Datenstrukturen und Algorithmen

Vorlesung 17: Kürzeste Pfade (K24)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

http://moves.rwth-aachen.de/teaching/ss-15/dsal/

18. Juni 2015

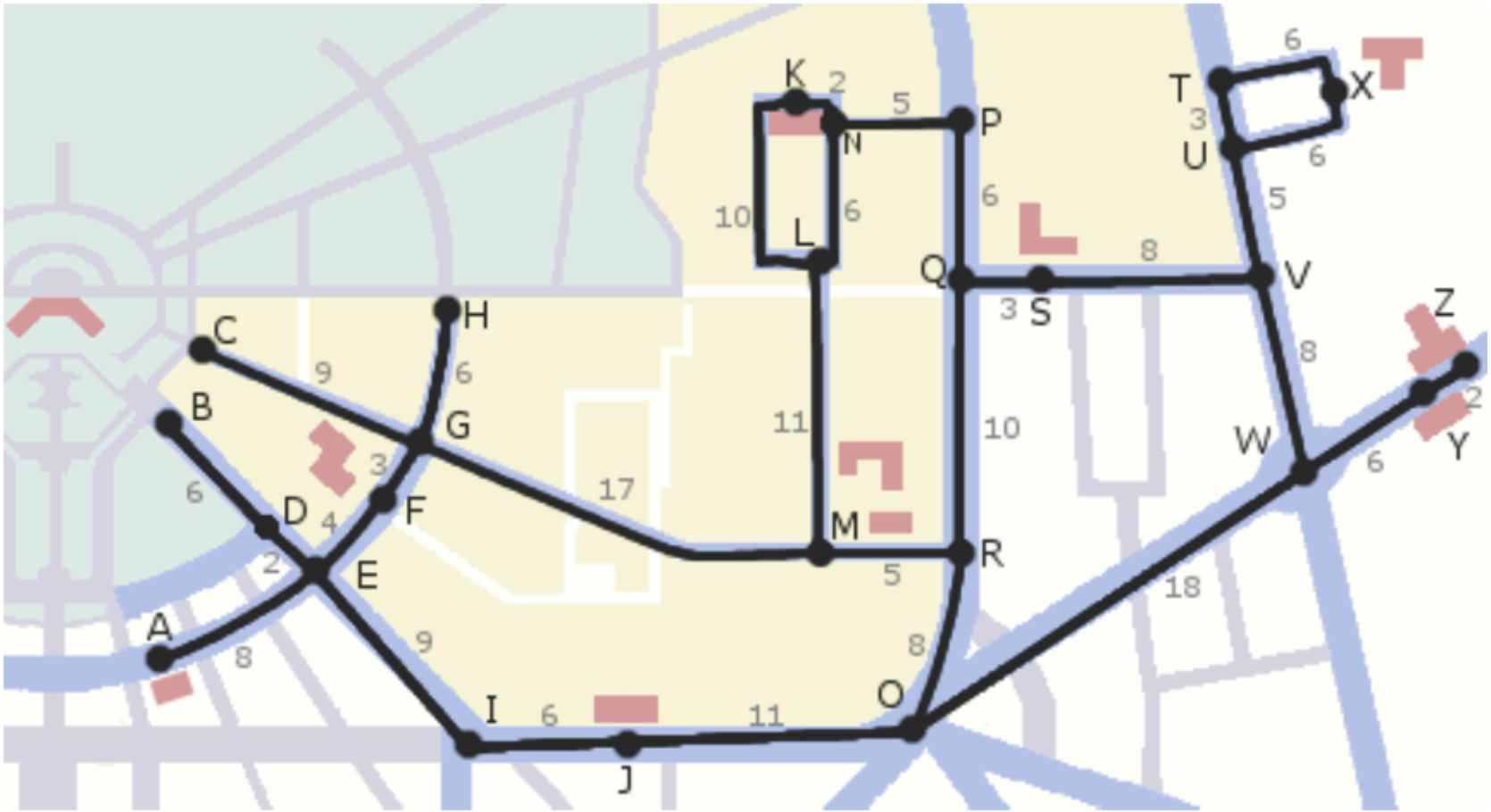
RWTH AACHEN UNIVERSITY

Übersicht

Übersicht

Das Rechenproblem: kürzeste Pfade

Das Rechenproblem: kürzeste Pfade



Andere Rechenprobleme: kürzester Weg

Beispiel (kürzester Weg)

Eingabe:
1. Eine Straßenkarte, auf der der Abstand zwischen jedem Paar benachbarter Kreuzungen eingezeichnet ist,
2. eine Startkreuzung s, und
3. eine Zielkreuzung t.

Ausgabe: Der kürzeste Weg von s nach t.

Kürzeste Pfade

Gegeben sei ein (kanten-)gewichteter Graph $G = (V, E, W)$.

Kürzeste Pfade

Gegeben sei ein (kanten-)gewichteter Graph $G = (V, E, W)$.

Das Gewicht eines Pfades ist die Summe der Gewichte seiner Kanten.

Kürzeste Pfade

Gegeben sei ein (kanten-)gewichteter Graph $G = (V, E, W)$.

Das Gewicht eines Pfades ist die Summe der Gewichte seiner Kanten.

Ein kürzester Pfad von einem Knoten $s \in V$ zu einem anderen Knoten $v \in V$ ist ein Pfad von s nach v mit minimalem Gewicht.

Kürzeste Pfade

Gegeben sei ein (kanten-)gewichteter Graph $G = (V, E, W)$.

Das Gewicht eines Pfades ist die Summe der Gewichte seiner Kanten.

