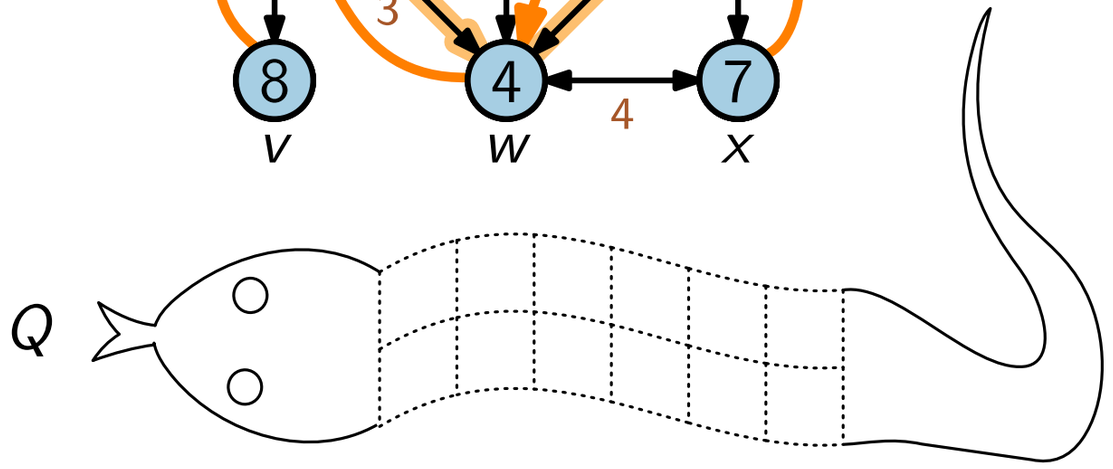



```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

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

Dijkstra

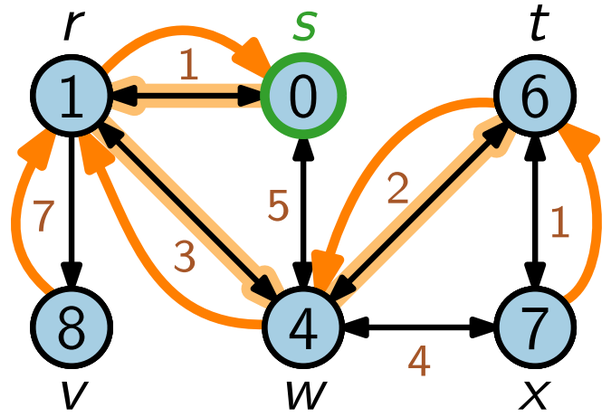


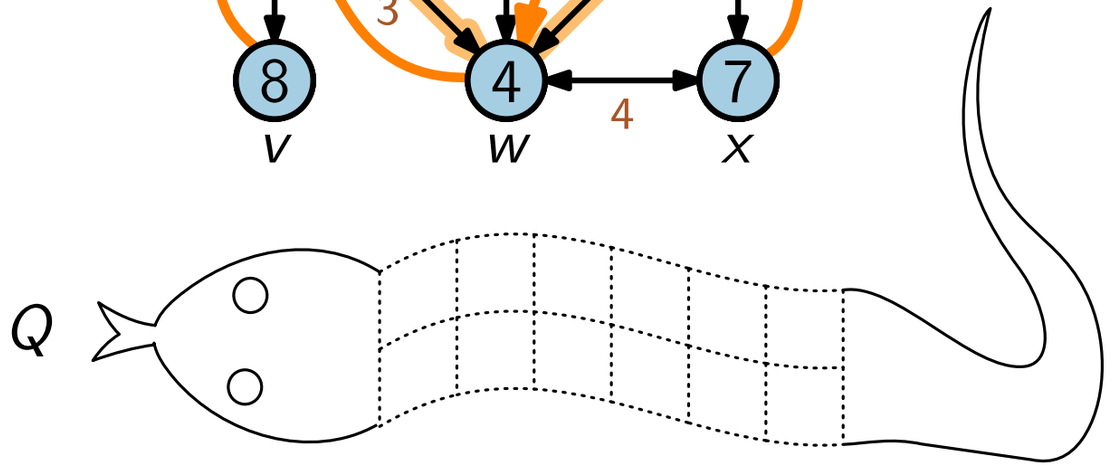



```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

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

Dijkstra






```
Dijkstra(Graph G, Edge weights w, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)

    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

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

Demo. `https://algo.uni-trier.de/demos/graphtraversal.html`

Dijkstra – die Laufzeit

```
Dijkstra(WeightedGraph G, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

Dijkstra – die Laufzeit

```
Dijkstra(WeightedGraph G, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

Dijkstra – die Laufzeit

```
Dijkstra(WeightedGraph G, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

$\mathcal{O}(\quad)$ Zeit

Dijkstra – die Laufzeit

```
Dijkstra(WeightedGraph G, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

$\mathcal{O}(V)$ Zeit

Dijkstra – die Laufzeit

```
Dijkstra(WeightedGraph G, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

Abk. für $O(|V(G)|)$

$\mathcal{O}(V)$ Zeit

Dijkstra – die Laufzeit

```
Dijkstra(WeightedGraph G, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

Abk. für $O(|V(G)|)$

$\mathcal{O}(V)$ Zeit

Dijkstra – die Laufzeit

```
Dijkstra(WeightedGraph G, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

Abk. für $O(|V(G)|)$

$\mathcal{O}(V)$ Zeit

genau ▢ mal

Dijkstra – die Laufzeit

```
Dijkstra(WeightedGraph G, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

Abk. für $O(|V(G)|)$

$\mathcal{O}(V)$ Zeit

genau $|V|$ mal

Dijkstra – die Laufzeit

```
Dijkstra(WeightedGraph G, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

Abk. für $O(|V(G)|)$

$\mathcal{O}(V)$ Zeit

genau $|V|$ mal

Wie oft wird der Schleifeninhalt aufgerufen?

