Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Lehrstuhl für
INFORMATIK I
Algorithmen & Komplexität

Institut für Informatik

Algorithmen und Datenstrukturen

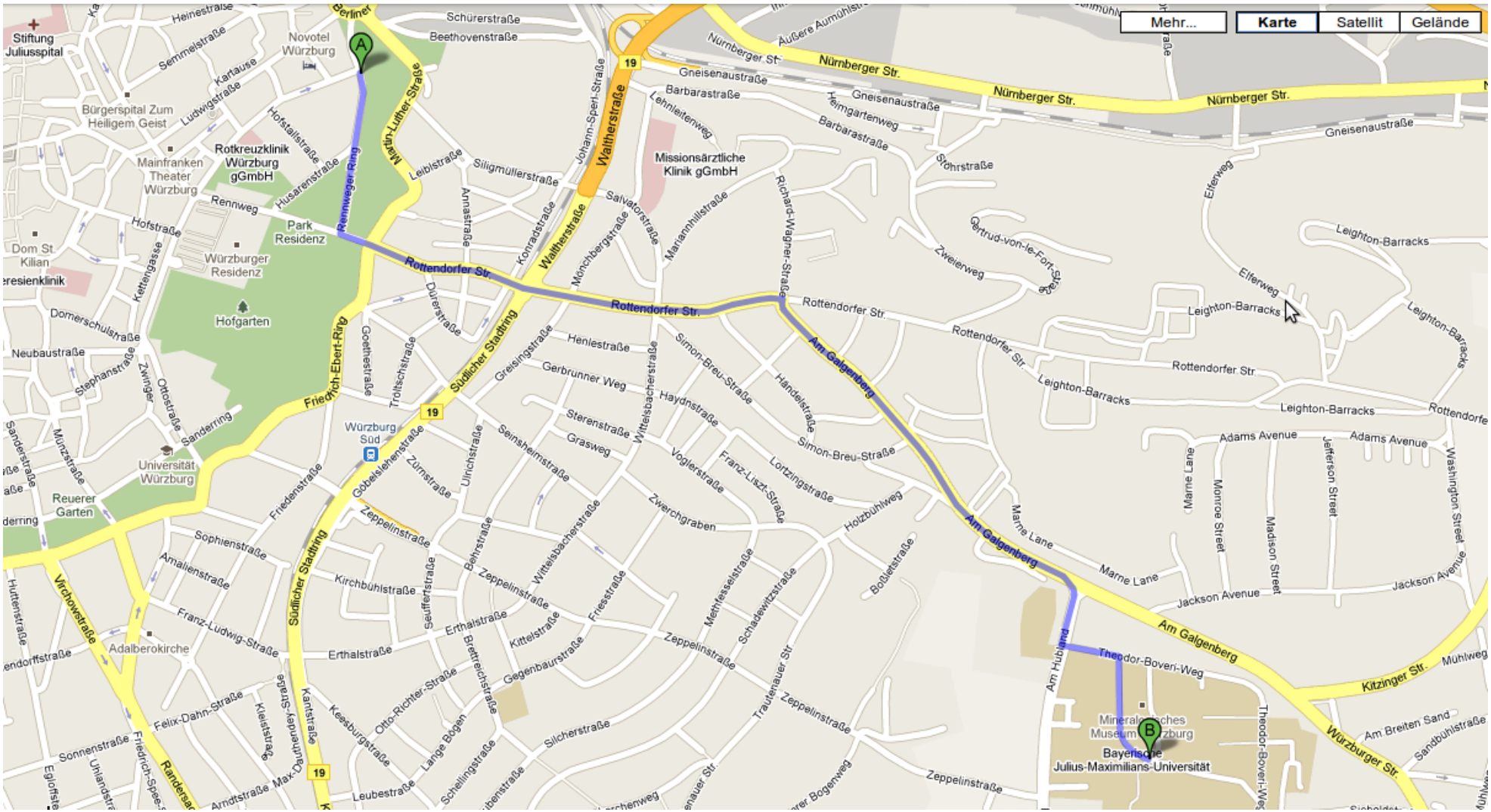
Wintersemester 2023/24

19. Vorlesung

Kürzeste Wege & Dijkstras Algorithmus

Alexander Wolff *Lehrstuhl für Informatik I*

Wozu kürzeste Wege?



Modellierung des Problems Routenplanung

Straßenkreuzung $\rightarrow$

Straßenabschnitt $\rightarrow$

Einbahnstraßenabschnitt $\rightarrow$

Fahrtzeit für Abschnitt e $\rightarrow$

Straßennetz $\rightarrow$

Start $\rightarrow$

Ziel $\rightarrow$

Start-Ziel-Route $\rightarrow$

Modellierung des Problems Routenplanung

Straßenkreuzung $\rightarrow$ Knoten

Straßenabschnitt $\rightarrow$

Einbahnstraßenabschnitt $\rightarrow$

Fahrtzeit für Abschnitt e $\rightarrow$

Straßennetz $\rightarrow$

Start $\rightarrow$

Ziel $\rightarrow$

Start-Ziel-Route $\rightarrow$

Modellierung des Problems Routenplanung

Straßenkreuzung $\rightarrow$ Knoten

Straßenabschnitt $\rightarrow$ zwei entgegengerichtete Kanten

Einbahnstraßenabschnitt $\rightarrow$

Fahrtzeit für Abschnitt e $\rightarrow$

Straßennetz $\rightarrow$

Start $\rightarrow$

Ziel $\rightarrow$

Start-Ziel-Route $\rightarrow$

Modellierung des Problems Routenplanung

Straßenkreuzung $\rightarrow$ Knoten

Straßenabschnitt $\rightarrow$ zwei entgegengerichtete Kanten

Einbahnstraßenabschnitt $\rightarrow$ in Fahrtrichtung gerichtete Kante

Fahrtzeit für Abschnitt e $\rightarrow$

Straßennetz $\rightarrow$

Start $\rightarrow$

Ziel $\rightarrow$

Start-Ziel-Route $\rightarrow$

Modellierung des Problems Routenplanung

Straßenkreuzung $\rightarrow$ Knoten

Straßenabschnitt $\rightarrow$ zwei entgegengerichtete Kanten

Einbahnstraßenabschnitt $\rightarrow$ in Fahrtrichtung gerichtete Kante

Fahrtzeit für Abschnitt e $\rightarrow$ Kantengewicht $w(e) \geq 0$

Straßennetz $\rightarrow$

Start $\rightarrow$

Ziel $\rightarrow$

Start-Ziel-Route $\rightarrow$

Modellierung des Problems Routenplanung

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

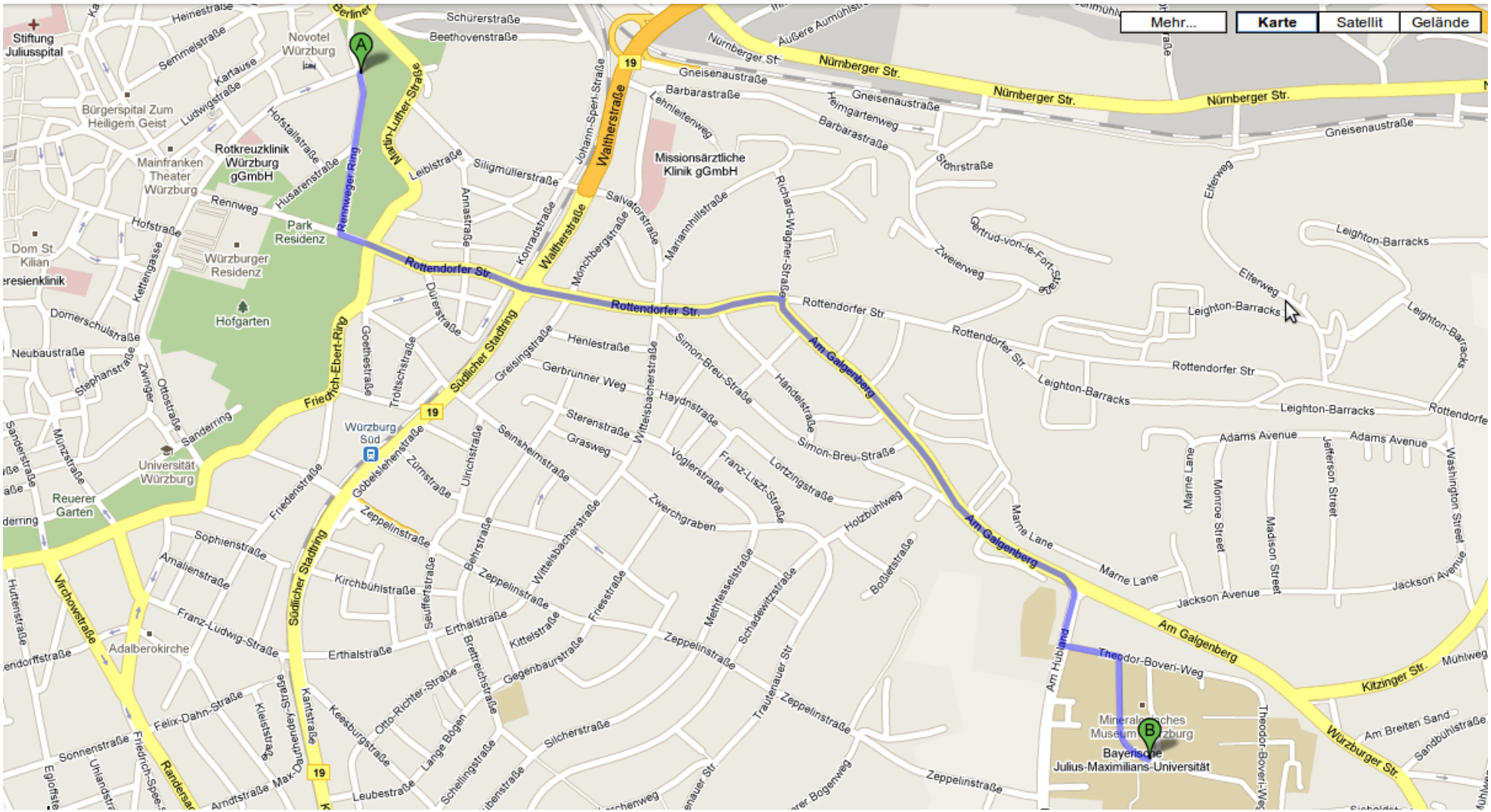
Modellierung des Problems Routenplanung

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

Modellierung des Problems Routenplanung

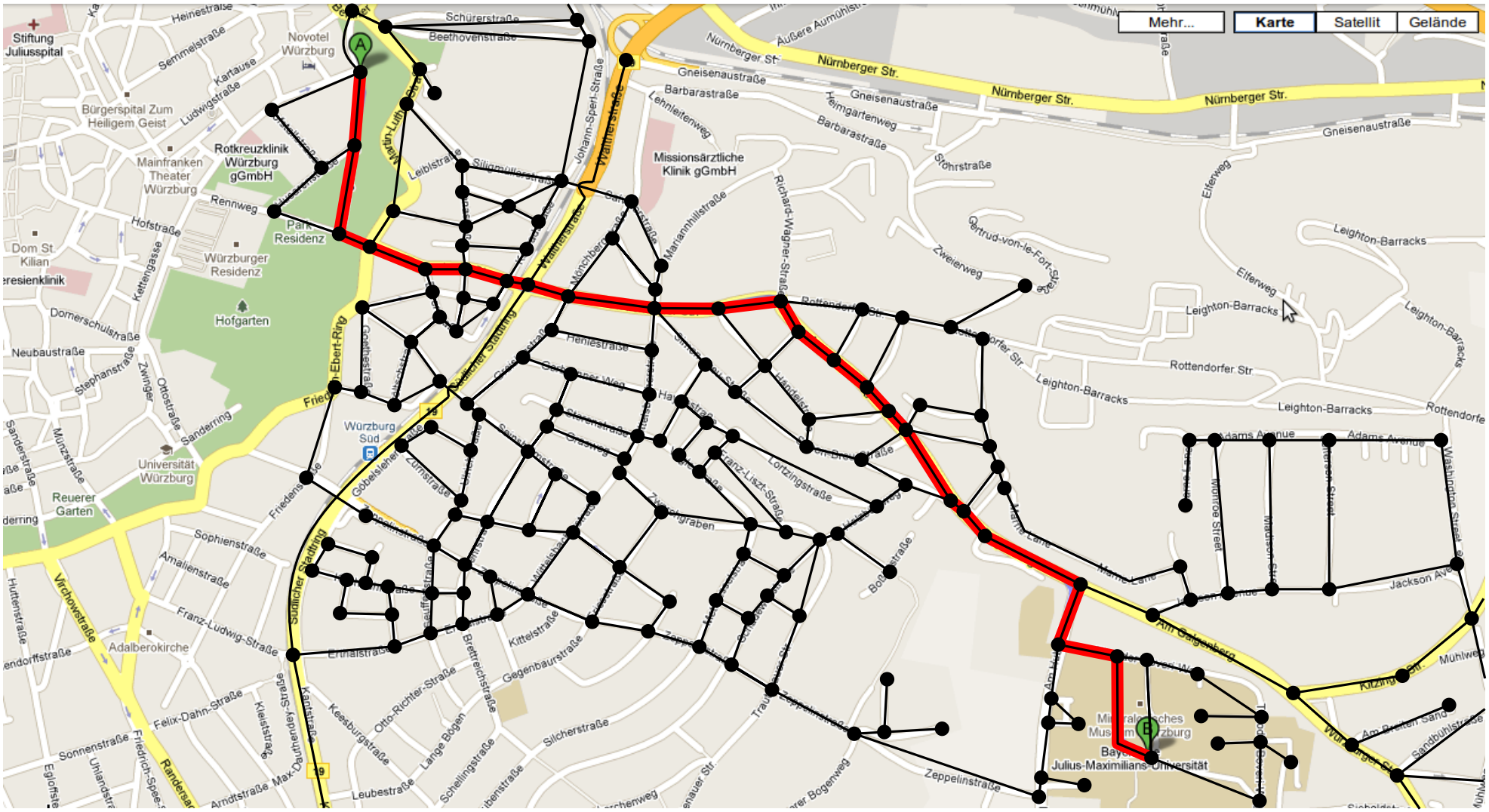
Straßenkreuzung	$\rightarrow$ Knoten
Straßenabschnitt	$\rightarrow$ zwei entgegengerichtete Kanten
Einbahnstraßenabschnitt	$\rightarrow$ in Fahrtrichtung gerichtete Kante
Fahrtzeit für Abschnitt e	$\rightarrow$ Kantengewicht $w(e) \geq 0$
Straßennetz	$\rightarrow$ gerichteter, gewichteter und zusammenhängender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

Wozu kürzeste Wege?

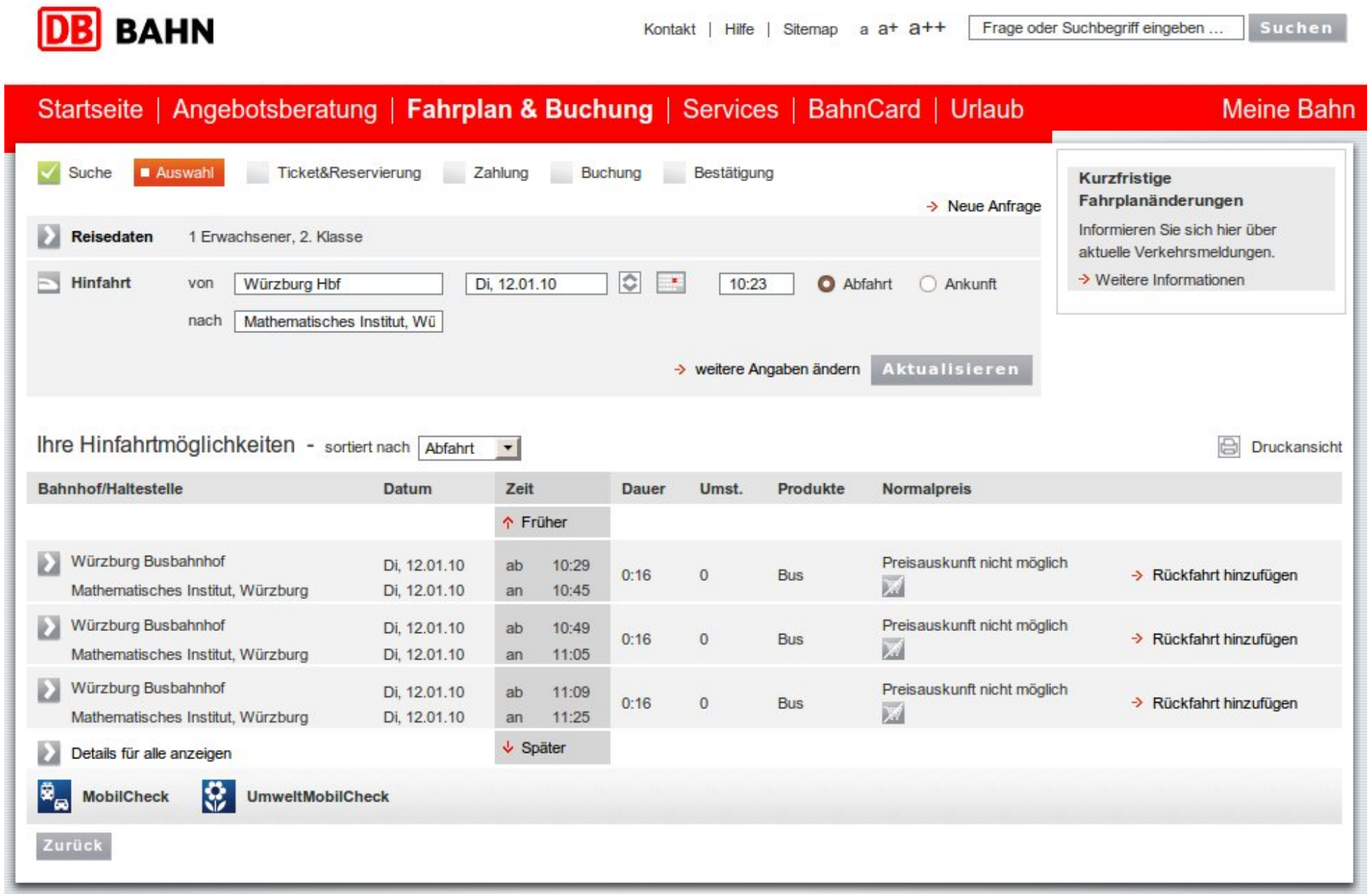


Wozu kürzeste Wege?