Ein kürzester Pfad von einem Knoten $s \in V$ zu einem anderen Knoten $v \in V$ ist ein Pfad von s nach v mit minimalem Gewicht.

Sei im Folgenden $\delta : (V \times V) \to (\mathbb{R} \cup \{+\texttt{inf}\})$ eine Funktion, sodaß:

- $\delta(s, v)$ ist das Gewicht des kürzesten Pfades von s nach v, und
- $\delta(s, v) = +\texttt{inf}$ gdw. v von s nicht erreichbar ist.

Kürzeste Pfade

Es gibt verschiedene Varianten:

- Kürzeste Pfade von einem Startknoten s zu allen anderen Knoten: Single-Source Shortest Paths (SSSP).

Kürzeste Pfade

Es gibt verschiedene Varianten:

- Kürzeste Pfade von einem Startknoten s zu allen anderen Knoten: Single-Source Shortest Paths (SSSP).
- Kürzeste Pfade von allen Knoten zu einem Zielknoten t.

Kürzeste Pfade

Es gibt verschiedene Varianten:

- Kürzeste Pfade von einem Startknoten s zu allen anderen Knoten: Single-Source Shortest Paths (SSSP).
- Kürzeste Pfade von allen Knoten zu einem Zielknoten t.
  Lässt sich auf SSSP zurückführen.

Kürzeste Pfade

Es gibt verschiedene Varianten:

- Kürzeste Pfade von einem Startknoten s zu allen anderen Knoten: Single-Source Shortest Paths (SSSP).
- Kürzeste Pfade von allen Knoten zu einem Zielknoten t.
  Lässt sich auf SSSP zurückführen.
- Kürzeste Pfade für *ein* festes Knotenpaar u, v.

Kürzeste Pfade

Es gibt verschiedene Varianten:

- Kürzeste Pfade von einem Startknoten s zu allen anderen Knoten: Single-Source Shortest Paths (SSSP).
- Kürzeste Pfade von allen Knoten zu einem Zielknoten t.
  Lässt sich auf SSSP zurückführen.
- Kürzeste Pfade für *ein* festes Knotenpaar u, v.
  Es ist kein Algorithmus bekannt, der asymptotisch schneller als der beste SSSP-Algorithmus ist.

Kürzeste Pfade

Es gibt verschiedene Varianten:

- Kürzeste Pfade von einem Startknoten s zu allen anderen Knoten: Single-Source Shortest Paths (SSSP).
- Kürzeste Pfade von allen Knoten zu einem Zielknoten t.
  Lässt sich auf SSSP zurückführen.
- Kürzeste Pfade für *ein* festes Knotenpaar u, v.
  Es ist kein Algorithmus bekannt, der asymptotisch schneller als der beste SSSP-Algorithmus ist.
- Kürzeste Pfade für *alle* Knotenpaare.

Kürzeste Pfade

Es gibt verschiedene Varianten:

- Kürzeste Pfade von einem Startknoten s zu allen anderen Knoten: Single-Source Shortest Paths (SSSP).
- Kürzeste Pfade von allen Knoten zu einem Zielknoten t.
  Lässt sich auf SSSP zurückführen.
- Kürzeste Pfade für *ein* festes Knotenpaar u, v.
  Es ist kein Algorithmus bekannt, der asymptotisch schneller als der beste SSSP-Algorithmus ist.
- Kürzeste Pfade für *alle* Knotenpaare.
  All-Pairs Shortest Paths (zweiter Teil dieser Vorlesung).

Übersicht

Single-Source Shortest Paths

Problem (Single-Source Shortest Path)

Für einen gegebenen Knoten $s \in V$ (die Quelle / source), bestimme für jeden anderen Knoten $t \in V$, der aus s erreichbar ist, einen kürzesten Pfad von s zu t.

Übersicht

Der Bellman-Ford Algorithmus

- Kürzeste Pfade bei einem einzigen Startknoten.

Der Bellman-Ford Algorithmus

- Kürzeste Pfade bei einem einzigen Startknoten.
- Erlaubt negative Kantengewichte.

Der Bellman-Ford Algorithmus

- Kürzeste Pfade bei einem einzigen Startknoten.
- Erlaubt negative Kantengewichte.
- Er zeigt an, ob es einen Zyklus mit negativem Gewicht gibt, der vom Startknoten aus erreichbar ist.

Der Bellman-Ford Algorithmus

- Kürzeste Pfade bei einem einzigen Startknoten.
- Erlaubt negative Kantengewichte.
- Er zeigt an, ob es einen Zyklus mit negativem Gewicht gibt, der vom Startknoten aus erreichbar ist.
- Falls ein solcher Zyklus gefunden wird, gibt es keine Lösung
  - (da die Gewichte der kürzesten Pfade nicht mehr wohldefiniert sind).

Der Bellman-Ford Algorithmus

- Kürzeste Pfade bei einem einzigen Startknoten.
- Erlaubt negative Kantengewichte.
- Er zeigt an, ob es einen Zyklus mit negativem Gewicht gibt, der vom Startknoten aus erreichbar ist.
- Falls ein solcher Zyklus gefunden wird, gibt es keine Lösung
  - (da die Gewichte der kürzesten Pfade nicht mehr wohldefiniert sind).
- Sonst verbessert der Algorithmus iterativ für jeden Knoten v eine obere Grenze `dist[v]` an $\delta(s, v)$, bis das Minimum gefunden wird.

