

# Kap. 6.6: Kürzeste Wege



Professor Dr. Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

Fakultät für Informatik, TU Dortmund

21./22. VO DAP2 SS 2009 7./9. Juli 2009

# Nachtest für Ausnahmefälle

- **Di 14. Juli 2009, 16:00 Uhr, OH14, R. 202**
- Anmeldung bis 9. Juli erforderlich via Email an Petra Mutzel (Nicola Beume ist nicht da)
- Stoff: Alles bis inkl. Hashing (inkl. 1.-10. Übungsblatt)
- Teilnahmevoraussetzungen:
  - Attest/Entschuldigt beim 1. oder 2. Übungstest
  - oder besondere Härtefälle anerkannt in meiner Sprechstunde Di 7.7.09, 14:15 Uhr-15:15 Uhr oder via email bis 9.7.09

# Kürzeste Wege - Überblick

- Single-source-shortest-path (SSSP)
  - Algorithmus von ***Dijkstra***
- All-pair-shortest-paths
  - Algorithmus von Floyd-Warshall

# Motivation

„Was gibt es heute Besonderes?“

Dijkstra oder Wie funktioniert mein Navi?

„Und wenn mir das zu schwer ist?“

Einfacher Algorithmus für APSP

# Kürzeste Wege

Achtung: in diesem Abschnitt  
**gerichtete gewichtete** Graphen!



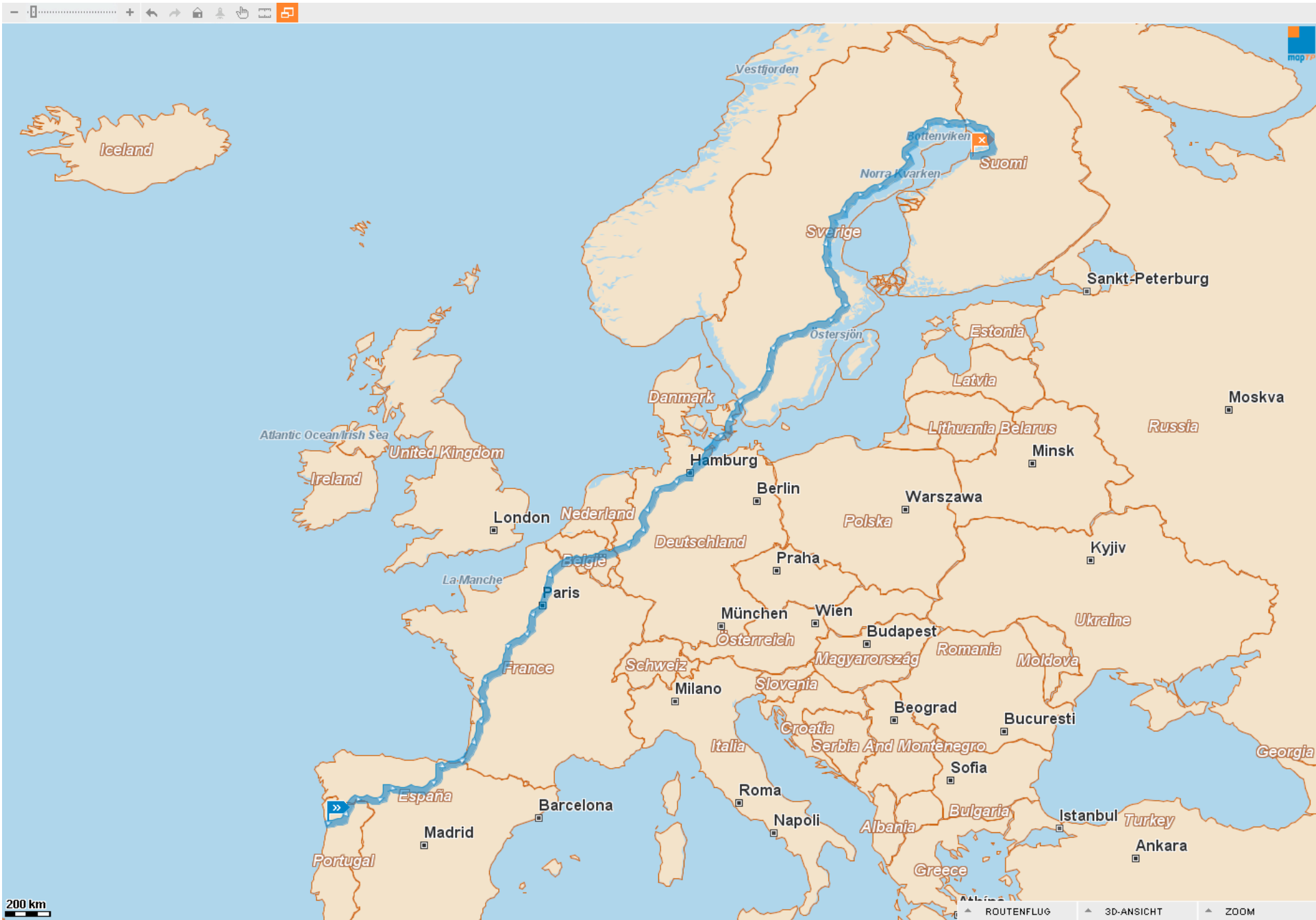
## Kürzeste Wege Problem

*Gegeben:* gerichteter Graph  $G=(V,A)$

Gewichtsfunktion  $w : A \rightarrow \mathbb{R}$  hier:  $w \geq 0$

Keine Mehrfachkanten

*Gesucht:* Der bzgl. Gewicht  $w$  kürzeste Weg von Startknoten zu Zielknoten.



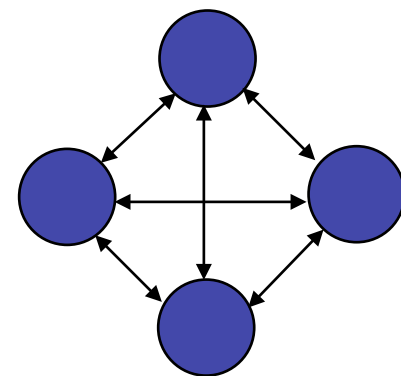
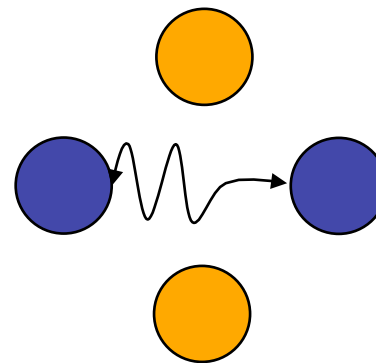
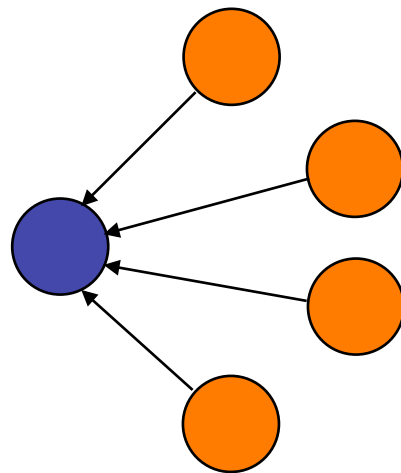
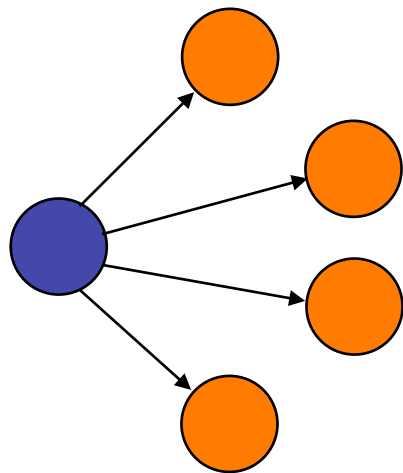
# Anwendungen

Direkte und indirekte (Teilproblem) Anwendungen:

- Routenplaner (Streckenlänge)
- Auskunftssysteme für Bus und Bahn (Zeit)
- Berechnung minimaler Flüsse in Netzwerken
- DNA Sequenz Analyse
- ...

# Kürzeste Wege Probleme

- Single Source Shortest Path (SSSP) ←
- Single Destination Shortest Path
- Single Pair Shortest Path
- All Pairs Shortest Path (APSP) ←





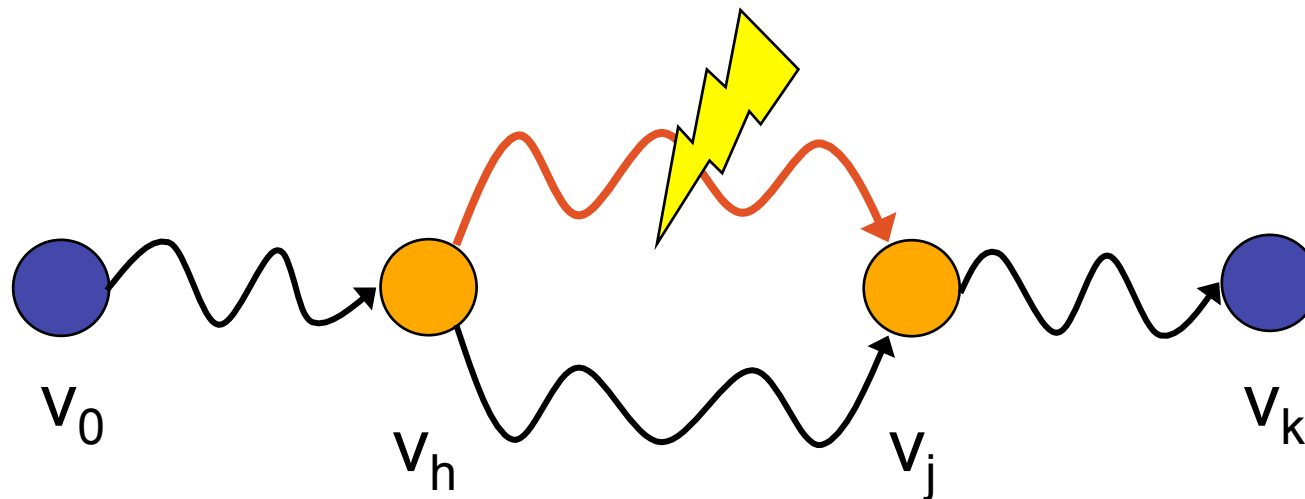
# 6.1 Single Source Shortest Path (SSSP)