Wozu kürzeste Wege?



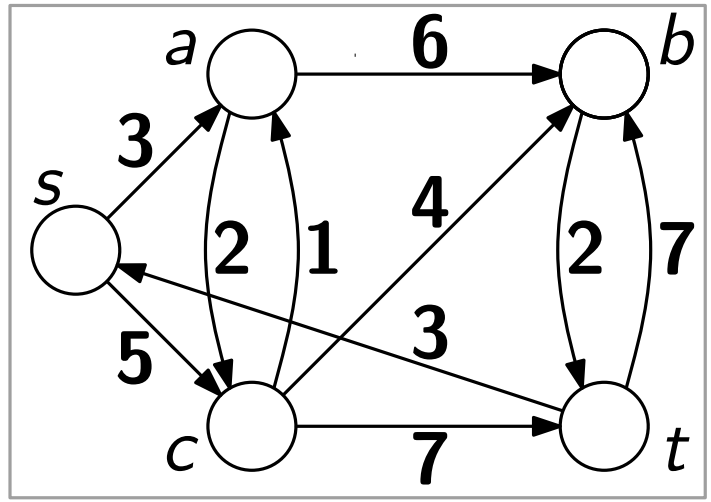
Wozu kürzeste Wege? (II)



Was ist das Problem?

Eingabe:

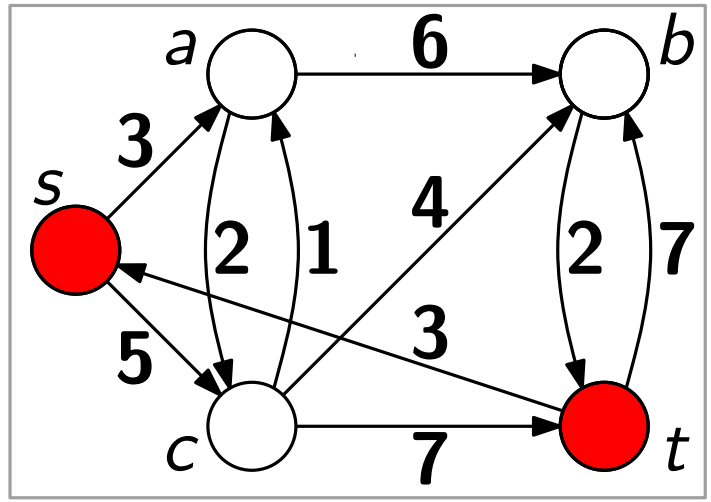
- gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen **Kantengewichten** $w\colon E \to \mathbb{Q}_0^+$,



Was ist das Problem?

Eingabe:

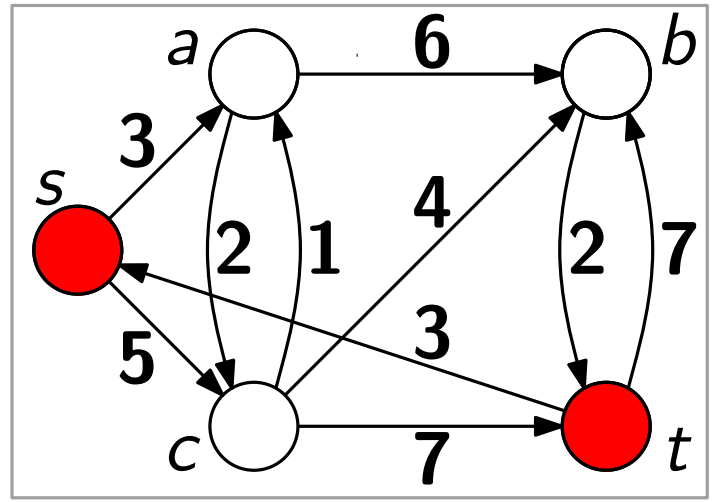
- gerichteter, zusammenhängender Graph $G = (V, E)$ mit nicht-negativen **Kantengewichten** $w\colon E \to \mathbb{Q}_0^+$,
- Knoten s und t



Was ist das Problem?

Eingabe:

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



Ausgabe:

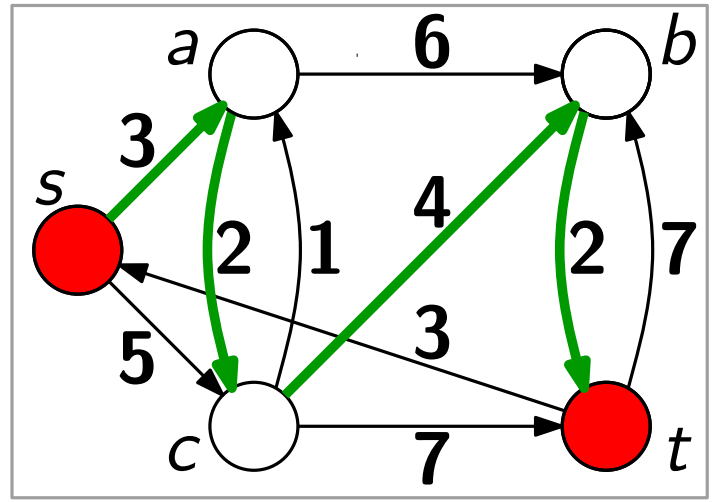
- kürzester s-t-Weg W in G, d.h. $\sum_{e \in W} w(e)$ minimal.

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

Was ist das Problem?

Eingabe:

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



Ausgabe:

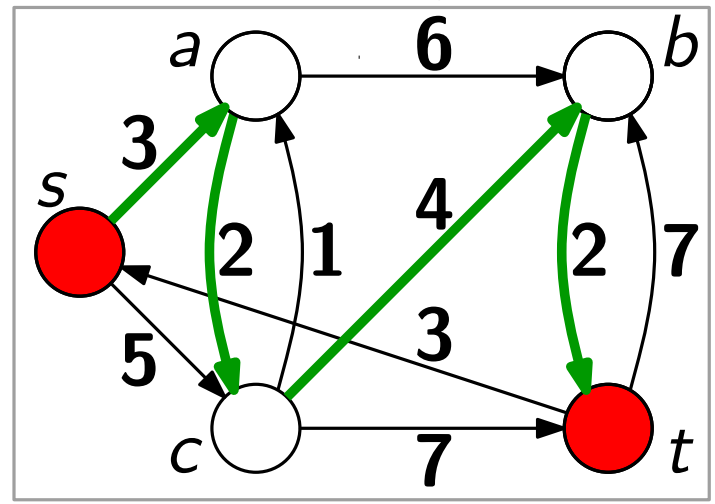
- kürzester s-t-Weg W in G, d.h. $\sum_{e \in W} w(e)$ minimal.

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

Was ist das Problem?

Eingabe:

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



Ausgabe:

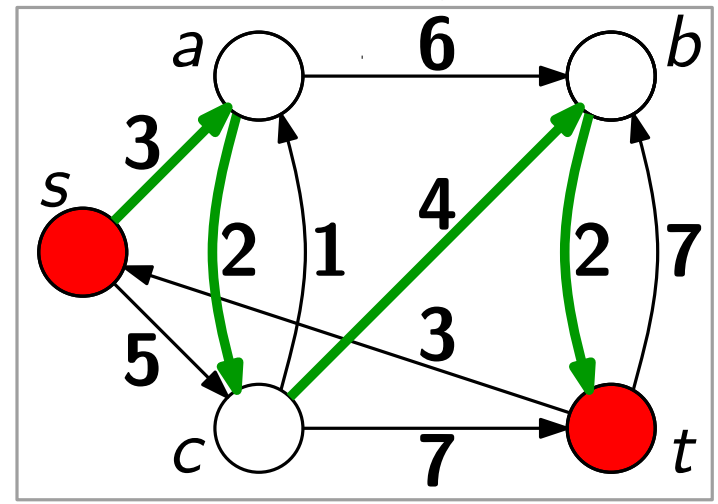
- kürzeste~~r~~ s-t-Wege W_t in G, $\underbrace{\text{für alle } t \in V}$ d.h. $\sum_{e \in W} w(e)$ minimal.

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

Was ist das Problem?

Eingabe:

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



Ausgabe:

- kürzeste~~r~~ s-t-Wege W_t in G, $\underbrace{\text{für alle } t \in V}$ d.h. $\sum_{e \in W} w(e)$ minimal.

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

Nebenbemerkung: *Analoge* Berechnungsverfahren?

Dijkstra – BFS mit Gewichten

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

Dijkstra – BFS mit Gewichten

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

```
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 = gray
        v.d = u.d + 1
        v.π = u
        Q.Enqueue(v)
    u.color = black
```

Dijkstra – BFS mit Gewichten

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

```
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 = gray
              v.d = u.d + 1
              v.π = u
              Q.Enqueue(v)
      u.color = black
```

Dijkstra – BFS mit Gewichten

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

```
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 = gray
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = black
```

Dijkstra – BFS mit Gewichten

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

```
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 = gray
                v.d = u.d + 1
                v.π = u
                Q.Enqueue(v)
        u.color = black
```

Dijkstra – BFS mit Gewichten

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

```
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 = gray
              v.d = u.d + 1
              v.π = u
              Q.Enqueue(v)
      u.color = black
```

Dijkstra – BFS mit Gewichten

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

```
Relax(u, v; w)
```

Was muss in dieser Unterroutine passieren?

Schreiben Sie's auf!

```
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 = gray
              v.d = u.d + 1
              v.π = u
              Q.Enqueue(v)
      u.color = black
```

Dijkstra – BFS mit Gewichten

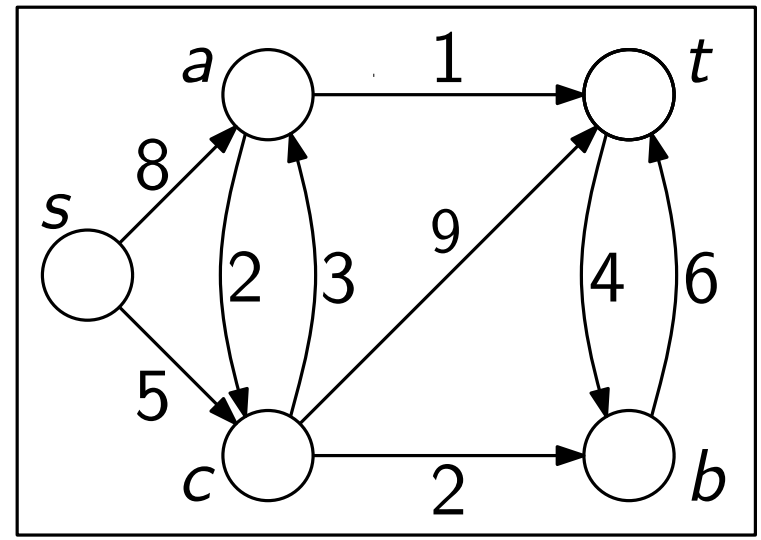
```
Dijkstra(WeightedGraph G = (V, E; w), Vertex s)
  Initialize(G, s)
  Q = new PriorityQueue(V, d)        // Gewichtung
  while not Q.Empty() do
      u = Q.ExtractMin()
      foreach v ∈ Adj[u] do
          Relax(u, v; w)
      u.color = black
```

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

```
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 = gray
              v.d = u.d + 1
              v.π = u
              Q.Enqueue(v)
      u.color = black
```