Dijkstra – die Laufzeit

```
Dijkstra(WeightedGraph G, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

Abk. für $O(|V(G)|)$

$\mathcal{O}(V)$ Zeit

genau $|V|$ mal

Wie oft wird der Schleifeninhalt aufgerufen?

Für jeden Knoten u von G genau $|\text{Adj}[u]|$ mal,

Dijkstra – die Laufzeit

```
Dijkstra(WeightedGraph G, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

Abk. für $O(|V(G)|)$

$\mathcal{O}(V)$ Zeit

genau $|V|$ mal

Wie oft wird der Schleifeninhalt aufgerufen?

Für jeden Knoten u von G genau $|\text{Adj}[u]| = \deg^{(\text{out-})}(u)$ mal,

Dijkstra – die Laufzeit

```
Dijkstra(WeightedGraph G, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

Abk. für $O(|V(G)|)$

$\mathcal{O}(V)$ Zeit

genau $|V|$ mal

Wie oft wird der Schleifeninhalt aufgerufen?

Für jeden Knoten u von G genau $|\text{Adj}[u]| = \deg^{(\text{out-})}(u)$ mal,

also insg. $\Theta(E)$ mal.

Dijkstra – die Laufzeit

```
Dijkstra(WeightedGraph G, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

Abk. für $O(|V(G)|)$

$\mathcal{O}(V)$ Zeit

genau $|V|$ mal

Wie oft wird der Schleifeninhalt aufgerufen?

Für jeden Knoten u von G genau $|\text{Adj}[u]| = \deg^{(\text{out-})}(u)$ mal,

also insg. $\Theta(E)$ mal.

Also wird DecreaseKey mal aufgerufen.

Dijkstra – die Laufzeit

```
Dijkstra(WeightedGraph G, Vertex s)
    Initialize(G, s)
    Q = new PriorityQueue(V(G), d)
    while not Q.Empty() do
        u = Q.ExtractMin()
        foreach v ∈ Adj[u] do
            if v.d > u.d + w(u, v) then
                v.color = red
                v.d = u.d + w(u, v)
                v.π = u
                Q.DecreaseKey(v, v.d)
        u.color = blue
```

Abk. für $O(|V(G)|)$

$\mathcal{O}(V)$ Zeit

genau $|V|$ mal

Wie oft wird der Schleifeninhalt aufgerufen?

Für jeden Knoten u von G genau $|\text{Adj}[u]| = \deg^{(\text{out-})}(u)$ mal,

also insg. $\Theta(E)$ mal.

Also wird DecreaseKey $\mathcal{O}(E)$ mal aufgerufen.

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\|V\|, \|E\|)$

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

n = (max.) Anzahl der Elemente in der PriorityQueue

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\|V\|, \|E\|)$

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\|V\|, \|E\|)$

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\|V\|, \|E\|)$
als unsortiertes Feld			

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\|V\|, \|E\|)$
als unsortiertes Feld	$\mathcal{O}(n)$		

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\|V\|, \|E\|)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^\star$	

*) Das geht, weil

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\|V\|, \|E\|)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^\star$	

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz ($\rightarrow$ Direktzugriff)

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\|V\|, \|E\|)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^\star$	$\mathcal{O}(V^2 + E)$

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz ($\rightarrow$ Direktzugriff)

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\|V\|, \|E\|)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^\star$	$\mathcal{O}(V^2 + E)$
als Heap			

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz ($\rightarrow$ Direktzugriff)

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\|V\|, \|E\|)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^\star$	$\mathcal{O}(V^2 + E)$
als Heap	$\mathcal{O}(\log n)$		

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz ($\rightarrow$ Direktzugriff)

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\lvert V\rvert, \lvert E\rvert)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^{\star}$	$\mathcal{O}(V^2 + E)$
als Heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$ **	

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz ($\rightarrow$ Direktzugriff)

**) Das geht, obwohl

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\|V\|, \|E\|)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^{\star}$	$\mathcal{O}(V^2 + E)$
als Heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$ **	

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz ($\rightarrow$ Direktzugriff)

**) Das geht, obwohl wir im Heap nicht suchen können (!). Wir merken uns ständig für jeden Knoten, wo er im Heap steht.

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\|V\|, \|E\|)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^{\star}$	$\mathcal{O}(V^2 + E)$
als Heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)^{\star\star}$	$\mathcal{O}((E + V) \log V)$

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz ($\rightarrow$ Direktzugriff)

**) Das geht, obwohl wir im Heap nicht suchen können (!). Wir merken uns ständig für jeden Knoten, wo er im Heap steht.

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\|V\|, \|E\|)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^{\star}$	$\mathcal{O}(V^2 + E)$
als Heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)^{\star\star}$	$\mathcal{O}((E + V) \log V)$
als FibonacciHeap			

