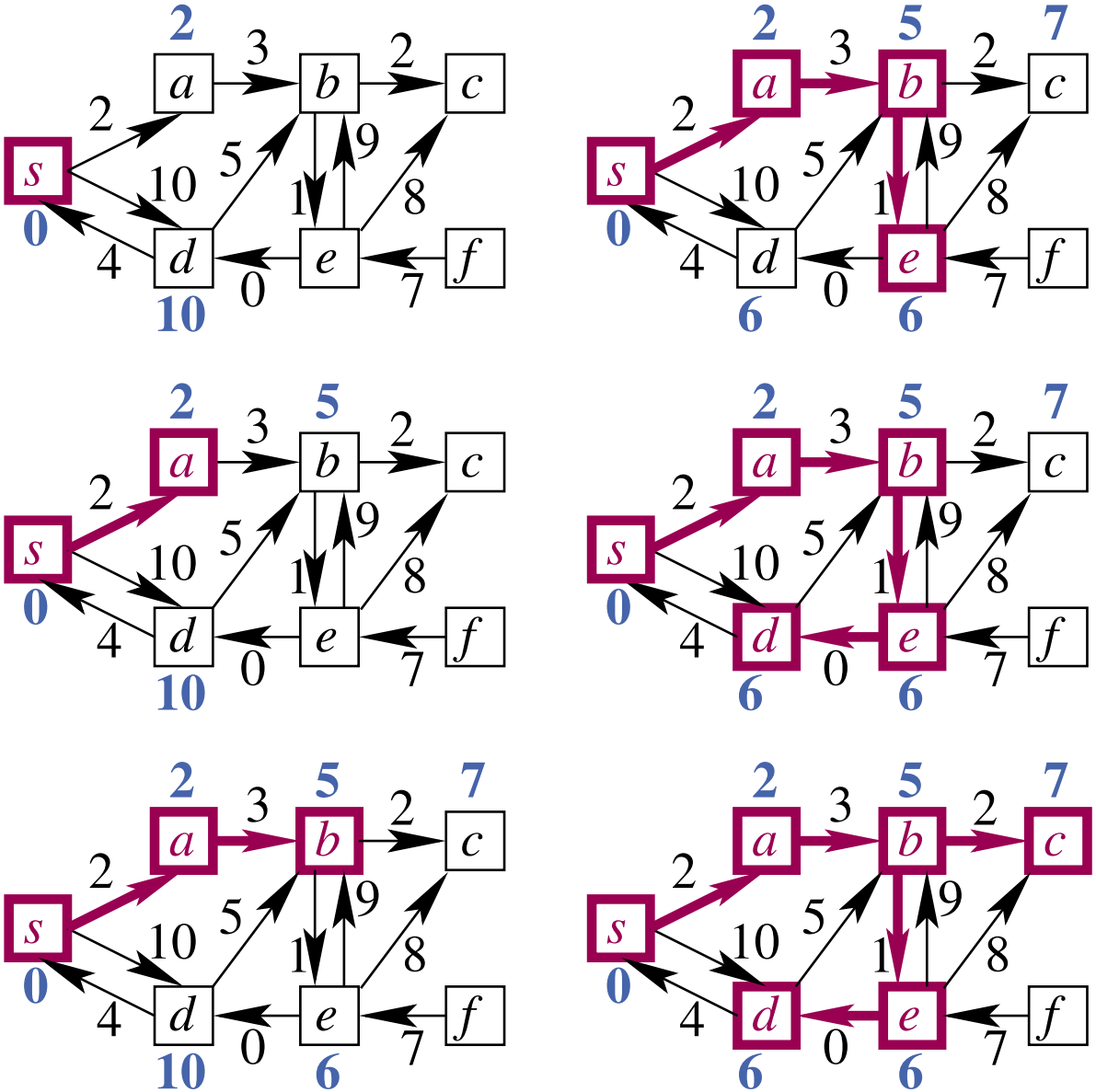
Dijkstras Algorithmus: Pseudocode

initialize d, parent
all nodes are non-scanned
while $\exists$ non-scanned node u with $d[u] < \infty$
    $u :=$ non-scanned node v with minimal $d[v]$
    relax all edges (u, v) out of u
    u is scanned now

Behauptung: Am Ende definiert d die optimalen Entfernungen und parent die zugehörigen Wege