Dijkstra – ein Beispiel

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

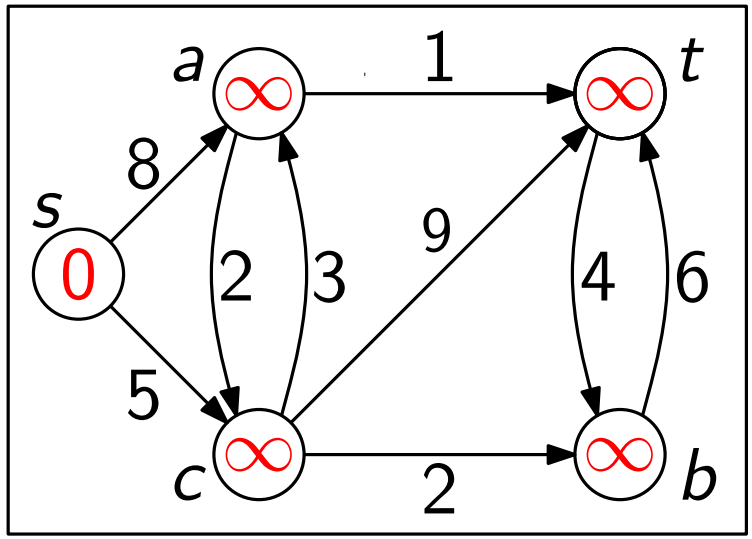



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

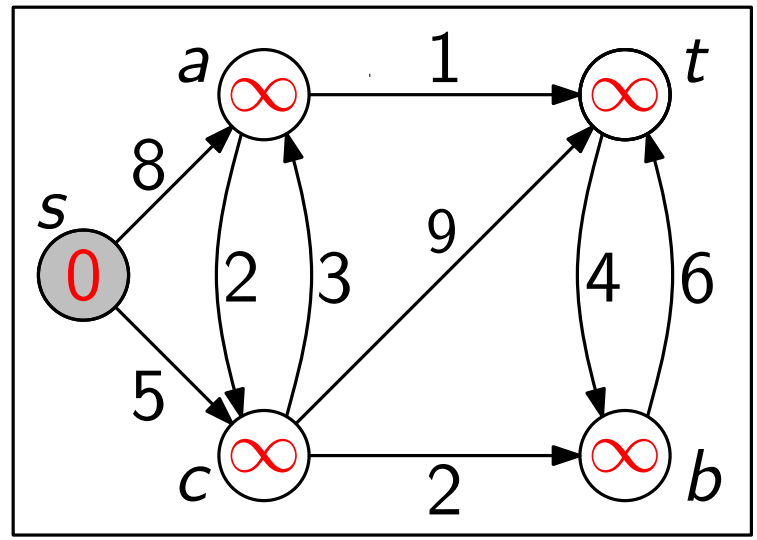



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

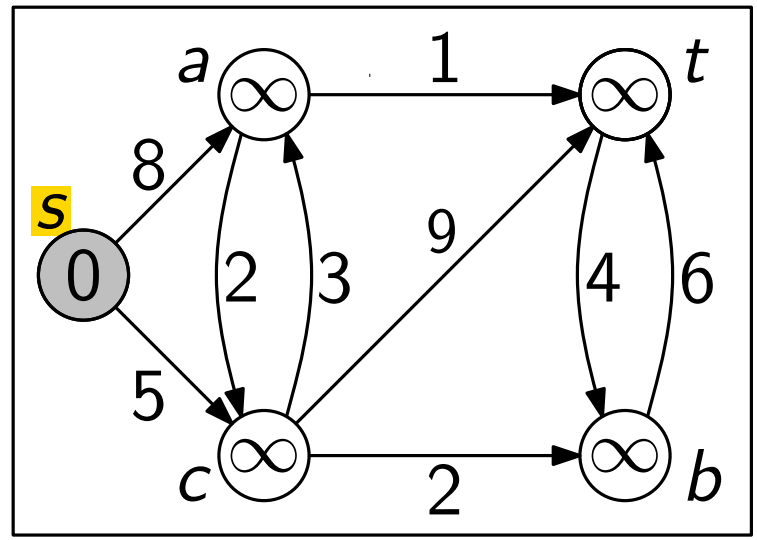



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

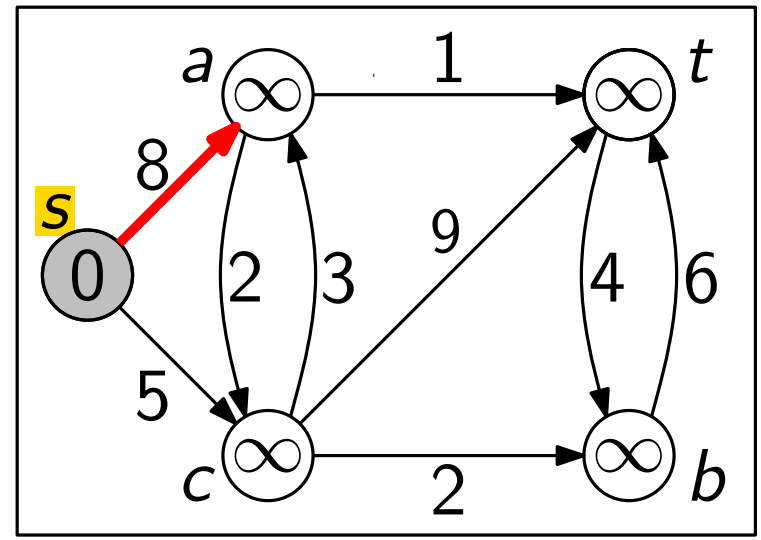



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

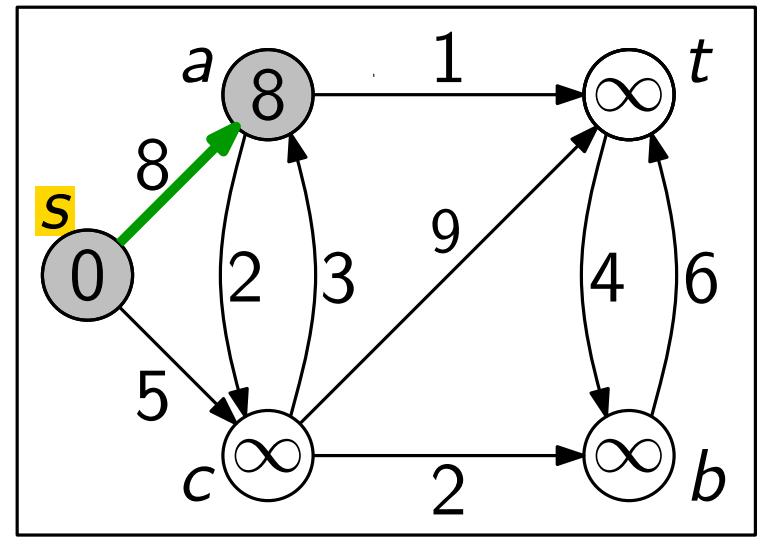



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

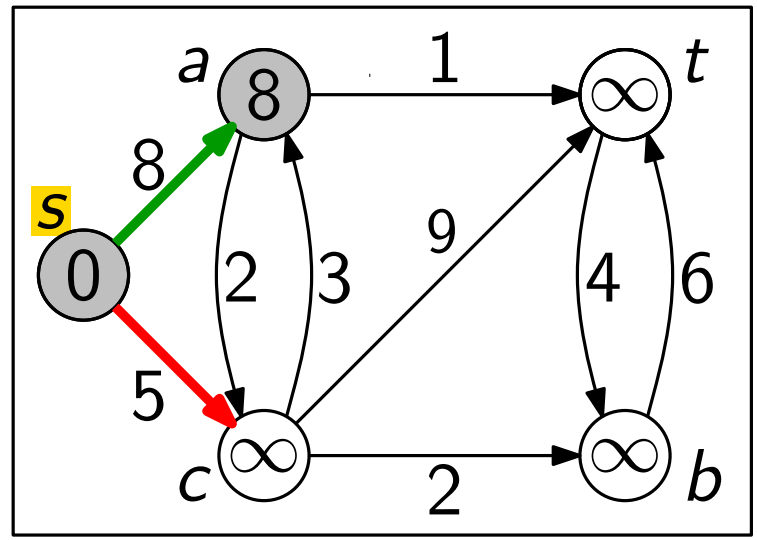



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

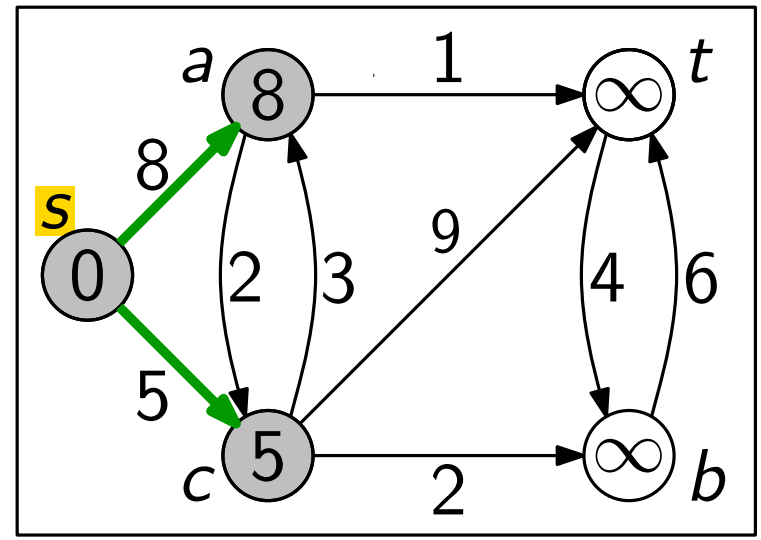



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

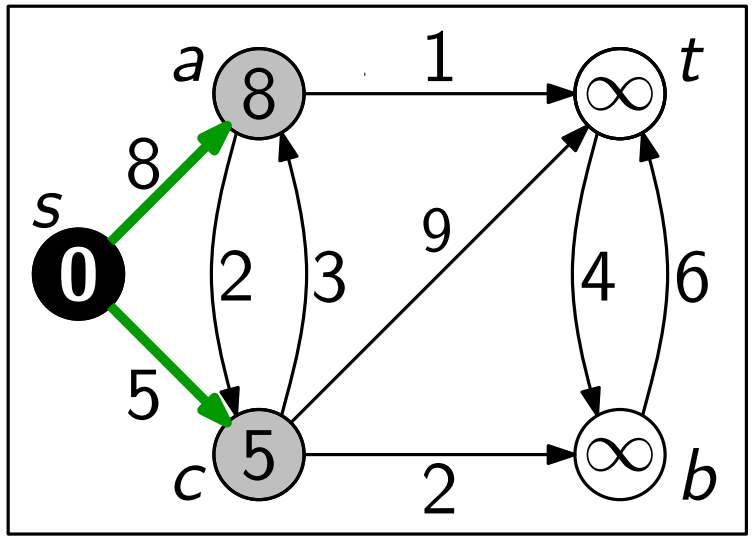



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

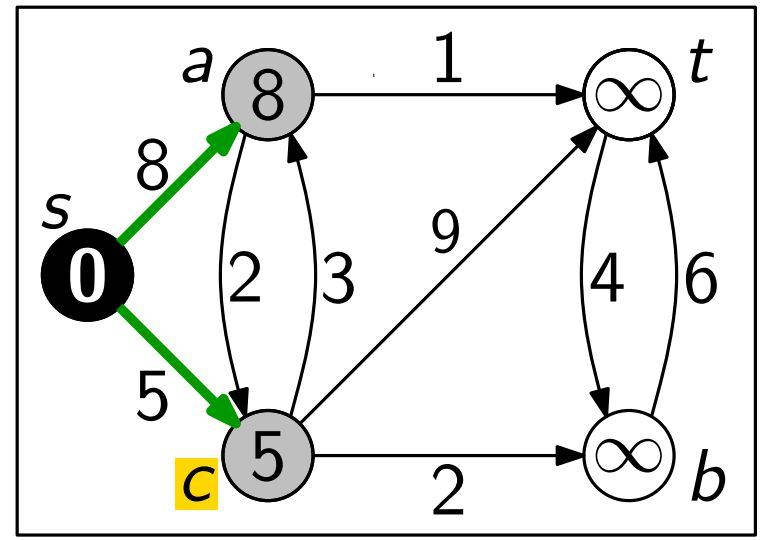



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

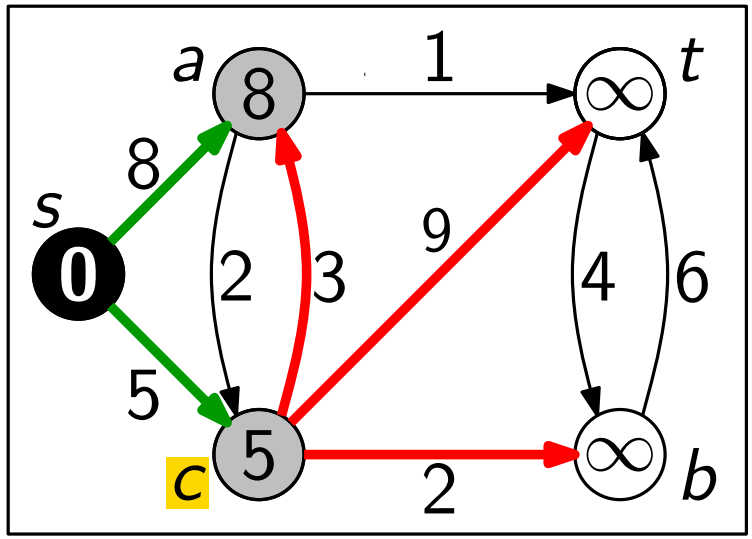



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

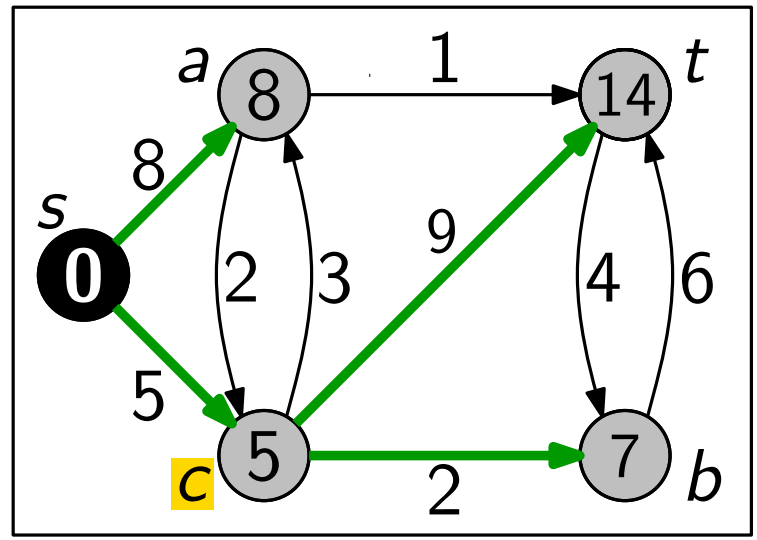



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

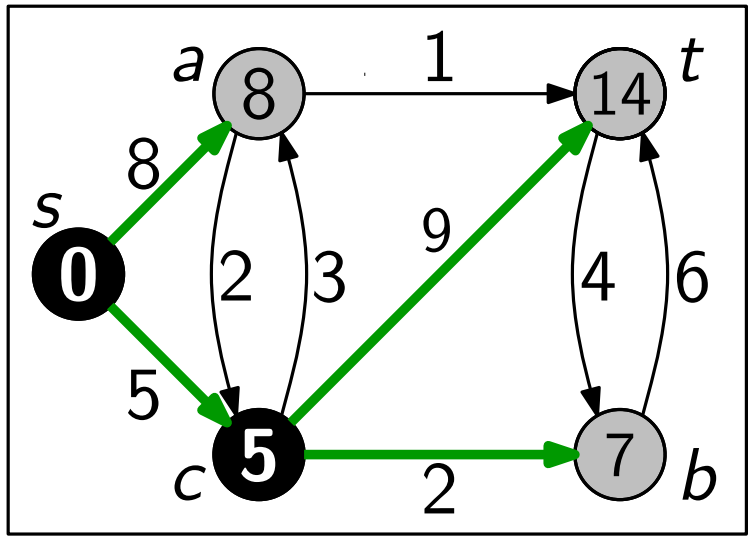



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

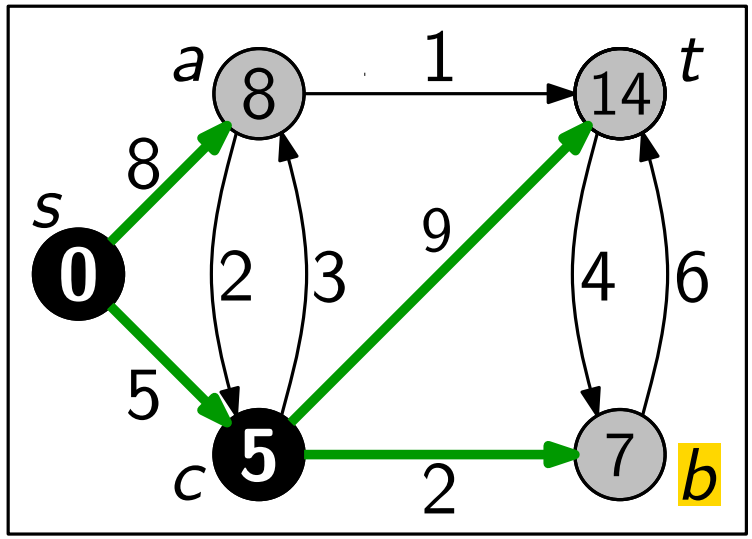



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

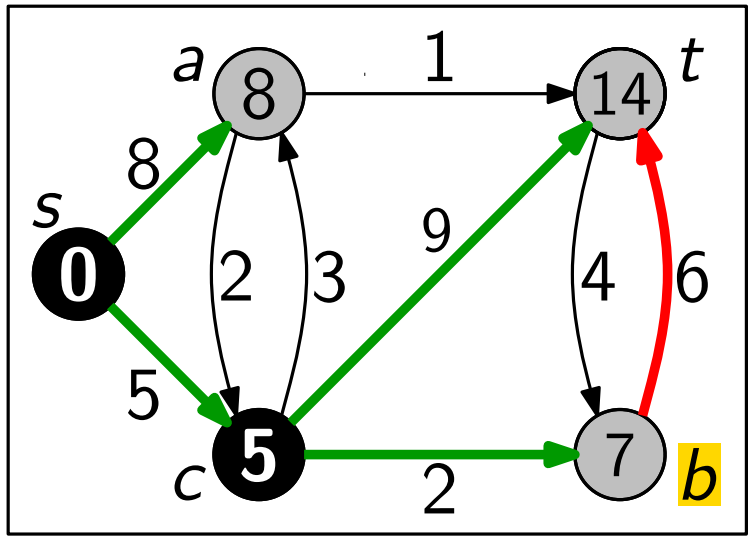



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

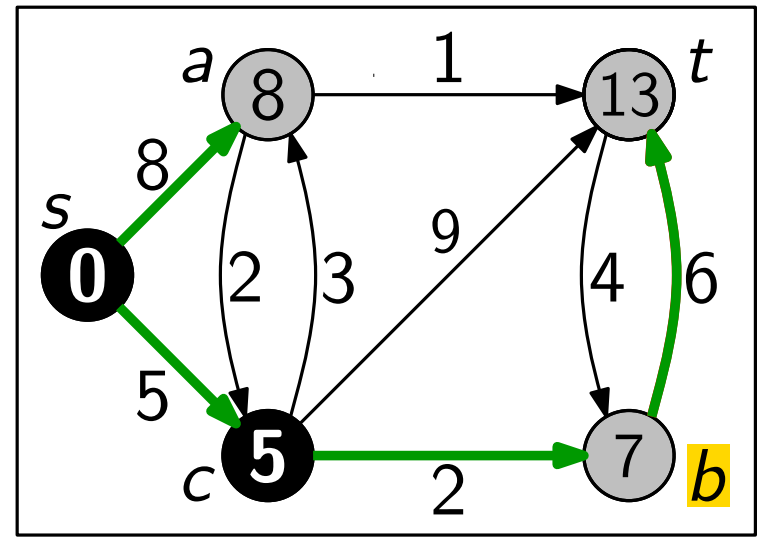



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

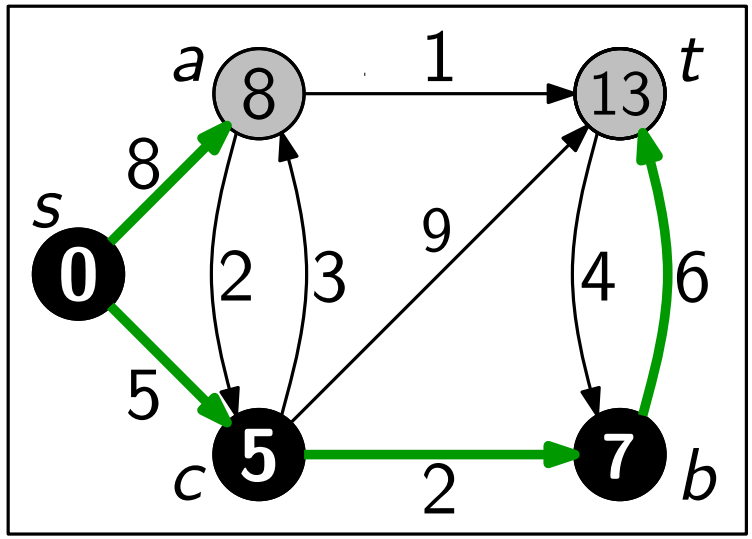



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

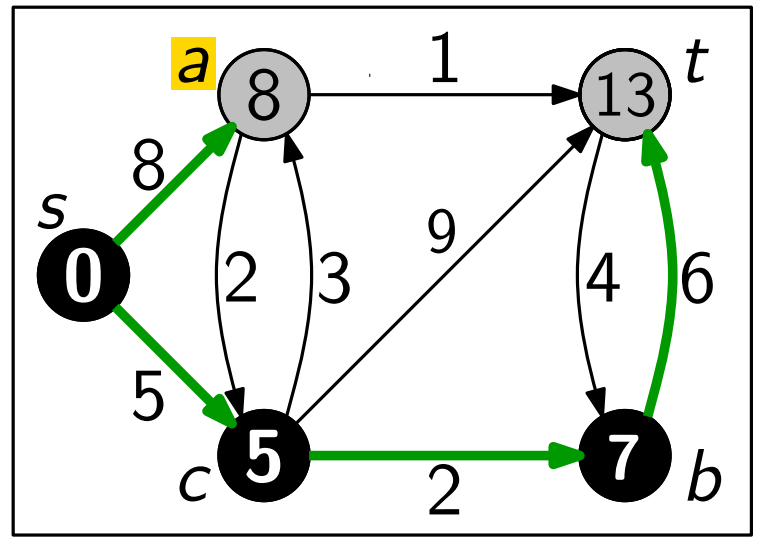



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

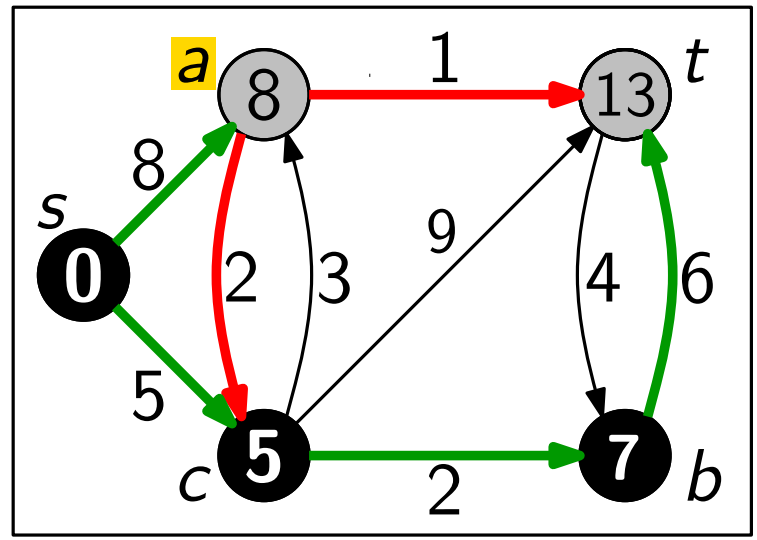



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

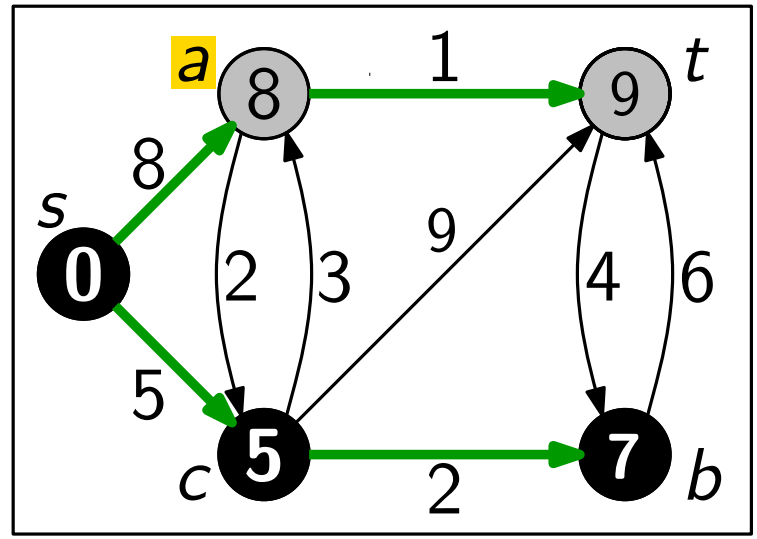



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

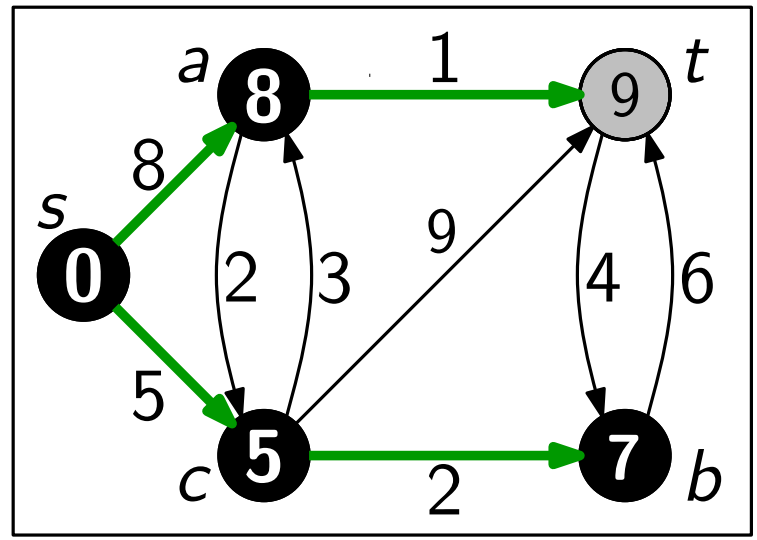



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

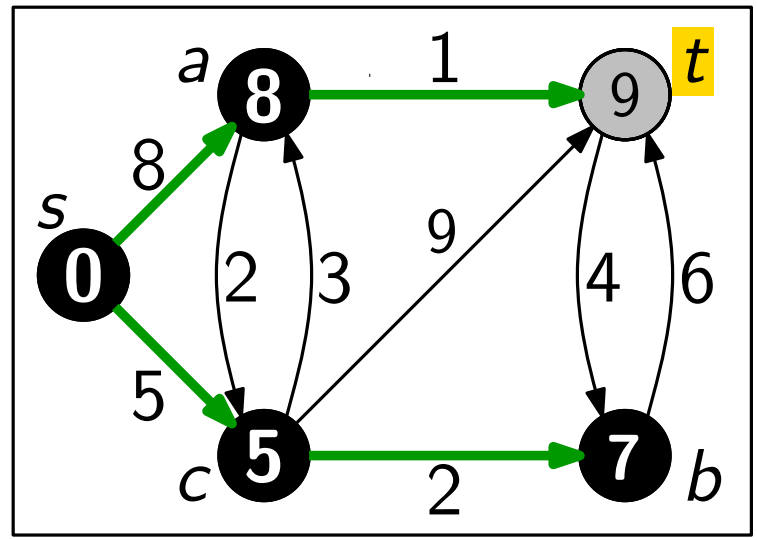



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

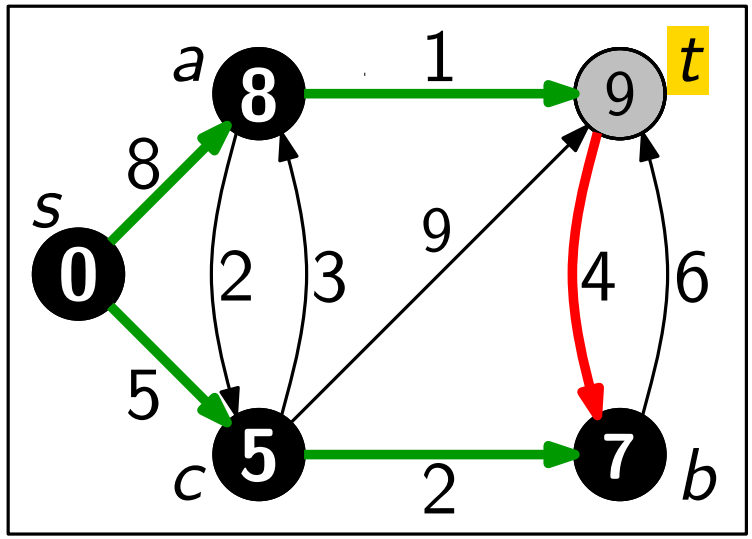



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

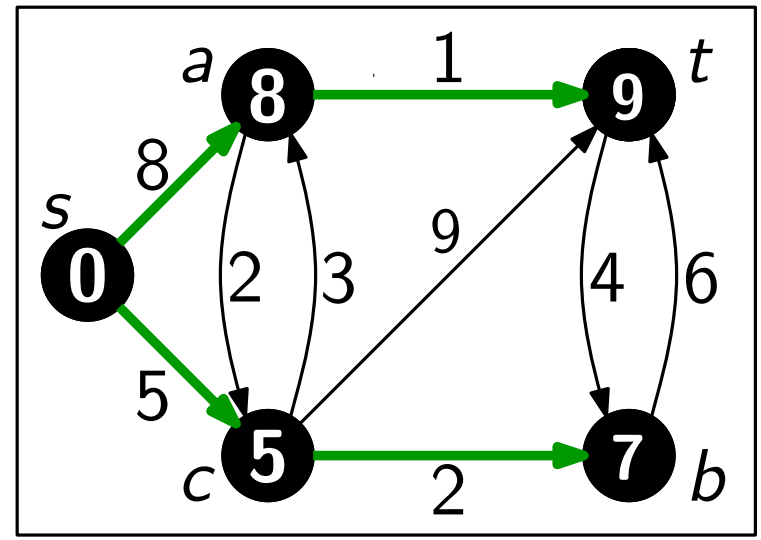



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

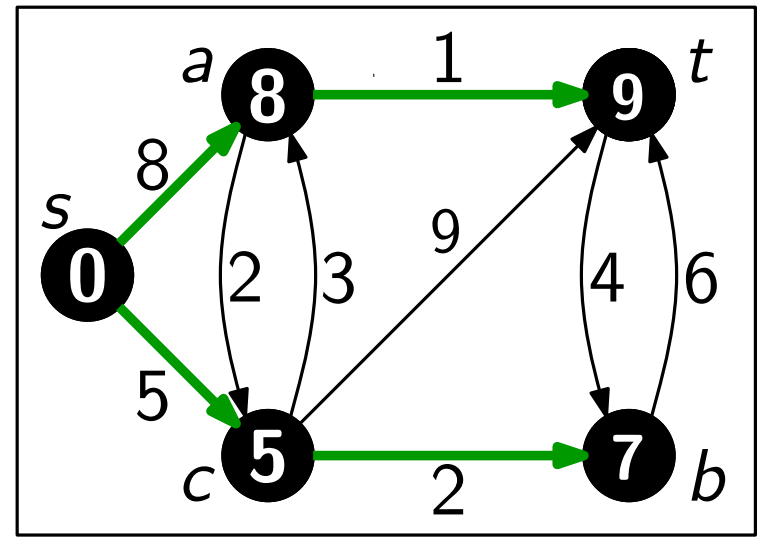



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

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

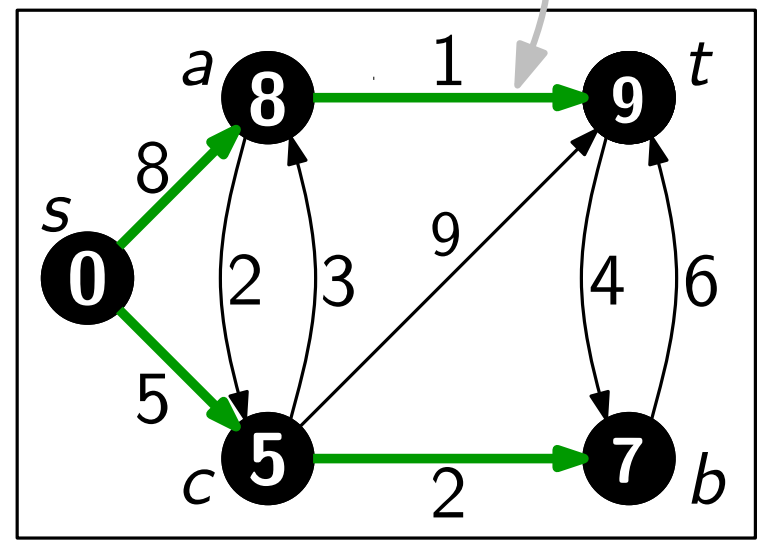



```
Relax(u, v; w)
  if v.d > u.d + w(u, v) then
      v.color = gray
      v.d = u.d + w(u, v)
      v.π = u
      Q.DecreaseKey(v, v.d)
```

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

Dijkstra – ein Beispiel

Kürzester-Wege-Baum mit Wurzel s




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

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

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

Dijkstra – die Laufzeit

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

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

Dijkstra – die Laufzeit

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

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

Dijkstra – die Laufzeit

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

$O(\quad)$ Zeit

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

Dijkstra – die Laufzeit

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

$O(V)$ Zeit

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

Dijkstra – die Laufzeit

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

Abk. für $O(|V|)$

$O(V)$ Zeit

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

Dijkstra – die Laufzeit

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

Abk. für $O(|V|)$

$O(V)$ Zeit

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

Dijkstra – die Laufzeit

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

Abk. für $O(|V|)$

$O(V)$ Zeit

genau $|V|$ mal

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

Dijkstra – die Laufzeit