Der Bellman-Ford Algorithmus

- Kürzeste Pfade bei einem einzigen Startknoten.
- Erlaubt negative Kantengewichte.
- Er zeigt an, ob es einen Zyklus mit negativem Gewicht gibt, der vom Startknoten aus erreichbar ist.
- Falls ein solcher Zyklus gefunden wird, gibt es keine Lösung
  - (da die Gewichte der kürzesten Pfade nicht mehr wohldefiniert sind).
- Sonst verbessert der Algorithmus iterativ für jeden Knoten v eine obere Grenze `dist[v]` an $\delta(s, v)$, bis das Minimum gefunden wird.
- Kürzeste Pfade können nach Terminierung mittels die im Array `prev` gespeicherten Vorgängerknoten entlang eines kürzesten Pfades konstruiert werden.

Bellman-Ford: Idee

- Initialisierung: `dist[v]=+inf`, `dist[start]=0`.

Bellman-Ford: Idee

- Initialisierung: `dist[v]=+inf`, `dist[start]=0`.
- Für alle Kanten $(v, w) \in E$:
  - Relaxierung: Ist das bisher bekannte Gewicht `dist[w]` größer als `dist[v]+W(v,w)`, dann verbessere `dist[w]` auf diesen Wert.

Bellman-Ford: Idee

- Initialisierung: `dist[v]=+inf`, `dist[start]=0`.
- Für alle Kanten $(v, w) \in E$:
  - Relaxierung: Ist das bisher bekannte Gewicht `dist[w]` größer als `dist[v]+W(v,w)`, dann verbessere `dist[w]` auf diesen Wert.
- Wiederhole den vorigen Schritt bis sich nichts mehr ändert, bzw. breche ab, falls ein negativer Zyklus gefunden wurde.

Bellman-Ford: Idee

- Initialisierung: `dist[v]=+inf`, `dist[start]=0`.
- Für alle Kanten $(v, w) \in E$:
  - Relaxierung: Ist das bisher bekannte Gewicht `dist[w]` größer als `dist[v]+W(v,w)`, dann verbessere `dist[w]` auf diesen Wert.
- Wiederhole den vorigen Schritt bis sich nichts mehr ändert, bzw. breche ab, falls ein negativer Zyklus gefunden wurde.

Korrektheit vom Bellman-Ford Algorithmus

Wenn nach $|V|-1$ Wiederholungen noch Verbesserungen möglich sind, dann gibt es einen negativen Zyklus. Andernfalls `dist[v]` $= \delta(s, v)$ für alle Knoten $v \in V$.

Beweis.

Beweisidee: Ein Pfad ohne Zyklen in (V, E, W) hat eine Länge $\leqslant |V|-1$. □

Bellman-Ford in Pseudo-Code

```
1  bool bellmanFord(List adj[n], int n, int start,
2       int &dist[n], int &prev[n]) {
3    for (int v = 0; v < n; v++) { // für alle Knoten v
4      dist[v] = +inf; // +inf ist initiale Entfernung von start zu v
5      prev[v] = -1; // v hat noch kein Vorgänger
6    }
7    dist[start] = 0; // die Entfernung von start zu start ist 0
8    for (int i = 1; i < n; i++) // n-1 Durchläufe
9      for (int v = 0; v < n; v++) // für alle Knoten v
10       foreach (edge in adj[v]) // für jede aus v ausgehende Kante
11         if (dist[edge.target] > dist[v] + edge.weight) {
12           dist[edge.target] = dist[v] + edge.weight;
13           prev[edge.target] = v;
14         } // Relaxierung
15   for (int v = 0; v < n; v++) // für alle Knoten v
16     foreach (edge in adj[v]) // für jede aus v ausgehende Kante
17       if (dist[edge.target] > dist[v] + edge.weight)
18         return false; // es existiert Zyklus mit negativem Gewicht
19   return true;
20 }
```

Bellman-Ford in Pseudo-Code

```
1  bool bellmanFord(List adj[n], int n, int start,
2       int &dist[n], int &prev[n]) {
3    for (int v = 0; v < n; v++) { // für alle Knoten v
4      dist[v] = +inf; // +inf ist initiale Entfernung von start zu v
5      prev[v] = -1; // v hat noch kein Vorgänger
6    }
7    dist[start] = 0; // die Entfernung von start zu start ist 0
8    for (int i = 1; i < n; i++) // n-1 Durchläufe
9      for (int v = 0; v < n; v++) // für alle Knoten v
10       foreach (edge in adj[v]) // für jede aus v ausgehende Kante
11         if (dist[edge.target] > dist[v] + edge.weight) {
12           dist[edge.target] = dist[v] + edge.weight;
13           prev[edge.target] = v;
14         } // Relaxierung
15   for (int v = 0; v < n; v++) // für alle Knoten v
16     foreach (edge in adj[v]) // für jede aus v ausgehende Kante
17       if (dist[edge.target] > dist[v] + edge.weight)
18         return false; // es existiert Zyklus mit negativem Gewicht
19   return true;
20 }
```