- Bekannt: BFS für ungewichtete kürzeste Wege (USSSP)
- **Jetzt:** Kürzeste Wege mit Kantengewichten:
  - Gerichteter Graph  $G = (V, A)$
  - Kantengewichte  $w(e) \in \mathbf{R}$  (Strecke, Fahrzeit)
  - **Startknoten**  $s$
  - Weglänge  $w(p) := \sum_{i=1, \dots, k} w(e_i)$  für  $p = v_0, e_1, \dots, e_k, v_k$

# Optimale Substruktur

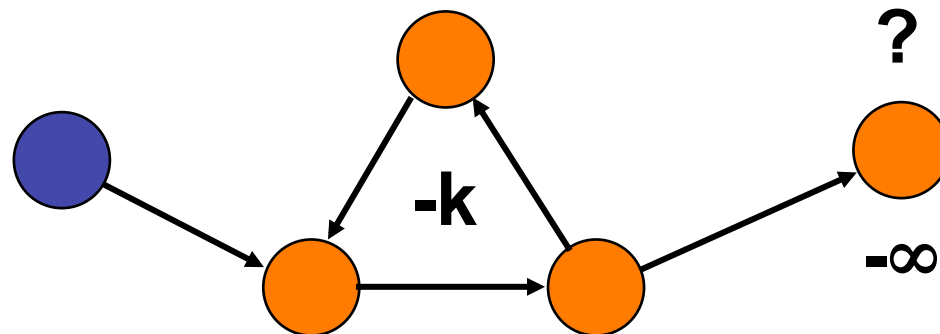
Pfad über  $v_0=s, v_1, \dots, v_k$  kürzester Weg von  $s$   
nach  $v_k$

$\Rightarrow$  Teilweg  $v_h \rightarrow v_j$  kürzester Weg von  $v_h$  nach  $v_j$



# Single Source Shortest Path

Keine negativen Kreise!!



⇒ Kürzeste Wege von Knoten  $s$  bilden immer einen Baum

Spezialfall:  $w(e) \geq 0$  (Strecke, Fahrtzeit)

# Algorithmus von Dijkstra

Analogien mit BFS und Prim:

- *BFS/Prim* lässt *einzelnen* Baum wachsen: Neue Kante verbindet Baum mit Rest
- *Dijkstra* bildet **Kürzeste-Wege Baum (SPT)** ausgehend von Wurzel  $s$ , aber andere Auswahl
- Greedy, ADS: Priority Queue

Bezeichnungen

- Vorgänger von Knoten  $v$  im SPT:  $\pi[v]$
- Kanten, die Knotenmenge  $S$  verlassen:  $A(S)$
- Min. Abstand vom Startknoten zu Knoten  $v$ :  $d[v]$

# Idee des Dijkstra-Algorithmus

1.  $S := \{s\}$  //  $S$  Knoten im SPT
2.  $d[s] := 0; \pi[s] := nil$  //  $s$  Wurzel
3. **while**  $A(S) \neq \emptyset$  **do** // erreichbare Knoten
4. // Optimale Substruktur  
Sei  $e = (u, v) \in A(S)$  mit  $d[u] + w(e)$  minimal  
genauer: s. später:  
PQ und edge scanning
5.  $S := S \cup \{v\}$
6.  $d[v] := d[u] + w(e)$
7.  $\pi[v] := u$
8. **end while**

# Korrektheit

- Knoten ausserhalb von  $S$  sind nur über Knoten bzw. Pfade in  $S$  erreichbar
- Dijkstra wählt Minimum der Weglänge aus  $S$  hinaus und erweitert  $S$
- $w$  nicht-negativ: Spätere Weglängen können nur größer sein als Minimum!
- Optimale Substruktur: Kürzester Weg besteht aus kürzestem Weg plus Kante

# Korrektheit

Induktion über die Kanten  $e_1, \dots, e_n$ , die der Algorithmus für  $v$  wählt,  $e_i = (u_i, v_i)$ ,  $v_0 := s$ .

Zeige:  $v_0, \dots, v_i$  ist kürzester Weg von  $v_0$  nach  $v_i$

$i=0$ : Keine Kante, korrekt

$1 \leq i \leq n$ : Annahme  $v_0, \dots, v_j$  kürzester Weg für  $j < i$ .

Dijkstra wählt Weg mit Kosten  $d[u_i] + w(e_i)$ .

Jeder andere Weg  $v_0 \rightarrow v_i$  beginnt mit Knoten aus der Menge  $V_i := \{v_0, \dots, v_{i-1}\}$  gefolgt von einer Kante  $(x, y) \in A(V_{i-1})$ . Alleine das Wegstück von  $v_0$  nach  $y$

hat aber bereits mindestens Kosten

$d[u_i] + w(e_i)$ , da  $e_i$  als Minimum gewählt wurde.

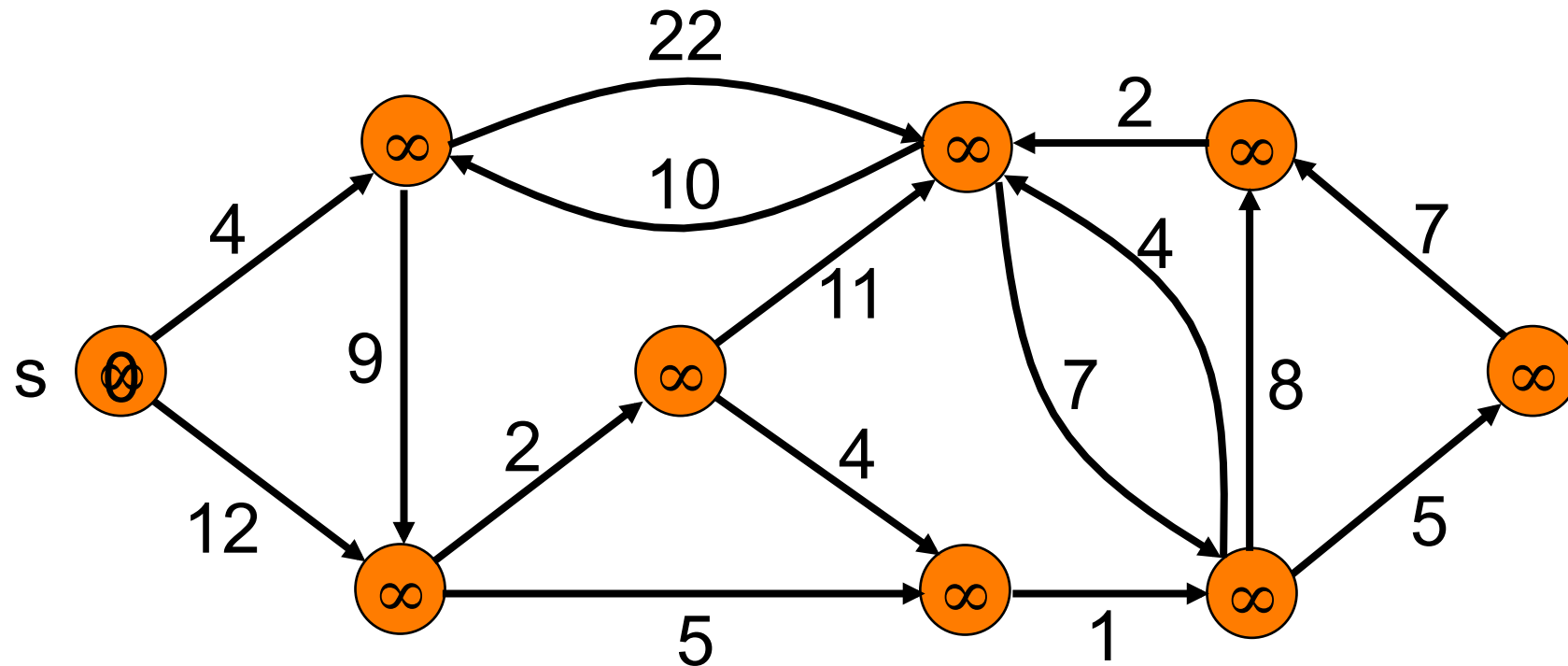
- Alle erreichbaren Knoten werden auch erreicht

# Realisierung von Dijkstra

- Knotenmenge  $S$ : Kürzester Weg mit Länge  $d[v]$  bereits ermittelt
  - Knotenmenge  $V \setminus S$ : Speichere die Knoten mit der Priorität „*vorläufige* Werte für den Abstand zu  $s$ “ (obere Schranke  $\delta(v) \geq d[v]$ ) in *Priority Queue*, und aktualisiere die Priorität, falls ein günstigerer Weg gefunden wird,  $\delta(s) = 0$
1. Wähle mit **EXTRACTMIN** Knoten  $u$  mit minimalem Abstandswert  $\delta(u)$ . Damit gilt  $d(u) = \delta(u)$ . Start:  $\delta(s) = d[s] = 0$
  2. Aktualisiere für ausgehende Kanten  $(u, v)$  Abstandswerte der Endknoten  $\delta(v)$  (*edge scanning*, relaxieren) mit **DECREASEPRIORITY** und Vorgänger  $\pi[v]$