Beispiel



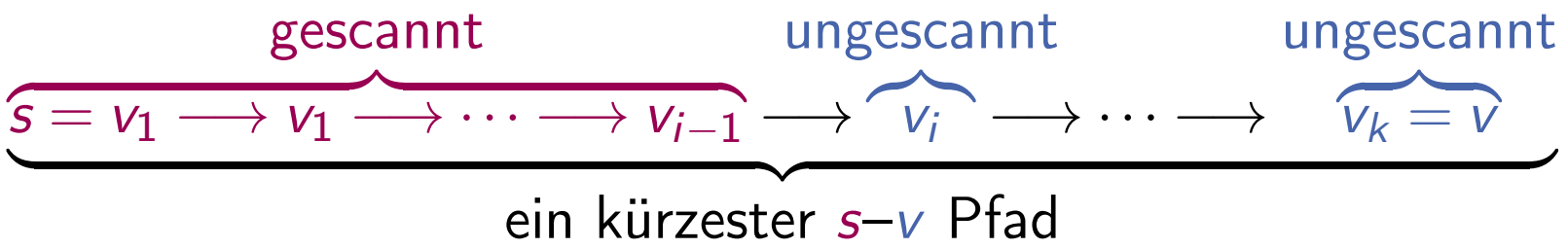
Annahme: alle Kosten nicht-negativ!

Wir zeigen: $\forall v \in V$:

- v erreichbar $\Longrightarrow$ v wird irgendwann gescannt
- v gescannt $\Longrightarrow \mu(v) = d[v]$

v erreichbar $\Longrightarrow$ v wird irgendwann gescannt

Annahme: v ist erreichbar, aber wird nicht gescannt



$\Longrightarrow$ v_{i-1} wird gescannt

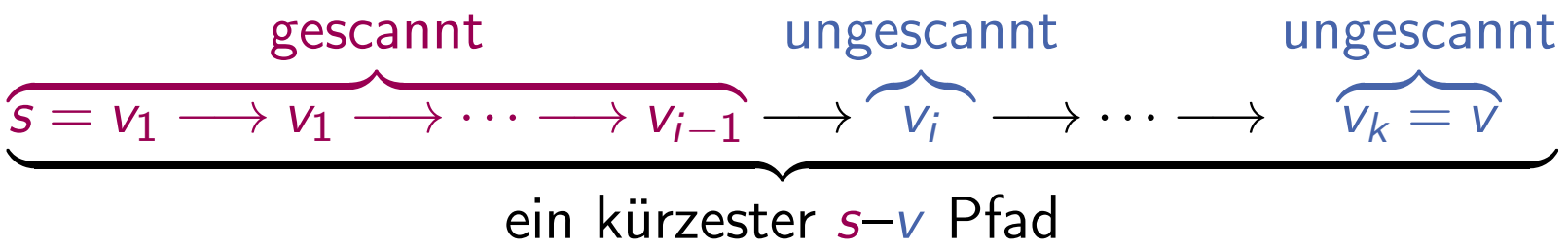
$\Longrightarrow$ Kante $v_{i-1} \longrightarrow v_i$ wird relaxiert

$\Longrightarrow$ $d[v_i] < \infty$

Widerspruch – nur Knoten x mit $d[x] = \infty$ werden nie gescannt $\square$?

v erreichbar $\Longrightarrow$ v wird irgendwann gescannt

Annahme: v ist erreichbar, aber wird nicht gescannt



$\Longrightarrow$ v_{i-1} wird gescannt

$\Longrightarrow$ Kante $v_{i-1} \longrightarrow v_i$ wird relaxiert

$\Longrightarrow$ $d[v_i] < \infty$

Widerspruch – nur Knoten x mit $d[x] = \infty$ werden nie gescannt

Ups: Spezialfall $i = 1$?

Kann auch nicht sein.

$v_1 = s$ wird nach Initialisierung gescannt. □

v gescannt $\Longrightarrow \mu(v) = d[v]$

Annahme: v gescannt und $\mu(v) < d[v]$
OBdA: v ist der erste gescannte Knoten mit $\mu(v) < d[v]$.
$t :=$ Scan-Zeit von v

$$\underbrace{\overbrace{s = v_1 \longrightarrow v_1 \longrightarrow \cdots \longrightarrow v_{i-1}}^{\text{Scan-Zeit} < t} \longrightarrow \overbrace{v_i}^{\text{Scan-Zeit} \geq t} \longrightarrow \cdots \longrightarrow \overbrace{v_k = v}^{\text{Scan-Zeit} = t}}_{\text{ein kürzester } s\text{–}v \text{ Pfad}}$$

Also gilt zur Zeit t:

$\mu(v_{i-1}) = d[v_{i-1}]$

$v_{i-1} \to v_i$ wurde relaxiert
$\overbrace{\Longrightarrow}$ $d[v_i] \leq d[v_{i-1}] + c(v_{i-1}, v_i) = \mu(v_i) \leq \mu(v) < d[v]$
$\Longrightarrow v_i$ wird vor v gescannt. Widerspruch!

