Algorithmentheorie

Daniel Neuen (Universität Bremen)

WiSe 2023/24

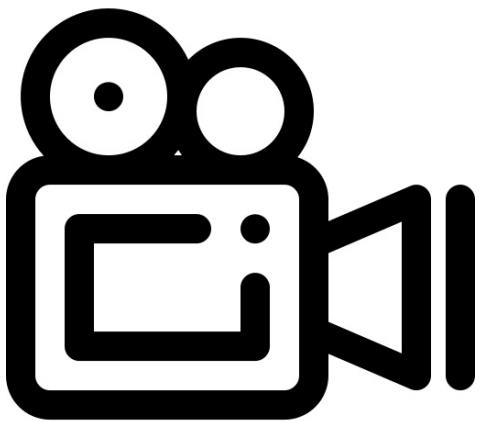
Kürzeste Wege in Graphen

8. Vorlesung

Aufzeichnung der Vorlesung

Diese Vorlesung wird aufgezeichnet und live gestreamt.

- Aufzeichnungen nur der Lehrenden durch sich selbst.
- Bei Rückfragen aus dem Auditorium und Diskussion bitte deutlich anzeigen, falls das Mikro stumm geschaltet werden soll.



Organisatorisches

Evaluation:

- Die Lehrevaluation läuft vom 27. Nov. bis zum 10. Dez.
- Bitte teilnehmen!

Folien ohne Overlays:

- Folien mit eingeschränkten Overlays (nur für Algorithmen Beispiele) im StudIP verfügbar (`handout` Version)

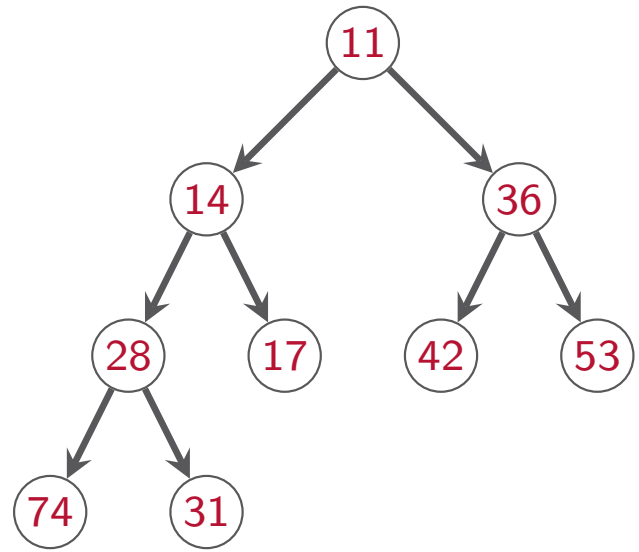
Klausur:

- Klausur findet am Montag, den 19.02.2024, 14:00-16:00 statt
- Wiederholungsklausur findet am Freitag, den 08.03.2024, 10:00-12:00 statt

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche Positionen i und n und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Falls $C[i] > \min(C[2i], C[2i+1])$, dann Vertausche Position i mit dem kleineren der beiden Elemente;
- Sonst prüfe ob $C[i] < C[\lfloor i/2 \rfloor]$; falls ja Vertausche beide Positionen
- Wiederhole rekursiv für das entsprechende Element
- Laufzeit: $\mathcal{O}(\log n)$

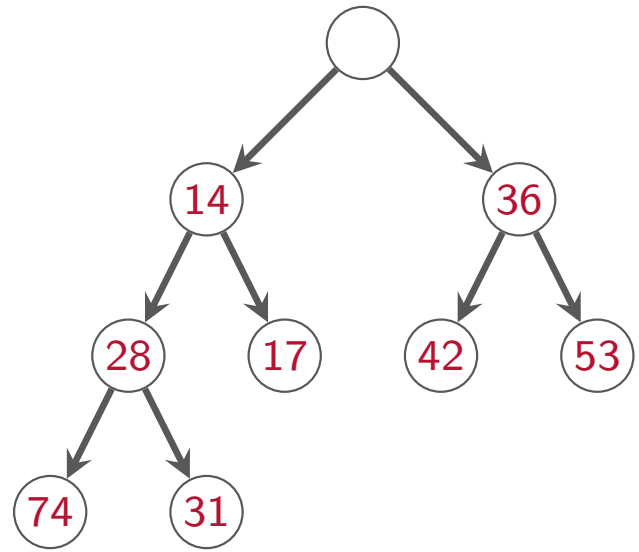


11	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche Positionen i und n und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Falls $C[i] > \min(C[2i], C[2i+1])$, dann Vertausche Position i mit dem kleineren der beiden Elemente;
- Sonst prüfe ob $C[i] < C[\lfloor i/2 \rfloor]$; falls ja Vertausche beide Positionen
- Wiederhole rekursiv für das entsprechende Element
- Laufzeit: $\mathcal{O}(\log n)$

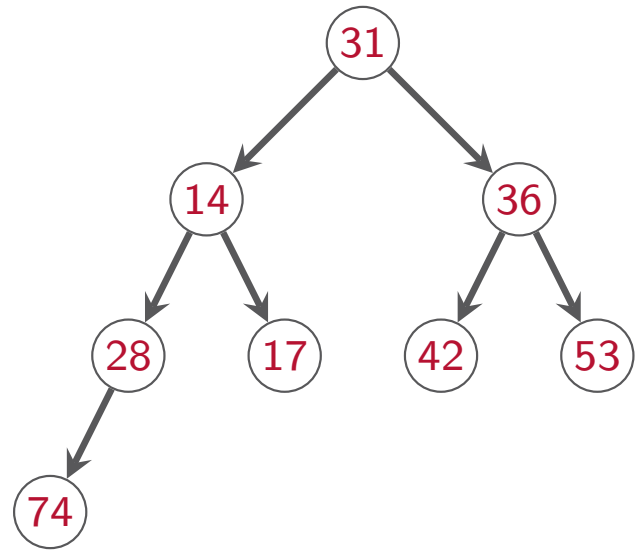


	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche Positionen i und n und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Falls $C[i] > \min(C[2i], C[2i+1])$, dann Vertausche Position i mit dem kleineren der beiden Elemente;
- Sonst prüfe ob $C[i] < C[\lfloor i/2 \rfloor]$; falls ja Vertausche beide Positionen
- Wiederhole rekursiv für das entsprechende Element
- Laufzeit: $\mathcal{O}(\log n)$

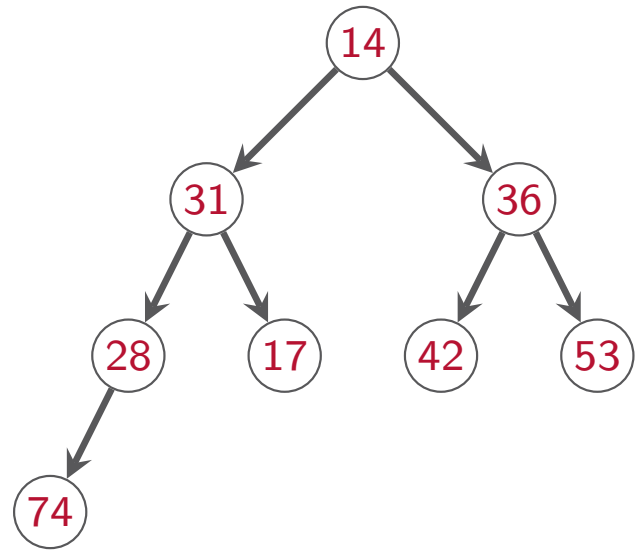


31	14	36	28	17	42	53	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche Positionen i und n und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Falls $C[i] > \min(C[2i], C[2i+1])$, dann Vertausche Position i mit dem kleineren der beiden Elemente;
- Sonst prüfe ob $C[i] < C[\lfloor i/2 \rfloor]$; falls ja Vertausche beide Positionen
- Wiederhole rekursiv für das entsprechende Element
- Laufzeit: $\mathcal{O}(\log n)$

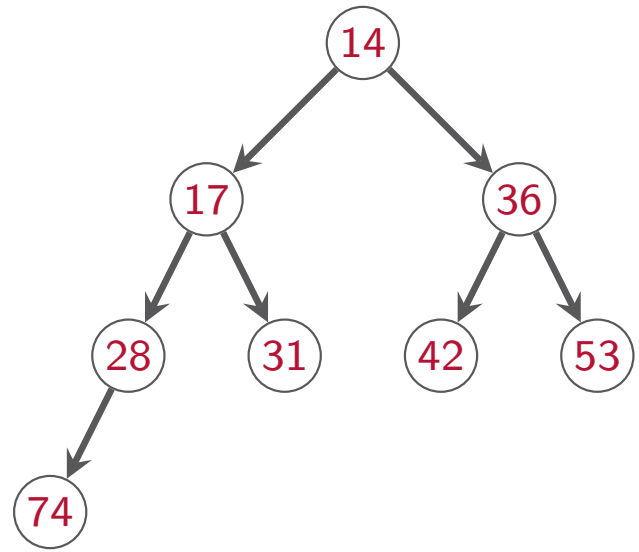


14	31	36	28	17	42	53	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche Positionen i und n und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Falls $C[i] > \min(C[2i], C[2i+1])$, dann Vertausche Position i mit dem kleineren der beiden Elemente;
- Sonst prüfe ob $C[i] < C[\lfloor i/2 \rfloor]$; falls ja Vertausche beide Positionen
- Wiederhole rekursiv für das entsprechende Element
- Laufzeit: $\mathcal{O}(\log n)$

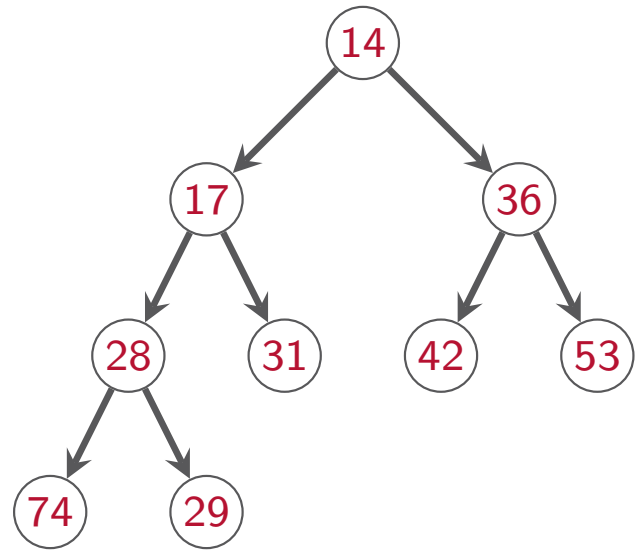


14	17	36	28	31	42	53	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche Positionen i und n und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Falls $C[i] > \min(C[2i], C[2i+1])$, dann Vertausche Position i mit dem kleineren der beiden Elemente;
- Sonst prüfe ob $C[i] < C[\lfloor i/2 \rfloor]$; falls ja Vertausche beide Positionen
- Wiederhole rekursiv für das entsprechende Element
- Laufzeit: $\mathcal{O}(\log n)$

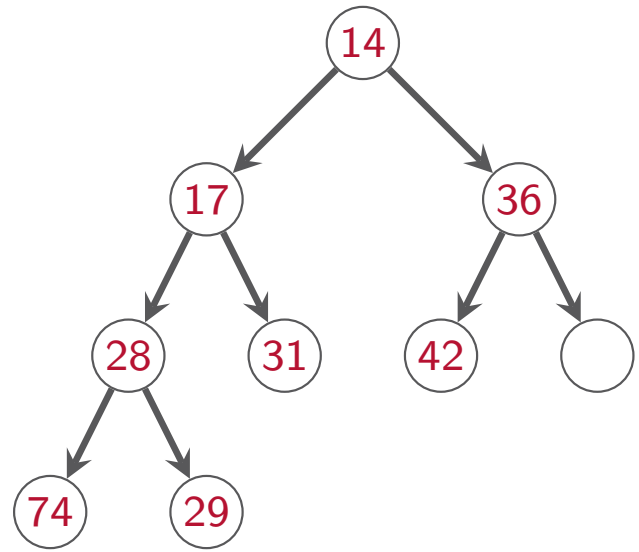


14	17	36	28	31	42	53	74	29
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche Positionen i und n und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Falls $C[i] > \min(C[2i], C[2i+1])$, dann Vertausche Position i mit dem kleineren der beiden Elemente;
- Sonst prüfe ob $C[i] < C[\lfloor i/2 \rfloor]$; falls ja Vertausche beide Positionen
- Wiederhole rekursiv für das entsprechende Element
- Laufzeit: $\mathcal{O}(\log n)$