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

Abk. für $O(|V|)$

$O(V)$ Zeit

genau $|V|$ mal

Wie oft wird Relax aufgerufen?

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

Dijkstra – die Laufzeit

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

Abk. für $O(|V|)$

$O(V)$ Zeit

genau $|V|$ mal

Wie oft wird Relax aufgerufen?

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

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

Dijkstra – die Laufzeit

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

Abk. für $O(|V|)$

$O(V)$ Zeit

genau $|V|$ mal

Wie oft wird Relax aufgerufen?

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

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

Dijkstra – die Laufzeit

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

Abk. für $O(|V|)$

$O(V)$ Zeit

genau $|V|$ mal

Wie oft wird Relax aufgerufen?

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

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

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

Dijkstra – die Laufzeit

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

Abk. für $O(|V|)$

$O(V)$ Zeit

genau $|V|$ mal

Wie oft wird Relax aufgerufen?

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

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

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

Also wird DecreaseKey [ ] mal aufgerufen.

Dijkstra – die Laufzeit

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

Abk. für $O(|V|)$

$O(V)$ Zeit

genau $|V|$ mal

Wie oft wird Relax aufgerufen?

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

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

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

Also wird DecreaseKey $O(E)$ mal aufgerufen.

Dijkstra – die Laufzeit

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

Abk. für $O(|V|)$

$O(V)$ Zeit

genau $|V|$ mal

Wie oft wird Relax aufgerufen?

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

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

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

Also wird DecreaseKey $O(E)$ mal aufgerufen.

Dijkstra – die Laufzeit

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

Dijkstra – die Laufzeit

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

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

Dijkstra – die Laufzeit

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

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

*) Das geht, weil

Dijkstra – die Laufzeit

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

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

*) 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 $O(V \cdot T_{\text{ExtractMin}}(|V|) + E \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

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

*) 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 $O(V \cdot T_{\text{ExtractMin}}(|V|) + E \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

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

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

*) 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 $O(V \cdot T_{\text{ExtractMin}}(|V|) + E \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

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

*) 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 $O(V \cdot T_{\text{ExtractMin}}(|V|) + E \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

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

*) 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 $O(V \cdot T_{\text{ExtractMin}}(|V|) + E \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

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

*) 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 $O(V \cdot T_{\text{ExtractMin}}(|V|) + E \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

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

*) 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 $O(V \cdot T_{\text{ExtractMin}}(|V|) + E \cdot T_{\text{DecreaseKey}}(|V|))$ Zeit.