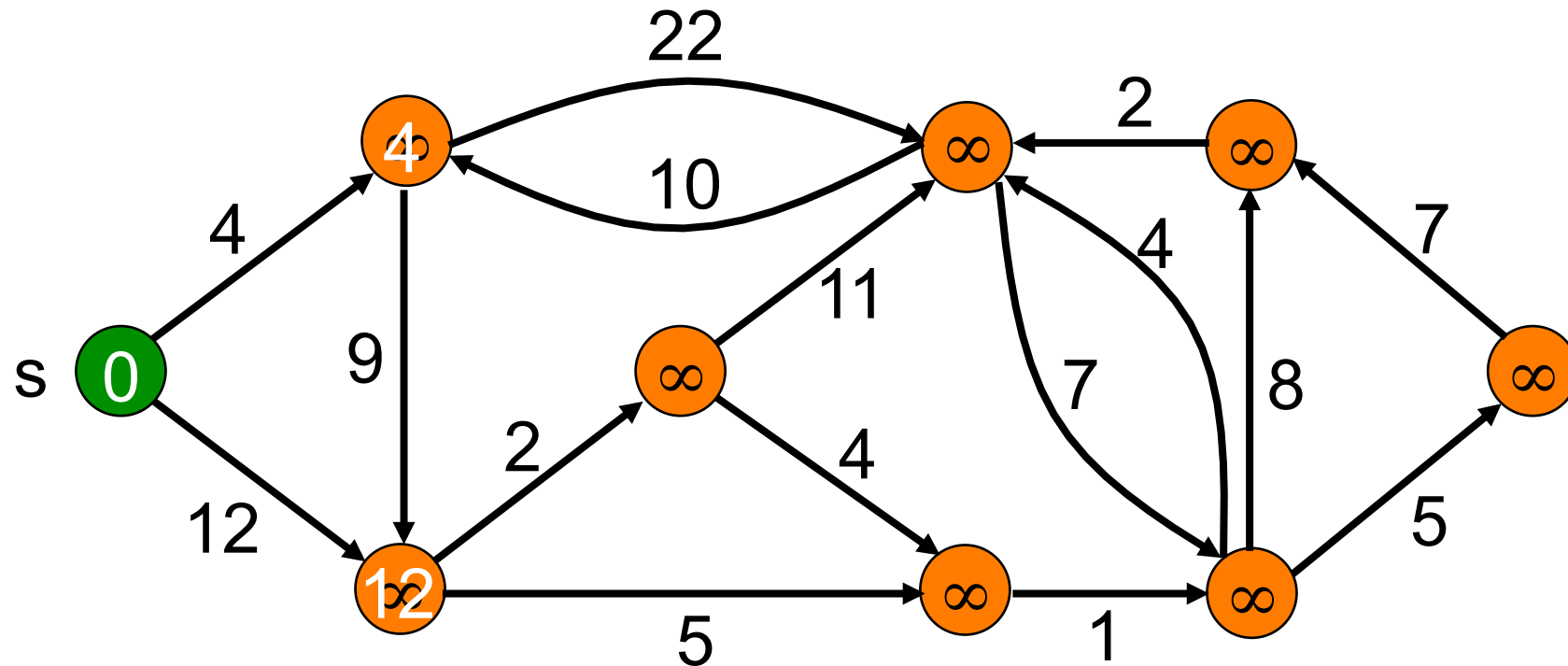


# Algorithmus von Dijkstra



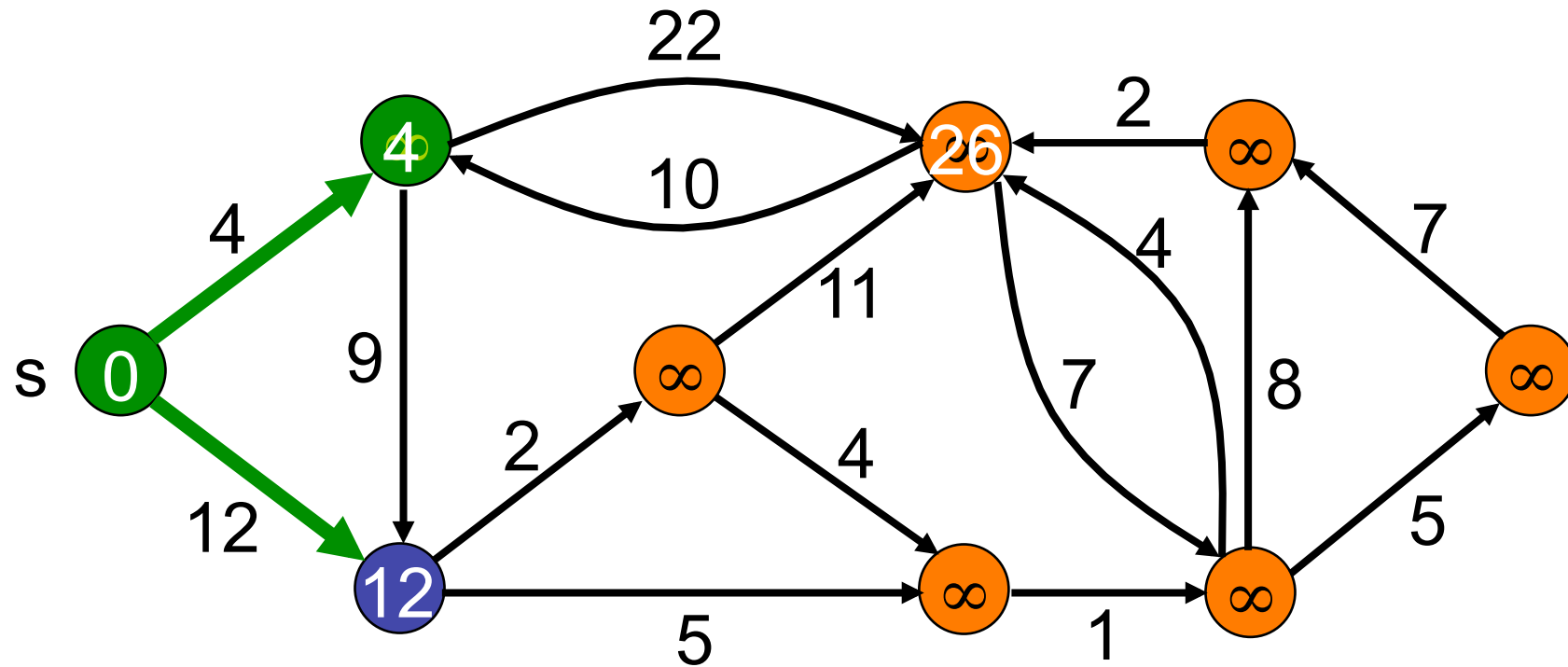
Priority Queue PQ: Knoten, Priorität Weglänge  
Kandidatenmenge K in PQ: Weg von s gefunden  
Abgeschlossene Knoten: Minimum aus PQ

# Algorithmus von Dijkstra



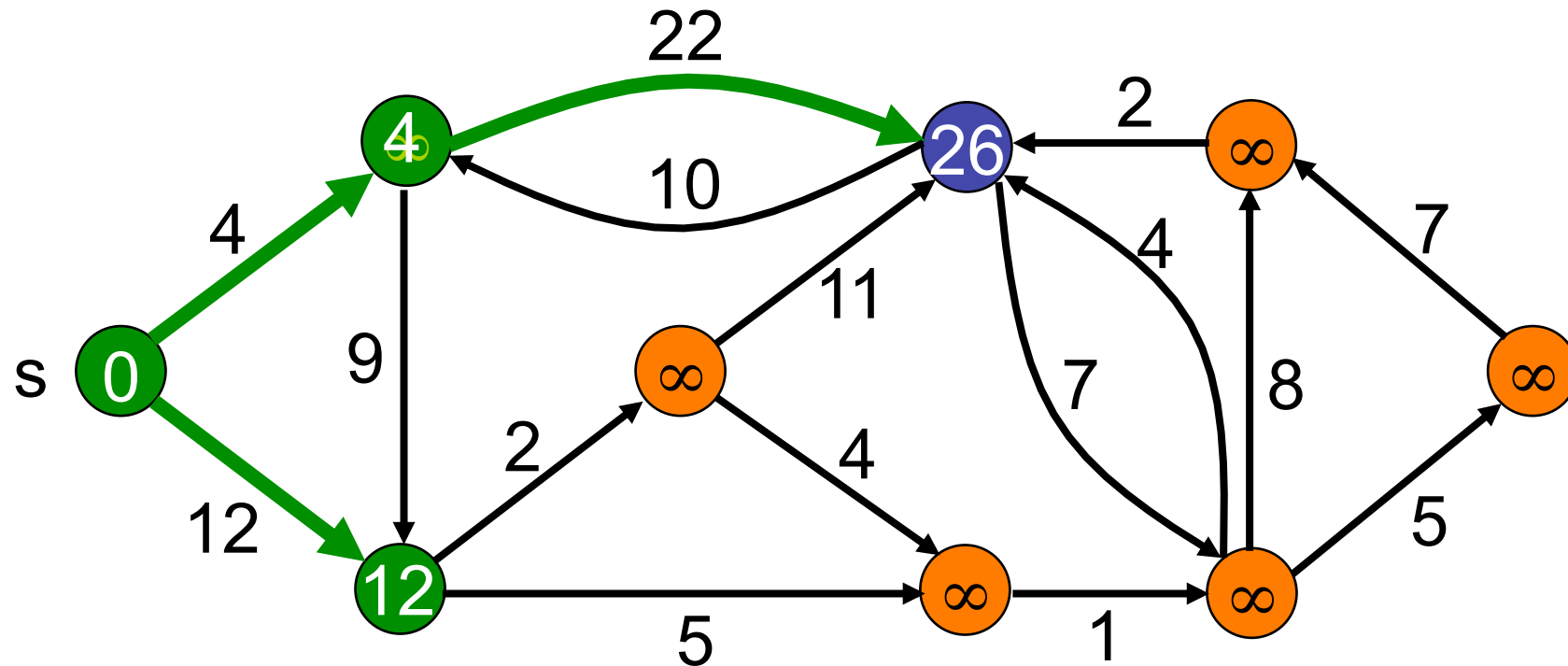
Ausgehende Kanten des gewählten Knoten  
Erreichte Kandidaten  
Vorgänger-Kanten in PQ:  $\pi$