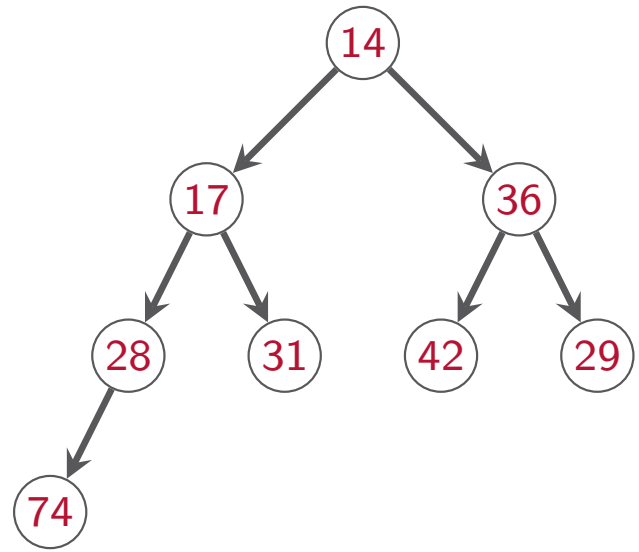


14	17	36	28	31	42		74	29
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche Positionen i und n und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Falls $C[i] > \min(C[2i], C[2i+1])$, dann Vertausche Position i mit dem kleineren der beiden Elemente;
- Sonst prüfe ob $C[i] < C[\lfloor i/2 \rfloor]$; falls ja Vertausche beide Positionen
- Wiederhole rekursiv für das entsprechende Element
- Laufzeit: $\mathcal{O}(\log n)$

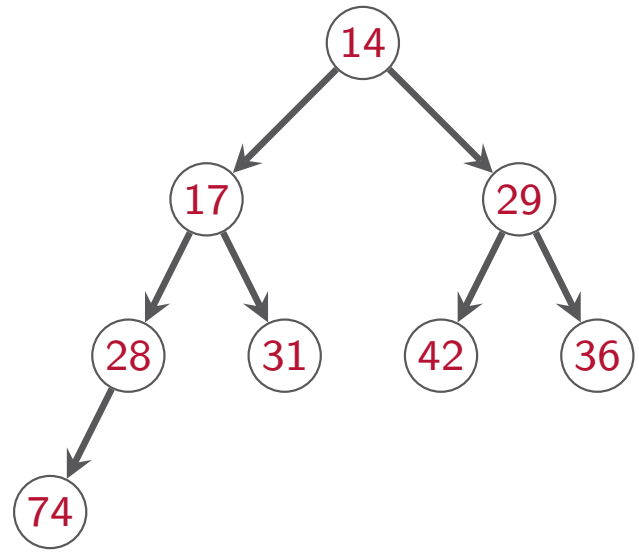


14	17	36	28	31	42	29	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

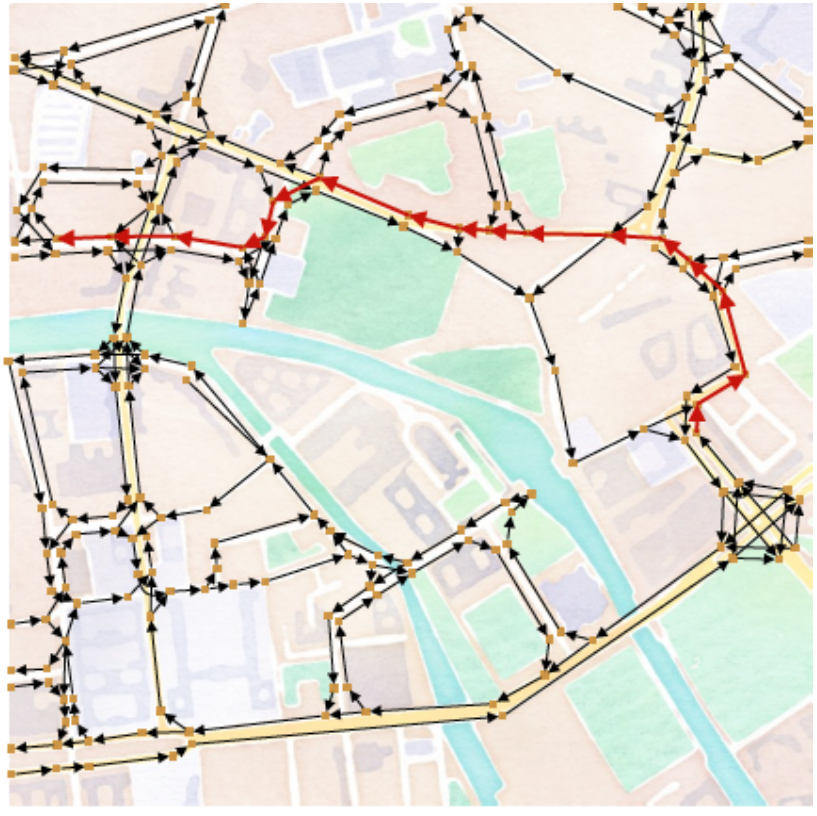
- Vertausche Positionen i und n und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Falls $C[i] > \min(C[2i], C[2i+1])$, dann Vertausche Position i mit dem kleineren der beiden Elemente;
- Sonst prüfe ob $C[i] < C[\lfloor i/2 \rfloor]$; falls ja Vertausche beide Positionen
- Wiederhole rekursiv für das entsprechende Element
- Laufzeit: $\mathcal{O}(\log n)$



14	17	29	28	31	42	36	74	
1	2	3	4	5	6	7	8	9

Kürzeste Wege in Graphen

Kürzeste Wege in Graphen



Weg und Kosten eines Weges

Definitionen

1. Ein Weg (auch Kantenzug) in einem gerichteten Graphen $G = (V, E)$ von einem Knoten v zu einem Knoten w ist eine endliche Folge von Knoten $v_1, v_2, ..., v_{k+1}$, sodass:

   $e_i = (v_i, v_{i+1}) \in E$ für $1 \leq i \leq k$ und
   $v = v_1$ und $w = v_{k+1}$.

2. Ein (gerichteter) Weg $W = v_1, \ldots, v_{k+1}$ mit Kanten $e_1, e_2, \ldots, e_k$ in einem gewichteten Graphen $G = (V, E, c)$ hat Kosten

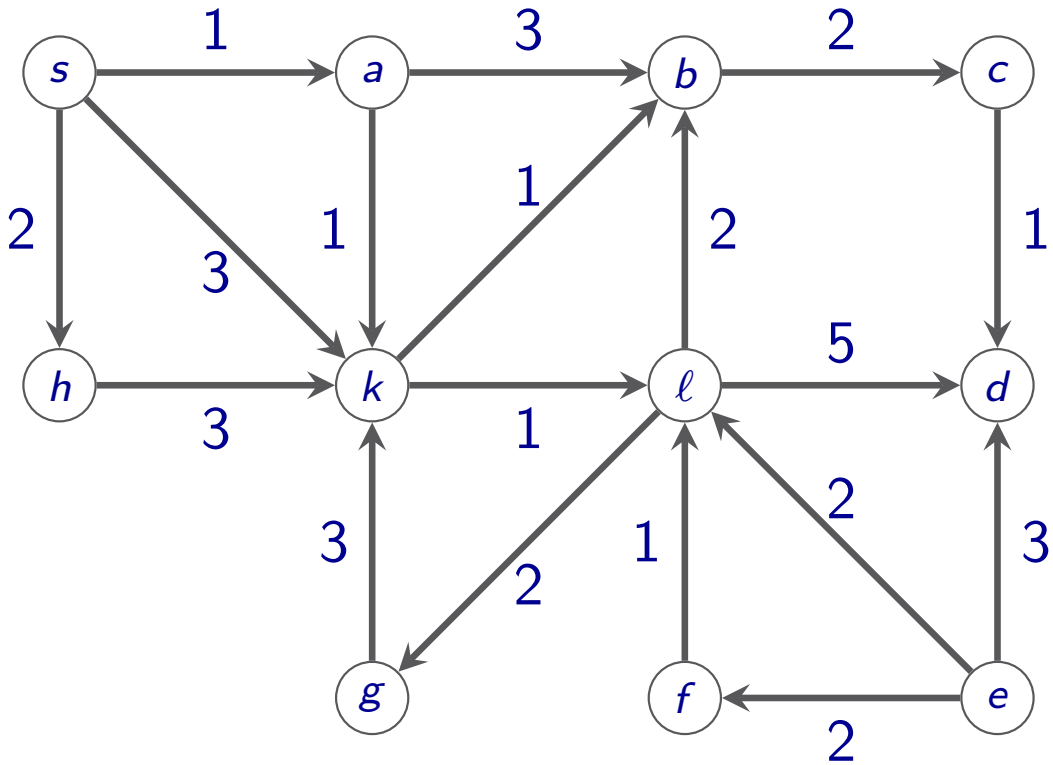
$$c(P) = \sum_{i=1}^{k} c(e_k).$$

3. Die Distanz zwischen $v, w \in V$ ist

$$d(v, w) := \min\{c(P) \mid P \text{ Weg von } v \text{ nach } w\}.$$

Sei $d(v, w) = \infty$, falls kein Weg von v nach w existiert; $d(v, v) = 0$.

Beispiel



$c(s, a, b, c) = ???$

$d(s, c) = ???$

$d(s, s) = ???$

$d(s, e) = ???$

Das Kürzeste-Wege-Problem

Gegeben: gewichteter Digraph $G = (V, E, c)$ und ein Startknoten $s \in V$.

Gesucht: für jedes $v \in V$ ein kürzester Weg von s nach v und die Distanz $d(s, v)$. (single-source shortest-paths)

Viele Fragestellungen (neben Routing) lassen sich als Kürzeste-Wege-Problem modellieren, z.B. zeitliche Kauf-Verkauf Entscheidungen, Arbitrage in Finanzmärkten, Sequencing, etc.

Beispiel: Kauf- und Verkaufentscheidungen

- Autopreis: 12.000 €, Wartungskosten abhängig vom Jahr (s. Tabelle)
- Jedes Jahr 2 Optionen: Halten oder Verkauf (Erlöse s. Tabelle) und Neukauf
- Seien c_{ij} die Kosten für Kauf eines Autos in Jahr i, Halten in den Jahren i bis $j-1$ und Verkauf zu Beginn von Jahr j, d.h.:

  $c_{ij} = 12.000 + (\text{Wartung in } i, i+1, \ldots, j-1) - (\text{Verkaufserlös zu Beginn von } j)$

Alter Auto (Jahre)	Wartung (in €)
0	2.000
1	4.000
2	5.000
3	9.000
4	12.000

Alter Auto (Jahre)	Erlös (in €)
1	7.000
2	6.000
3	2.000
4	1.000
5	0

Bsp. Kosten (in 1.000 €)

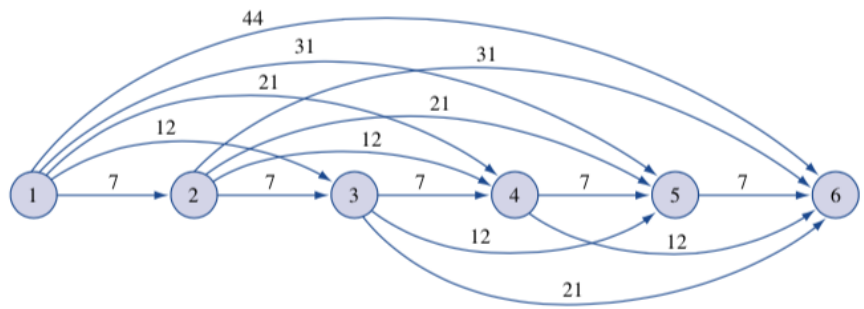
$c_{12} = 12 + 2 - 7 = 7$

$c_{13} = 12 + 2 + 4 - 6 = 12$

$c_{25} = 12 + 2 + 4 + 5 - 2 = 21$

$c_{35} = 12 + 2 + 4 - 6 = 12$

Die minimalen Kauf- und Haltungskosten für eine 5-Jahresperiode entsprechen der Länge des kürzesten Weges von 1 nach 6 im abgebildeten Graphen.



Arbitrage in Devisenmärkten

Gegeben sei eine Tabelle mit Wechselkursen, welches ist der beste Wechsel von 1 oz. Gold in Dollar?