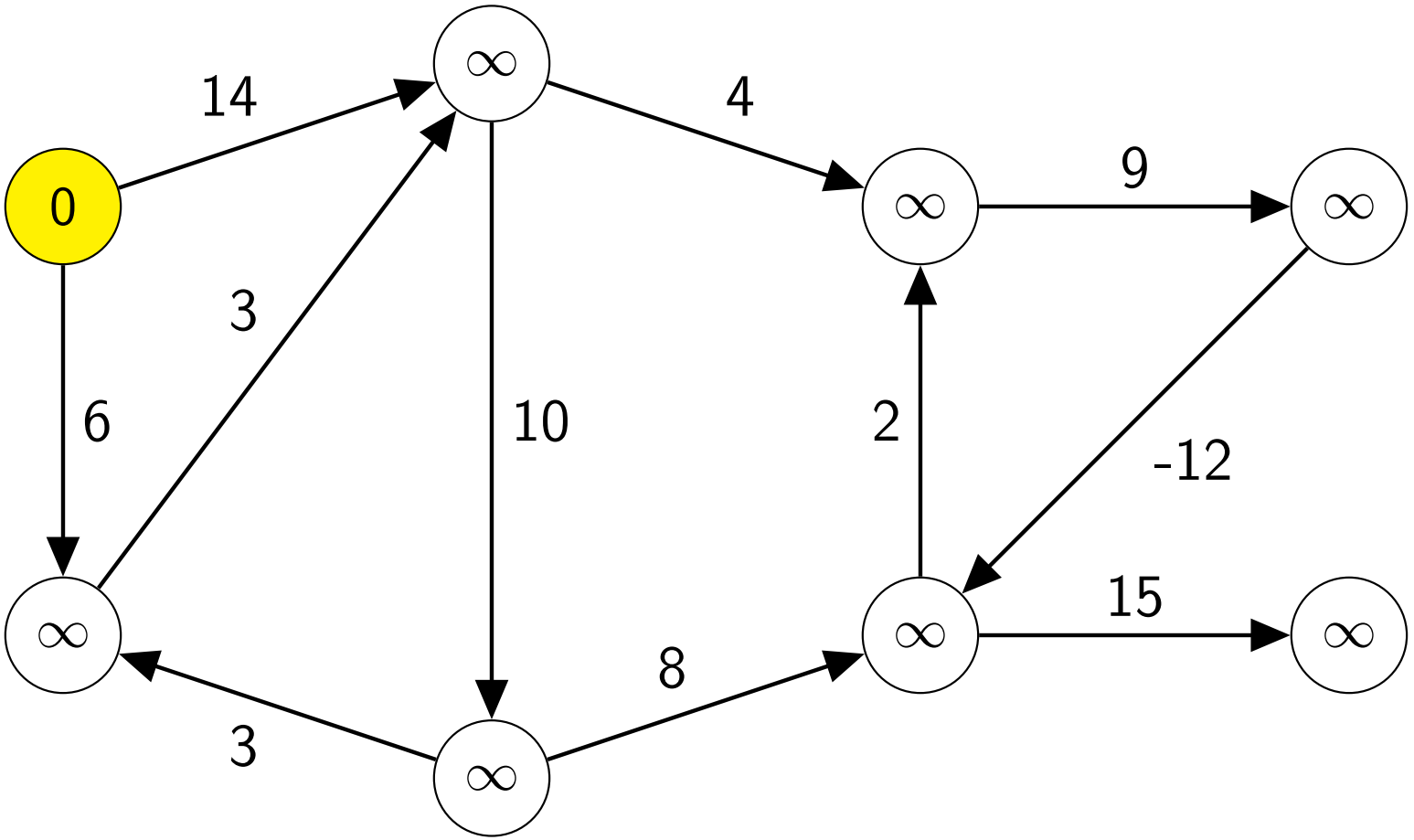
- Zeitkomplexität:

Bellman-Ford in Pseudo-Code

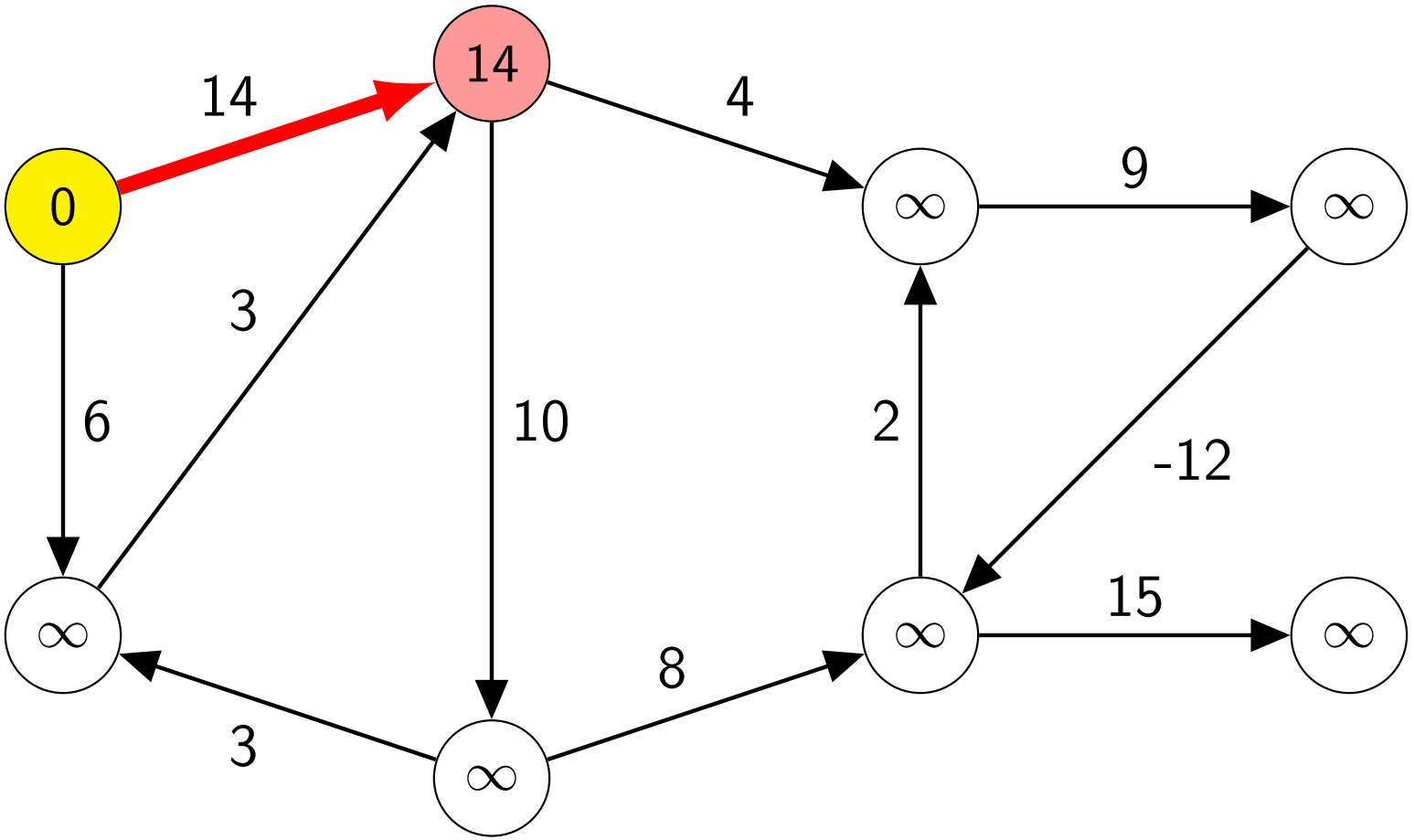
```
1  bool bellmanFord(List adj[n], int n, int start,
2       int &dist[n], int &prev[n]) {
3    for (int v = 0; v < n; v++) { // für alle Knoten v
4      dist[v] = +inf; // +inf ist initiale Entfernung von start zu v
5      prev[v] = -1; // v hat noch kein Vorgänger
6    }
7    dist[start] = 0; // die Entfernung von start zu start ist 0
8    for (int i = 1; i < n; i++) // n-1 Durchläufe
9      for (int v = 0; v < n; v++) // für alle Knoten v
10       foreach (edge in adj[v]) // für jede aus v ausgehende Kante
11         if (dist[edge.target] > dist[v] + edge.weight) {
12           dist[edge.target] = dist[v] + edge.weight;
13           prev[edge.target] = v;
14         } // Relaxierung
15   for (int v = 0; v < n; v++) // für alle Knoten v
16     foreach (edge in adj[v]) // für jede aus v ausgehende Kante
17       if (dist[edge.target] > dist[v] + edge.weight)
18         return false; // es existiert Zyklus mit negativem Gewicht
19   return true;
20 }
```

- Zeitkomplexität: $\mathcal{O}(|V| \cdot |E|)$.