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

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

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

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

Implementierung einer PriorityQueue	$T_{\text{ExtractMin}}(n)$	$T_{\text{DecreaseKey}}(n)$	$T_{\text{Dijkstra}}(\lvert V\rvert,\lvert E\rvert)$
als unsortiertes Feld	$O(n)$	$O(1)^{\star}$	$O(V^2+E)$
als Heap	$O(\log n)$	$O(\log n)^{\star\star}$	$O((E+V)\log V)$
als Fibonacci-Heap	$O(\log n)$ *amortisiert*	$O(1)$ *amortisiert*	$O(E+V\log V)$ *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 $w\colon E\rightarrow \mathbb{Q}_{\geq 0}$ kann man in $O(E+V\log V)$ Zeit die kürzesten Wege von einem zu allen Knoten berechnen (SSSP-Problem).

Dijkstra – die Korrektheit

Dijkstra – die Korrektheit

siehe [CLRS], Kapitel 24.3., Satz 24.6:
Korrektheisbeweis mittels Schleifeninvariante.

Dijkstra – die Korrektheit

siehe [CLRS], Kapitel 24.3., Satz 24.6:
Korrektheisbeweis mittels Schleifeninvariante.

oder

Dijkstra – die Korrektheit

siehe [CLRS], Kapitel 24.3., Satz 24.6:
Korrektheisbeweis mittels Schleifeninvariante.

oder

MIT-Vorlesungsmitschnitt von Erik Demaine:
`http://videolectures.net/mit6046jf05_demaine_lec17`

Wozu kürzeste Wege? (III) – SMSen

GHI

MNO

DEF

MNO

PQRS

MNO

ABC

TUV

GHI

JKL

Wozu kürzeste Wege? (III) – SMSen

GHI

MNO

DEF

MNO

PQRS

MNO

ABC

TUV

GHI

JKL

Wozu kürzeste Wege? (III) – SMSen

GHI

MNO

DEF

MNO

PQRS

MNO

ABC

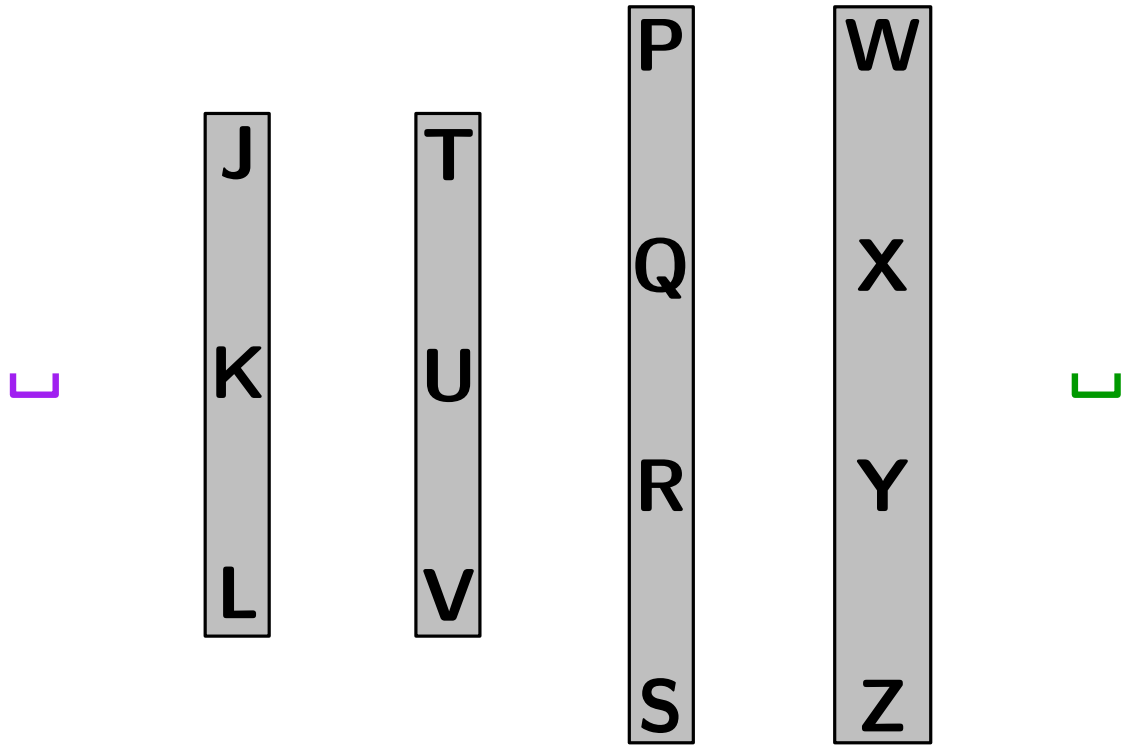
TUV

GHI

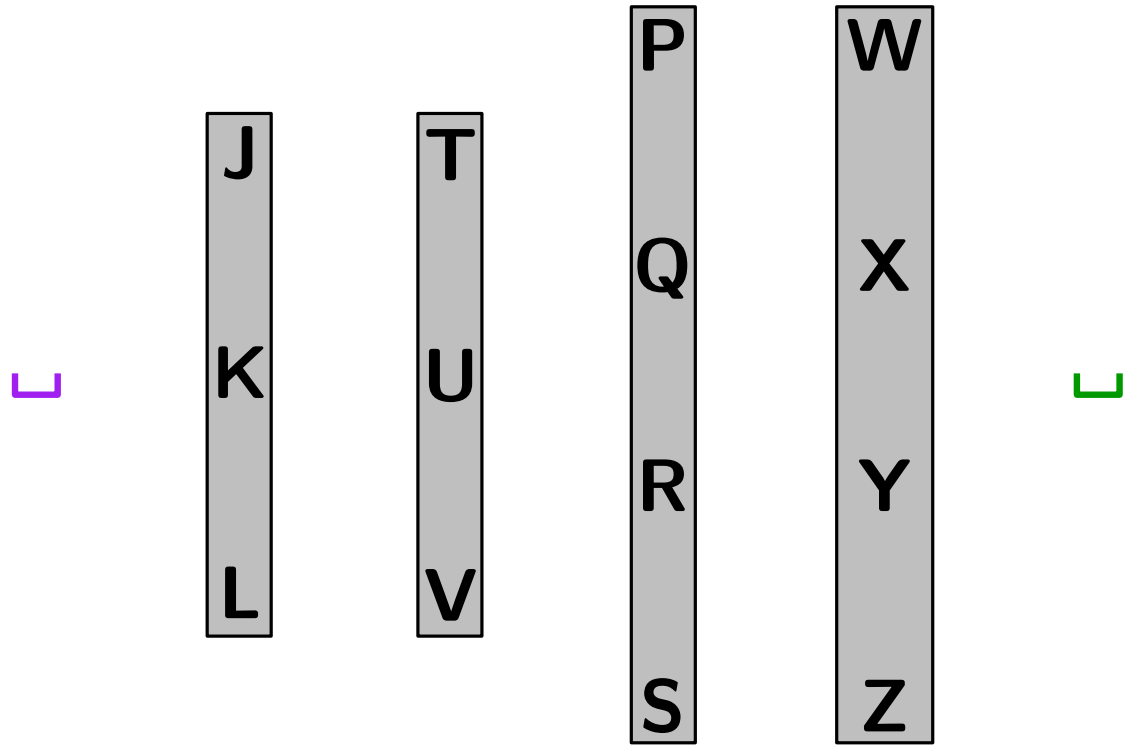
JKL

10:21 für T9

Modellierung – SMSen

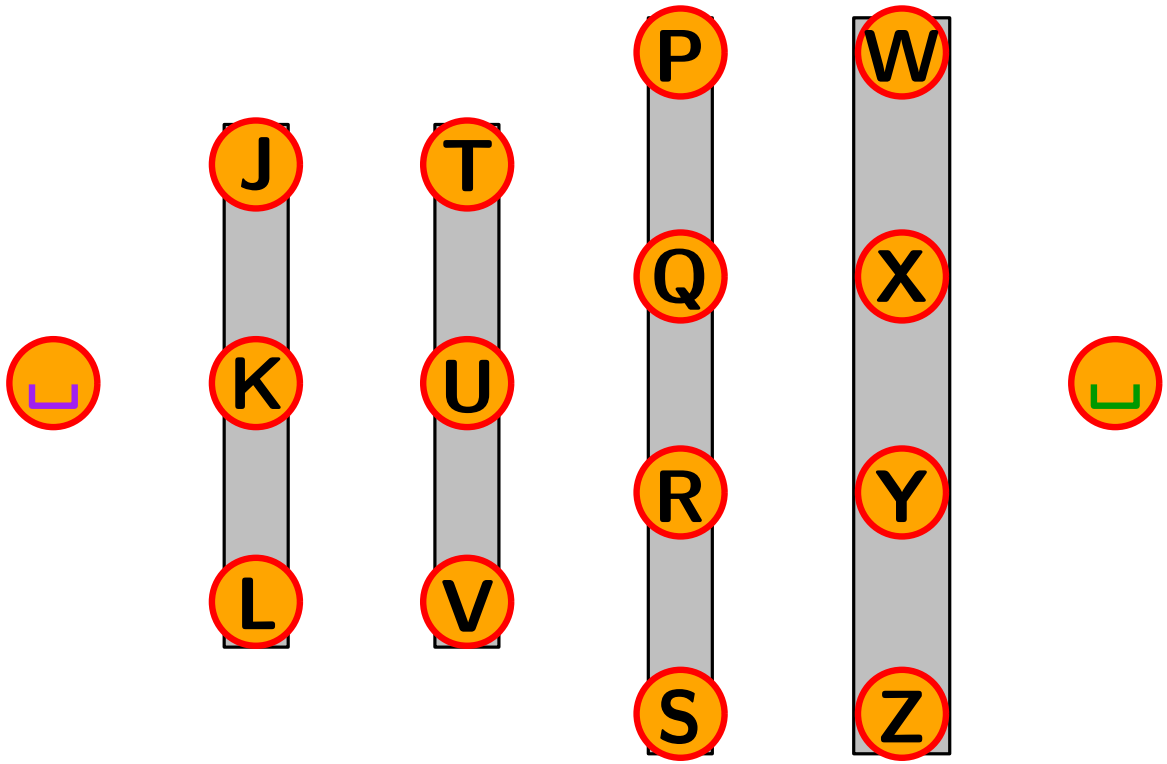


Modellierung – SMSen



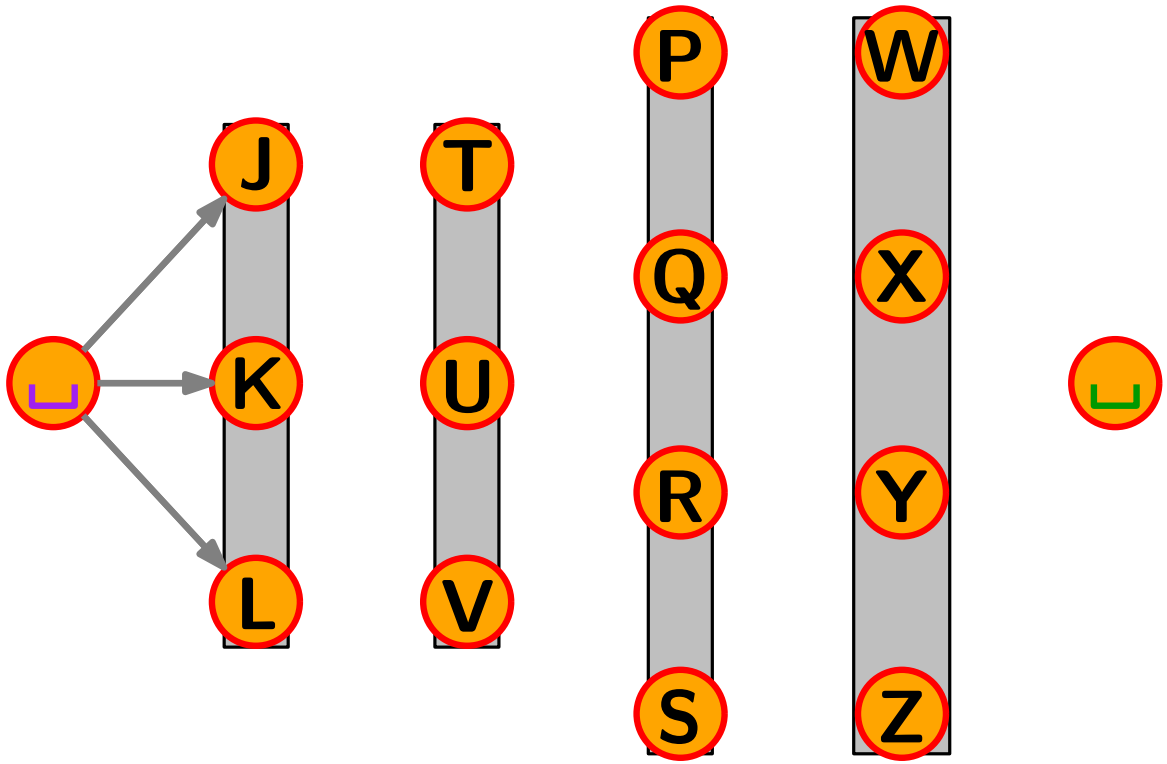
Graph:

Modellierung – SMSen



Graph: Knoten $\hat{=}$ Buchstaben

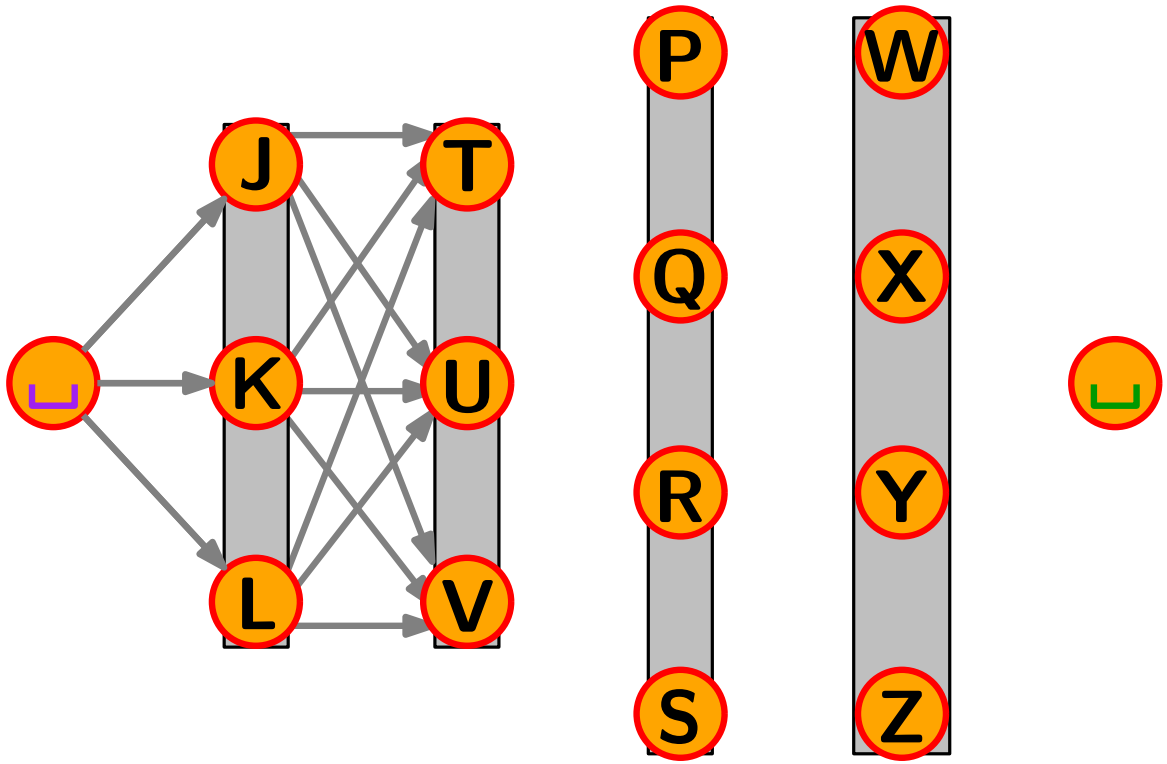
Modellierung – SMSen



Graph: Knoten $\hat{=}$ Buchstaben

Kanten $\hat{=}$ aufeinanderfolgende Buchst.