# Algorithmus von Dijkstra



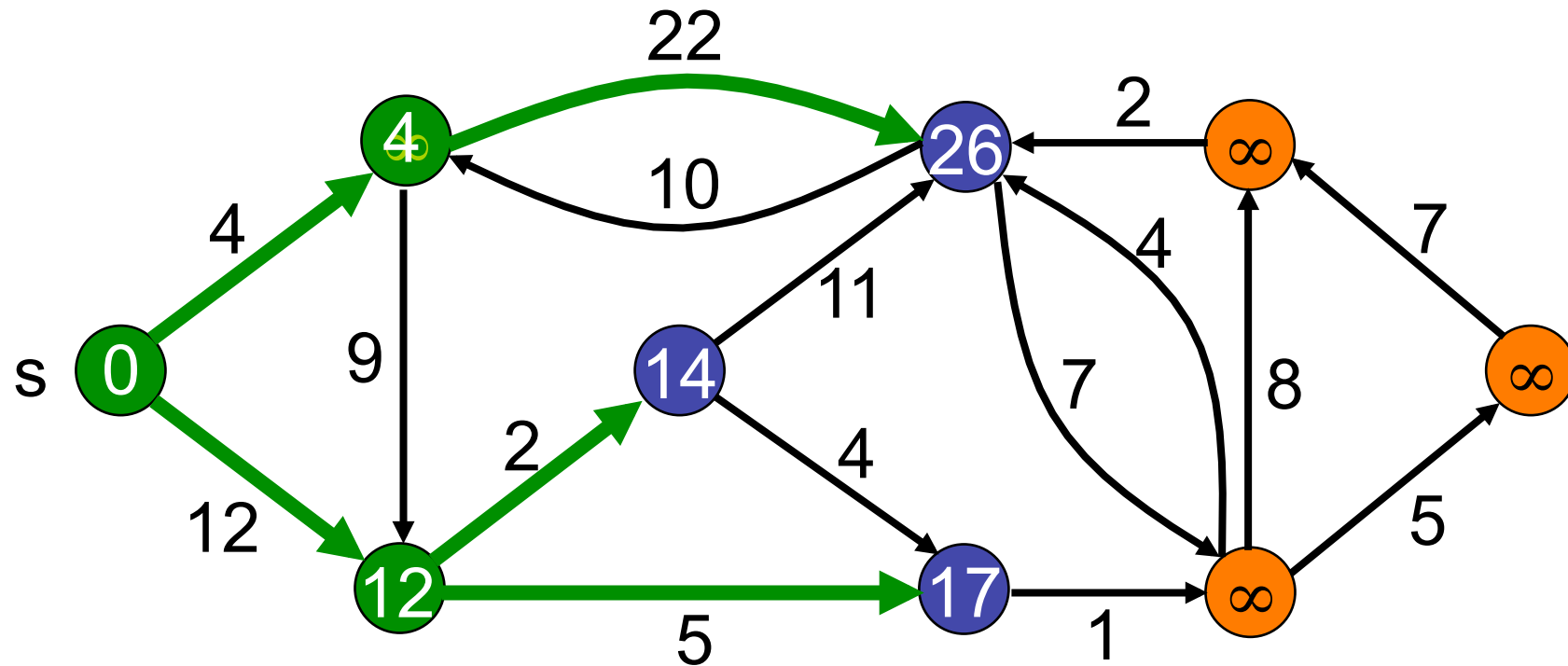
Ausgehende Kanten des gewählten Knoten  
Erreichte Kandidaten  
Vorgänger-Kanten in PQ:  $\pi$

# Algorithmus von Dijkstra



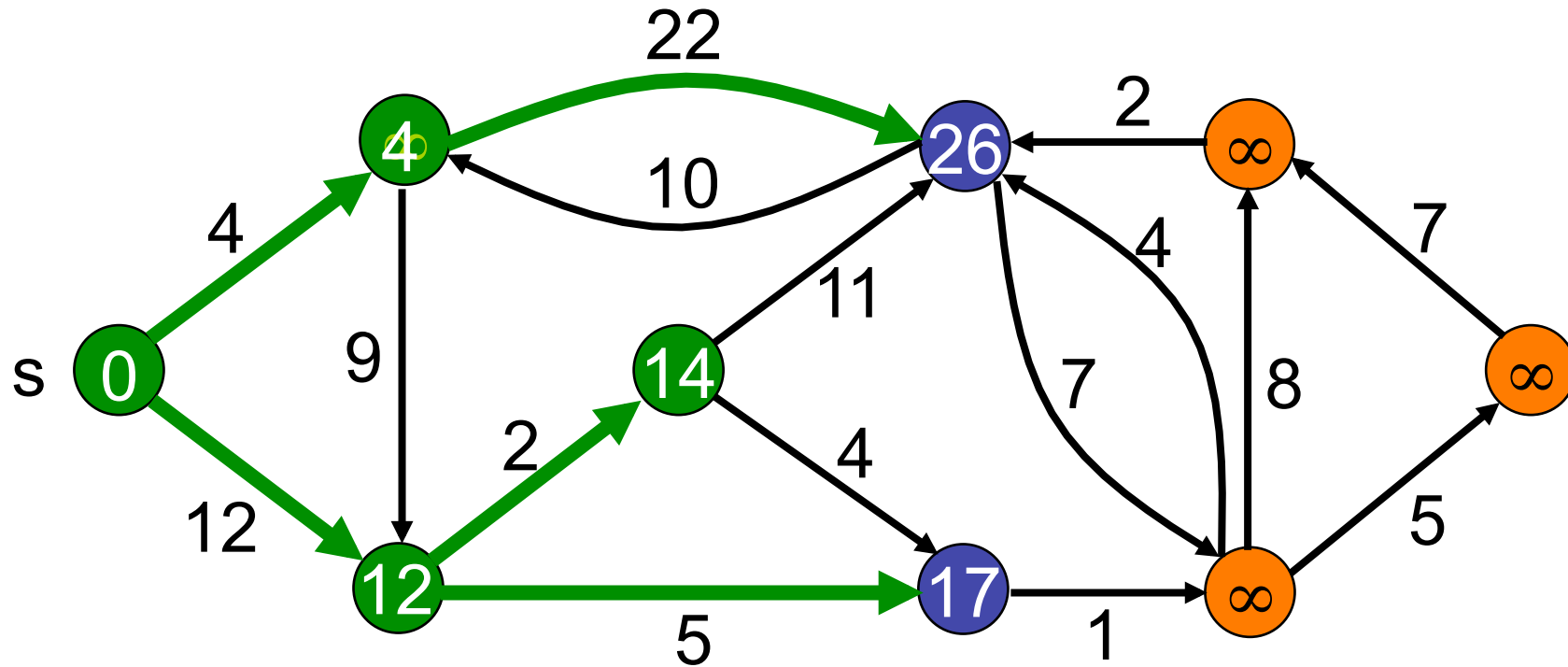
Ausgehende Kanten des gewählten Knoten  
Erreichte Kandidaten  
Vorgänger-Kanten in PQ:  $\pi$

# Algorithmus von Dijkstra



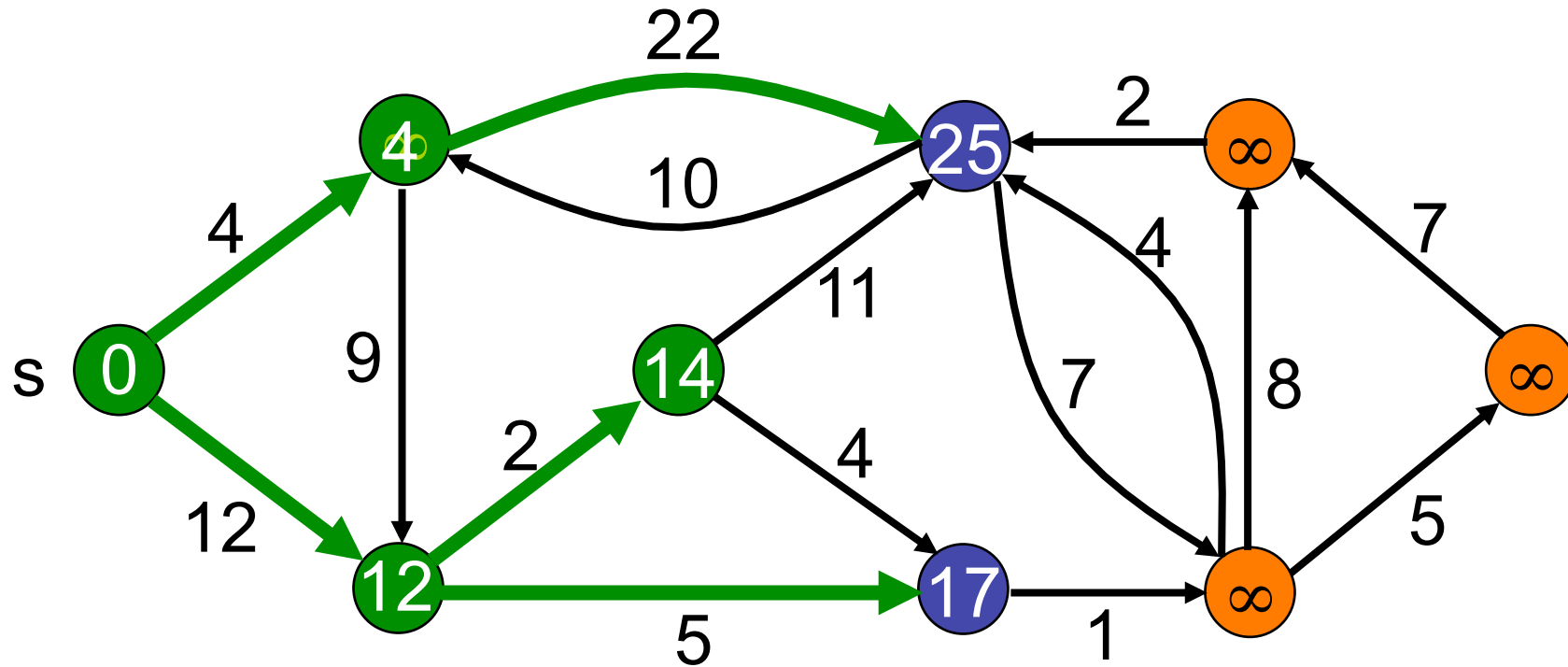
Ausgehende Kanten des gewählten Knoten  
Erreichte Kandidaten  
Vorgänger-Kanten in PQ:  $\pi$