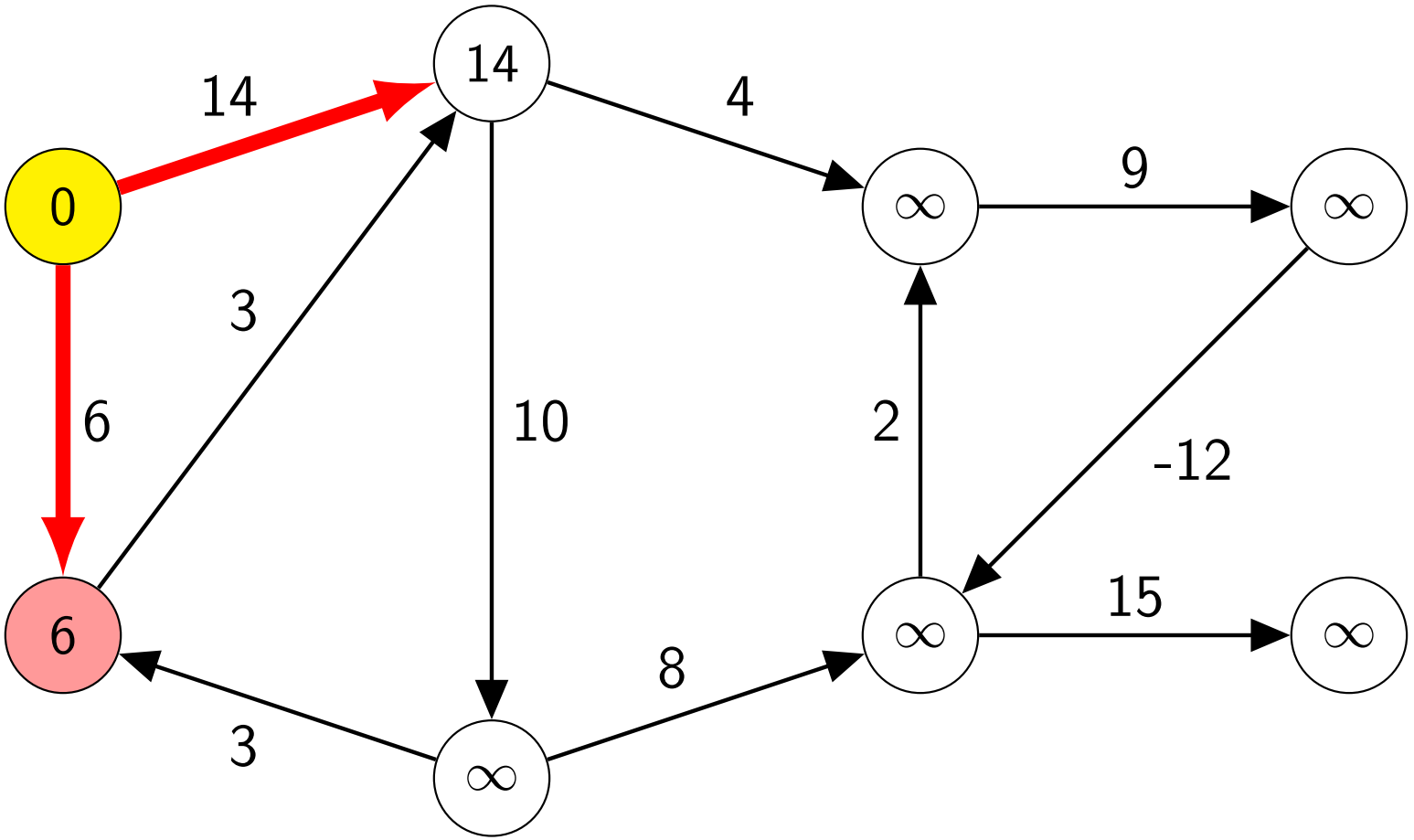
Bellman-Ford – Beispiel



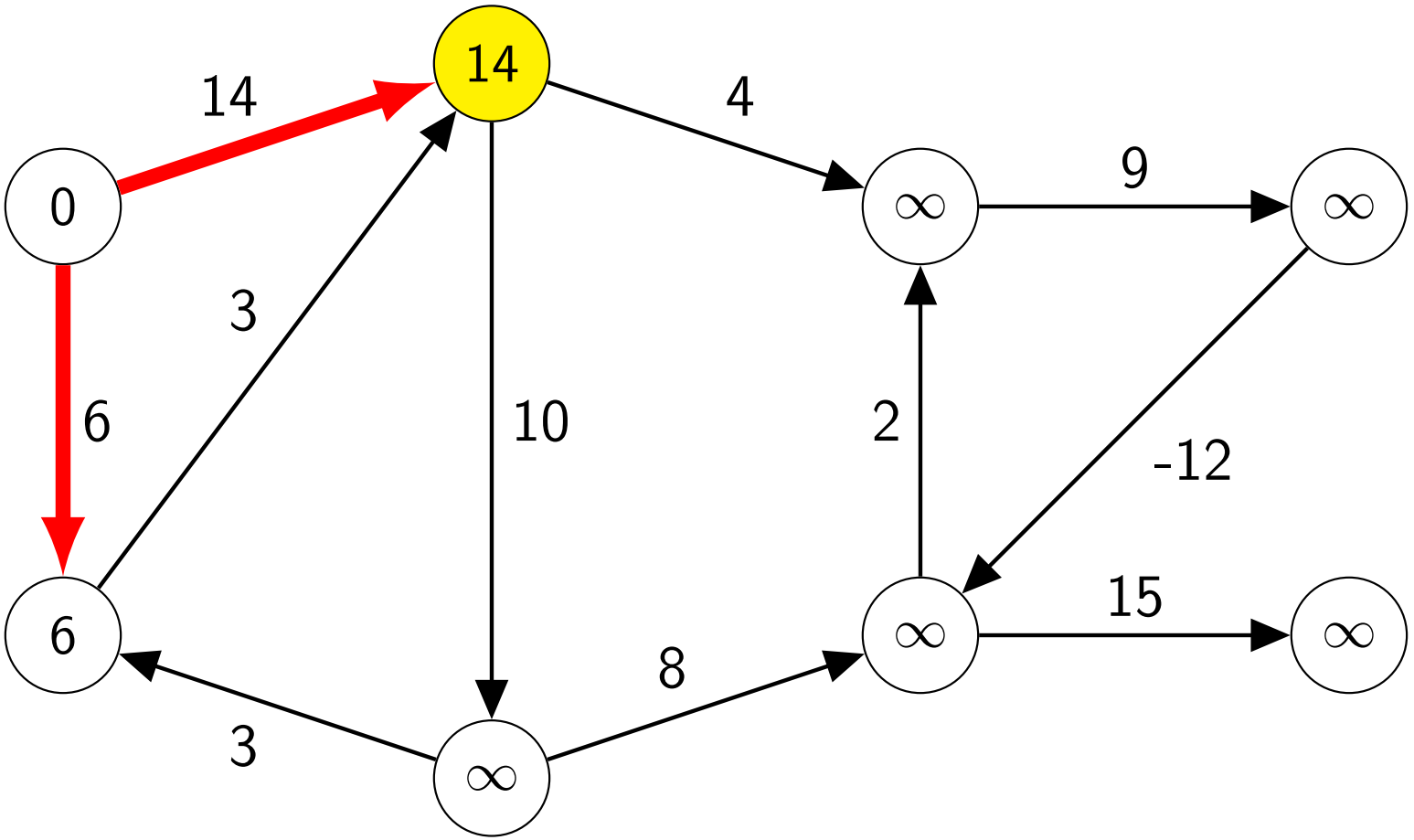