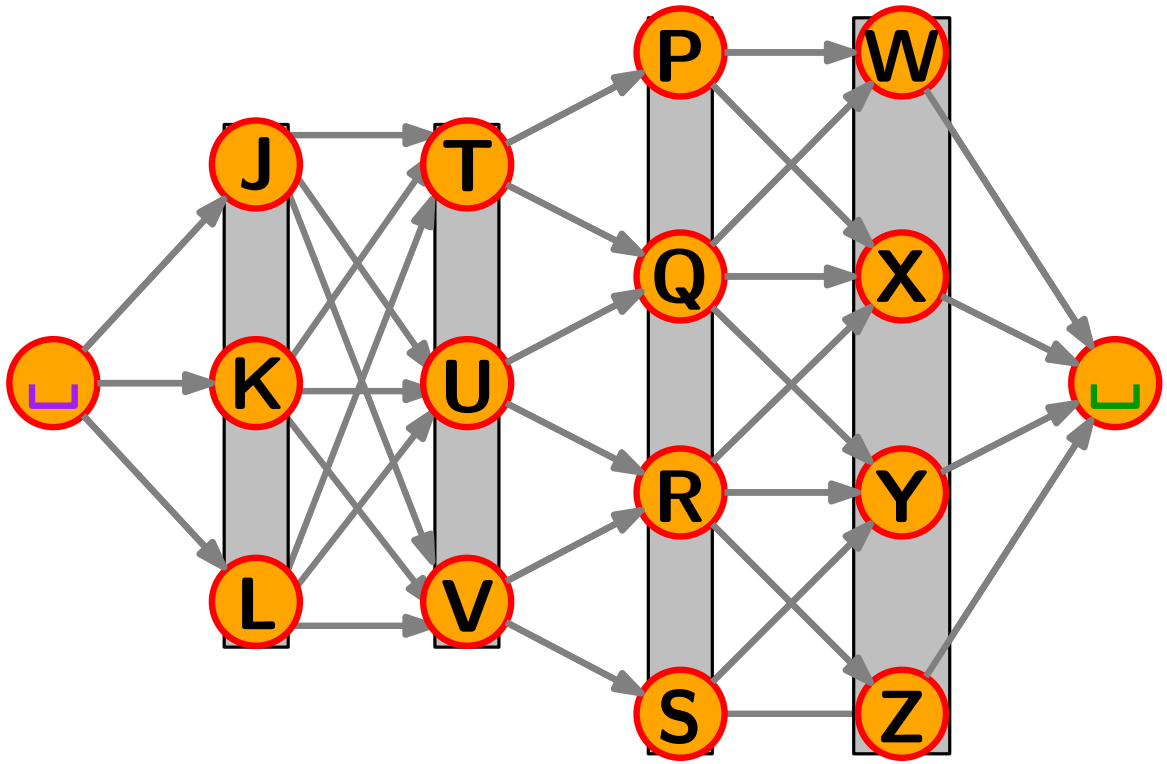
Modellierung – SMSen



Graph: Knoten $\hat{=}$ Buchstaben

Kanten $\hat{=}$ aufeinanderfolgende Buchst.

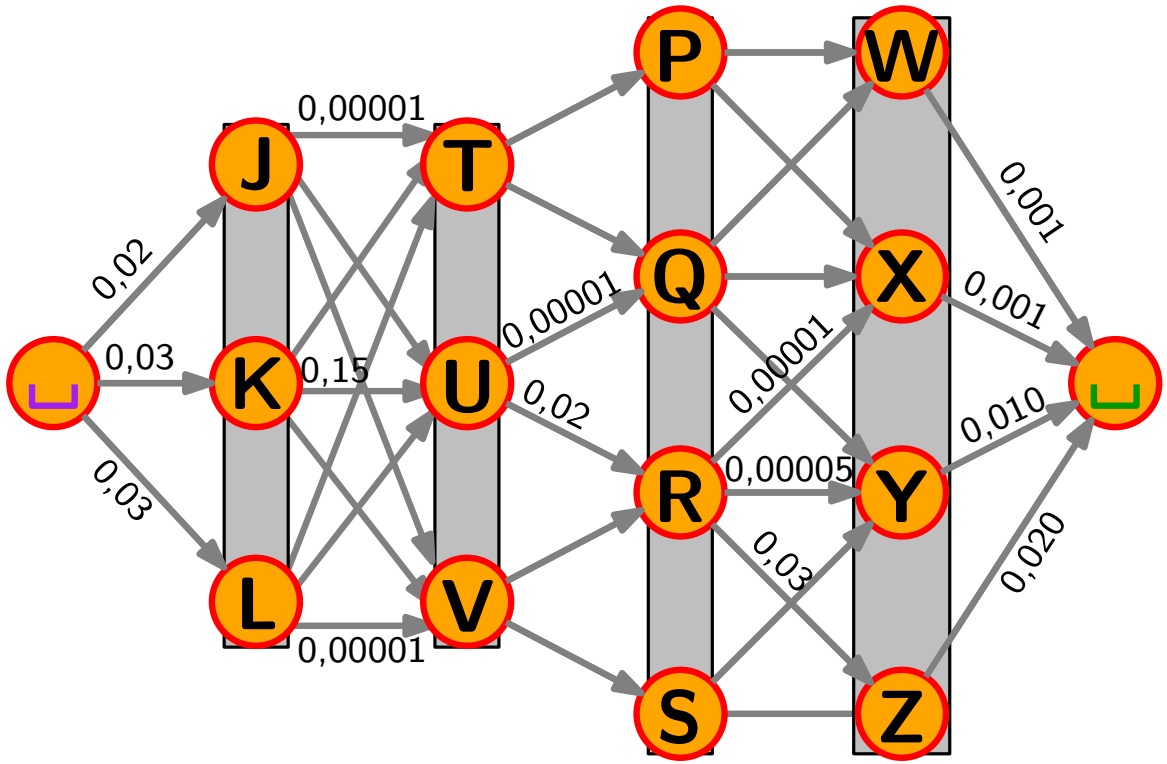
Modellierung – SMSen



Graph: Knoten $\hat{=}$ Buchstaben

Kanten $\hat{=}$ aufeinanderfolgende Buchst.

Modellierung – SMSen

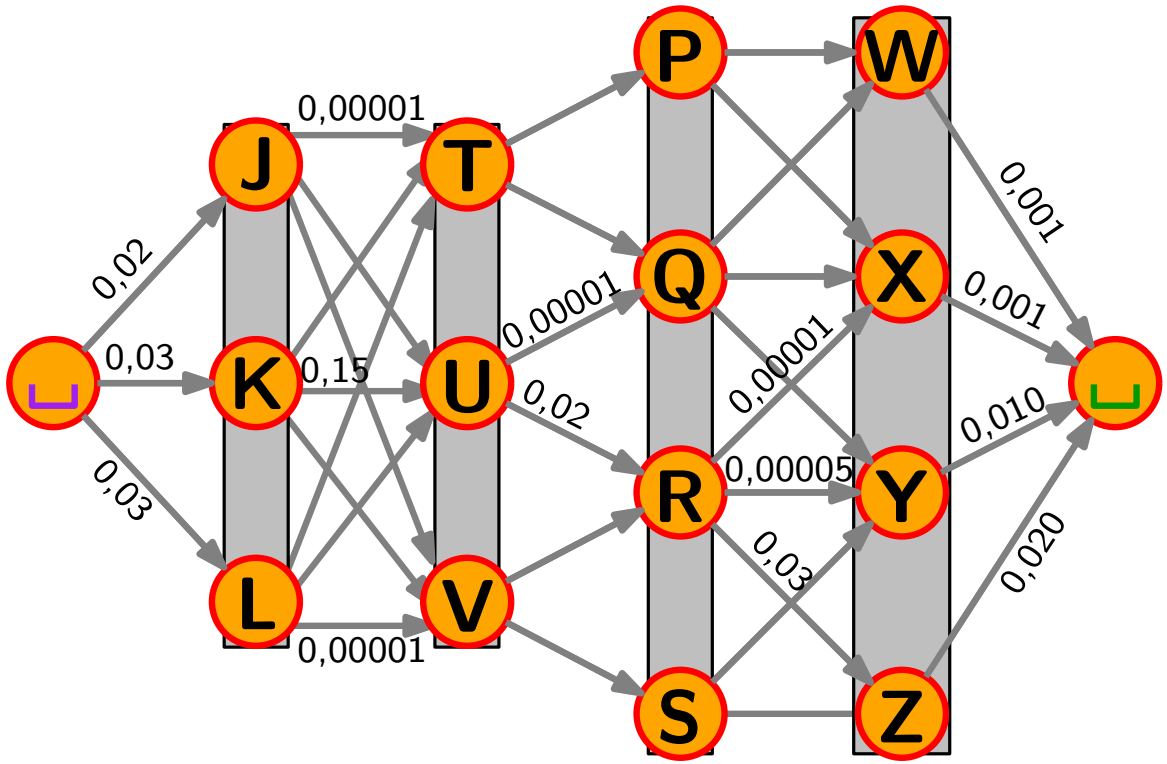


Graph: Knoten $\hat{=}$ Buchstaben

Kanten $\hat{=}$ aufeinanderfolgende Buchst.

Gewichte $\hat{=}$ Wahrscheinlichkeiten w

Modellierung – SMSen

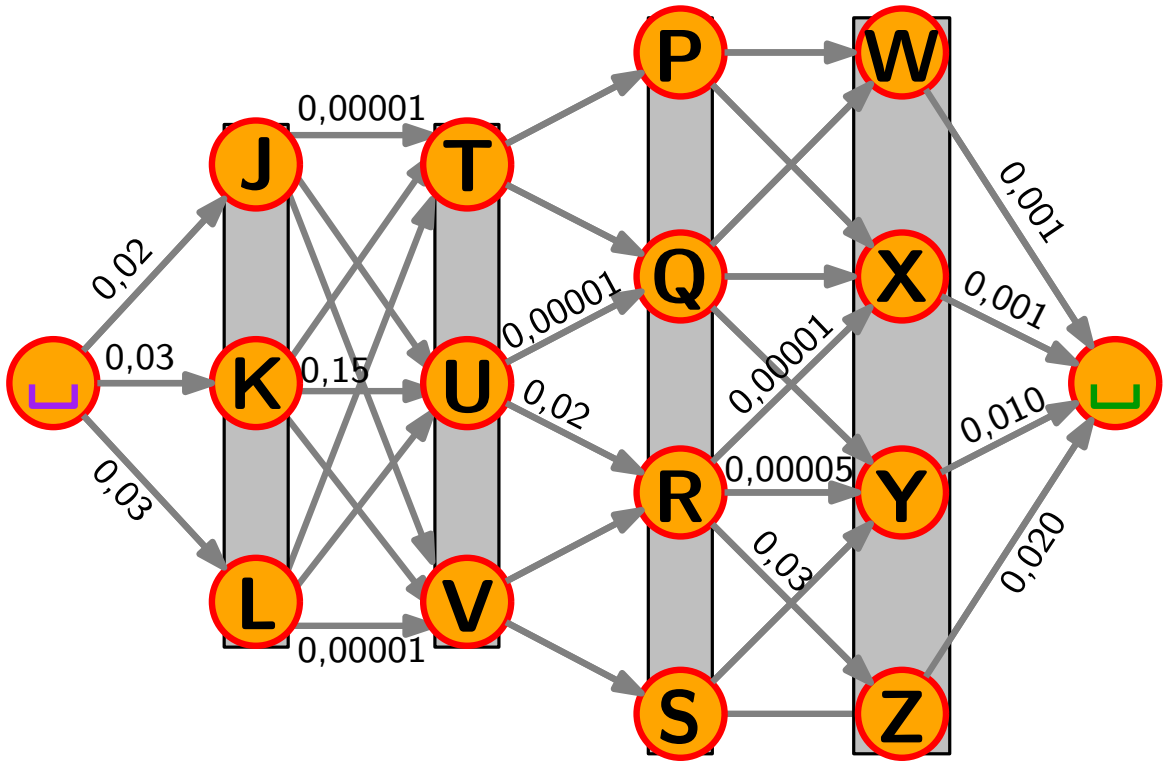


Graph: Knoten $\hat{=}$ Buchstaben

Kanten $\hat{=}$ aufeinanderfolgende Buchst.

Gewichte $\hat{=}$ Wahrscheinlichkeiten w / Häufigkeiten

Modellierung – SMSen



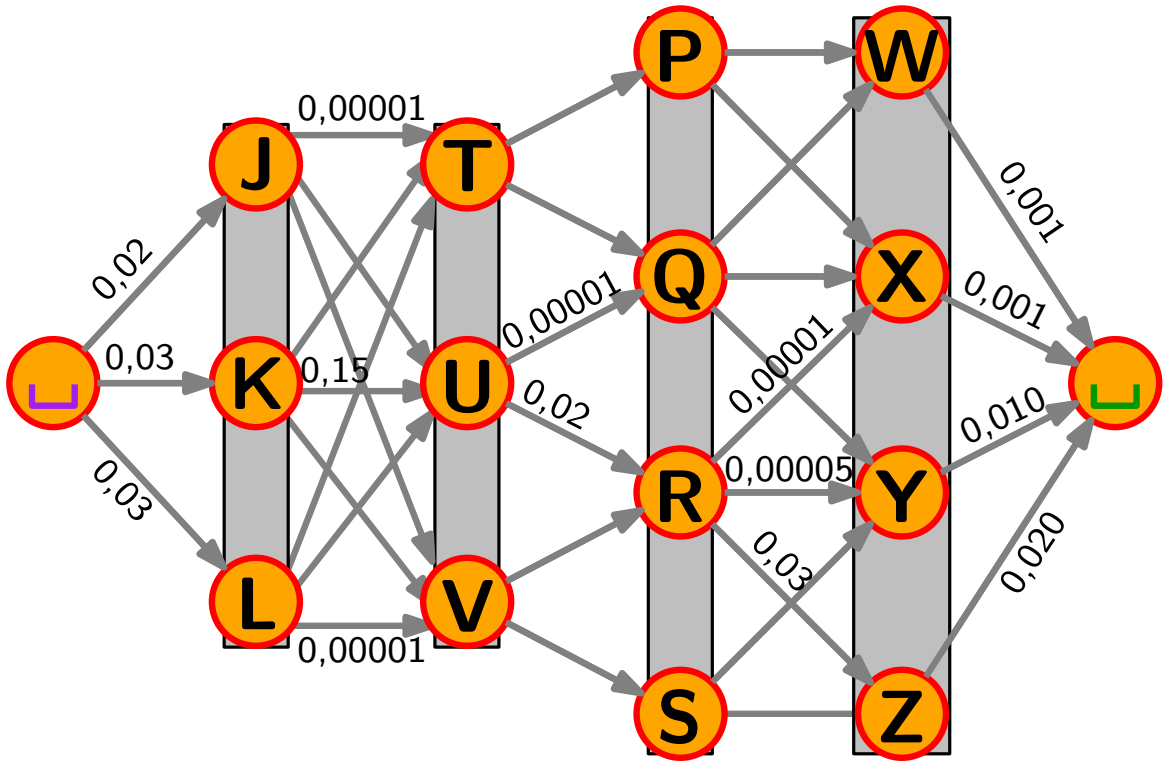
Graph: Knoten $\hat{=}$ Buchstaben

Kanten $\hat{=}$ aufeinanderfolgende Buchst.