# Algorithmus von Dijkstra



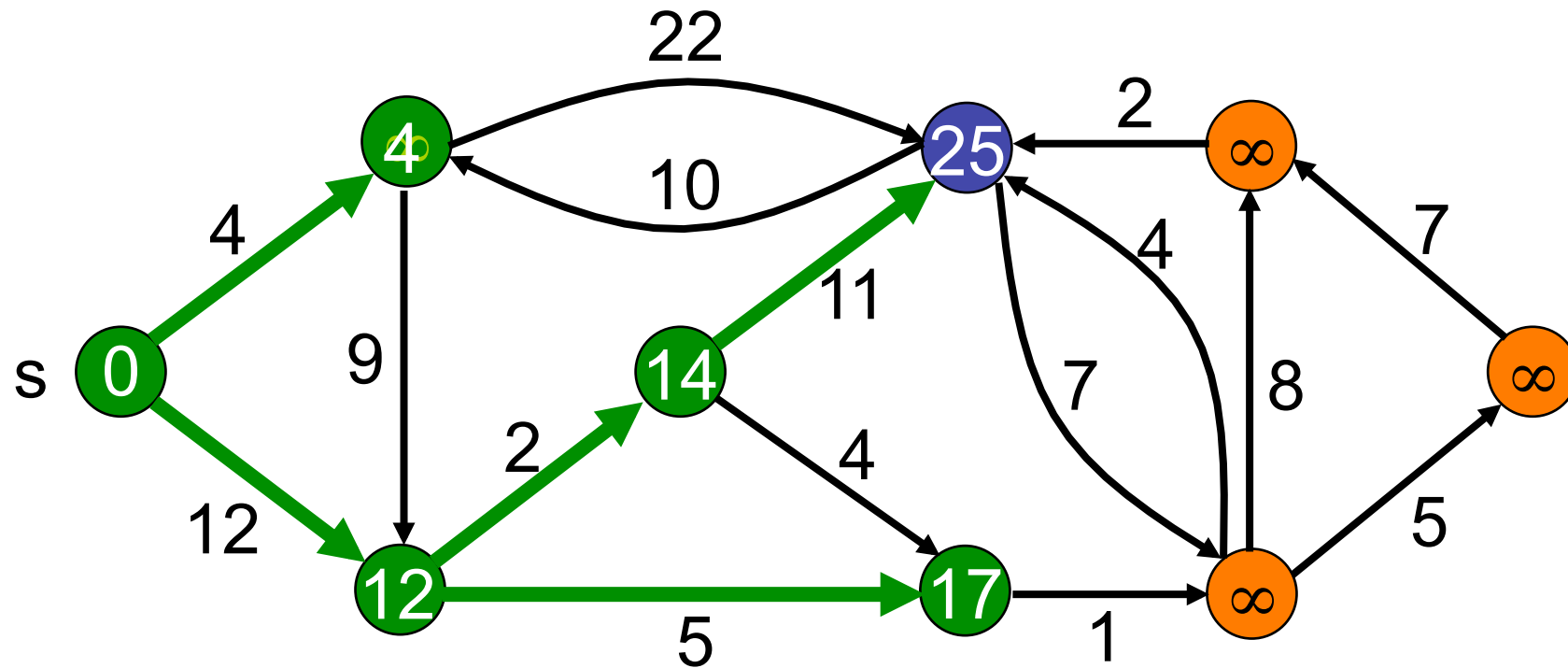
Ausgehende Kanten des gewählten Knoten  
Erreichte Kandidaten  
Vorgänger-Kanten:  $\pi$

# Algorithmus von Dijkstra



Ausgehende Kanten des gewählten Knoten  
Erreichte Kandidaten  
Vorgänger-Kanten:  $\pi$