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz ($\rightarrow$ Direktzugriff)

**) Das geht, obwohl wir im Heap nicht suchen können (!). Wir merken uns ständig für jeden Knoten, wo er im Heap steht.

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\lvert V\rvert, \lvert E\rvert)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^{\star}$	$\mathcal{O}(V^2 + E)$
als Heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)^{\star\star}$	$\mathcal{O}((E + V) \log V)$
als FibonacciHeap	$\mathcal{O}(\log n)$ amortisiert		

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz ($\rightarrow$ Direktzugriff)

**) Das geht, obwohl wir im Heap nicht suchen können (!). Wir merken uns ständig für jeden Knoten, wo er im Heap steht.

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G = (V, E)$, läuft Dijkstras Alg. in $\Theta(|V| \cdot T_{\text{ExtractMin}}(|V|) + |E| \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\|V\|, \|E\|)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^{\star}$	$\mathcal{O}(V^2 + E)$
als Heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)^{\star\star}$	$\mathcal{O}((E + V) \log V)$
als FibonacciHeap	$\mathcal{O}(\log n)$ amortisiert	$\mathcal{O}(1)$ amortisiert	

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz ($\rightarrow$ Direktzugriff)

**) Das geht, obwohl wir im Heap nicht suchen können (!). Wir merken uns ständig für jeden Knoten, wo er im Heap steht.

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G=(V,E)$, läuft Dijkstras Alg. in $\Theta(|V|\cdot T_{\text{ExtractMin}}(|V|) + |E|\cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\lvert V\rvert,\lvert E\rvert)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^{\star}$	$\mathcal{O}(V^2+E)$
als Heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)^{\star\star}$	$\mathcal{O}((E+V)\log V)$
als FibonacciHeap	$\mathcal{O}(\log n)$ amortisiert	$\mathcal{O}(1)$ amortisiert	$\mathcal{O}(E+V\log V)$ im Worst-Case!

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz ($\rightarrow$ Direktzugriff)

**) Das geht, obwohl wir im Heap nicht suchen können (!). Wir merken uns ständig für jeden Knoten, wo er im Heap steht.

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G=(V,E)$, läuft Dijkstras Alg. in $\Theta(|V|\cdot T_{\text{ExtractMin}}(|V|) + |E|\cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\lvert V\rvert,\lvert E\rvert)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^{\star}$	$\mathcal{O}(V^2+E)$
als Heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)^{\star\star}$	$\mathcal{O}((E+V)\log V)$
als FibonacciHeap	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(E+V\log V)$

siehe Master-Vorlesung **Fortgeschrittene Algorithmen**e!

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz ($\rightarrow$ Direktzugriff)

**) Das geht, obwohl wir im Heap nicht suchen können (!). Wir merken uns ständig für jeden Knoten, wo er im Heap steht.

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G=(V,E)$, läuft Dijkstras Alg. in $\Theta(|V|\cdot T_{\text{ExtractMin}}(|V|) + |E|\cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\lvert V\rvert,\lvert E\rvert)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^{\star}$	$\mathcal{O}(V^2+E)$
als Heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)^{\star\star}$	$\mathcal{O}((E+V)\log V)$
als FibonacciHeap	$\mathcal{O}(\log n)$ amortisiert	$\mathcal{O}(1)$ amortisiert	$\mathcal{O}(E+V\log V)$ im Worst-Case!

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz ($\rightarrow$ Direktzugriff)

**) Das geht, obwohl wir im Heap nicht suchen können (!). Wir merken uns ständig für jeden Knoten, wo er im Heap steht.

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G=(V,E)$, läuft Dijkstras Alg. in $\Theta(|V|\cdot T_{\text{ExtractMin}}(|V|) + |E|\cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\lvert V\rvert,\lvert E\rvert)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^{\star}$	$\mathcal{O}(V^2+E)$
als Heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)^{\star\star}$	$\mathcal{O}((E+V)\log V)$
als FibonacciHeap	$\mathcal{O}(\log n)$ amortisiert	$\mathcal{O}(1)$ amortisiert	$\mathcal{O}(E+V\log V)$ im Worst-Case!

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz ($\rightarrow$ Direktzugriff)
**) Das geht, obwohl wir im Heap nicht suchen können (!). Wir merken uns ständig für jeden Knoten, wo er im Heap steht.

Korollar. In einem Graphen $G=(V,E;w)$ mit $w\colon E\to\mathbb{Q}_{\geq 0}$ kann man in $\mathcal{O}(E+V\log V)$ Zeit die kürzesten Wege von einem zu allen Knoten berechnen (SSSP-Problem).

Dijkstra – die Laufzeit

Satz. Gegeben ein Graph $G=(V,E)$, läuft Dijkstras Alg. in $\Theta(|V|\cdot T_{\text{ExtractMin}}(|V|) + |E|\cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\lvert V\rvert,\lvert E\rvert)$
als unsortiertes Feld	$\mathcal{O}(n)$	$\mathcal{O}(1)^{\star}$	$\mathcal{O}(V^2+E)$
als Heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)^{\star\star}$	$\mathcal{O}((E+V)\log V)$
als FibonacciHeap	$\mathcal{O}(\log n)$ amortisiert	$\mathcal{O}(1)$ amortisiert	$\mathcal{O}(E+V\log V)$ im Worst-Case!