Wieder: Spezialfall $i = 1$ unmöglich. □

Implementierung?

initialize d, parent
all nodes are non-scanned
while $\exists$ non-scanned node u with $d[u] < \infty$
    $u :=$ non-scanned node v with minimal $d[v]$
    relax all edges (u, v) out of u
    u is scanned now

Wichtigste Operation: finde u

Prioritätsliste

Wir speichern ungescannte erreichte Knoten in addressierbarer Prioritätsliste Q.

Schlüssel ist $d[v]$.

Knoten speichern handles. oder gleich items

Implementierung $\approx$ BFS mit PQ statt FIFO

Function Dijkstra(s : NodeId) : NodeArray$\times$NodeArray
// returns (d, parent)

Initialisierung:

$d = \langle \infty, \ldots, \infty \rangle$: NodeArray **of** $\mathbb{R} \cup \{\infty\}$
// tentative distance from root

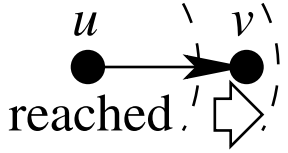
parent$= \langle \bot, \ldots, \bot \rangle$: NodeArray **of** NodeId

parent[s]:= s // self-loop signals root

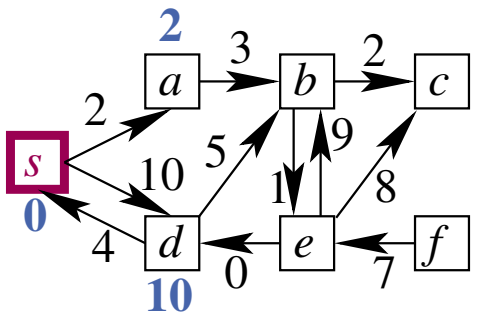
Q : NodePQ // unscanned reached nodes

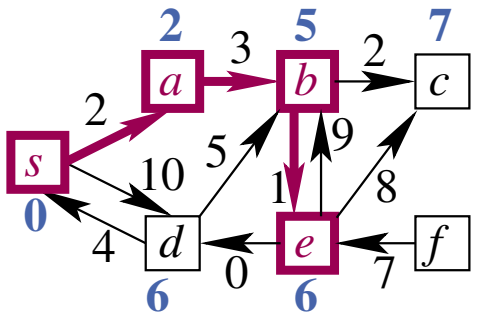
$d[s] := 0$; Q.insert(s)

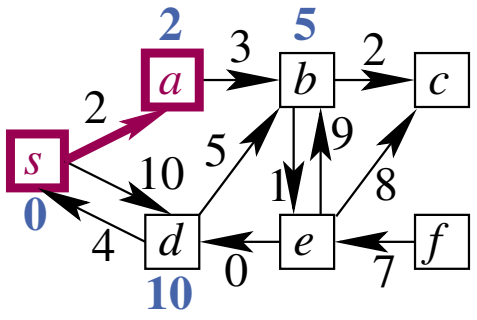
```
Function Dijkstra(s : NodeId) : NodeArray×NodeArray
    d = ⟨∞,...,∞⟩; parent[s]:= s; d[s] := 0;   Q.insert(s)
    while Q ≠ ∅ do
        u := Q.deleteMin
        // scan u
        foreach edge e = (u,v) ∈ E do
            if d[u] + c(e) < d[v] then                    // relax
                d[v]:= d[u] + c(e)
                parent[v] := u                            // update tree
                if v ∈ Q then Q.decreaseKey(v)
                else Q.insert(v)
    return (d, parent)
```