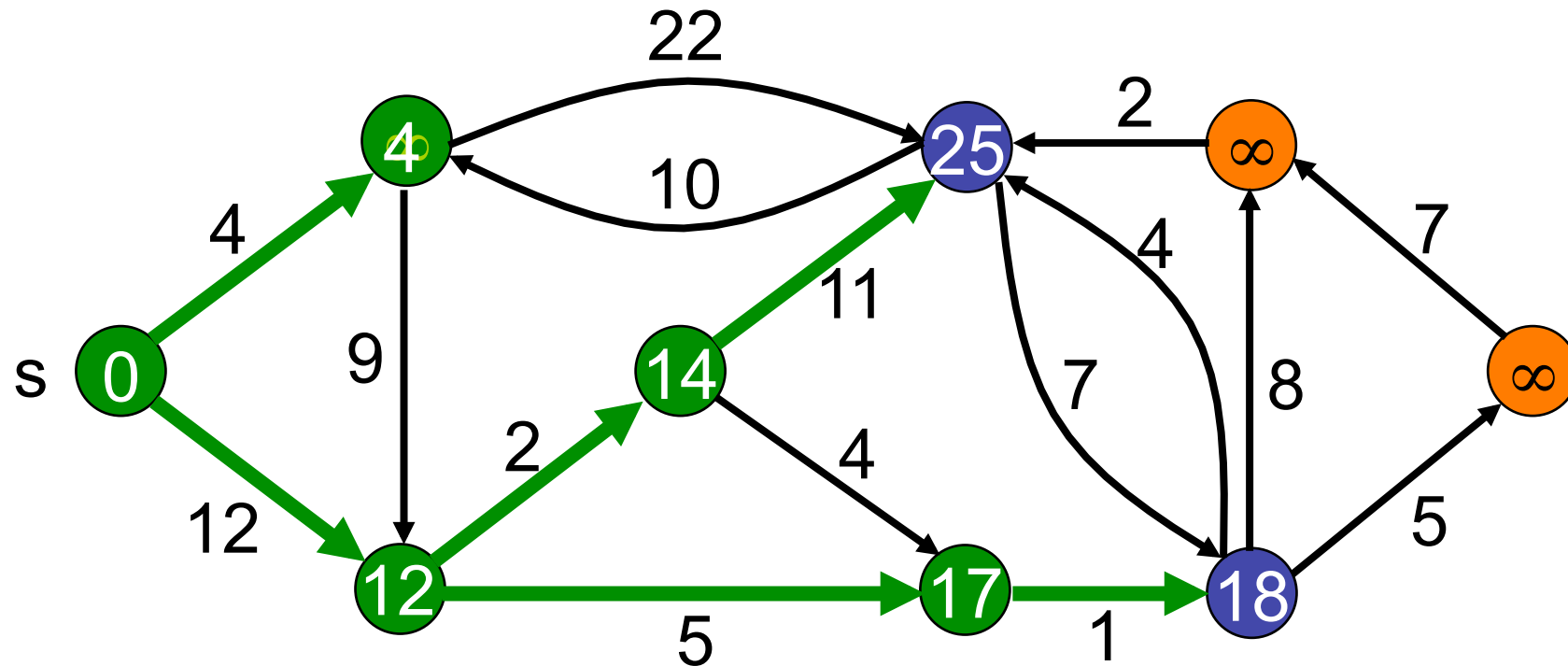
# Algorithmus von Dijkstra



**Ausgehende Kanten** des gewählten Knoten  
**Erreichte Kandidaten**  
**Vorgänger-Kanten:**  $\pi$



# Algorithmus von Dijkstra

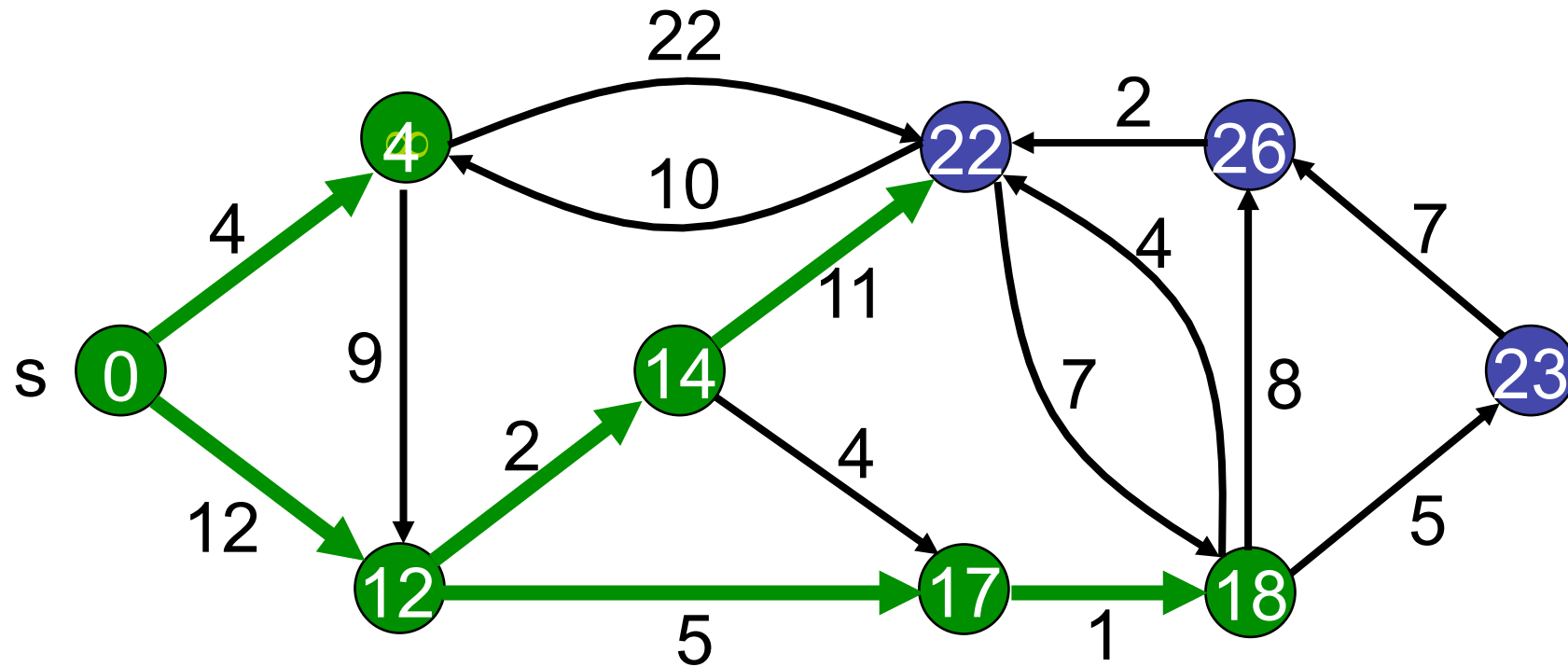


Ausgehende Kanten des gewählten Knoten

Erreichte Kandidaten

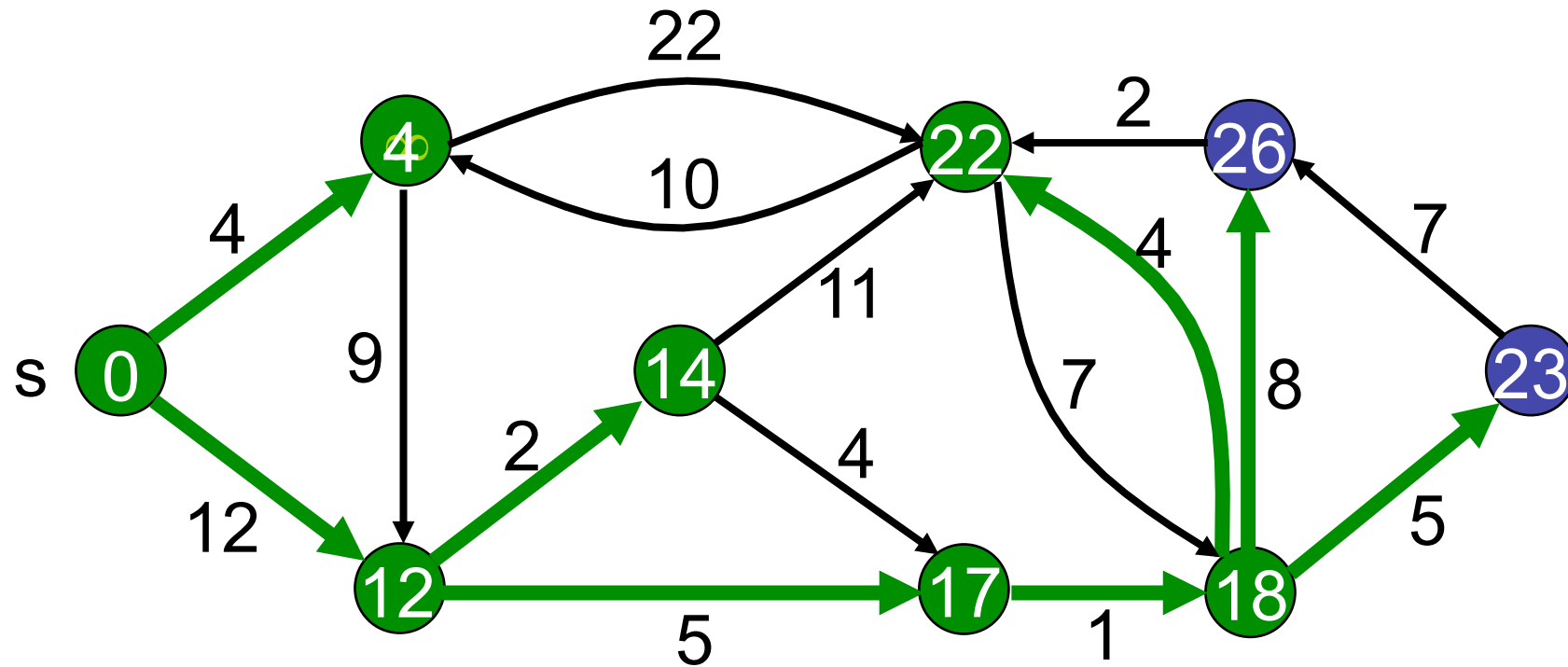
Vorgänger-Kanten:  $\pi$

# Algorithmus von Dijkstra



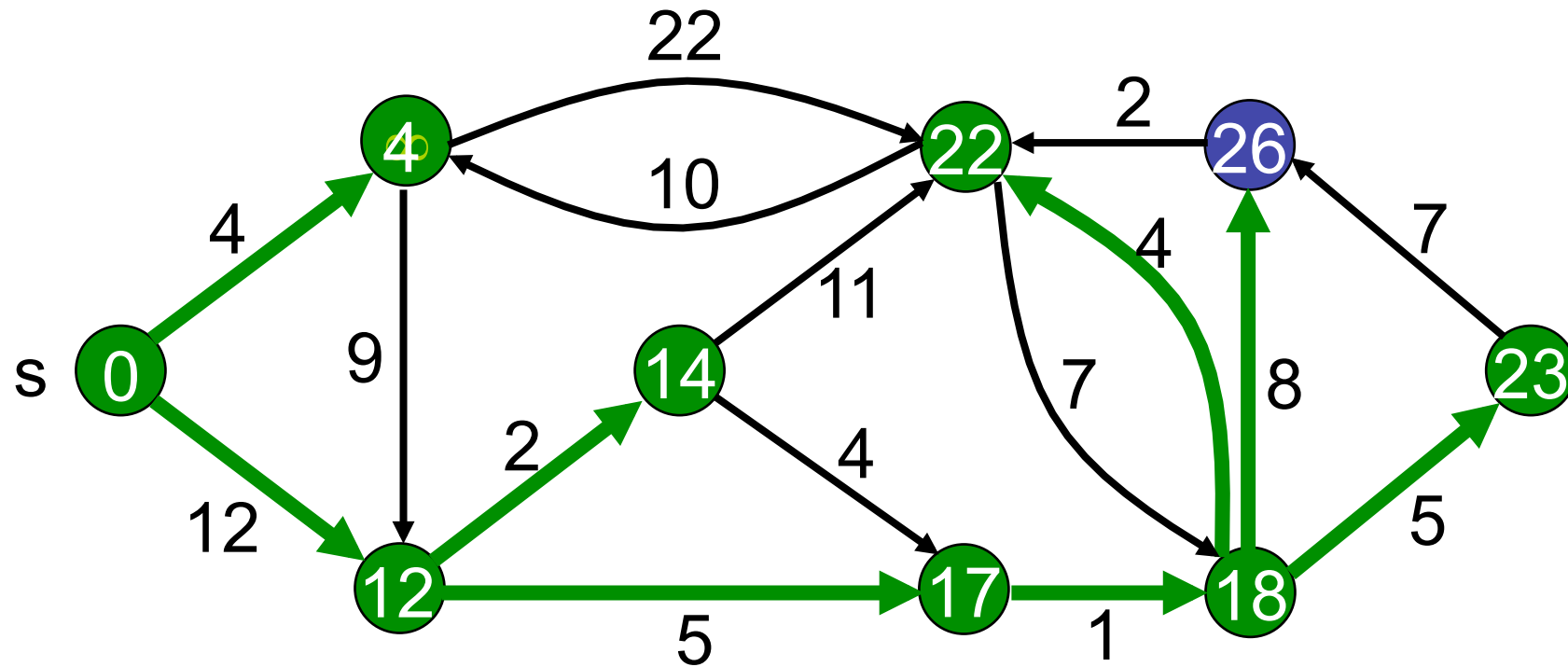
Ausgehende Kanten des gewählten Knoten  
Erreichte Kandidaten  
Vorgänger-Kanten:  $\pi$

# Algorithmus von Dijkstra



Ausgehende Kanten des gewählten Knoten  
Erreichte Kandidaten  
Vorgänger-Kanten:  $\pi$

# Algorithmus von Dijkstra



Ausgehende Kanten des gewählten Knoten  
Erreichte Kandidaten  
Vorgänger-Kanten:  $\pi$

# Pseudo-Code: Initialisierung

$G=(V, A), w: A \rightarrow R_0^+$

- ➔ (1) var  $\pi[V]$ , PriorityQueue  $Q$ ,  $\text{pos}[V]$
- ➔ (2) **for each**  $u \in V \setminus \{s\}$  **do** {
- ➔ (3)      $\text{pos}[u] := Q.\text{INSERT}(\infty, u)$
- ➔ (4)      $\pi[u] := \text{nil}$
- ➔ (5) }
- ➔ (6)  $\text{pos}[s] := Q.\text{INSERT}(0, s)$
- ➔ (7)  $\pi[s] := \text{nil}$

# Pseudo-Code

```
(8) while not Q.ISEMPTY() do {
(9)   ( $d_u, u$ ) := Q.EXTRACTMIN(); //  $d_u$  Abstand  $s$  zu  $u$ 
(10)  pos[ $u$ ] := nil // Minimum entfernt
(11)  for all  $e = (u, v) \in A^+(u)$  do { // Erreichbare Knoten
(12)    if  $d_u + w(e) < Q.PRIORITY(pos[v])$  then {
(13)      Q.DECREASEPRIORITY(pos[ $v$ ],  $d_u + w(e)$ )
(14)       $\pi[v] := u$ 
(15)    }
(16)  }
(17) }
```

# Analyse

Laufzeit abhängig von Datenstruktur:

- Kosten der ExtractMin und vor allem der DecreasePriority Operationen
- Wir betrachten Binary Heaps

*Spezielle PQ mit amortisiert konstanter  
DecreasePriority Zeit:*

***Fibonacci-Heaps***

# Analyse

- Aufbau des Heaps in  $O(|V|)$
- $|V|$  Durchläufe der while-Schleife Z. 8 mit EXTRACTMIN  $\Rightarrow O(|V| \log |V|)$
- $|A|$  Aufrufe von DECREASEPRIORITY max.  $\Rightarrow O(|A| \log |V|)$