*) Das geht, weil wir bei ExtractMin Lücken im Feld lassen; daher bleiben die Schlüssel an ihrem Platz ($\rightarrow$ Direktzugriff)
**) Das geht, obwohl wir im Heap nicht suchen können (!). Wir merken uns ständig für jeden Knoten, wo er im Heap steht.

Korollar. In einem Graphen $G=(V,E;w)$ mit $w\colon E\to\mathbb{Q}_{\geq 0}$ kann man in $\mathcal{O}(E+V\log V)$ Zeit die kürzesten Wege von einem zu allen Knoten berechnen (SSSP-Problem).

Dijkstra – die Korrektheit

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Annahme: Es gibt eine Nummerierung $v_1, \ldots, v_n$ der Knotenmenge von G, so dass
$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Annahme: Es gibt eine Nummerierung $v_1, \ldots, v_n$ der Knotenmenge von G, so dass
$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

Im Allgemeinen gilt nur $\leq$

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Annahme: Es gibt eine Nummerierung $v_1, \ldots, v_n$ der Knotenmenge von G, so dass
$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

0. Schleifeninvariante

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Annahme: Es gibt eine Nummerierung $v_1, \ldots, v_n$ der Knotenmenge von G, so dass
$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

0. Schleifeninvariante

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Annahme: Es gibt eine Nummerierung $v_1, \ldots, v_n$ der Knotenmenge von G, so dass
$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Annahme: Es gibt eine Nummerierung $v_1, \ldots, v_n$ der Knotenmenge von G, so dass
$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Annahme: Es gibt eine Nummerierung $v_1, \ldots, v_n$ der Knotenmenge von G, so dass
$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Annahme: Es gibt eine Nummerierung $v_1, \ldots, v_n$ der Knotenmenge von G, so dass
$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Annahme: Es gibt eine Nummerierung $v_1, \ldots, v_n$ der Knotenmenge von G, so dass
$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

- **I)** gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$
- **II)** enthält Q genau die Knoten $v_k, \ldots, v_n$
- **III)** gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ($k = 1$)

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Annahme: Es gibt eine Nummerierung $v_1, \ldots, v_n$ der Knotenmenge von G, so dass
$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

- **I)** gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$
- **II)** enthält Q genau die Knoten $v_k, \ldots, v_n$
- **III)** gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ($k = 1$)

Offensichtlich ist $v_1 = s$ und $\delta(s, s) = 0$.

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Annahme: Es gibt eine Nummerierung $v_1, \ldots, v_n$ der Knotenmenge von G, so dass
$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

- **I)** gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$
- **II)** enthält Q genau die Knoten $v_k, \ldots, v_n$
- **III)** gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ($k = 1$)

Offensichtlich ist $v_1 = s$ und $\delta(s, s) = 0$.

In Initialize(G, s) wird $s.d = 0$ gesetzt $\Rightarrow$ **I)**

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Annahme: Es gibt eine Nummerierung $v_1, \ldots, v_n$ der Knotenmenge von G, so dass
$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ($k = 1$)

Offensichtlich ist $v_1 = s$ und $\delta(s, s) = 0$.

In Initialize(G, s) wird $s.d = 0$ gesetzt $\Rightarrow$ **I)**

Alle Knoten sind in Q $\Rightarrow$ **II)**

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Annahme: Es gibt eine Nummerierung $v_1, \ldots, v_n$ der Knotenmenge von G, so dass
$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ($k = 1$)

Offensichtlich ist $v_1 = s$ und $\delta(s, s) = 0$.

In Initialize(G, s) wird $s.d = 0$ gesetzt $\Rightarrow$ **I)**

Alle Knoten sind in Q $\Rightarrow$ **II)**

Für jeden Knoten $v \neq s$ wird $v.d = \infty$ gesetzt

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Annahme: Es gibt eine Nummerierung $v_1, \ldots, v_n$ der Knotenmenge von G, so dass
$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ($k = 1$)

Offensichtlich ist $v_1 = s$ und $\delta(s, s) = 0$.

In Initialize(G, s) wird $s.d = 0$ gesetzt $\Rightarrow$ **I)**

Alle Knoten sind in Q $\Rightarrow$ **II)**

Für jeden Knoten $v \neq s$ wird $v.d = \infty$ gesetzt $\Rightarrow$ **III)**

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

Dijkstra – die Korrektheit

Definition. Für einen (un)gerichteten Graphen G und $u, v \in V(G)$ sei
$\delta(u, v) :=$ Länge eines kürzesten u-v-Wegs
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Annahme: Es gibt eine Nummerierung $v_1, \ldots, v_n$ der Knotenmenge von G, so dass
$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ($k = 1$) ✓

Offensichtlich ist $v_1 = s$ und $\delta(s, s) = 0$.

In Initialize(G, s) wird $s.d = 0$ gesetzt $\Rightarrow$ **I)**

Alle Knoten sind in Q $\Rightarrow$ **II)**

Für jeden Knoten $v \neq s$ wird $v.d = \infty$ gesetzt $\Rightarrow$ **III)**

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte kürzesten s-v_k-Pfad P.

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

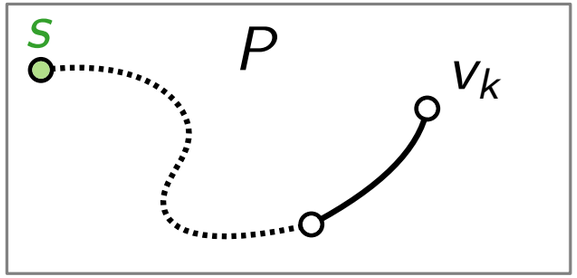
II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte kürzesten s-v_k-Pfad P.