- 1 oz. Gold $\rightarrow$ \$327, 25
- 1 oz. Gold $\rightarrow$ £208, 10 und £1 $\rightarrow$ \$1, 5714 also
  1 oz. Gold $\rightarrow$ \$(208, 10 $\cdot$ 1, 5714) = \$327
- 1 oz. Gold $\rightarrow$ 455, 2 Francs $\rightarrow$ 304, 39 Euro $\rightarrow$ \$327, 28

Currency	£	Euro	¥	Franc	$	Gold
UK Pound	1.0000	0.6853	0.005290	0.4569	0.6368	208.100
Euro	1.4599	1.0000	0.007721	0.6677	0.9303	304.028
Japanese Yen	189.050	129.520	1.0000	85.4694	120.400	39346.7
Swiss Franc	2.1904	1.4978	0.011574	1.0000	1.3929	455.200
US Dollar	1.5714	1.0752	0.008309	0.7182	1.0000	327.250
Gold (oz.)	0.004816	0.003295	0.0000255	0.002201	0.003065	1.0000

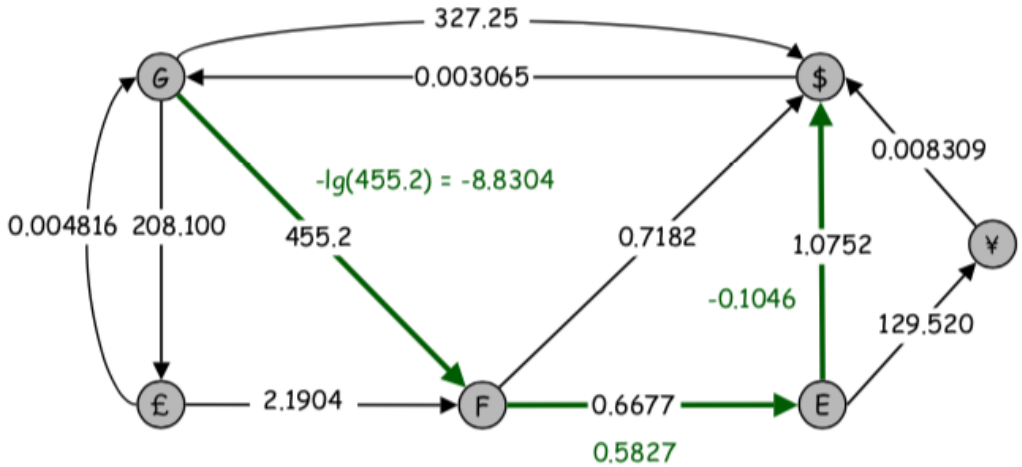
Arbitrage = Ausnutzen von Wechselkursen

Beispiel: Arbitrage in Devisenmärkten

Formulierung als Graph mit

- einem Knoten pro Währung und
- einer Kante für eine mögliche Transaktion mit Gewicht entsprechend dem Wechselkurs.

Finde einen Weg, der das Produkt der Gewichte maximiert.



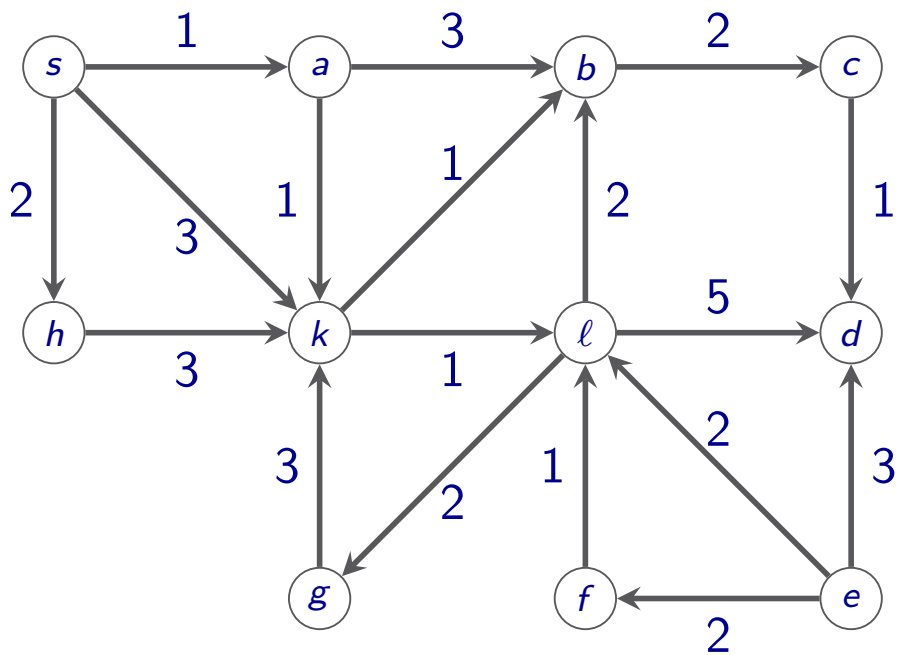
Reduktion auf das Kürzeste-Wege Problem:

- Sei γ_{uv} Wechselkurs, dann setze Kantengewicht $c_{uv} = -\log \gamma_{uv}$.
- Kürzester (u, v)-Weg entspricht der besten Wechselsequenz u in v.

Das Kürzeste-Wege-Problem

Gegeben: gewichteter Digraph $G = (V, E, c)$ und ein Startknoten $s \in V$.

Gesucht: für jedes $v \in V$ ein kürzester Weg von s nach v und die Distanz $d(s, v)$. (single-source shortest-paths)



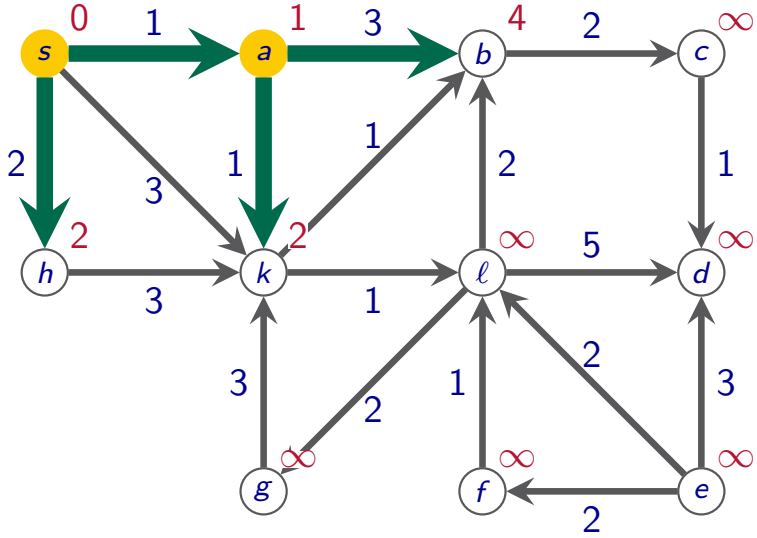
Dijkstra's Algorithmus

Kürzeste-Wege-Problem mit nicht-negativen Kantenkosten

Edsger W. Dijkstra ("Daik-stra"), 1930–2002
niederländischer Informatiker

Idee Dijkstra's Algorithmus

- Baue sukzessive Menge S von explorierten Knoten auf (für die schon ein kürzester Weg von Startknoten s bekannt ist).
- $dist(v)$ vorläufige Distanz von s nach v
  Initialisierung: $dist(s) = 0$. Für alle $v \neq s$ sei $dist(v) = \infty$.
- Pro Runde: Bestimme Knoten $u \in V \setminus S$ mit min. vorläufiger Distanz $dist(u)$. Füge u zu S hinzu und friere $dist(u)$ ein.



- Aktualisiere $dist$-Werte (wenn ein anderer Weg gefunden)

$$dist(v) \leftarrow \min\{dist(v), dist(u) + c(u, v)\}.$$

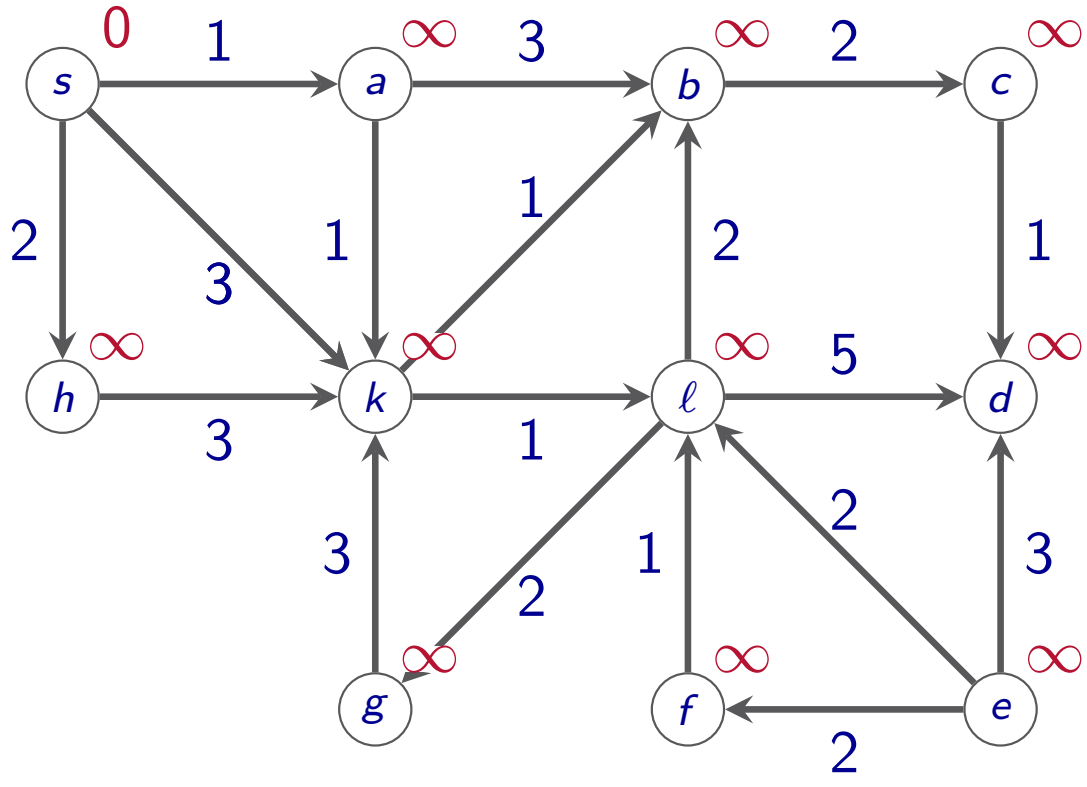
Dijkstra's Algorithmus

Dijkstra's Algorithmus

Input : gewichteter Digraph $G = (V, E, c)$, Startknoten $s \in V$
Output: $d(s, v)$ für alle $v \in V$ (= Kosten des günstigsten Weges von s zu allen Knoten in G)

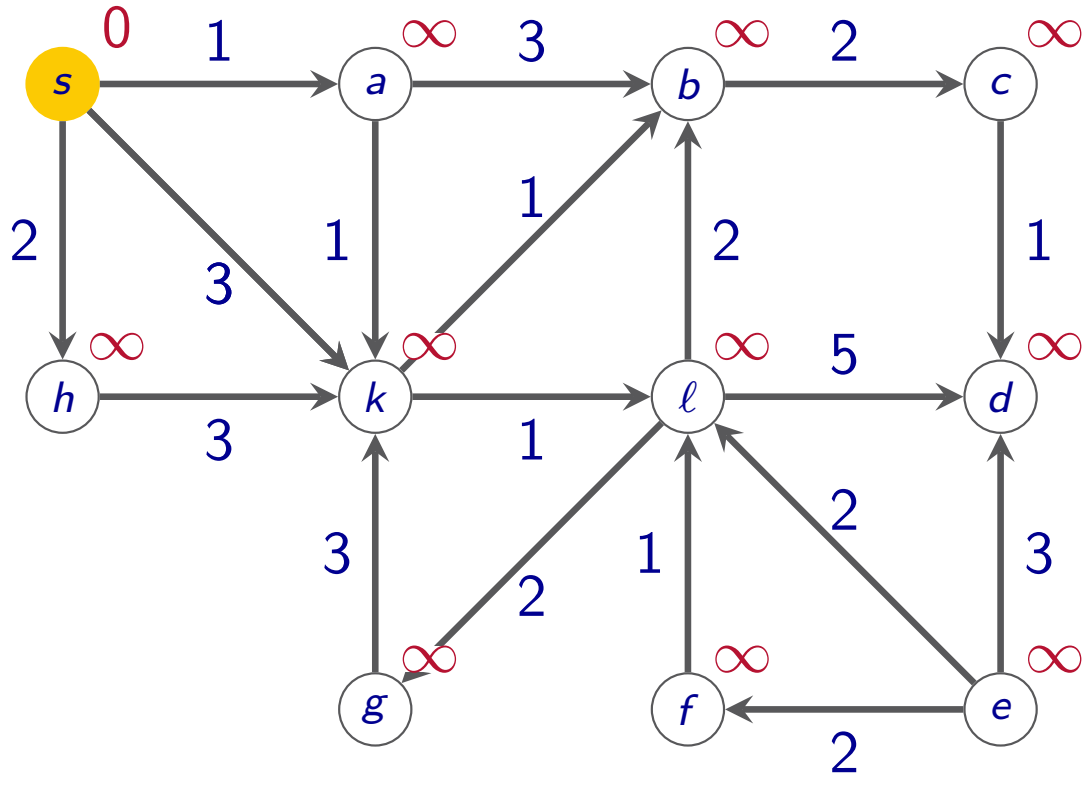
1 **Initialisiere** $S \leftarrow \emptyset$, $dist(s) \leftarrow 0$ und $\forall v \in V \setminus \{s\} : dist(v) \leftarrow \infty$
2 **while** $S \neq V$ **do**
3 Finde $u \in V \setminus S$ mit minimaler vorläufiger Distanz $dist(u)$
4 $S \leftarrow S \cup \{u\}$
5 **for** $v \in V \setminus S$ mit $(u, v) \in E$ **do**
6 $dist(v) \leftarrow \min\{dist(v), dist(u) + c(u, v)\}$
7 **return** $dist(v)$ für alle $v \in V$