Gesamtlaufzeit  $O((|V|+|A|)\log|V|)$



## 6.6.2 All Pairs Shortest Paths

All-Pairs Shortest Paths (APSP)	
<i>Gegeben:</i>	gerichteter Graph $G = (V, A)$ Gewichtsfunktion $w : A \rightarrow \mathbb{R}_0^+$
<i>Gesucht:</i>	ein kürzester Weg von $u$ nach $v$ für jedes Paar $u, v \in V$

# Algorithmen für APSP

## Idee:

- Aufruf von Dijkstra für alle Knoten  $v \in V$
- Laufzeit:  $O(|V| (|V| + |E|) \log |V|)$ 
  - für dünne Graphen:  $O(|V|^2 \log |V|)$
  - für dichte Graphen:  $O(|V|^3 \log |V|)$

jetzt: schnellerer Algorithmus für dichte Graphen

# Algorithmus von Floyd-Warshall

## Idee: Löse eingeschränkte Teilprobleme:

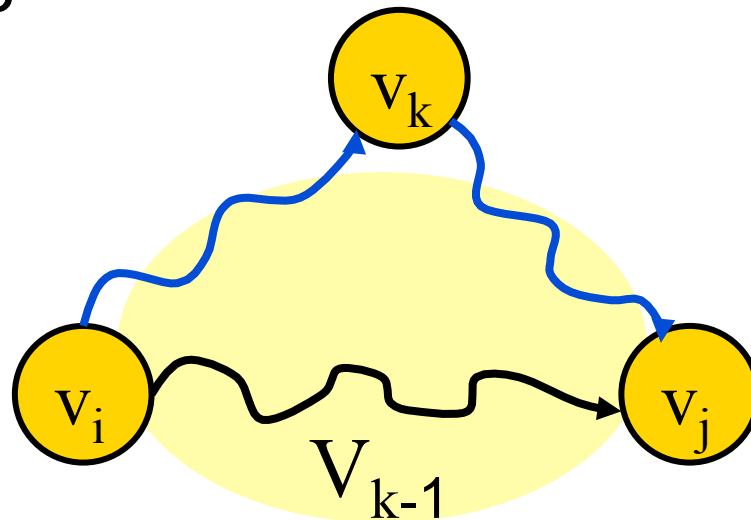
- Sei  $V_k := \{v_1, \dots, v_k\}$  die Menge der ersten  $k$  Knoten
- Finde kürzeste Wege, die nur Knoten aus  $V_k$  als Zwischenknoten benutzen dürfen
- Zwischenknoten sind alle Knoten eines Weges außer die beiden Endknoten
- Sei  $d_{ij}^{(k)}$  die Länge eines kürzesten Weges von  $v_i$  nach  $v_j$ , der nur Knoten aus  $V_k$  als Zwischenknoten benutzt

# Berechnung von $d_{ij}^{(k)}$

## Berechnung von $d_{ij}^{(k)}$ :

- $d_{ij}^{(0)} = w(v_i, v_j)$
- Bereits berechnet:  $d_{ij}^{(k-1)}$  für alle  $1 \leq i, j \leq n$

Für einen kürzesten Weg von  $v_i$  nach  $v_j$  gibt es zwei Möglichkeiten:



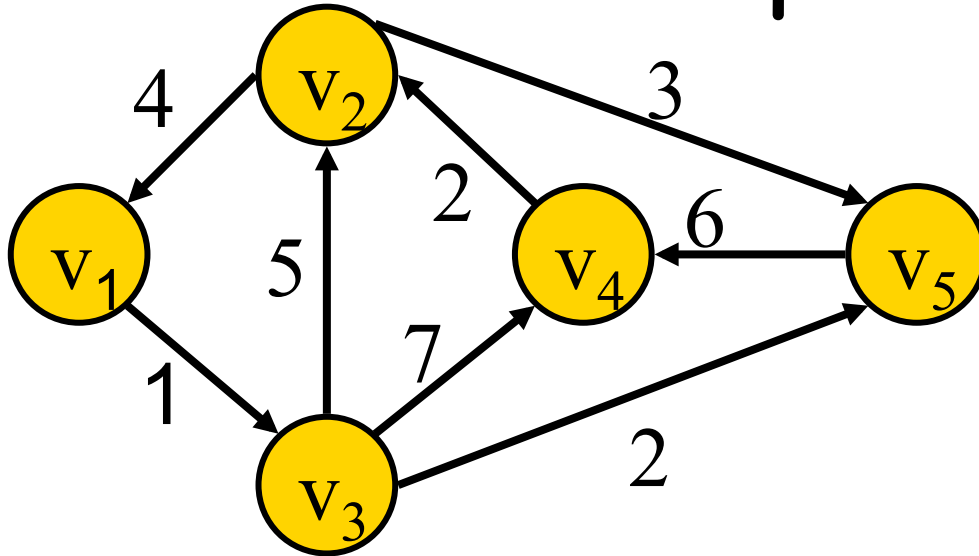
# Berechnung von $d_{ij}^{(k)}$

Zwei Möglichkeiten für kürzesten Weg von  $v_i$  nach  $v_j$ :

- Der Weg  $p$  benutzt  $v_k$  nicht als Zwischenknoten:  
dann ist  $d_{ij}^{(k)} = d_{ij}^{(k-1)}$
- Der Weg  $p$  benutzt  $v_k$  als Zwischenknoten: dann setzt sich  $p$  aus einem kürzesten Weg von  $v_i$  nach  $v_k$  und einem von  $v_k$  nach  $v_j$  zusammen, die jeweils nur Zwischenknoten aus  $V_{k-1}$  benutzen:  
 $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

- $d_{ij}^{(k)} := \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
- Für  $k=n$ : optimale Wege gefunden

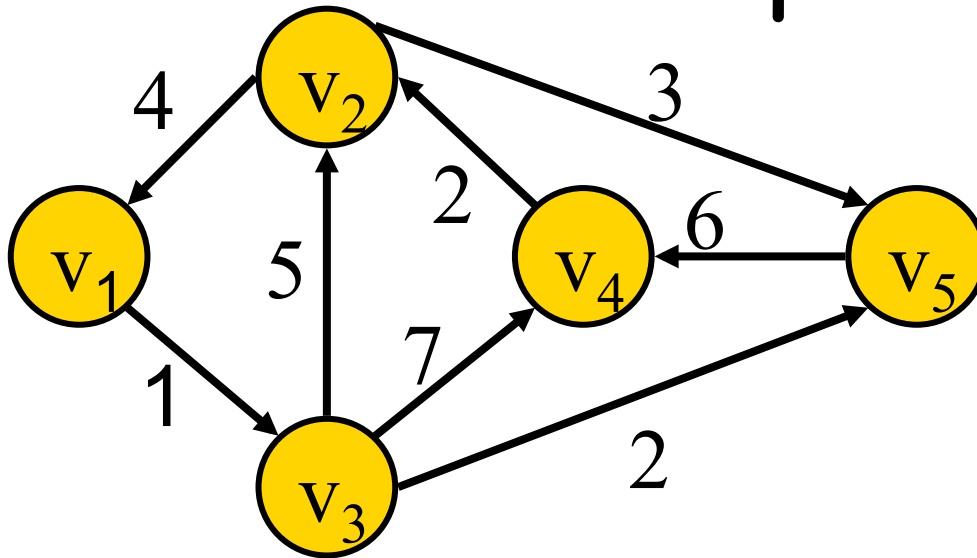
# Beispiel



$D^{(0)}$

	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$
$v_1$	0	$\infty$	1	$\infty$	$\infty$
$v_2$	4	0	$\infty$	$\infty$	3
$v_3$	$\infty$	5	0	7	2
$v_4$	$\infty$	2	$\infty$	0	$\infty$
$v_5$	$\infty$	$\infty$	$\infty$	6	0

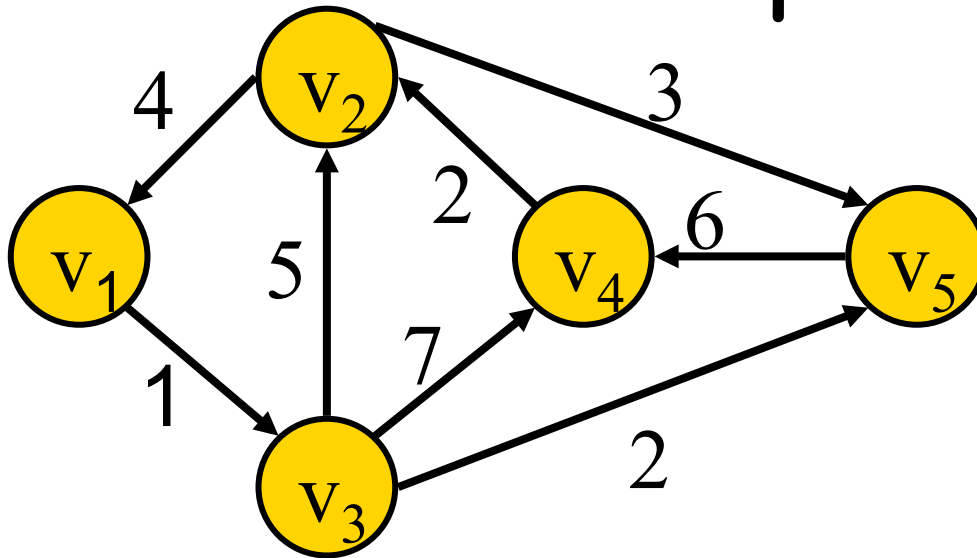
# Beispiel



$$D^{(0)} = \begin{pmatrix} 0 & \infty & 1 & \infty & \infty \\ 4 & 0 & \infty & \infty & 3 \\ \infty & 5 & 0 & 7 & 2 \\ \infty & 2 & \infty & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & \infty & 1 & \infty & \infty \\ 4 & 0 & 5 & \infty & 3 \\ \infty & 5 & 0 & 7 & 2 \\ \infty & 2 & \infty & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