Gewichte $\hat{=}$ Wahrscheinlichkeiten w / Häufigkeiten

Gesucht:

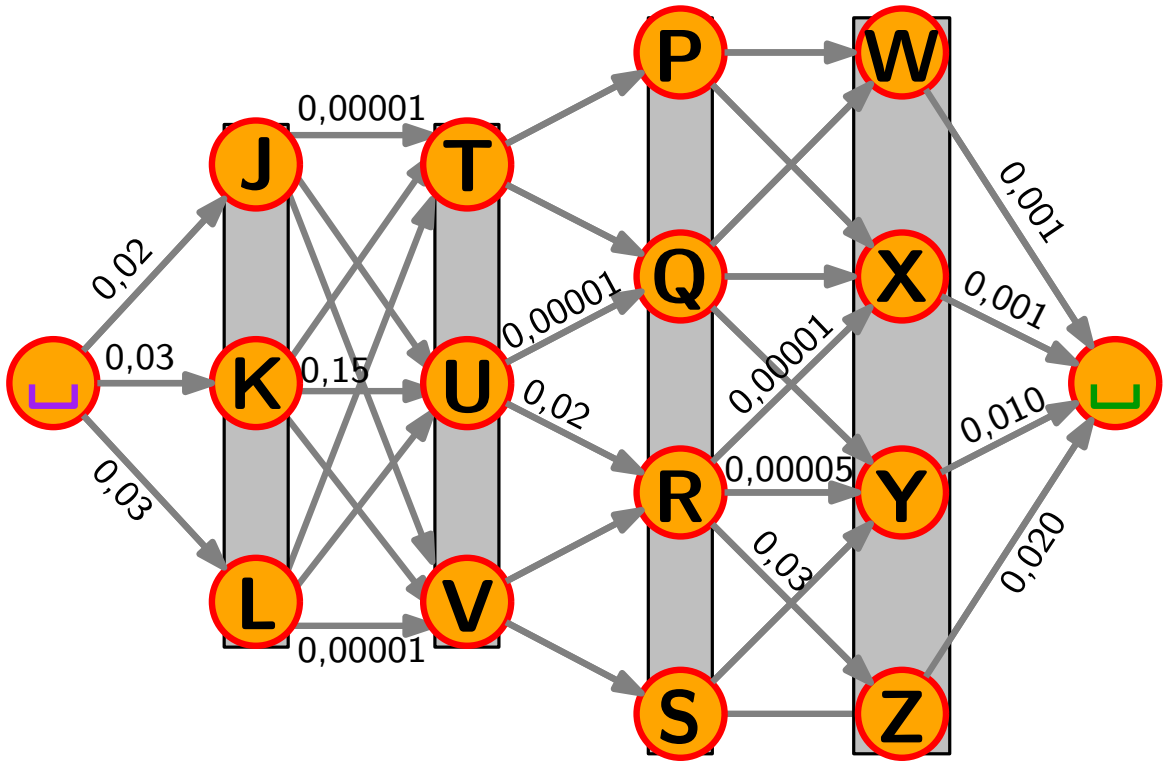
Modellierung – SMSen



Graph: Knoten $\hat{=}$ Buchstaben
Kanten $\hat{=}$ aufeinanderfolgende Buchst.
Gewichte $\hat{=}$ Wahrscheinlichkeiten w / Häufigkeiten

Gesucht: Weg P von ␣ nach ␣ mit *größter* WK

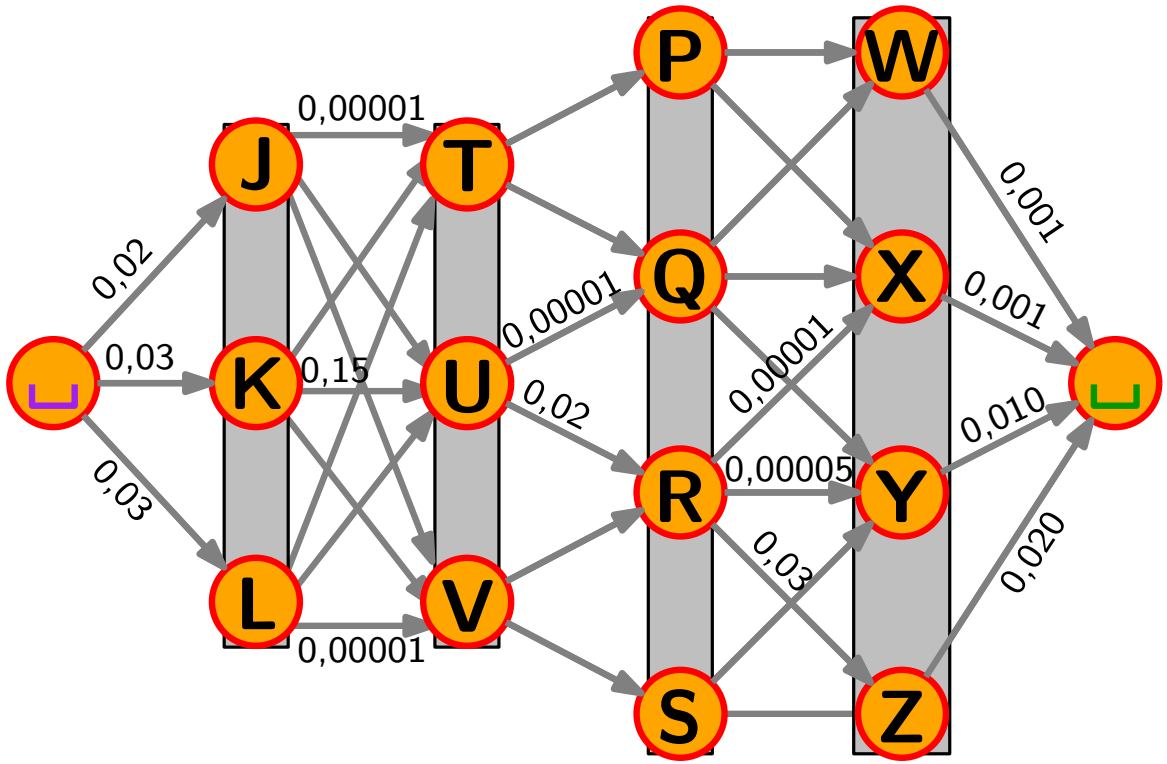
Modellierung – SMSen



Graph: Knoten $\hat{=}$ Buchstaben
Kanten $\hat{=}$ aufeinanderfolgende Buchst.
Gewichte $\hat{=}$ Wahrscheinlichkeiten w / Häufigkeiten

Gesucht: Weg P von ␣ nach ␣ mit *größter* WK $(= \prod_{e \in P} w(e))$

Modellierung – SMSen

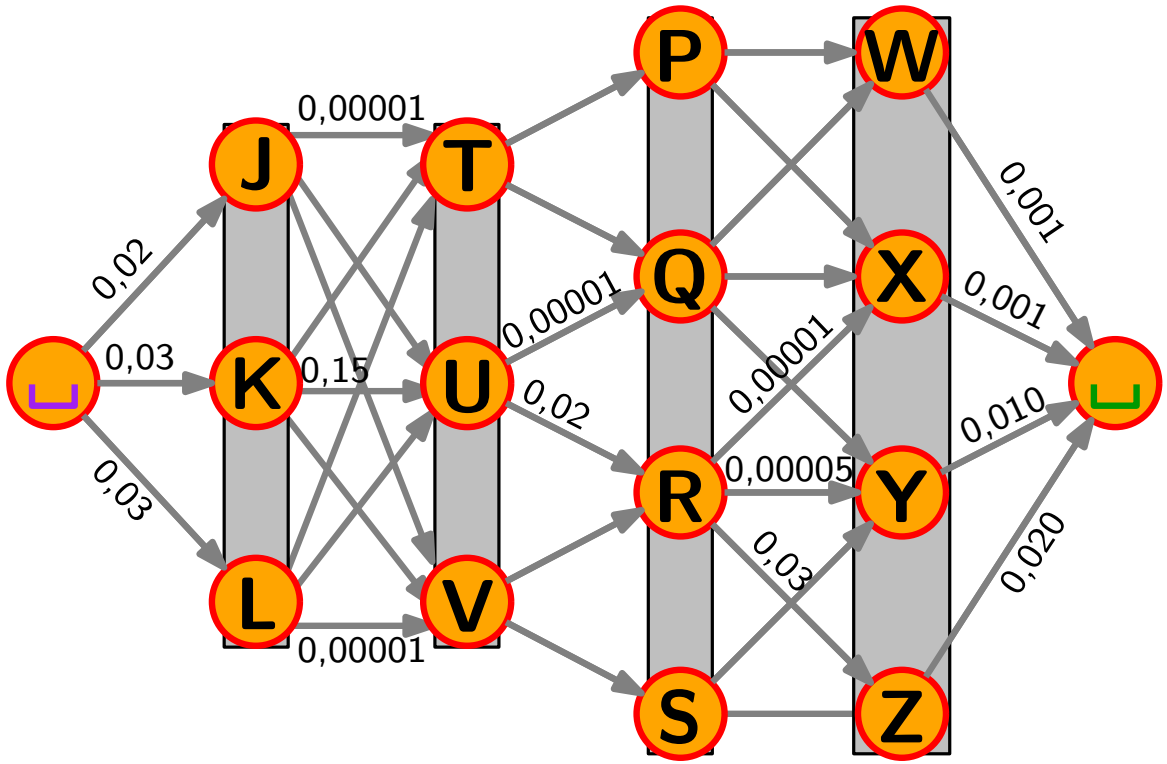


Graph: Knoten $\hat{=}$ Buchstaben
Kanten $\hat{=}$ aufeinanderfolgende Buchst.
Gewichte $\hat{=}$ Wahrscheinlichkeiten w / Häufigkeiten

Gesucht: Weg P von ␣ nach ␣ mit *größter* WK $(= \prod_{e \in P} w(e))$

Lösung: *dynamisches Programmieren...*

Modellierung – SMSen



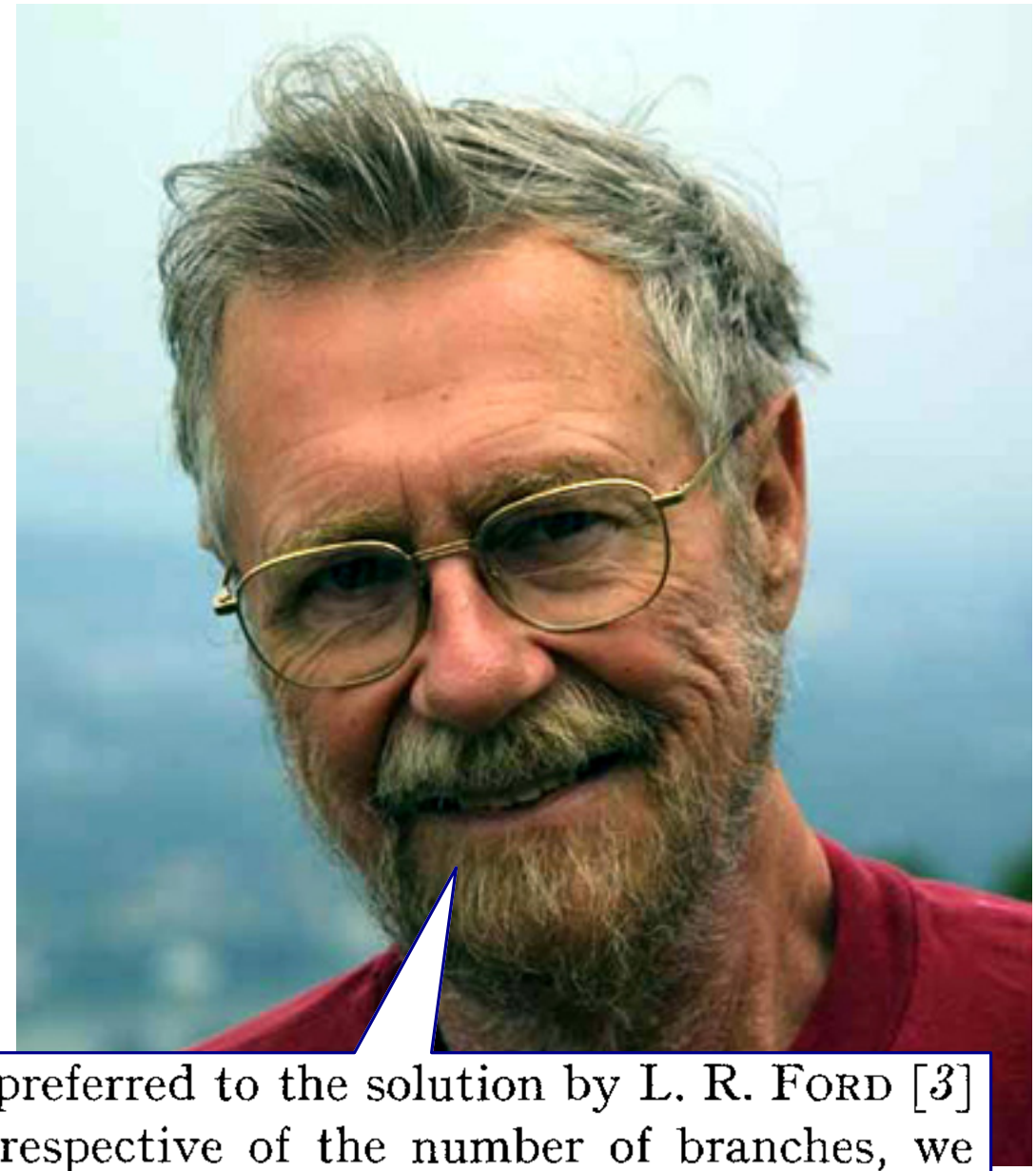
Graph: Knoten $\hat{=}$ Buchstaben
Kanten $\hat{=}$ aufeinanderfolgende Buchst.
Gewichte $\hat{=}$ Wahrscheinlichkeiten w / Häufigkeiten

Gesucht: Weg P von ␣ nach ␣ mit *größter* WK $(= \prod_{e \in P} w(e))$

Lösung: *dynamisches Programmieren. . .* [*kommt noch!*]

Literatur

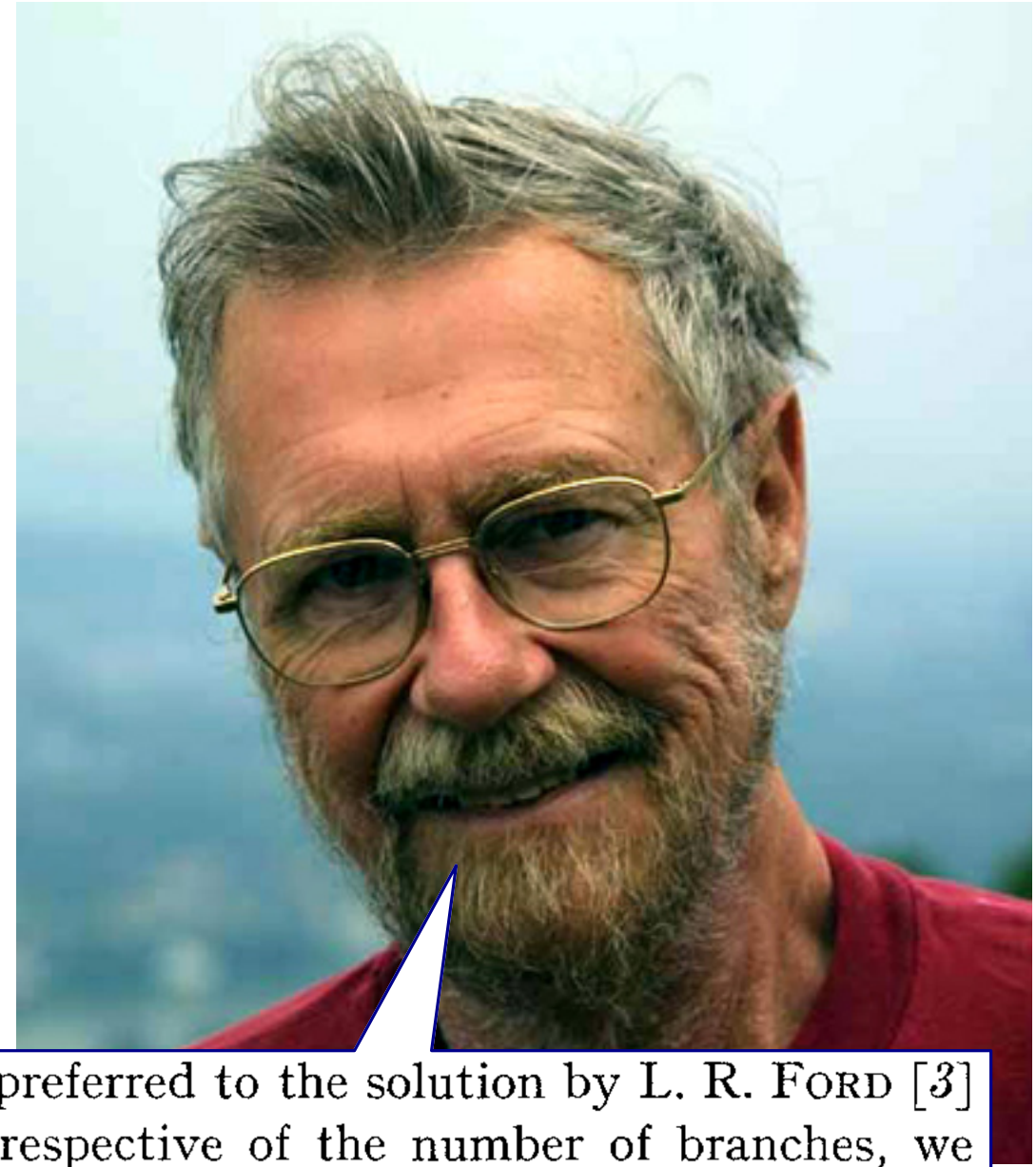
- **A note on two problems in connexion with graphs.**
  Edsger Wybe Dijkstra:
  Numerische Mathematik,
  Band 1, S. 269–271, 1959.



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.

Literatur

- **A note on two problems in connexion with graphs.**
  Edsger Wybe Dijkstra:
  Numerische Mathematik,
  Band 1, S. 269–271, 1959.