Beispiel



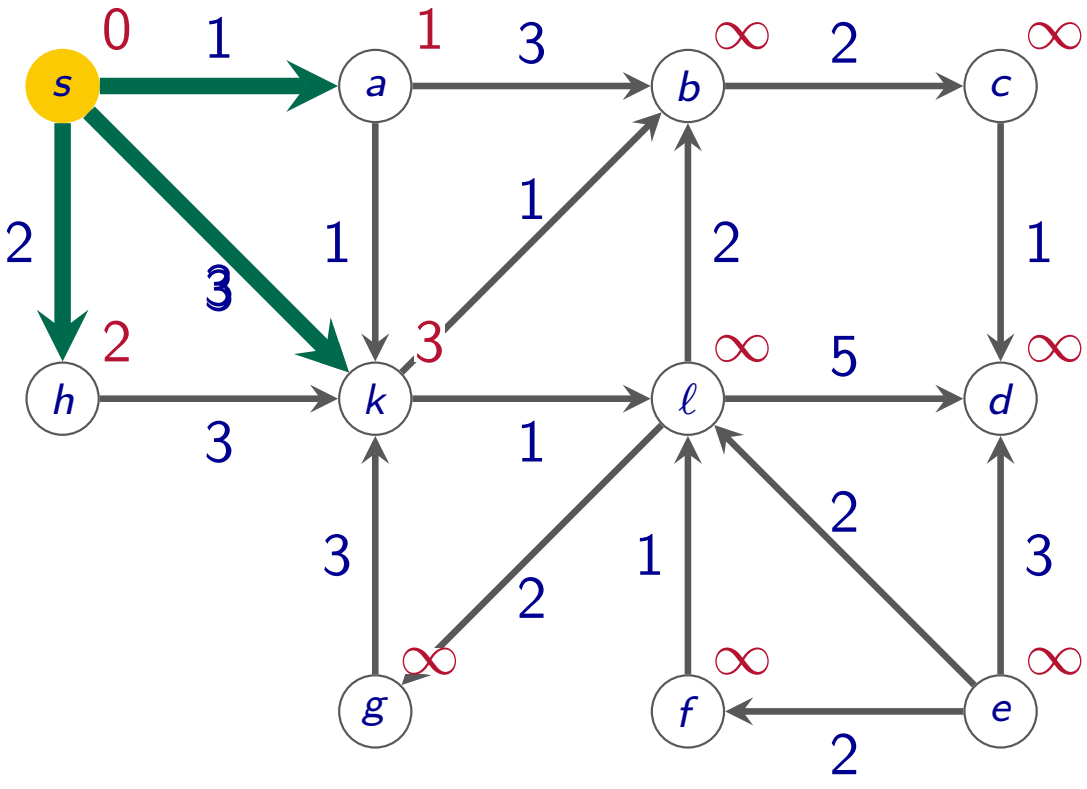
- Initialisiere: $dist(s) = 0$, $dist(v) = \infty$, $S = \emptyset$

Beispiel



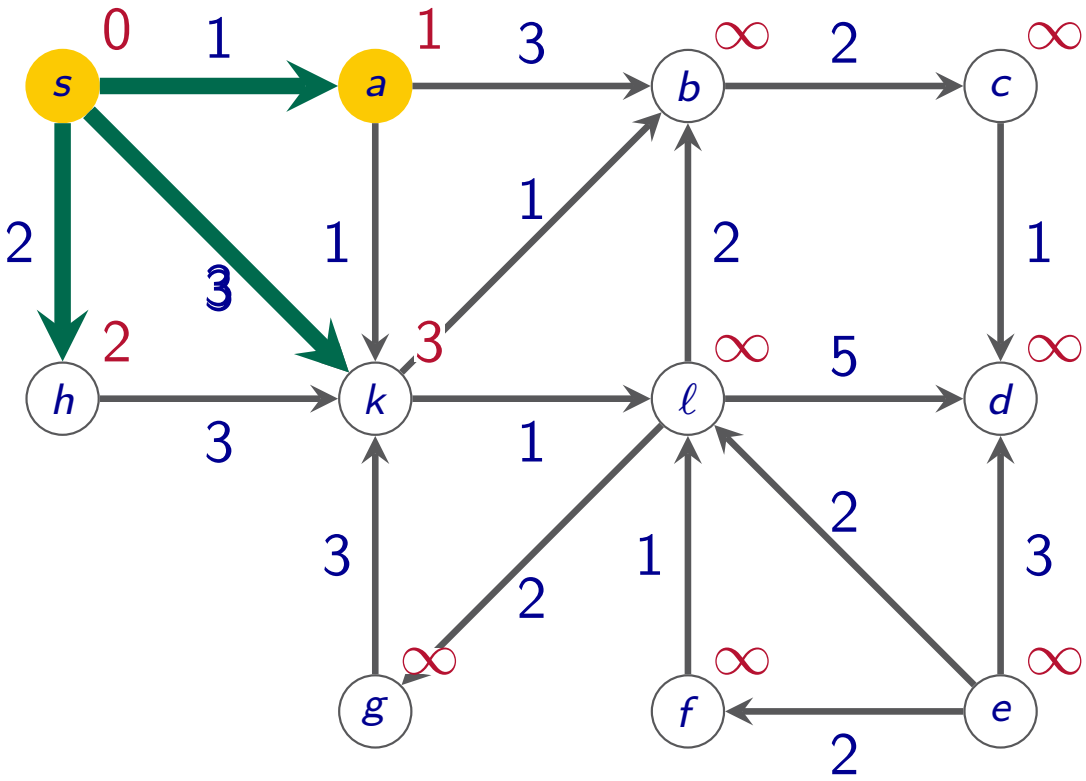
- Wähle s, da $dist(s) = 0$ minimal.

Beispiel



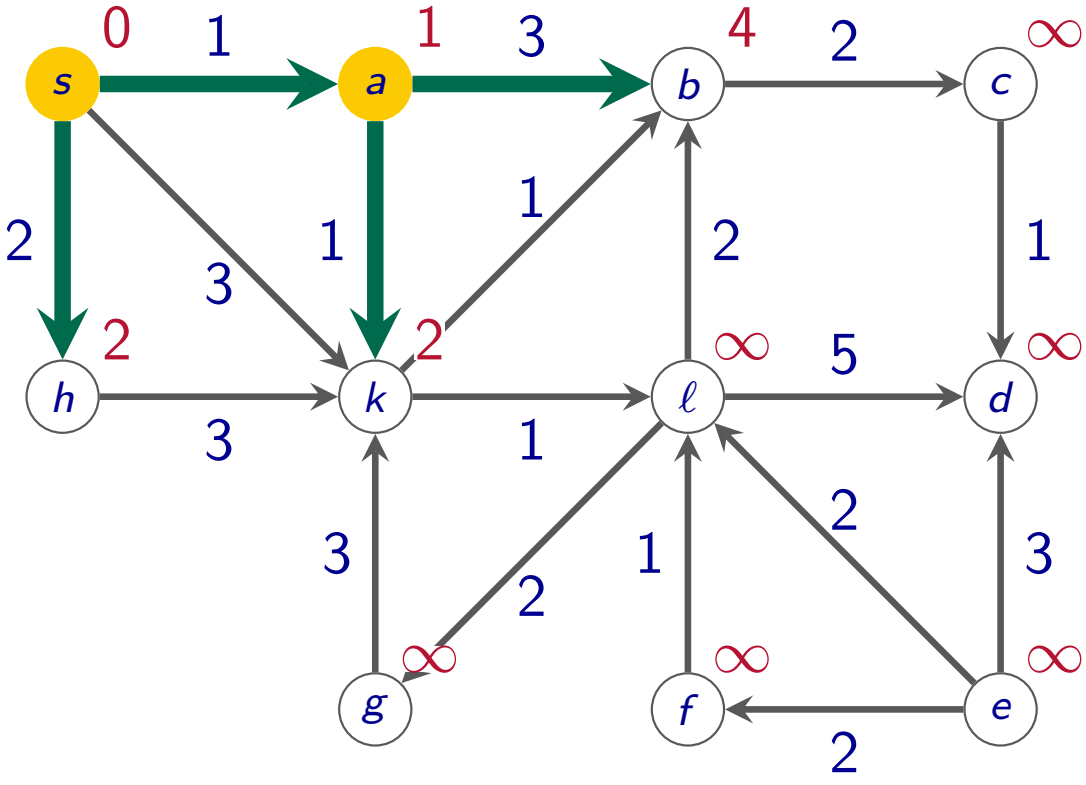
- Wege zu a, h und k neu, deshalb überschreibe $dist$.

Beispiel



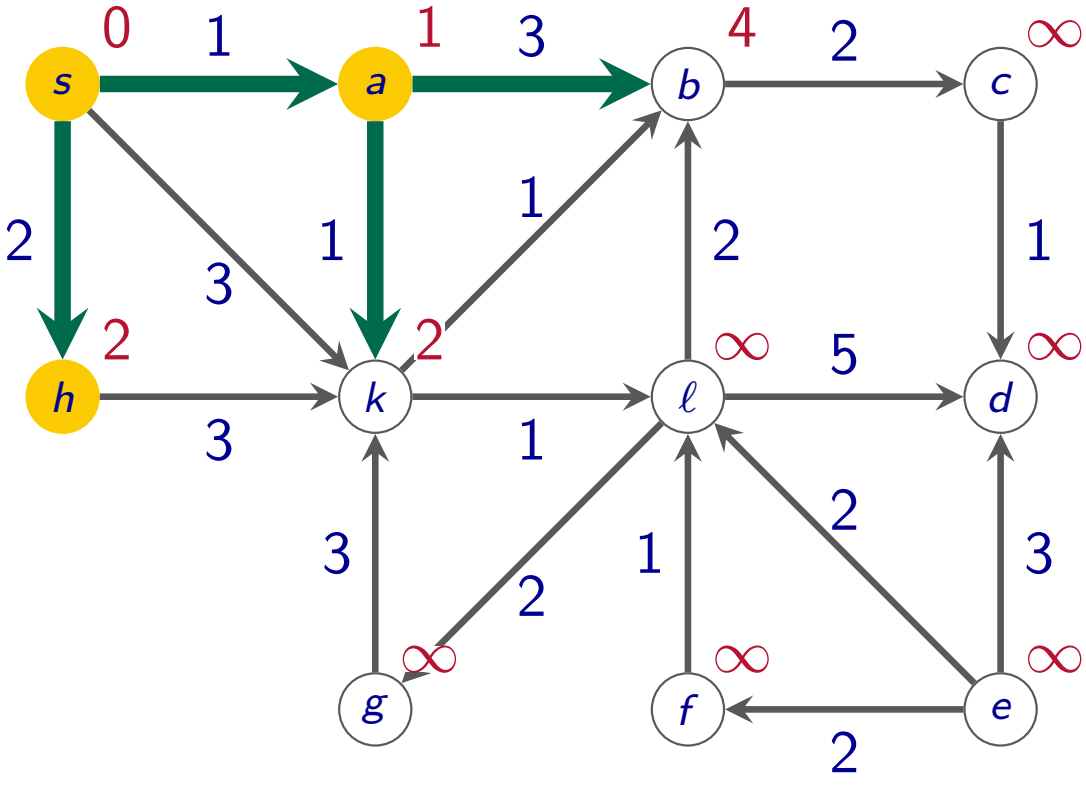
- Wähle a, da $dist(a) = 1$ minimal unter Knoten $V \setminus S$.
- $dist(k) = \min\{3, 1+1\} = 2$, deshalb kürzester Weg zu k nun über a.
- Weg zu b ist neu, $dist(b) = 1 + 3$.