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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

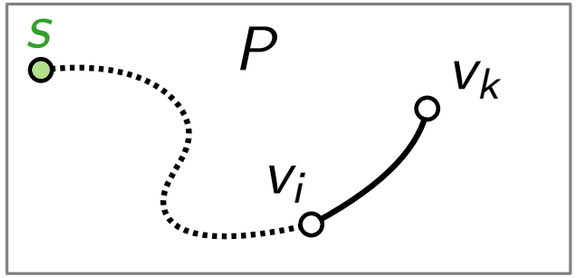
III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte kürzesten s-v_k-Pfad P.

Sei v_i der Vorgänger von v_k auf P.




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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

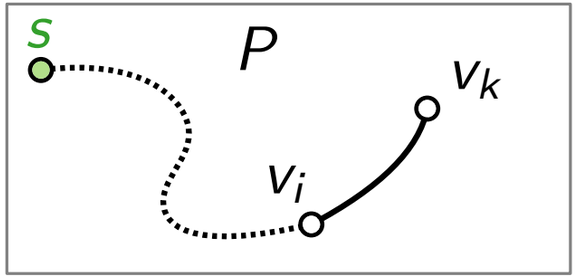
1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte kürzesten s-v_k-Pfad P.

Sei v_i der Vorgänger von v_k auf P.

Dann ist $i < k$.




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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

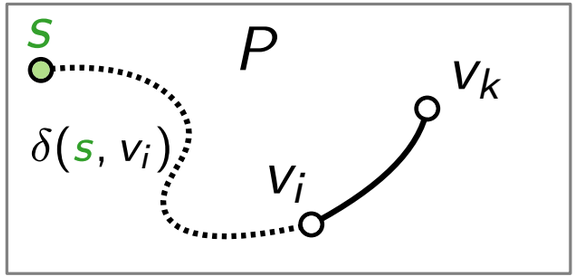
1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte kürzesten s-v_k-Pfad P.

Sei v_i der Vorgänger von v_k auf P.

Dann ist $i < k$.




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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

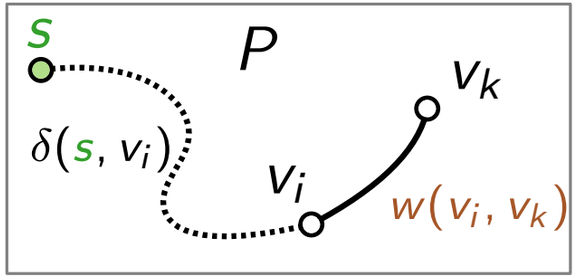
1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte kürzesten s-v_k-Pfad P.

Sei v_i der Vorgänger von v_k auf P.

Dann ist $i < k$.




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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

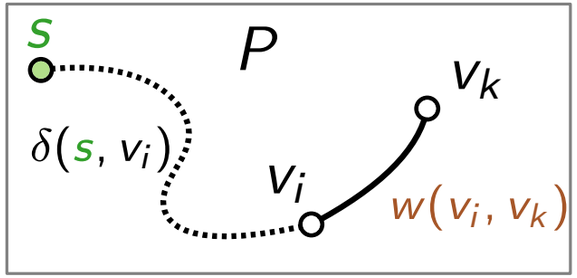
1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte kürzesten s-v_k-Pfad P.

Sei v_i der Vorgänger von v_k auf P.

Dann ist $i < k$. **I)** $\Rightarrow v_i.d = \delta(s, v_i)$




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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

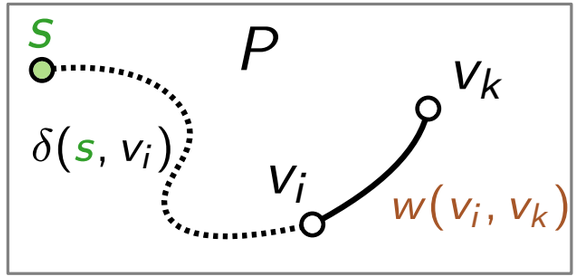
I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung



Betrachte kürzesten s-v_k-Pfad P.

Sei v_i der Vorgänger von v_k auf P.

Dann ist $i < k$. **I)** $\Rightarrow v_i.d = \delta(s, v_i)$

Betrachte i-ten Schleifendurchlauf mit $u = v_i$, $v = v_k$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

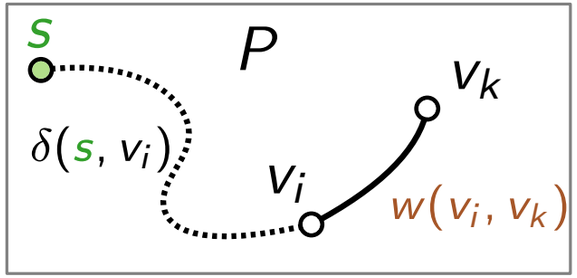
I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung



Betrachte kürzesten s-v_k-Pfad P.

Sei v_i der Vorgänger von v_k auf P.