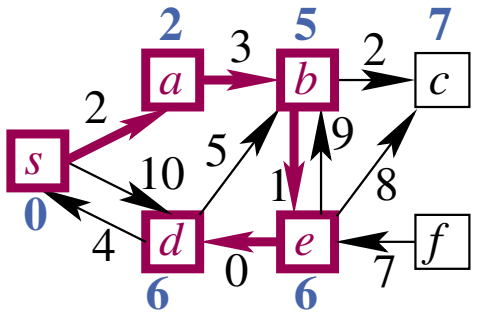


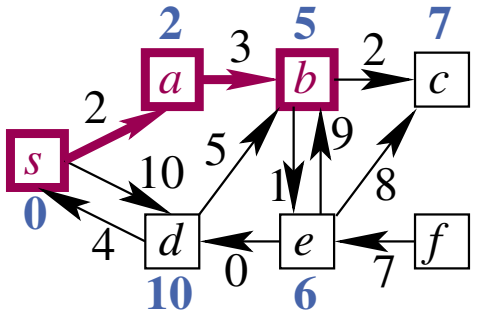
Beispiel

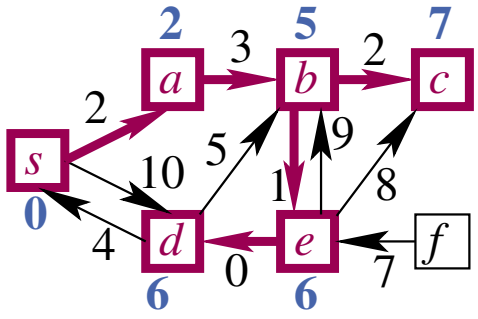












Operation	Queue
insert(s)	$\langle(s,0)\rangle$
deleteMin$\rightsquigarrow(s,0)$	$\langle\rangle$
relax $s \xrightarrow{2} a$	$\langle(a,2)\rangle$
relax $s \xrightarrow{10} d$	$\langle(a,2),(d,10)\rangle$
deleteMin$\rightsquigarrow(a,2)$	$\langle(d,10)\rangle$
relax $a \xrightarrow{3} b$	$\langle(b,5),(d,10)\rangle$
deleteMin$\rightsquigarrow(b,5)$	$\langle(d,10)\rangle$
relax $b \xrightarrow{2} c$	$\langle(c,7),(d,10)\rangle$
relax $b \xrightarrow{1} e$	$\langle(e,6),(c,7),(d,10)$
deleteMin$\rightsquigarrow(e,6)$	$\langle(c,7),(d,10)\rangle$
relax $e \xrightarrow{9} b$	$\langle(c,7),(d,10)\rangle$
relax $e \xrightarrow{8} c$	$\langle(c,7),(d,10)\rangle$
relax $e \xrightarrow{0} d$	$\langle(d,6),(c,7)\rangle$
deleteMin$\rightsquigarrow(d,6)$	$\langle(c,7)\rangle$
relax $d \xrightarrow{4} s$	$\langle(c,7)\rangle$
relax $d \xrightarrow{5} b$	$\langle(c,7)\rangle$
deleteMin$\rightsquigarrow(c,7)$	$\langle\rangle$

Dijkstra: Laufzeit

Function Dijkstra(s : NodeId) : NodeArray×NodeArray
Initialisierung:
$d=\langle\infty,\ldots,\infty\rangle$: NodeArray **of** $\mathbb{R}\cup\{\infty\}$ // $O(n)$
parent$=\langle\bot,\ldots,\bot\rangle$: NodeArray **of** NodeId // $O(n)$
parent[s]:= s
Q : NodePQ // unscanned reached nodes, $O(n)$
$d[s]$:= 0; Q.insert(s)

Dijkstra: Laufzeit

```
Function Dijkstra(s : NodeId) : NodeArray×NodeArray
    d = {∞,...,∞}; parent[s]:= s; d[s] := 0;   Q.insert(s)        // O(n)
    while Q ≠ ∅ do
        u := Q.deleteMin                                          // ≤ n×
        foreach edge e = (u,v) ∈ E do                             // ≤ m×
            if d[u] + c(e) < d[v] then                            // ≤ m×
                d[v]:= d[u] + c(e)                                // ≤ m×
                parent[v] := u                                    // ≤ m×
                if v ∈ Q then Q.decreaseKey(v)                    // ≤ m×
                else Q.insert(v)                                  // ≤ n×
    return (d, parent)
```

Dijkstra: Laufzeit

```
Function Dijkstra(s : NodeId) : NodeArray×NodeArray
    d = {∞,...,∞}; parent[s]:= s; d[s] := 0;   Q.insert(s)        // O(n)
    while Q ≠ ∅ do
        u := Q.deleteMin                                          // ≤ n×
        foreach edge e = (u,v) ∈ E do                             // ≤ m×
            if d[u] + c(e) < d[v] then                            // ≤ m×
                d[v]:= d[u] + c(e)                                // ≤ m×
                parent[v] := u                                    // ≤ m×
                if v ∈ Q then Q.decreaseKey(v)                    // ≤ m×
                else Q.insert(v)                                  // ≤ n×
    return (d, parent)
```