Bellman-Ford – Beispiel



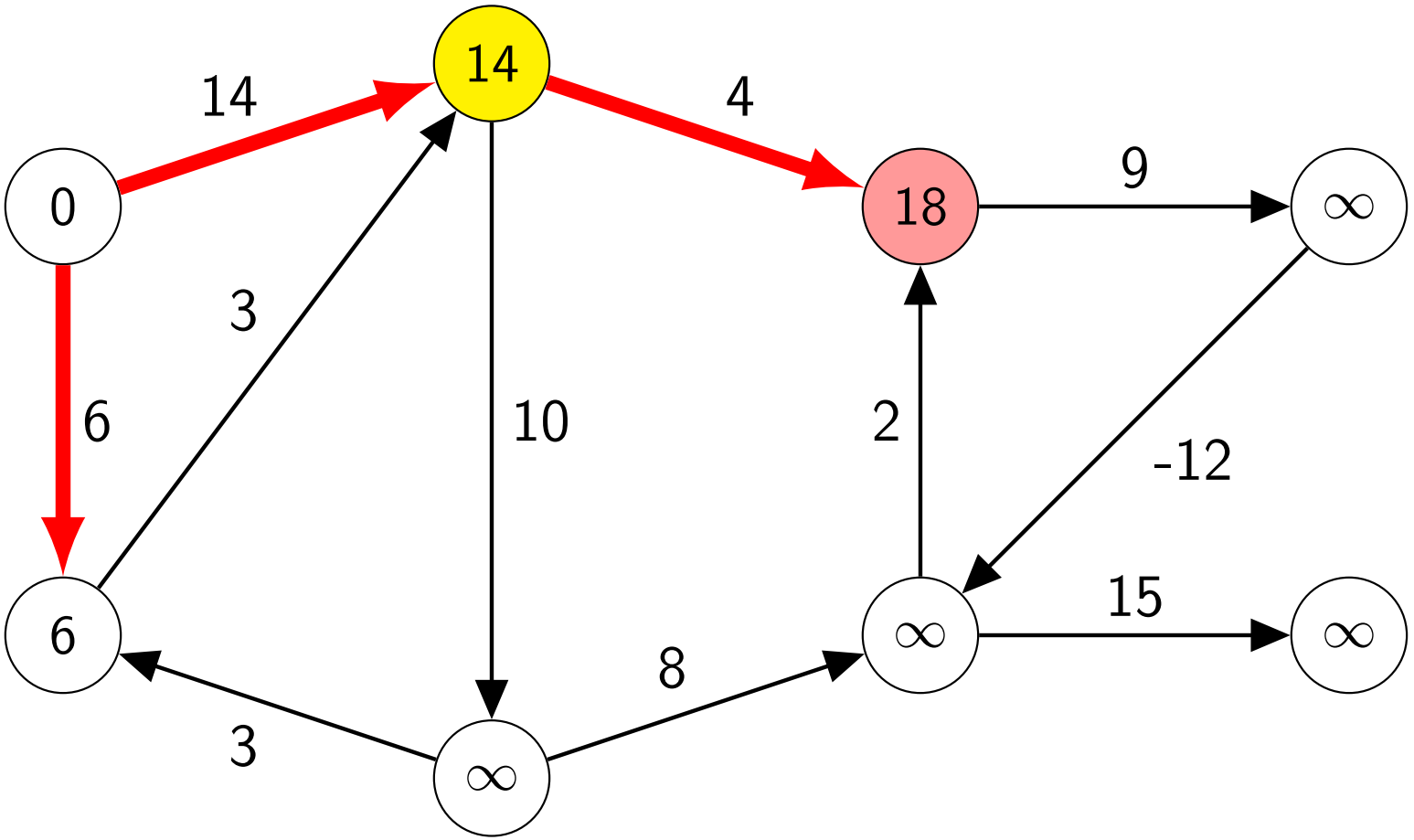
Bellman-Ford – Beispiel



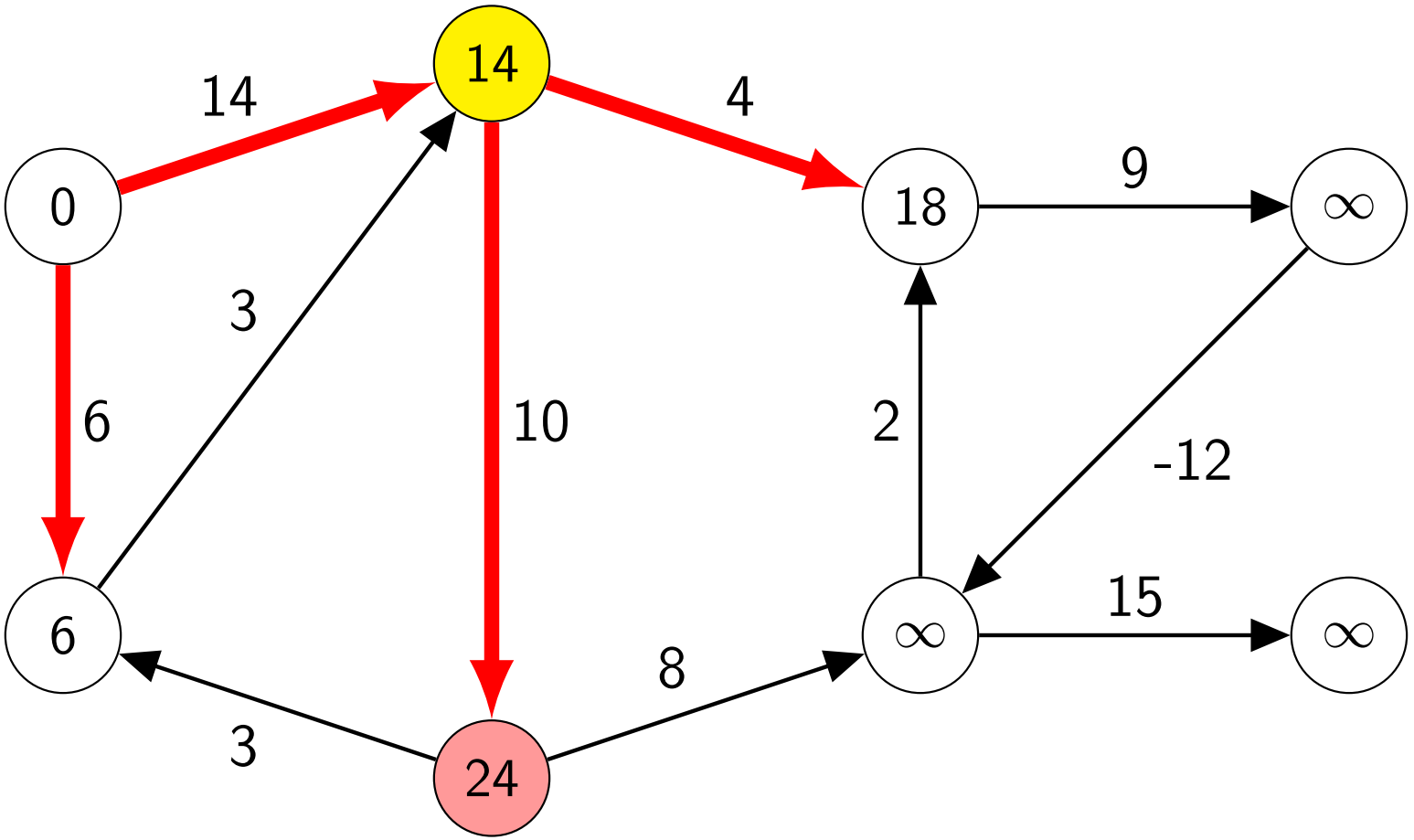