Dann ist $i < k$. **I)** $\Rightarrow v_i.d = \delta(s, v_i)$

Betrachte i-ten Schleifendurchlauf mit $u = v_i$, $v = v_k$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

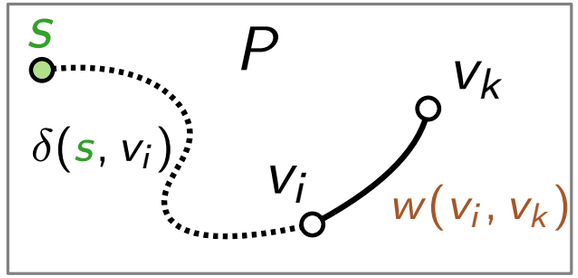
I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung



Betrachte kürzesten s-v_k-Pfad P.

Sei v_i der Vorgänger von v_k auf P.

Dann ist $i < k$. **I)** $\Rightarrow v_i.d = \delta(s, v_i)$

Betrachte i-ten Schleifendurchlauf mit $u = v_i$, $v = v_k$

$\Rightarrow v_k.d = v_i.d + w(v_i, v_k)$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

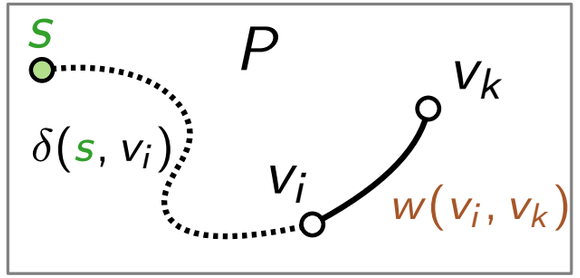
I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung



Betrachte kürzesten s-v_k-Pfad P.

Sei v_i der Vorgänger von v_k auf P.

Dann ist $i < k$. **I)** $\Rightarrow$ $v_i.d = \delta(s, v_i)$

Betrachte i-ten Schleifendurchlauf mit $u = v_i$, $v = v_k$

$\Rightarrow v_k.d = v_i.d + w(v_i, v_k)$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

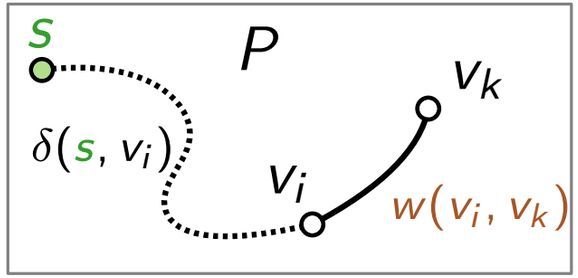
I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung



Betrachte kürzesten s-v_k-Pfad P.

Sei v_i der Vorgänger von v_k auf P.

Dann ist $i < k$. **I)** $\Rightarrow$ $v_i.d = \delta(s, v_i)$

Betrachte i-ten Schleifendurchlauf mit $u = v_i$, $v = v_k$

$\Rightarrow v_k.d = v_i.d + w(v_i, v_k) = \delta(s, v_i) + w(v_i, v_k)$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

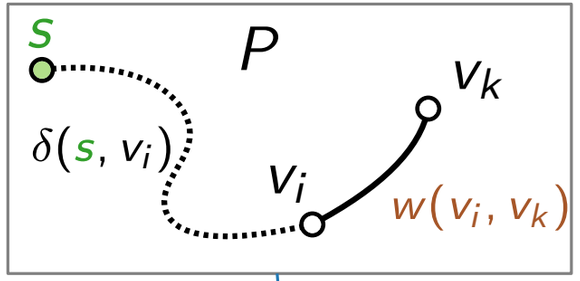
I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung



Betrachte kürzesten s-v_k-Pfad P.

Sei v_i der Vorgänger von v_k auf P.

Dann ist $i < k$. **I)** $\Rightarrow$ $v_i.d = \delta(s, v_i)$

Betrachte i-ten Schleifendurchlauf mit $u = v_i$, $v = v_k$

$$\Rightarrow v_k.d = v_i.d + w(v_i, v_k) = \delta(s, v_i) + w(v_i, v_k) = \delta(s, v_k)$$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

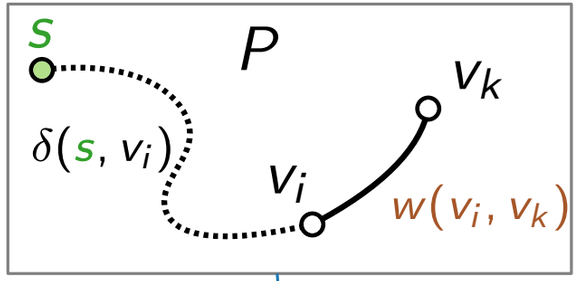
I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung



Betrachte kürzesten s-v_k-Pfad P.

Sei v_i der Vorgänger von v_k auf P.

Dann ist $i < k$. **I)** $\Rightarrow$ $v_i.d = \delta(s, v_i)$

Betrachte i-ten Schleifendurchlauf mit $u = v_i$, $v = v_k$

$$\Rightarrow v_k.d = v_i.d + w(v_i, v_k) = \delta(s, v_i) + w(v_i, v_k) = \delta(s, v_k) \Rightarrow \textbf{I)}$$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

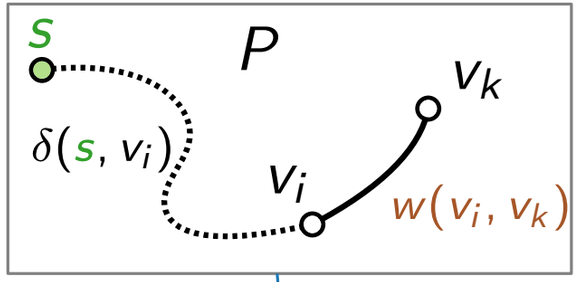
I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung



Betrachte kürzesten s-v_k-Pfad P.