Insgesamt:

$$T_{\text{Dijkstra}} = \mathrm{O}\big(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))\big)$$

Laufzeit

Dijkstras ursprüngliche Implementierung: „naiv“

- insert: $\mathrm{O}(1)$ — $d[v] := d[u] + c(u,v)$
- decreaseKey: $\mathrm{O}(1)$ — $d[v] := d[u] + c(u,v)$
- deleteMin: $\mathrm{O}(n)$ — d komplett durchsuchen

$$
\begin{aligned}
T_{\text{Dijkstra}} &= \mathrm{O}\big(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))\big) \\
T_{\text{Dijkstra59}} &= \mathrm{O}(m \cdot 1 + n \cdot (n+1)) \\
&= \mathrm{O}\big(m + n^2\big)
\end{aligned}
$$

Laufzeit

Bessere Implementierung mit Binary-Heap-Prioritätslisten:

- insert: $\mathrm{O}(\log n)$
- decreaseKey: $\mathrm{O}(\log n)$
- deleteMin: $\mathrm{O}(\log n)$

$$
\begin{aligned}
T_{\text{Dijkstra}} &= \mathrm{O}\big(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))\big) \\
T_{\text{DijkstraBHp}} &= \mathrm{O}(m \cdot \log n + n \cdot (\log n + \log n)) \\
&= \mathrm{O}((m+n)\log n)
\end{aligned}
$$

Laufzeit

(Noch) besser mit Fibonacci-Heapprioritätslisten:

- insert: $\mathrm{O}(1)$
- decreaseKey: $\mathrm{O}(1)$ (amortisiert)
- deleteMin: $\mathrm{O}(\log n)$ (amortisiert)

$$
\begin{aligned}
T_{\text{Dijkstra}} &= \mathrm{O}\big(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))\big) \\
T_{\text{DijkstraFib}} &= \mathrm{O}(m \cdot 1 + n \cdot (\log n + 1)) \\
&= \mathrm{O}(m + n \log n)
\end{aligned}
$$

Aber: konstante Faktoren in $\mathrm{O}(\cdot)$ sind hier größer!

Analyse im Mittel

Modell: Kantengewichte sind „zufällig“ auf die Kanten verteilt
Dann gilt:

$$\mathrm{E}[T_{\mathsf{DijkstraBH(ea)p}}] = \mathrm{O}\left(m + n \log n \log \frac{m}{n}\right)$$

Beweis: In Algorithmen II

Monotone ganzzahlige Prioritätslisten

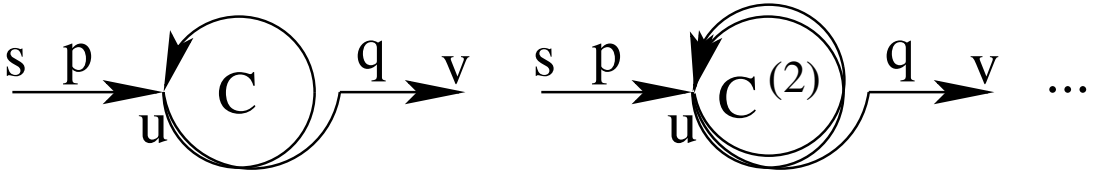
Beobachtung: In Dijkstras Algorithmus steigt das Minimum in der Prioritätsliste monoton.

Das kann man ausnutzen. $\rightsquigarrow$ schnellere Algorithmen
u.U. bis herunter zu $O(m+n)$.