Beispiel



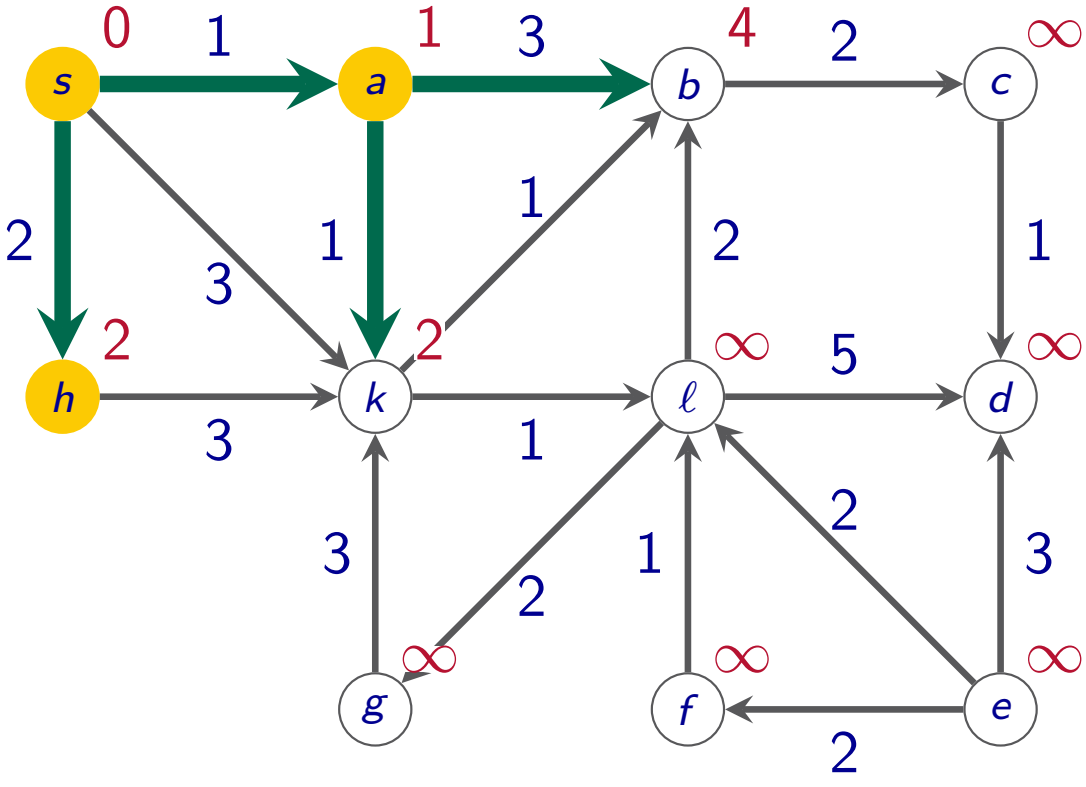
- Wähle a, da $dist(a) = 1$ minimal unter Knoten $V \setminus S$.
- $dist(k) = \min\{3, 1+1\} = 2$, deshalb kürzester Weg zu k nun über a.
- Weg zu b ist neu, $dist(b) = 1 + 3$.

Beispiel



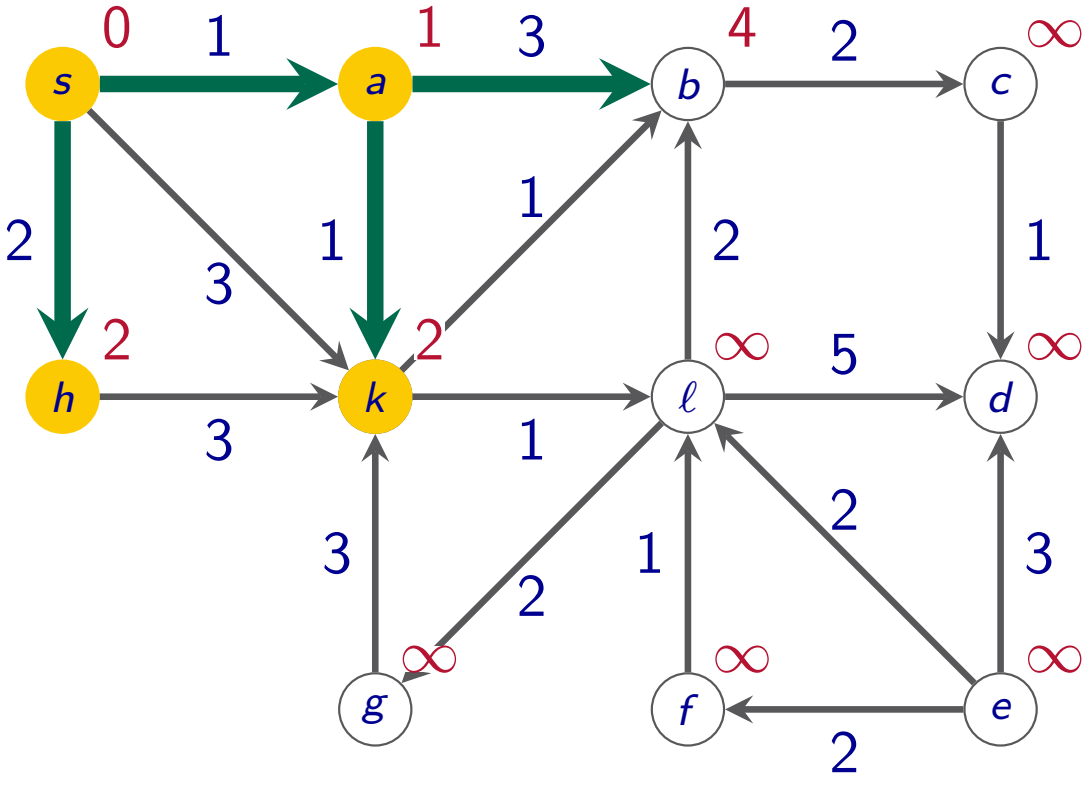
- Wähle h.
- $dist(k) = \min\{2, 2+3\} = 2$, deshalb ändern wir den bisher kürzesten Weg zu k (und das entsprechende *dist*-Label) nicht.

Beispiel



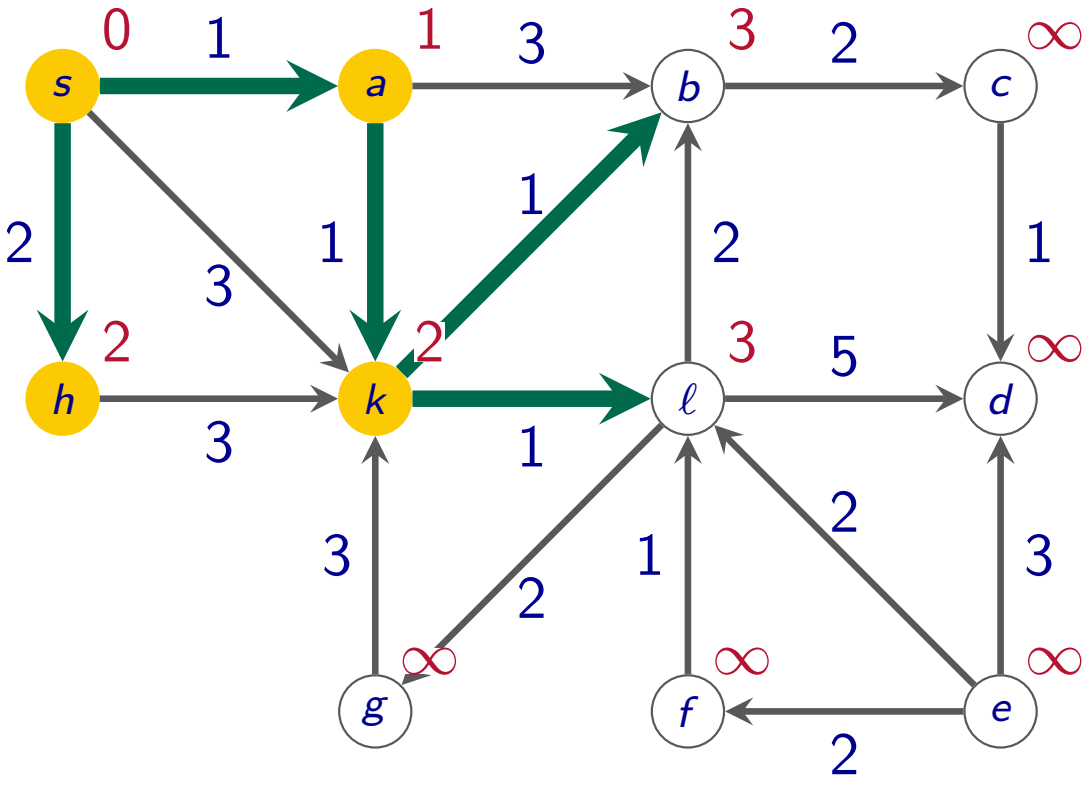
- Wähle h.
- $dist(k) = \min\{2, 2+3\} = 2$, deshalb ändern wir den bisher kürzesten Weg zu k (und das entsprechende *dist*-Label) nicht.

Beispiel



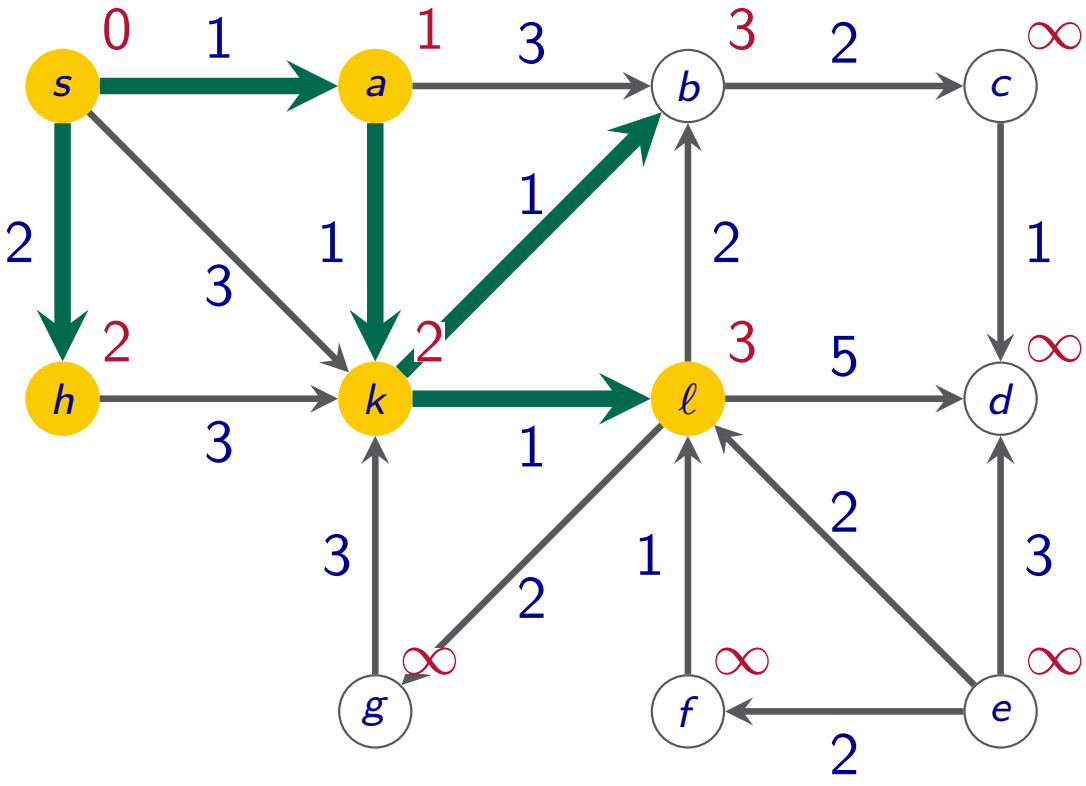
- Wähle k.
- $dist(b) = \min\{4, 2+1\} = 3$, deshalb kürzester Weg zu b nun über k.
- Weg zu ℓ ist neu, $dist(\ell) = 2+1$.

Beispiel



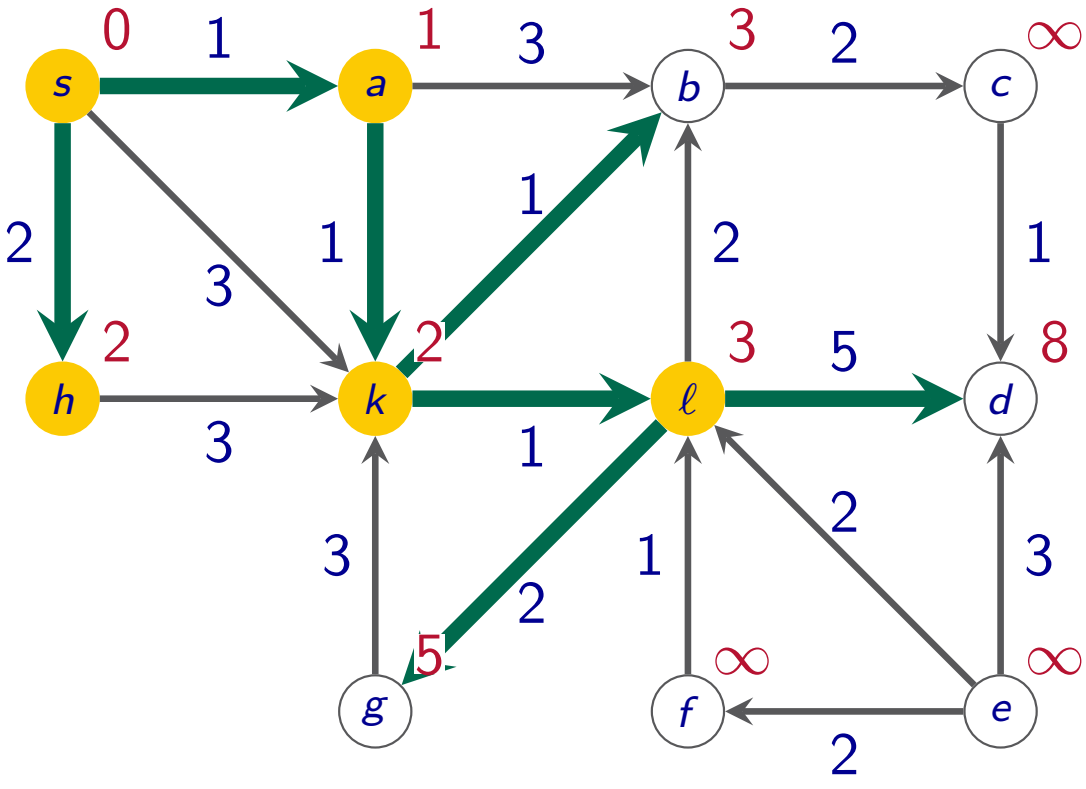
- Wähle k.
- $dist(b) = \min\{4, 2+1\} = 3$, deshalb kürzester Weg zu b nun über k.
- Weg zu ℓ ist neu, $dist(\ell) = 2+1$.