Sei v_i der Vorgänger von v_k auf P.

Dann ist $i < k$. **I)** $\Rightarrow$ $v_i.d = \delta(s, v_i)$

Betrachte i-ten Schleifendurchlauf mit $u = v_i$, $v = v_k$

$\Rightarrow v_k.d = v_i.d + w(v_i, v_k) = \delta(s, v_i) + w(v_i, v_k) = \delta(s, v_k)$ $\Rightarrow$ **I)**

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte $(k-1)$-ten Schleifendurchlauf

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte $(k-1)$-ten Schleifendurchlauf

II) $\Rightarrow$ Q enthielt genau die Knoten $v_{k-1}, \ldots, v_n$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte $(k-1)$-ten Schleifendurchlauf

II) $\Rightarrow$ Q enthielt genau die Knoten $v_{k-1}, \ldots, v_n$

III) $\Rightarrow$ es wurde genau v_{k-1} aus Q entfernt

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte $(k-1)$-ten Schleifendurchlauf

II) $\Rightarrow$ Q enthielt genau die Knoten $v_{k-1}, \ldots, v_n$

III) $\Rightarrow$ es wurde genau v_{k-1} aus Q entfernt $\Rightarrow$ **II)**

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓

II) enthält Q genau die Knoten $v_k, \ldots, v_n$ ✓

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte $(k-1)$-ten Schleifendurchlauf

II) $\Rightarrow$ Q enthielt genau die Knoten $v_{k-1}, \ldots, v_n$

III) $\Rightarrow$ es wurde genau v_{k-1} aus Q entfernt $\Rightarrow$ **II)**

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓

II) enthält Q genau die Knoten $v_k, \ldots, v_n$ ✓

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓

II) enthält Q genau die Knoten $v_k, \ldots, v_n$ ✓

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte Knoten v_ℓ mit $\ell > k$.

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓

II) enthält Q genau die Knoten $v_k, \ldots, v_n$ ✓

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte Knoten v_ℓ mit $\ell > k$.

Dann gilt $v_\ell.d \geq$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓

II) enthält Q genau die Knoten $v_k, \ldots, v_n$ ✓

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte Knoten v_ℓ mit $\ell > k$.

Dann gilt $v_\ell.d \geq \delta(s, v_\ell)$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓

II) enthält Q genau die Knoten $v_k, \ldots, v_n$ ✓

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte Knoten v_ℓ mit $\ell > k$.

Dann gilt $v_\ell.d \geq \delta(s, v_\ell)$

$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓

II) enthält Q genau die Knoten $v_k, \ldots, v_n$ ✓

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte Knoten v_ℓ mit $\ell > k$.

Dann gilt $v_\ell.d \geq \delta(s, v_\ell) > \delta(s, v_k) =$

$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓

II) enthält Q genau die Knoten $v_k, \ldots, v_n$ ✓

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte Knoten v_ℓ mit $\ell > k$.

Dann gilt $v_\ell.d \geq \delta(s, v_\ell) > \delta(s, v_k) = v_k.d$ (I)

$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

- **I)** gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓
- **II)** enthält Q genau die Knoten $v_k, \ldots, v_n$ ✓
- **III)** gilt $v_k = Q.\text{ExtractMin}()$ ✓

1. Initialisierung ✓

2. Aufrechterhaltung

Betrachte Knoten v_ℓ mit $\ell > k$.

Dann gilt $v_\ell.d \geq \delta(s, v_\ell) > \delta(s, v_k) = v_k.d$ (I)

$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

- **I)** gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$ ✓
- **II)** enthält Q genau die Knoten $v_k, \ldots, v_n$ ✓
- **III)** gilt $v_k = Q.\text{ExtractMin}()$ ✓

1. Initialisierung ✓

2. Aufrechterhaltung ✓

Betrachte Knoten v_ℓ mit $\ell > k$.

Dann gilt $v_\ell.d \geq \delta(s, v_\ell) > \delta(s, v_k) = v_k.d$ (I)

$\delta(s, v_1) < \delta(s, v_2) < \ldots < \delta(s, v_n)$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung ✓

3. Terminierung

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung ✓

3. Terminierung

I) $\Rightarrow$ $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq n$

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

Dijkstra – die Korrektheit

0. Schleifeninvariante

Zu Beginn des k-ten Durchlaufs der **while**-Schleife

I) gilt $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq k$

II) enthält Q genau die Knoten $v_k, \ldots, v_n$

III) gilt $v_k = Q.\text{ExtractMin}()$

1. Initialisierung ✓

2. Aufrechterhaltung ✓

3. Terminierung ✓

I) $\Rightarrow$ $v_i.d = \delta(s, v_i)$ für alle $1 \leq i \leq n$

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

Kürzeste Wege nach Dijkstra

Eingabe	Algorithmus	Laufzeit
ungewichteter Graph		
nicht-neg. Kantengew.		
azyklischer Graph		
negative Kantengew.		
für alle Knotenpaare		
+ negative Kantengew.		
k kürzeste s-t-Wege		