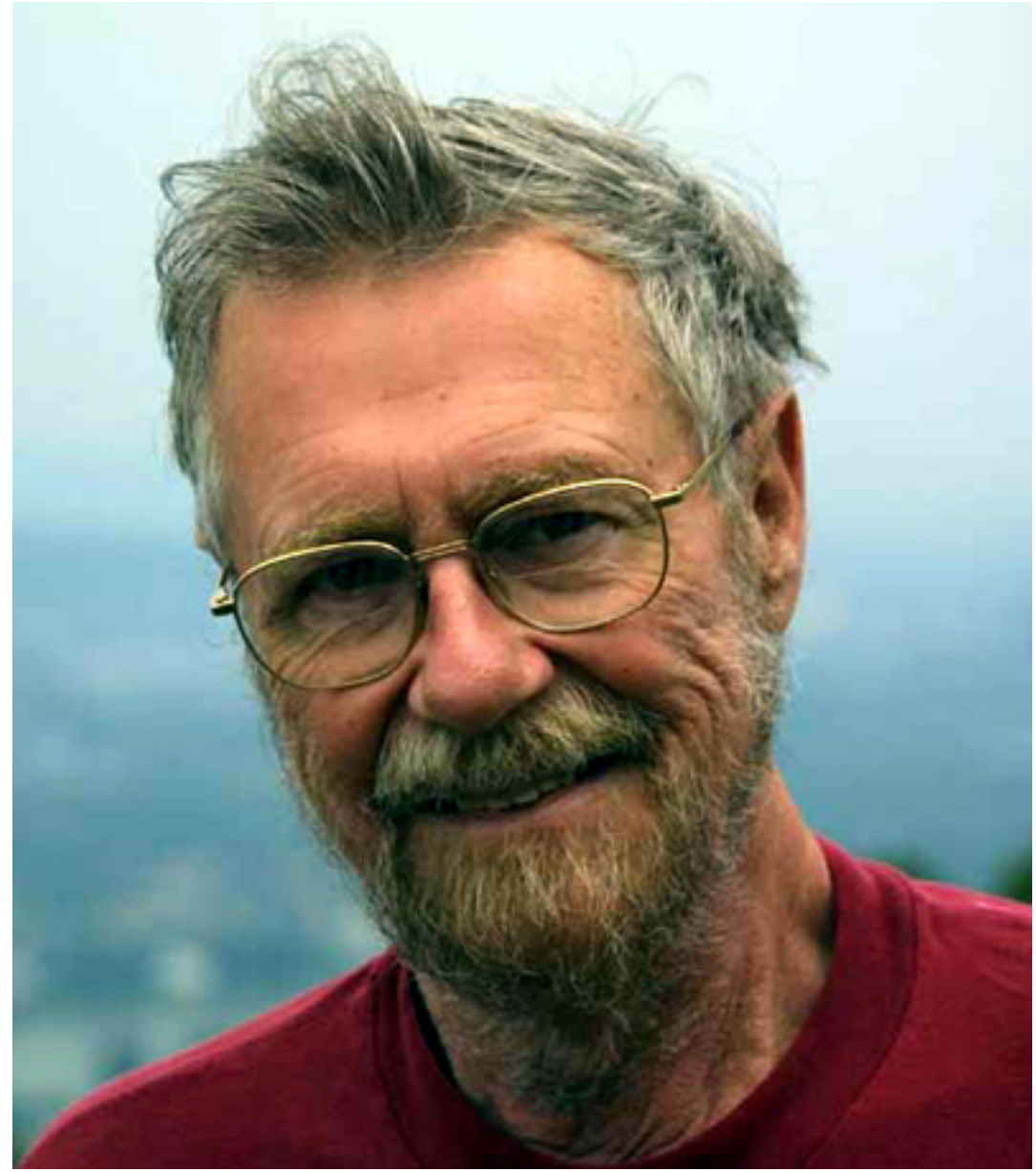


The solution given above is to be preferred to the solution by L. R. FORD [3] as described by C. BERGE [4], for, irrespective of the number of branches, we need not store the data for all branches simultaneously but only those for the branches in sets I and II, and this number is always less than n.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

Literatur

- **A note on two problems in connexion with graphs.**
  Edsger Wybe Dijkstra:
  Numerische Mathematik,
  Band 1, S. 269–271, 1959.

- **Das Geheimnis des kürzesten Weges.**
  Ein mathematisches Abenteuer.
  Peter Gritzmann und
  René Brandenberg:
  Springer-Verlag, 3. Aufl., 2005.



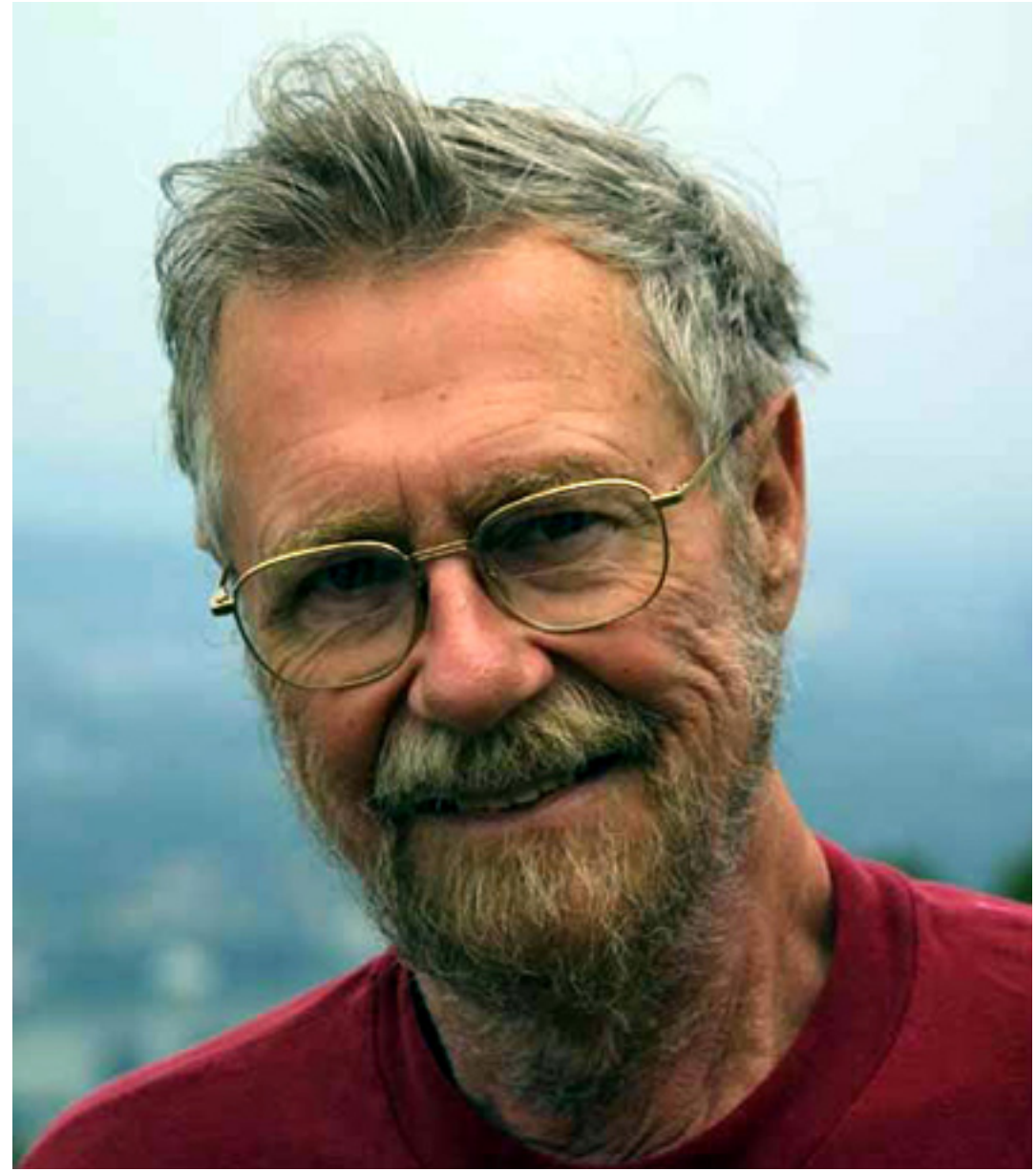
Edsger Wybe Dijkstra
* 1930 in Rotterdam
† 2002 in Nuenen, Niederlande

Literatur

- **A note on two problems in connexion with graphs.**
  Edsger Wybe Dijkstra:
  Numerische Mathematik,
  Band 1, S. 269–271, 1959.

- **Das Geheimnis des kürzesten Weges.**
  Ein mathematisches Abenteuer.
  Peter Gritzmann und
  René Brandenberg:
  Springer-Verlag, 3. Aufl., 2005.

Beide Werke sind über die UB *frei zugänglich* und über unsere WueCampus-Seite verlinkt!



Edsger Wybe Dijkstra
* 1930 in Rotterdam
† 2002 in Nuenen, Niederlande

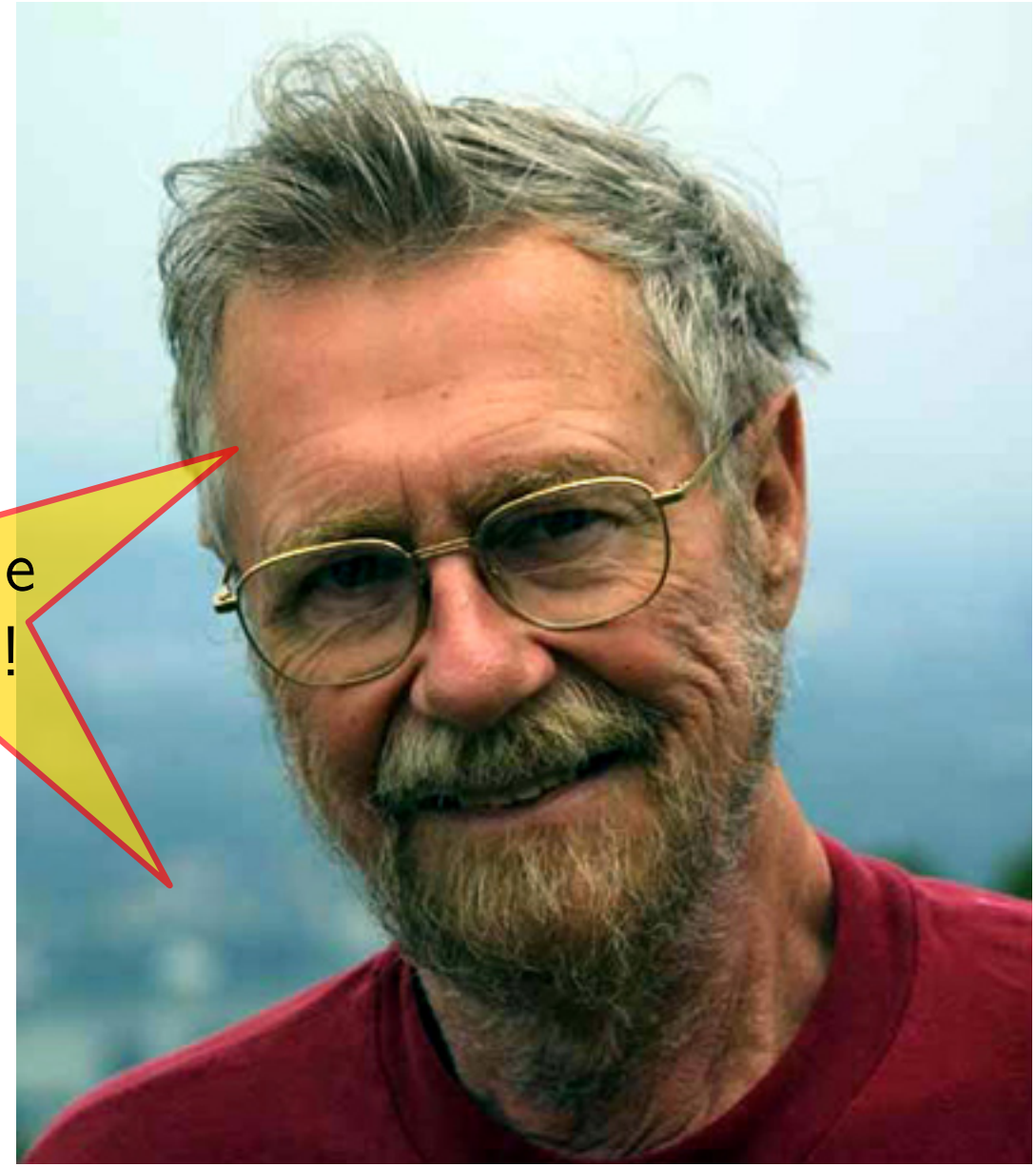
Literatur

- **A note on two problems in connexion with graphs.**
  Edsger Wybe Dijkstra:
  Numerische Mathematik,
  Band 1, S. 269–271, 1959.

- **Das Geheimnis des kürzesten Weges.**
  Ein mathematisches Abenteuer.
  Peter Gritzmann und
  René Brandenberg:
  Springer-Verlag, 3. Aufl., 2005.

Lesen Sie mal rein!

Beide Werke sind über die UB *frei zugänglich* und über unsere WueCampus-Seite verlinkt!



Edsger Wybe Dijkstra
* 1930 in Rotterdam
† 2002 in Nuenen, Niederlande

Kürzeste Wege nach Dijkstra

Eingabe	*Algorithmus*	*Laufzeit*
ungewichteter Graph	Breitensuche	$O(E+V)$ *letztes Mal* ✓
nicht-neg. Kantengew.	Dijkstra	$O(E+V\log V)$
azyklischer Graph	topol. Sortieren	$O(E+V)$
negative Kantengew.	Bellman-Ford	$O(EV)$
für alle Knotenpaare	$\lvert V\rvert \times$ Dijkstra	$O(V(E+V\log V))$
+ negative Kantengew.	Floyd-Warshall	$O(V^3)$
	Johnson	$O(V(E+V\log V))$
k kürzeste s-t-Wege	Eppstein	$O(k+E+V\log V)$

Kürzeste Wege nach Dijkstra

Eingabe	*Algorithmus*	*Laufzeit*	
ungewichteter Graph	Breitensuche	$O(E+V)$	*letztes Mal* ✓
nicht-neg. Kantengew.	Dijkstra	$O(E+V\log V)$	✓
azyklischer Graph	topol. Sortieren	$O(E+V)$	
negative Kantengew.	Bellman-Ford	$O(EV)$	
für alle Knotenpaare	$\lvert V\rvert \times$ Dijkstra	$O(V(E+V\log V))$	
+ negative Kantengew.	Floyd-Warshall	$O(V^3)$	
	Johnson	$O(V(E+V\log V))$	
k kürzeste s-t-Wege	Eppstein	$O(k+E+V\log V)$	

Kürzeste Wege nach Dijkstra

Eingabe	*Algorithmus*	*Laufzeit*	
ungewichteter Graph	Breitensuche	$O(E+V)$	*letztes Mal* ✓
nicht-neg. Kantengew.	Dijkstra	$O(E+V\log V)$	✓
azyklischer Graph	topol. Sortieren	$O(E+V)$	*nächstes Mal!* ✓
negative Kantengew.	Bellman-Ford	$O(EV)$	
für alle Knotenpaare	$\lvert V\rvert \times$ Dijkstra	$O(V(E+V\log V))$	
+ negative Kantengew.	Floyd-Warshall	$O(V^3)$	
	Johnson	$O(V(E+V\log V))$	
k kürzeste s-t-Wege	Eppstein	$O(k+E+V\log V)$	

Kürzeste Wege nach Dijkstra

Eingabe	*Algorithmus*	*Laufzeit*	
ungewichteter Graph	Breitensuche	$O(E+V)$	*letztes Mal* ✓
nicht-neg. Kantengew.	Dijkstra	$O(E+V\log V)$	✓
azyklischer Graph	topol. Sortieren	$O(E+V)$	*nächstes Mal!* ✓
negative Kantengew.	Bellman-Ford	$O(EV)$	Vorlesung Adv. Algorithms (M.Sc.)
für alle Knotenpaare	$\lvert V\rvert \times$ Dijkstra	$O(V(E+V\log V))$	
+ negative Kantengew.	Floyd-Warshall	$O(V^3)$	
	Johnson	$O(V(E+V\log V))$	
k kürzeste s-t-Wege	Eppstein	$O(k+E+V\log V)$	

Kürzeste Wege nach Dijkstra

Eingabe	*Algorithmus*	*Laufzeit*	
ungewichteter Graph	Breitensuche	$O(E+V)$	*letztes Mal* ✓
nicht-neg. Kantengew.	Dijkstra	$O(E+V\log V)$	✓
azyklischer Graph	topol. Sortieren	$O(E+V)$	*nächstes Mal!* ✓
negative Kantengew.	Bellman-Ford	$O(EV)$	Vorlesung Adv. Algorithms (M.Sc.)
für alle Knotenpaare	$\lvert V\rvert \times$ Dijkstra	$O(V(E+V\log V))$	✓
+ negative Kantengew.	Floyd-Warshall	$O(V^3)$	
	Johnson	$O(V(E+V\log V))$	
k kürzeste s-t-Wege	Eppstein	$O(k+E+V\log V)$	