Beispiel



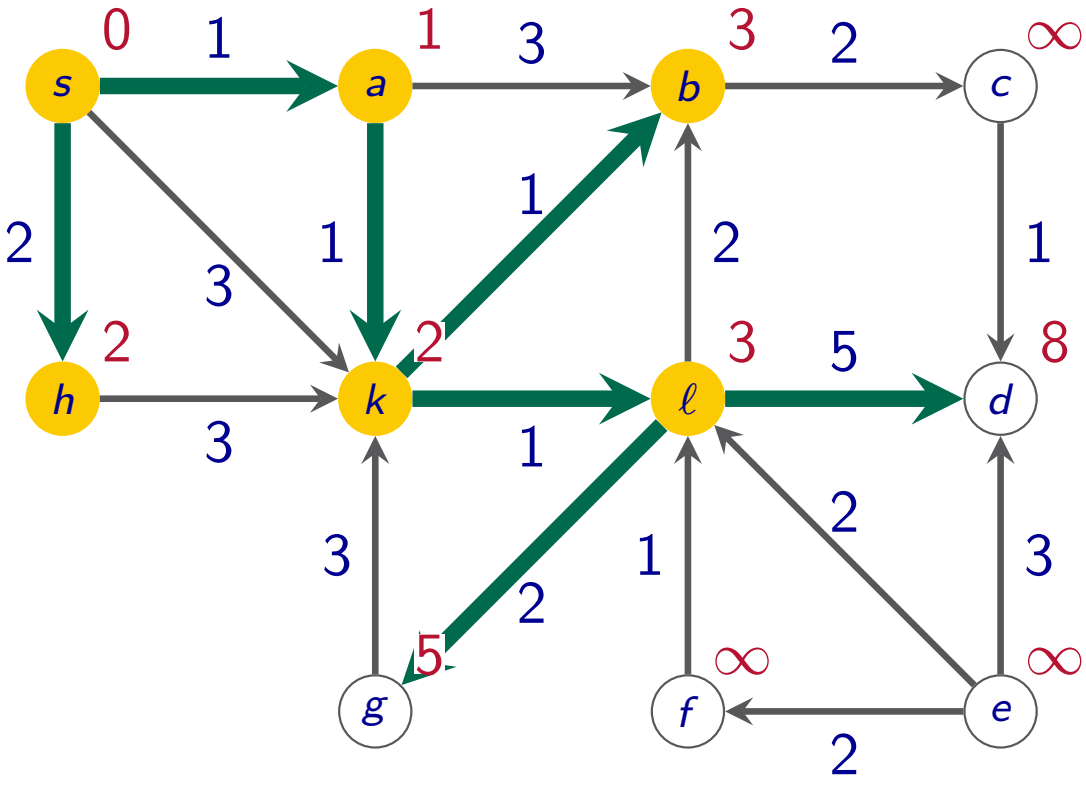
- Wähle ℓ.
- $dist(b) = \min\{3, 3+2\} = 3$, deshalb ändern wir den kürzesten Weg zu b nicht.
- Wege zu g und d sind neu.

Beispiel



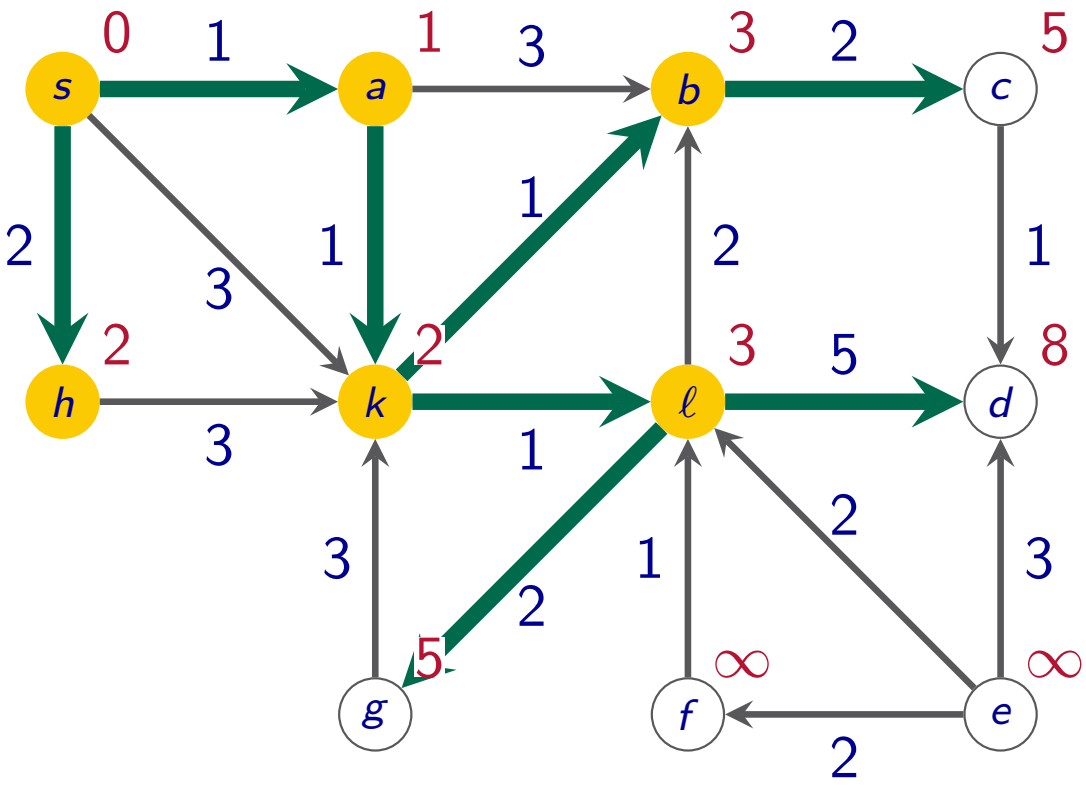
- Wähle ℓ.
- $dist(b) = \min\{3, 3+2\} = 3$, deshalb ändern wir den kürzesten Weg zu b nicht.
- Wege zu g und d sind neu.

Beispiel



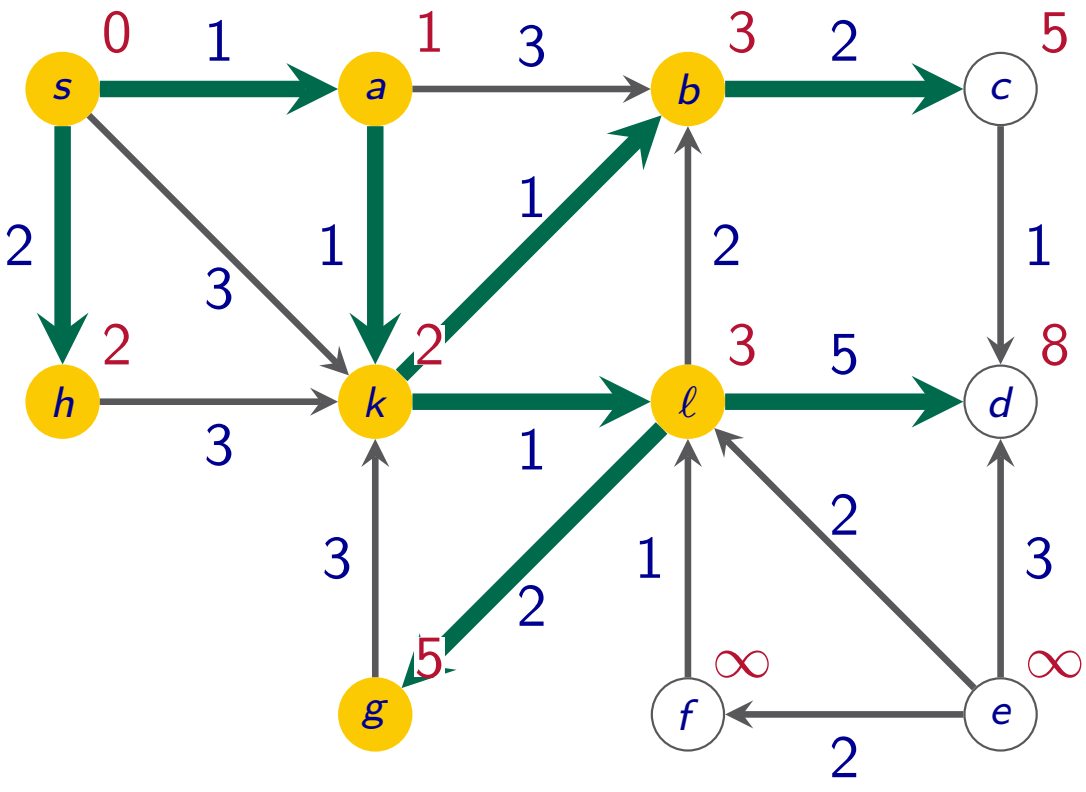
- Wähle b.
- $dist(b) = \min\{3, 3+2\} = 3$, deshalb ändern wir den kürzesten Weg zu b nicht.
- Weg zu c ist neu, also setze $dist(c) = 3 + 2$.

Beispiel



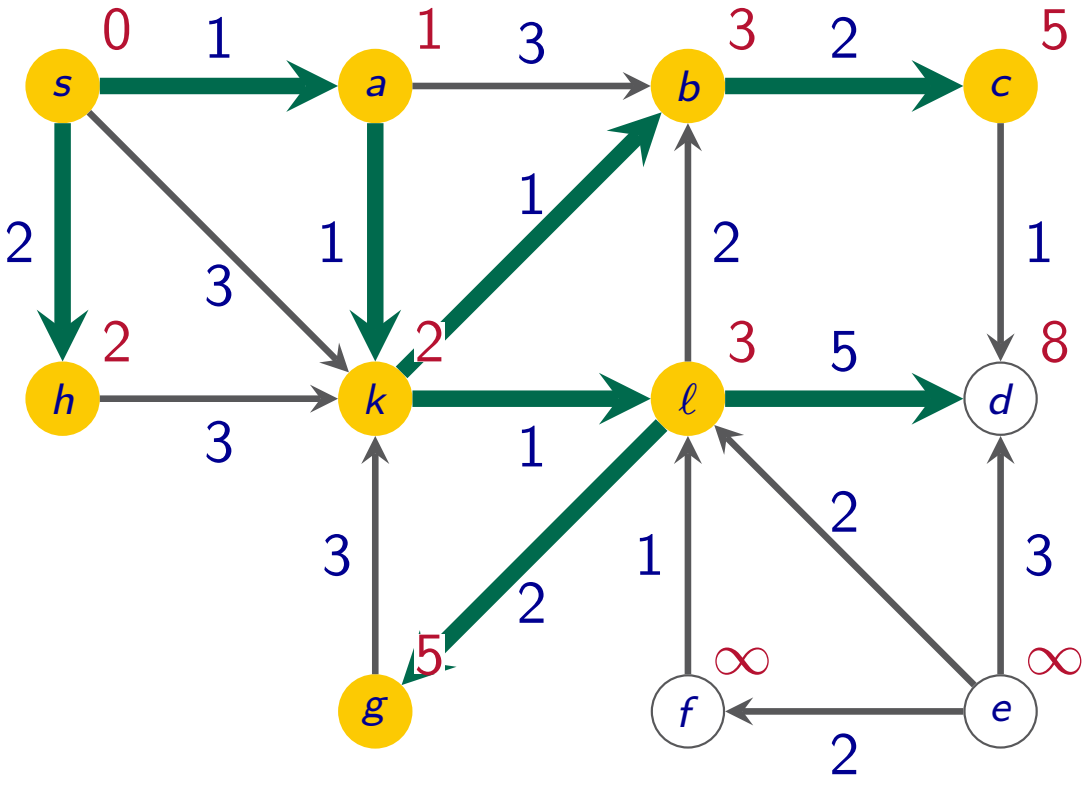
- Wähle b.
- $dist(b) = \min\{3, 3+2\} = 3$, deshalb ändern wir den kürzesten Weg zu b nicht.
- Weg zu c ist neu, also setze $dist(c) = 3 + 2$.