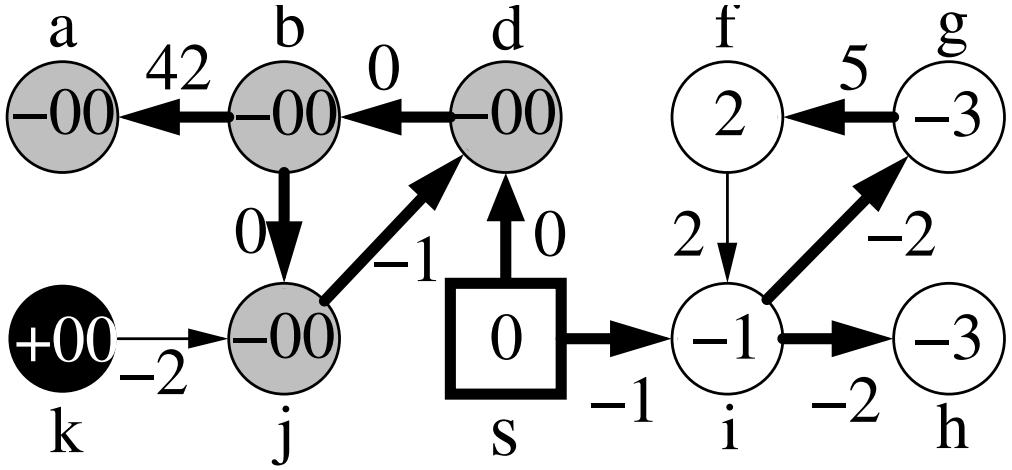
Details: in Algorithmen II

Negative Kosten

Was machen wir, wenn es Kanten mit negativen Kosten gibt?
Es kann Knoten geben mit $d[v] = -\infty$

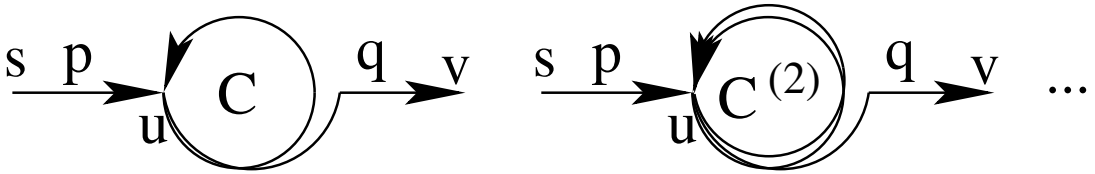


Wie finden wir heraus, welche das sind?
Endlosschleifen vermeiden!

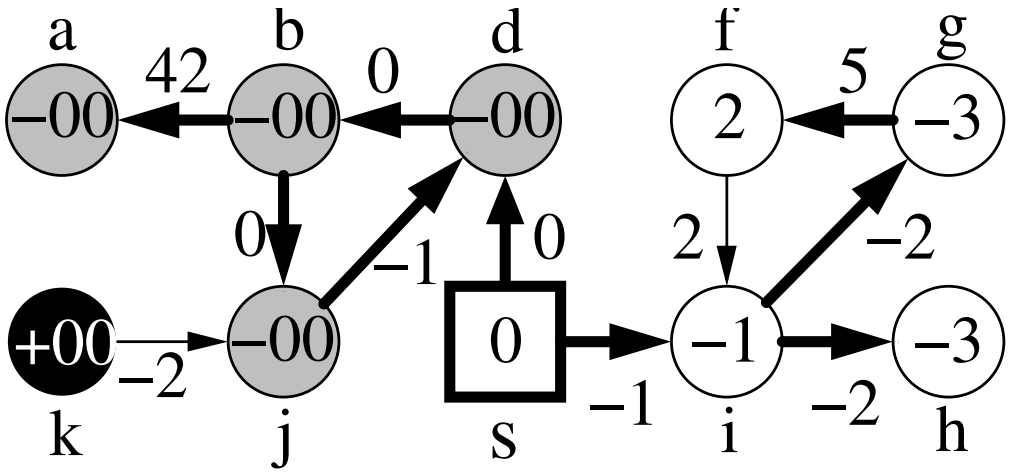


Zurück zu Basiskonzepten (Abschnitt 10.1 im Buch)

Lemma: $\exists$ kürzester s–v-Pfad $P \Longrightarrow P$ ist OBdA einfach (eng. simple)
Beweisidee: (Kontraposition)
Fall: v über negativen Kreis erreichbar $\Rightarrow$
$\neg\exists$ kürzester s–v-Pfad (sondern beliebig viele immer kürzere)



Sonst: betrachte beliebigen nicht-einfachen s–v-Pfad.
Alle Kreise streichen $\rightsquigarrow$ einfacher, nicht längerer Pfad. ■



Mehr Basiskonzepte

Übung, zeige:

Teilpfade kürzester Pfade sind selbst kürzeste Pfade

$$a-b-c-d \rightsquigarrow a-b, b-c, c-d, a-b-c, b-c-d$$

Übung: Kürzeste-Wege-Baum

Alle kürzeste Pfade von s aus zusammen bilden einen Baum, falls es keine negativen Kreise gibt.