Kürzeste Wege nach Dijkstra

Eingabe	Algorithmus	Laufzeit
ungewichteter Graph	Breitensuche	$\mathcal{O}(E + V)$
nicht-neg. Kantengew.		
azyklischer Graph		
negative Kantengew.		
für alle Knotenpaare		
+ negative Kantengew.		
k kürzeste s-t-Wege		

Vorlesung 18

Kürzeste Wege nach Dijkstra

Eingabe	Algorithmus	Laufzeit	
ungewichteter Graph	Breitensuche	$\mathcal{O}(E + V)$	Vorlesung 18
nicht-neg. Kantengew.	Dijkstra	$\mathcal{O}(E + V \log V)$	heute
azyklischer Graph			
negative Kantengew.			
für alle Knotenpaare			
+ negative Kantengew.			
k kürzeste s-t-Wege			

Kürzeste Wege nach Dijkstra

Eingabe	Algorithmus	Laufzeit	
ungewichteter Graph	Breitensuche	$\mathcal{O}(E + V)$	Vorlesung 18
nicht-neg. Kantengew.	Dijkstra	$\mathcal{O}(E + V \log V)$	heute
azyklischer Graph	Topol. Sortieren	$\mathcal{O}(E + V)$	Vorlesung 20
negative Kantengew.			
für alle Knotenpaare			
+ negative Kantengew.			
k kürzeste s-t-Wege			

Kürzeste Wege nach Dijkstra

Eingabe	Algorithmus	Laufzeit	
ungewichteter Graph	Breitensuche	$\mathcal{O}(E + V)$	Vorlesung 18
nicht-neg. Kantengew.	Dijkstra	$\mathcal{O}(E + V \log V)$	heute
azyklischer Graph	Topol. Sortieren	$\mathcal{O}(E + V)$	Vorlesung 20
negative Kantengew.	Bellman-Ford	$\mathcal{O}(EV)$	
für alle Knotenpaare			
+ negative Kantengew.			
k kürzeste s-t-Wege			

Kürzeste Wege nach Dijkstra

Eingabe	Algorithmus	Laufzeit	
ungewichteter Graph	Breitensuche	$\mathcal{O}(E + V)$	Vorlesung 18
nicht-neg. Kantengew.	Dijkstra	$\mathcal{O}(E + V \log V)$	heute
azyklischer Graph	Topol. Sortieren	$\mathcal{O}(E + V)$	Vorlesung 20
negative Kantengew.	Bellman-Ford	$\mathcal{O}(EV)$	
für alle Knotenpaare	$\lvert V\rvert \times$ Dijkstra	$\mathcal{O}(V(E + V \log V))$	heute
+ negative Kantengew.			
k kürzeste s-t-Wege			

Kürzeste Wege nach Dijkstra

Eingabe	Algorithmus	Laufzeit	
ungewichteter Graph	Breitensuche	$\mathcal{O}(E + V)$	Vorlesung 18
nicht-neg. Kantengew.	Dijkstra	$\mathcal{O}(E + V \log V)$	heute
azyklischer Graph	Topol. Sortieren	$\mathcal{O}(E + V)$	Vorlesung 20
negative Kantengew.	Bellman-Ford	$\mathcal{O}(EV)$	
für alle Knotenpaare	$\lvert V\rvert \times$ Dijkstra	$\mathcal{O}(V(E + V \log V))$	heute
+ negative Kantengew.	Floyd-Warshall	$\mathcal{O}(V^3)$	
k kürzeste s-t-Wege			

Kürzeste Wege nach Dijkstra

Eingabe	Algorithmus	Laufzeit	
ungewichteter Graph	Breitensuche	$\mathcal{O}(E + V)$	Vorlesung 18
nicht-neg. Kantengew.	Dijkstra	$\mathcal{O}(E + V \log V)$	heute
azyklischer Graph	Topol. Sortieren	$\mathcal{O}(E + V)$	Vorlesung 20
negative Kantengew.	Bellman-Ford	$\mathcal{O}(EV)$	
für alle Knotenpaare	$\lvert V\rvert \times$ Dijkstra	$\mathcal{O}(V(E + V \log V))$	heute
+ negative Kantengew.	Floyd-Warshall	$\mathcal{O}(V^3)$	
	Johnson	$\mathcal{O}(V(E + V \log V))$	
k kürzeste s-t-Wege			

Kürzeste Wege nach Dijkstra

Eingabe	Algorithmus	Laufzeit	
ungewichteter Graph	Breitensuche	$\mathcal{O}(E + V)$	Vorlesung 18
nicht-neg. Kantengew.	Dijkstra	$\mathcal{O}(E + V \log V)$	heute
azyklischer Graph	Topol. Sortieren	$\mathcal{O}(E + V)$	Vorlesung 20
negative Kantengew.	Bellman-Ford	$\mathcal{O}(EV)$	
für alle Knotenpaare	$\lvert V\rvert \times$ Dijkstra	$\mathcal{O}(V(E + V \log V))$	heute
+ negative Kantengew.	Floyd-Warshall	$\mathcal{O}(V^3)$	
	Johnson	$\mathcal{O}(V(E + V \log V))$	
k kürzeste s-t-Wege	Eppstein	$\mathcal{O}(k + E + V \log V)$	