Beispiel



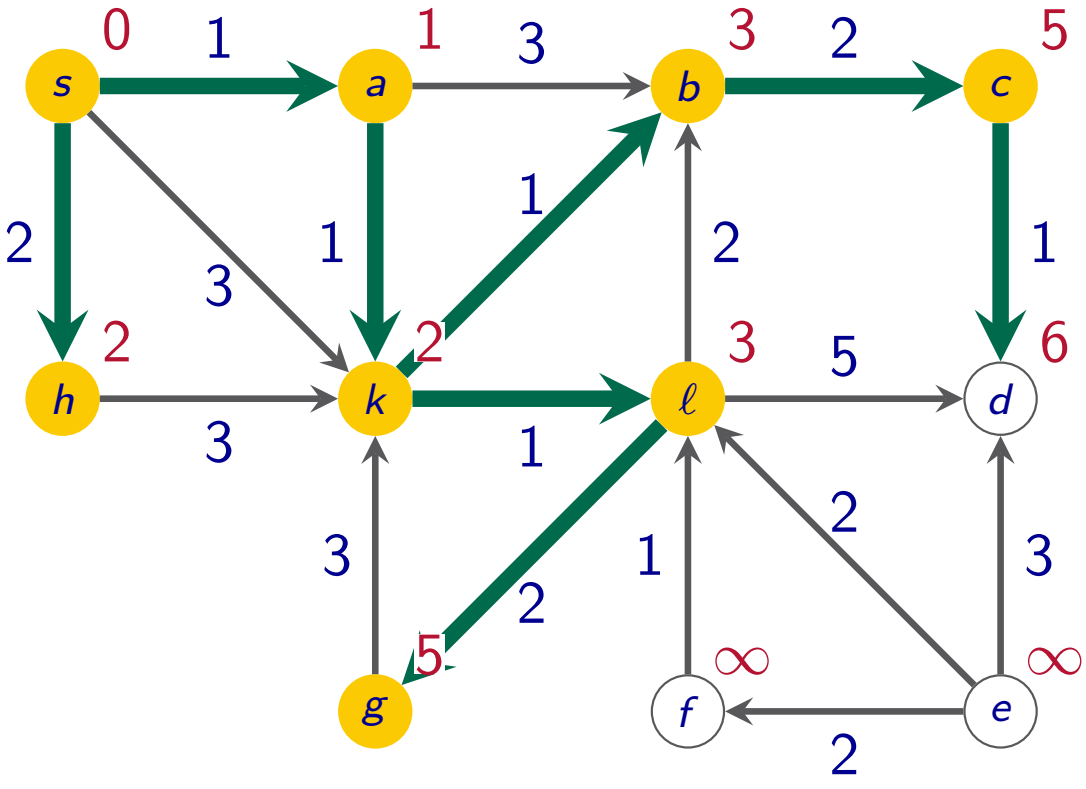
- Wähle g.
- $dist(k) = \min\{2, 5 + 3\} = 2$, deshalb ändern wir den kürzesten Weg zu k nicht.

Beispiel



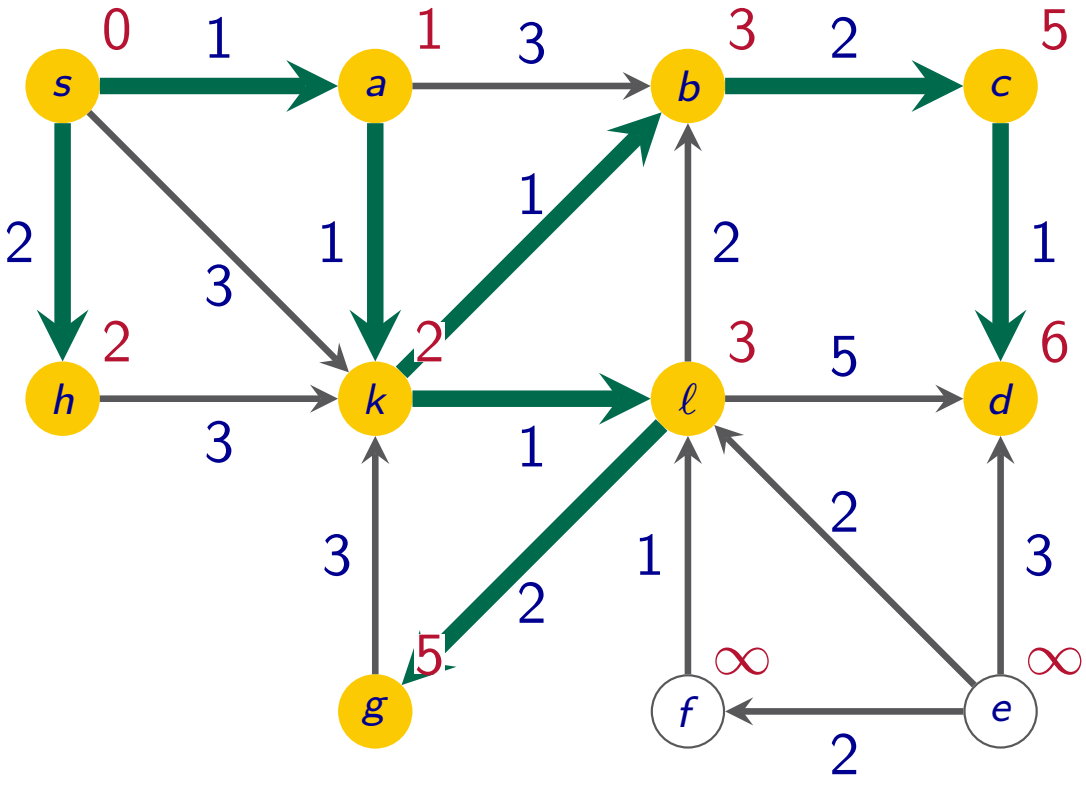
- Wähle c.
- $dist(d) = \min\{8, 5+1\} = 6$, deshalb kürzester Weg zu d nun über c.

Beispiel



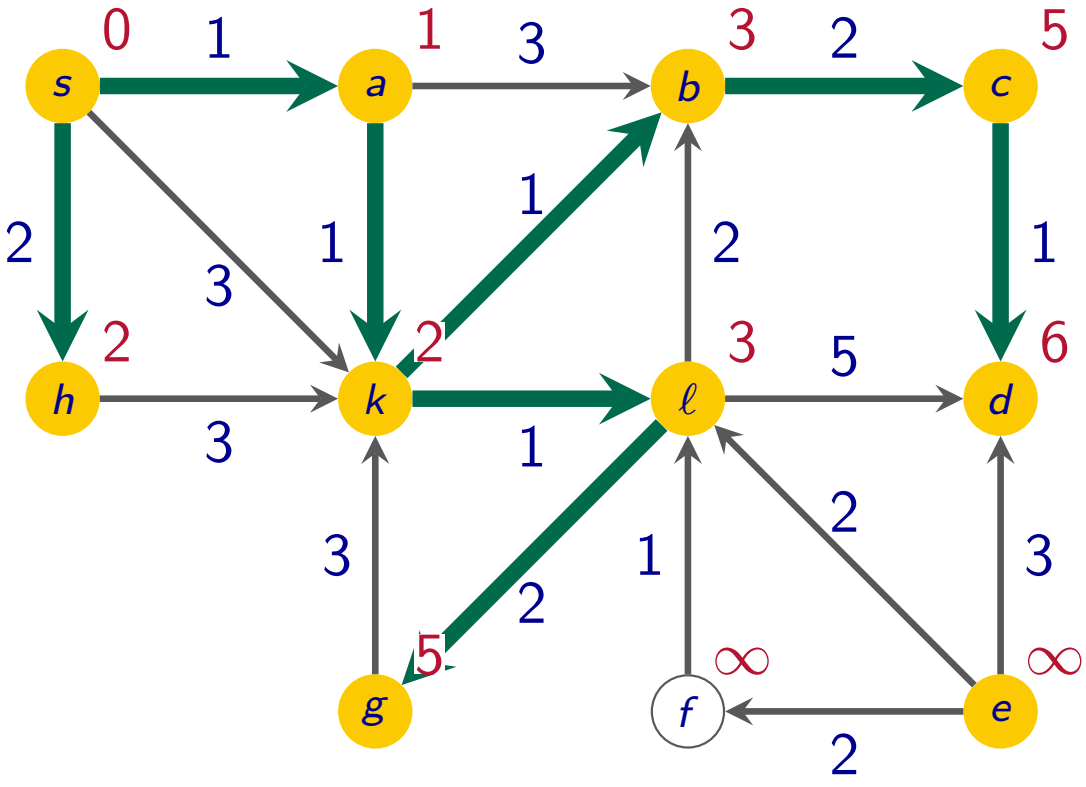
- Wähle c.
- $dist(d) = \min\{8, 5+1\} = 6$, deshalb kürzester Weg zu d nun über c.

Beispiel



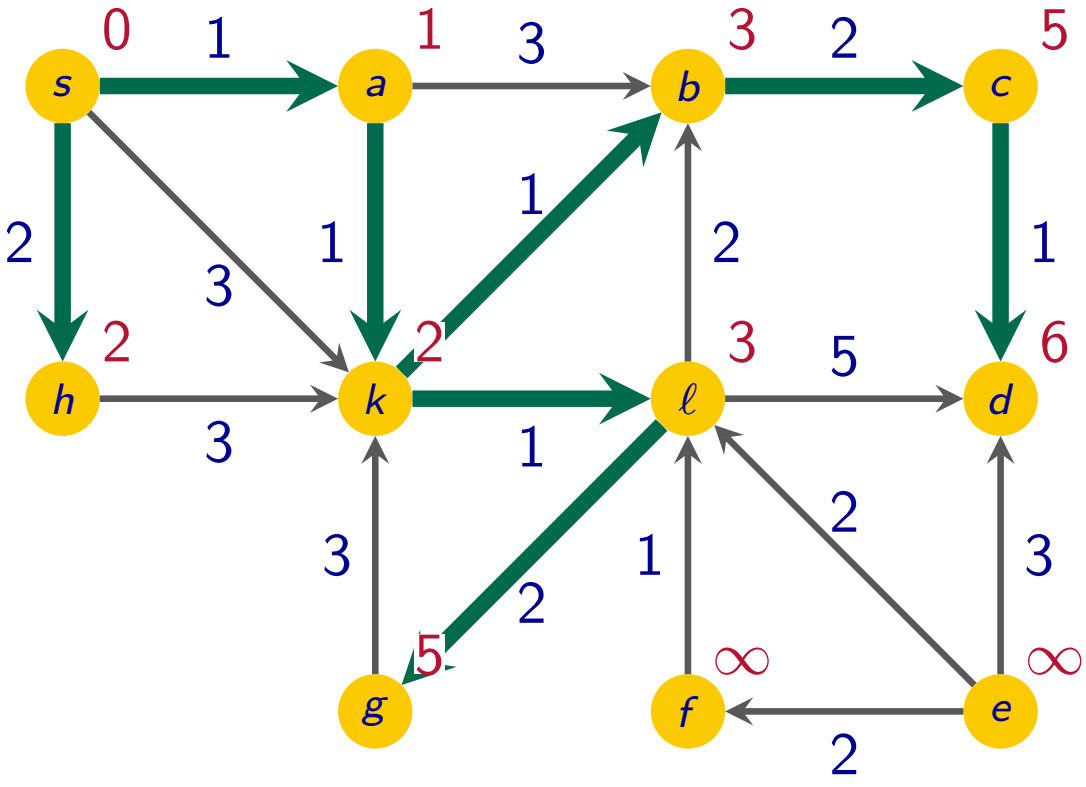
- Wähle d.
- d hat keine ausgehenden Kanten, deshalb sind wir mit d fertig.

Beispiel



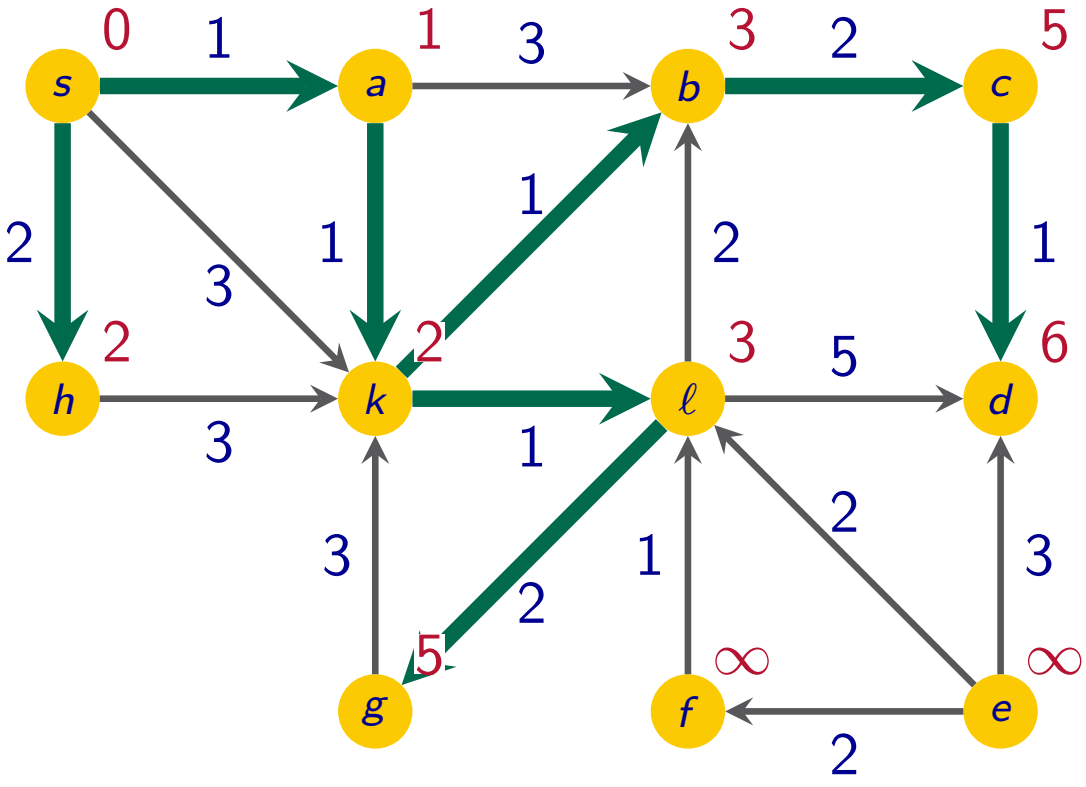
- Wähle e.
- $dist(e) = \infty$, deshalb ist keine Verbesserung von $dist(\ell)$ oder $dist(d)$ möglich.