# Beispiel

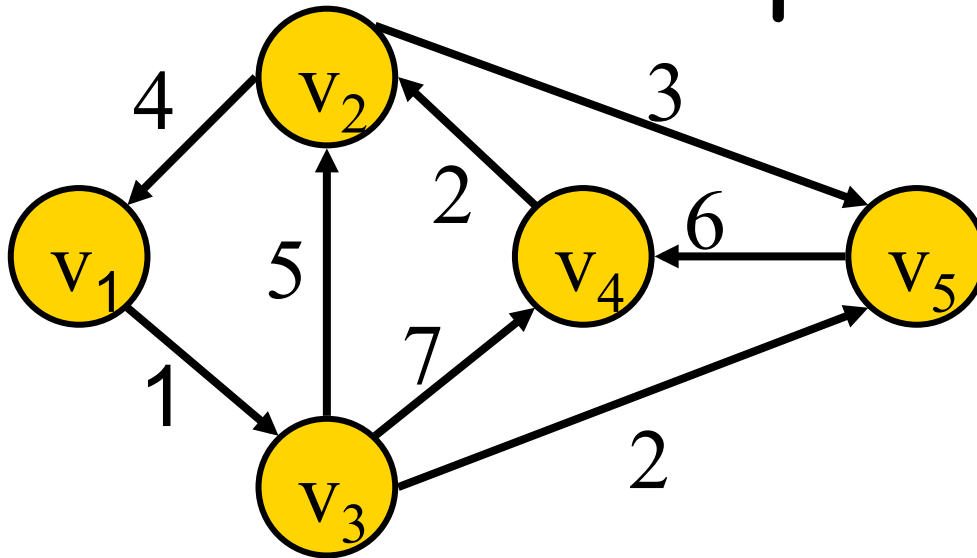


$$D^{(1)} = \begin{pmatrix} 0 & \infty & 1 & \infty & \infty \\ 4 & 0 & 5 & \infty & 3 \\ \infty & 5 & 0 & 7 & 2 \\ \infty & 2 & \infty & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & \infty & 1 & \infty & \infty \\ 4 & 0 & 5 & \infty & 3 \\ 9 & 5 & 0 & 7 & 2 \\ 6 & 2 & 7 & 0 & 5 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$



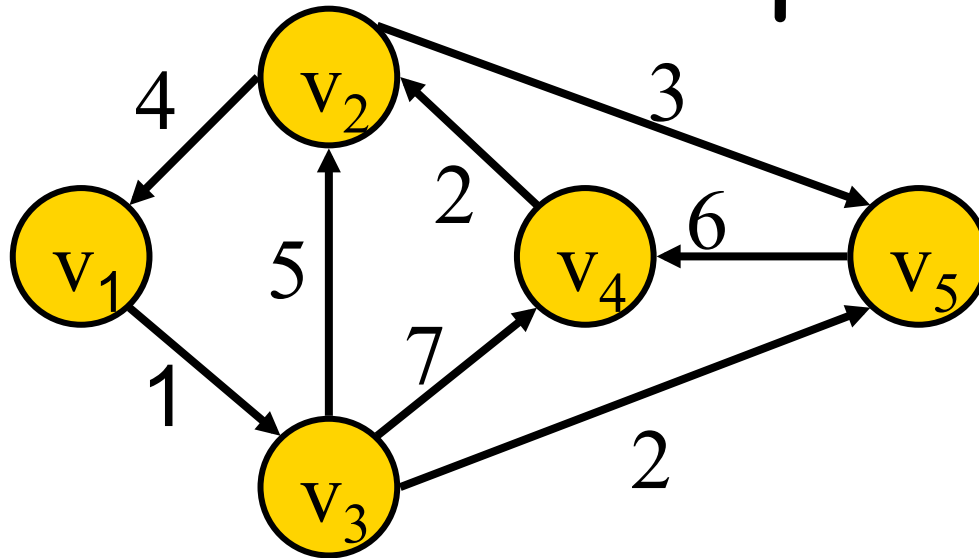
# Beispiel



$$D^{(2)} = \begin{pmatrix} 0 & \infty & 1 & \infty & \infty \\ 4 & 0 & 5 & \infty & 3 \\ 9 & 5 & 0 & 7 & 2 \\ 6 & 2 & 7 & 0 & 5 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 6 & 1 & 8 & 3 \\ 4 & 0 & 5 & 12 & 3 \\ 9 & 5 & 0 & 7 & 2 \\ 6 & 2 & 7 & 0 & 5 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

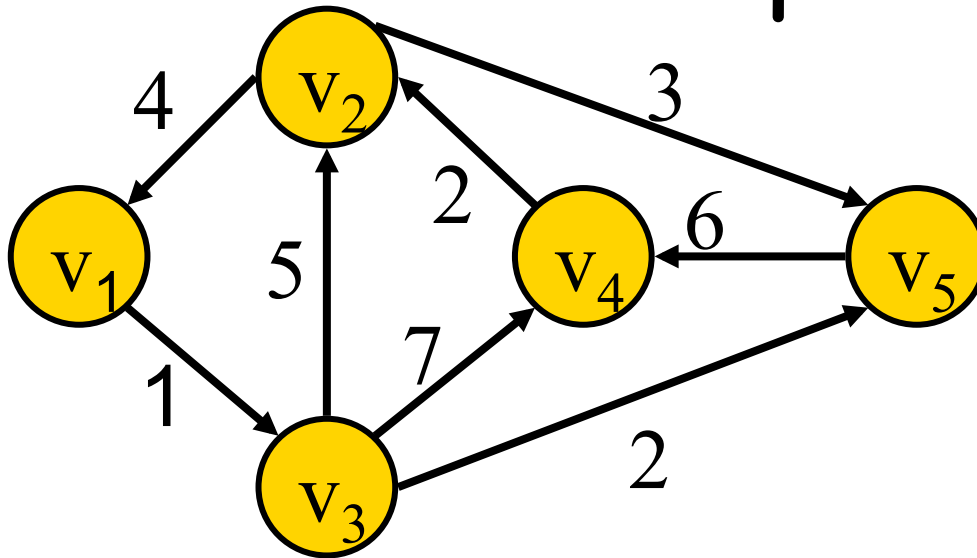
# Beispiel



$$D^{(3)} = \begin{pmatrix} 0 & 6 & 1 & 8 & 3 \\ 4 & 0 & 5 & 12 & 3 \\ 9 & 5 & 0 & 7 & 2 \\ 6 & 2 & 7 & 0 & 5 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 6 & 1 & 8 & 3 \\ 4 & 0 & 5 & 12 & 3 \\ 9 & 5 & 0 & 7 & 2 \\ 6 & 2 & 7 & 0 & 5 \\ 12 & 8 & 13 & 6 & 0 \end{pmatrix}$$

# Beispiel



$$D^{(4)} = \begin{pmatrix} 0 & 6 & 1 & 8 & 3 \\ 4 & 0 & 5 & 12 & 3 \\ 9 & 5 & 0 & 7 & 2 \\ 6 & 2 & 7 & 0 & 5 \\ 12 & 8 & 13 & 6 & 0 \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 6 & 1 & 8 & 3 \\ 4 & 0 & 5 & 9 & 3 \\ 9 & 5 & 0 & 7 & 2 \\ 6 & 2 & 7 & 0 & 5 \\ 12 & 8 & 13 & 6 & 0 \end{pmatrix}$$

# Algorithmus von Floyd-Warshall

```
(1) for i:=1 to n do
(2)   for j:=1 to n do
(3)      $d_{ij}^{(0)} := w(v_i, v_j)$ 
(4)   } }
(5) for k:=1 to n do
(6)   for i:=1 to n do
(7)     for j:=1 to n do
(8)        $d_{ij}^{(k)} := \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
(9)   } } }
```

# Diskussion

- Ist man auch an den Wegen selbst interessiert, dann: **Vorgängermatrix**:  $\pi_{ij}^{(k)}$  für jedes  $0 \leq k \leq n$
- **Initialisierung**:
  - Falls  $i=j$  oder  $w(v_i, v_j) = \infty$ , dann:  $\pi_{ij}^{(0)} := \text{nil}$
  - Sonst:  $\pi_{ij}^{(0)} := i$
- **Aktualisierung**:
  - Falls  $d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$
  - dann:  $\pi_{ij}^{(k)} := \pi_{ij}^{(k-1)}$
  - sonst:  $\pi_{ij}^{(k)} := \pi_{kj}^{(k-1)}$

# Laufzeit-Analyse

- Speicherplatzverbrauch für d-Matrizen:  $\Theta(|V|^3)$
- Man benötigt zur Berechnung von  $d_{ij}^{(k)}$  nur die Werte von  $d_{ij}^{(k-1)} \rightarrow \Theta(|V|^2)$  Speicherplatz,
- genauso für die  $\pi$ -Matrizen
- Laufzeit:  $\Theta(|V|^3)$

# Algorithmus von Floyd-Warshall

- Der Algorithmus von Floyd-Warshall berechnet das APSP-Problem in einer Laufzeit von  $\Theta(|V|^3)$  mit einem Speicherverbrauch von  $\Theta(|V|^2)$ .
- Der Algorithmus von Floyd-Warshall funktioniert auch mit negativen Kantenkosten, solange kein negativer Kreis in  $G$  enthalten ist.

# Diskussion

- Dijkstra-Algorithmus kann durch eine andere Implementierung der Priority-Queue in Laufzeit  $O(|V|^2+|E|)$  realisiert werden
- Dann: APSP für dichte Graphen:  $O(|V|^3)$

- Dijkstra-Algorithmus kann durch eine andere Implementierung der Priority-Queue (Fibonacci-Heaps) in Laufzeit  $O(|E|+|V| \log |V|)$  realisiert werden (aber eher theoretisch)
- Dann: APSP für dichte Graphen:  $O(|V|^3)$

Aber: Floyd-Warshall konzeptionell einfacher!

Für dünne Graphen: APSP mit Dijkstra schneller!