Beispiel



- Wähle f.
- $dist(f) = \infty$, deshalb ist keine Verbesserung von $dist(\ell)$ möglich.

Beispiel



- Algorithmus terminiert, da alle Knoten besucht wurden ($S = V$).

Analyse Dijkstra's Algorithmus

Satz

Sei $G = (V, E, c)$ mit $c : E \to \mathbb{R}_+$ und $|V| = n$ und $|E| = m$. Dann hat Dijkstra's Algorithmus eine Laufzeit von $\mathcal{O}((m + n) \log n)$ und berechnet die kürzeste Distanz $d(s, v)$ für alle $v \in V$.

Beweis.

Korrektheit: Zu zeigen: für alle $v \in V$: $dist(v) = d(s, v)$. (Tafel)

Analyse Dijkstra's Algorithmus

Laufzeit: Nutze Binary Heap für Knoten in $V \setminus S$ mit Priorität $dist(v)$

Dijkstra's Algorithmus

1	**Initialisiere** $S \leftarrow \emptyset$, $dist(s) \leftarrow 0$, $\forall v \in V \setminus \{s\} : dist(v) \leftarrow \infty$	$\mathcal{O}(n)$
2	**while** $S \neq V$ **do**	n mal
3	Finde $u \in V \setminus S$ mit minimaler vorläufiger Distanz $dist(u)$	$\mathcal{O}(\log n)$
4	$S \leftarrow S \cup \{u\}$	$\mathcal{O}(1)$
5	**for** $v \in V \setminus S$ mit $(u, v) \in E$ **do**	$\deg(u)$ mal
6	$dist(v) \leftarrow \min\{dist(v), dist(u) + c(u, v)\}$	$\mathcal{O}(\log n)$
7	**return** $dist(v)$ für alle $v \in V$	$\mathcal{O}(n)$

Insgesamt: $\mathcal{O}\Big(n \log n + \sum_{u \in V}(deg(u) \cdot \log n)\Big) = \mathcal{O}((n+m)\log n)$. $\square$

Bemerkungen

Laufzeit: Unter Verwendung von speziellen Prioritätswarteschlangen (Fibonacci-Heaps; Fredman, Tarjan 1987) lässt sich Dijkstra's Algorithmus implementieren mit einer Laufzeit

$$\mathcal{O}(m + n \log n).$$

$\rightarrow$ Lineare Laufzeit für Graphen mit $m = \Omega(n \log n)$ Kanten.

Wege: Wollen Wege und nicht nur Distanzen berechnen!

Merke Vorgänger $\text{pred}(v)$ zu jedem Knoten $v \in S$.
$+$ Backtracking

Dijkstra's Algorithmus (Wege)

Dijkstra's Algorithmus (Distanzen und Wege)

Input : gewichteter Digraph $G = (V, E, c)$, Startknoten $s \in V$
Output: $d(s, v)$ und entsprechender Weg für alle $v \in V$

1 **Initialisiere** $S \leftarrow \emptyset$, $dist(s) \leftarrow 0$ und $\forall v \in V \setminus \{s\} : dist(v) \leftarrow \infty$
2 **while** $S \neq V$ **do**
3 Finde $u \in V \setminus S$ mit minimaler vorläufiger Distanz $dist(u)$
4 $S \leftarrow S \cup \{u\}$
5 **for** $v \in V \setminus S$ mit $(u, v) \in E$ **do**
6 **if** $dist(u) + c(u, v) < dist(v)$ **then**
7 $dist(v) \leftarrow dist(u) + c(u, v)$ und $pred(v) \leftarrow u$
8 **return** $dist(v)$ und $pred(v)$ für alle $v \in V$

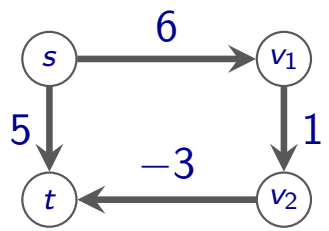
Bemerkungen II

- **Dijkstra ist ein Greedy Algorithmus**
  In jeder Runde wird als nächster einzubeziehender Knoten einer gewählt, dessen dist(v)-Wert minimal ist.

  $\longrightarrow$ Sehr ähnlich zu Prim's Algorithmus für das Finden von minimalen aufspannenden Bäumen (min c_e statt min dist(v)).

- **Sehr große Graphen – z.B. Routenplaner**:
  - Verallgemeinerung von Dijkstra's Algorithmus zu A*-Algorithmus (mit Heuristik für Suchrichtung).
  - Hierarchische Ansätze (suche kürzeste Wege zur Autobahn, dann suche nur auf Autobahnnetz weiter)

- **Negative Kantengewichte?** Dijkstra arbeitet nicht korrekt!



Kürzeste Wege

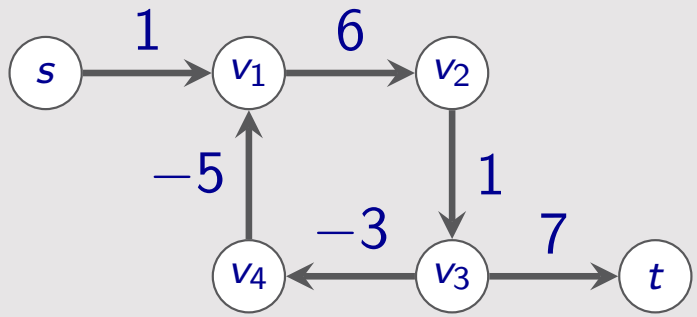
mit beliebigen Kantengewichten:

Bellmann-Ford Algorithmus

Negative Kreise

Definitionen

Ein negativer Kreis in einem gewichteten Graphen $G = (V, E, c)$ ist ein Kreis C mit negativer Kantenkostensumme, d.h. $\sum_{e \in C} c_e < 0$.



Lemma

In einem Graphen $G = (V, E, c)$ gilt für $u, v \in V$: $d(u, v) = -\infty$ genau dann, wenn ein Weg W von u nach v existiert mit einem Knoten $w \in W$ wobei w auf einem negativen Kreis liegt.

Annahme für das Kürzeste-Wege Problem: Graph G enthält keine negativen Kreise.

Teilwege kürzester Wege

Lemma

Sei $P = (v_1, \ldots, v_k)$ ein kürzester (v_1, v_k)-Weg. Dann ist für alle i, j mit $1 \leq i \leq j \leq k$: $P' = (v_i, \ldots, v_j)$ ein kürzester (v_i, v_j)-Weg.

Korollar

Sei $s \in V$ ein ausgezeichneter Knoten, dann gilt für alle $(u, v) \in E$:

$$d(s, v) \leq d(s, u) + c(u, v).$$

Satz

Falls G keine negativen Kreise enthält und ein (s, t)-Weg existiert, dann existiert auch ein kürzester (s, t)-Weg mit höchstens $n - 1$ Kanten.

Beweis. Bei mehr Kanten gäbe es einen Kreis, dessen Entfernung die Kosten nicht erhöht.

Algorithmus von Bellman und Ford

(auch Moore-Bellman-Ford Algorithmus)

Bellman-Ford Algorithmus

Input : gewichteter Digraph $G = (V, E, c)$, Startknoten $s \in V$
Output : $d(s, v)$ und kürzesten Weg für alle $v \in V$, oder Information dass G einen negativen Kreis enthält

1 **Initialisiere** $dist(s) \leftarrow 0$, $\forall v \in V \setminus \{s\} : dist(v) \leftarrow \infty$
2 **for** $k := 1, \ldots, n-1$ **do** — $n-1$ Runden
3 **for** $(u, v) \in E$ **do**
4 **if** $dist(u) + c(u, v) < dist(v)$ **then**
5 $dist(v) \leftarrow dist(u) + c(u, v)$ und $pred(v) \leftarrow u$
6 **for** $(u, v) \in E$ **do** — Negative-Kreise-Test
7 **if** $dist(u) + c(u, v) < dist(v)$ **then**
8 Ausgabe, dass negativer Kreis enthalten
9 **return** $dist(v)$ und $pred(v)$ für alle $v \in V$

Analyse Bellman-Ford

Lemma

Am Ende jeder Runde $k \in \{1, \ldots, n-1\}$ gilt für alle $v \in V$

$$dist(v) \leq \min\{c(P) \mid P \text{ ist } (s,v)\text{-Weg mit höchstens } k \text{ Kanten}\}.$$

Beweis. Betrachte ein beliebiges $v \in V$. Wenn kein (s,v)-Weg existiert, dann ist nichts zu zeigen.

Angenommen v ist von s erreichbar. Induktion über k.

- IA $k = 1$: klar.

  $\min \ldots < \infty$ nur für $P = (s,v)$; es gilt $dist(v) = c(s,v)$.
- IS: Betrachte (s,v)-Weg $P = (s = v_0, v_1, \ldots, v_{k-1}, v_k = v)$ mit genau k Kanten. Für den Teilweg $P' = (s, \ldots, v_{k-1})$ gilt nach IV $dist(v_{k-1}) \leq c(P')$. In Runde k wird auch Kante (v_{k-1}, v_k) betrachtet und es gilt

$$dist(v_k) \leq dist(v_{k-1}) + c(v_{k-1}, v_k) \leq c(P') + c(v_{k-1}, v_k) = c(P).$$

□

Analyse Bellman-Ford

Satz

Wenn G keine von s erreichbaren negativen Kreise enthält, so berechnet der Algorithmus $dist(v) = d(s, v)$ für jeden Knoten v. Er erkennt korrekt, ob G einen von s erreichbaren negativen Kreis enthält und hat Laufzeit $\mathcal{O}(nm)$.

Beweisskizze.

- Keine von s erreichbaren negativen Kreise: Betrachte $v \in V$ und einen kürzesten (s, v)-Weg P (falls ein solcher existiert, sonst gilt trivialerweise $dist(v) = d(s, v) = \infty$). Dieser hat $k \leq n - 1$ Kanten. Mit vorigem Lemma gilt nach Phase k, dass $dist(v) \leq c(P) = d(s, v)$. Labels werden nie erhöht und nur gesenkt, wenn ein Weg existiert.

Analyse Bellman-Ford

- Negative Kreise:
  (1) Wenn kein von s erreichbarer negativer Kreis ex., dann gilt $dist(v) = d(s,v) \Rightarrow dist(v) \leq dist(u) + c(u,v)$, $\forall (u,v) \in E$ und der Alg. meldet korrekt, dass kein von s erreichbarer neg. Kreis ex.

  (2) Wenn von s erreichbarer neg. Kreis $C = v_1, \ldots, v_{k+1}$ (mit $v_1 = v_{k+1}$) mit $\sum_{i=1}^{k} c(v_i, v_{i+1}) < 0$ ex., dann gibt der Alg. dies zurück: Angenommen nicht. Der Alg. hat geprüft: $dist(u) + c(u,v) \geq dist(v)$ für alle $(u,v) \in E$. Da $v_1 = v_{k+1}$, gilt $\sum_{i=1}^{k} dist(v_i) = \sum_{i=1}^{k} dist(v_{i+1})$. Also

$$\sum_{i=1}^{k} dist(v_{i+1}) > \sum_{i=1}^{k} dist(v_{i+1}) + \sum_{i=1}^{k} c(v_i, v_{i+1})$$

$$= \sum_{i=1}^{k} \big(dist(v_i) + c(v_i, v_{i+1})\big) \geq \sum_{i=1}^{k} dist(v_{i+1}),$$

  ein Widerspruch.
- Laufzeit: $n-1$ Runden, pro Runde m Kantenupdates □

Zusammenfassung

Kürzeste Wege

- Keine negativen Kantenkosten: Dijkstra Algorithmus
- Keine negativen Kreise: Bellman-Ford Algorithmus