Bellman-Ford – Beispiel



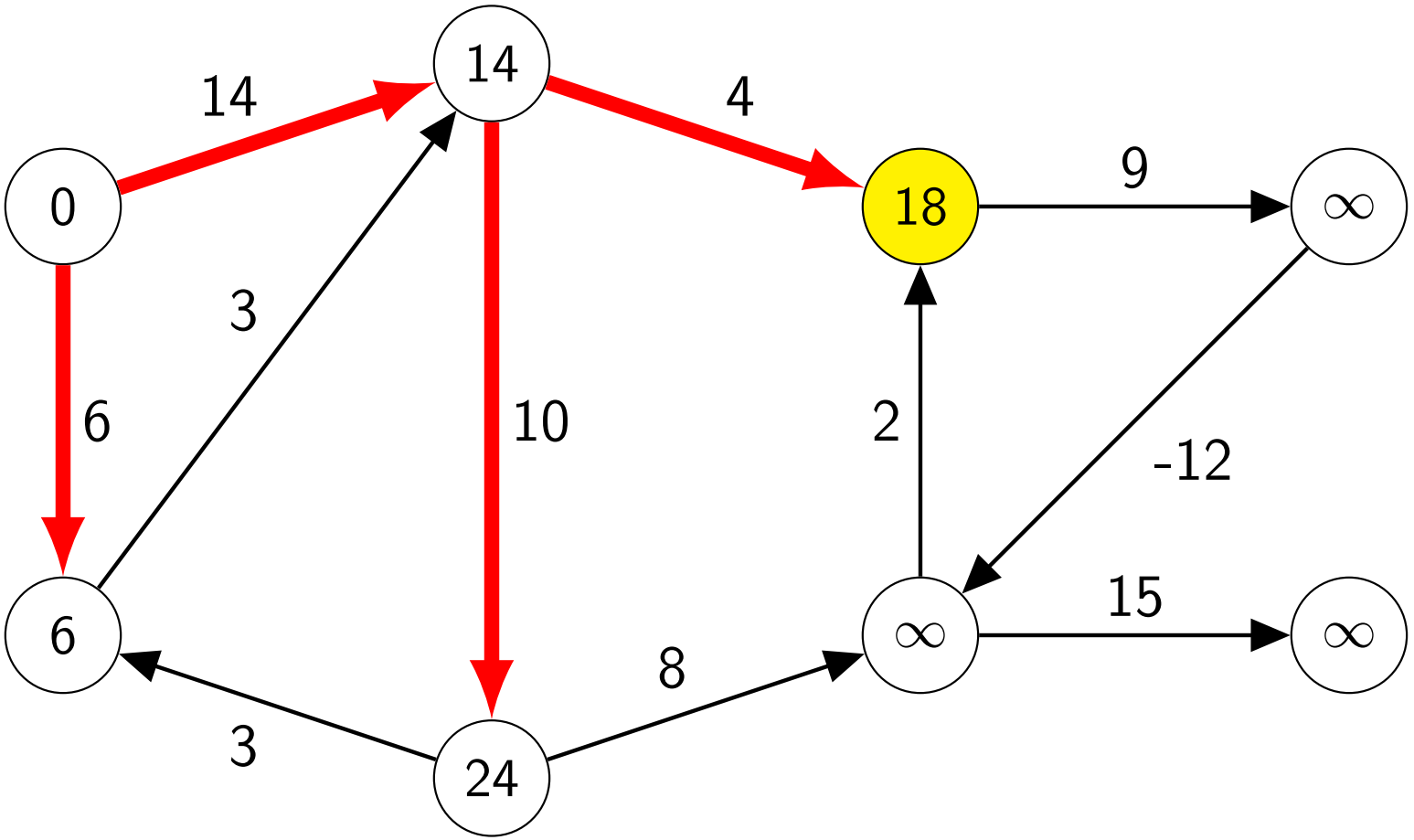
Bellman-Ford – Beispiel



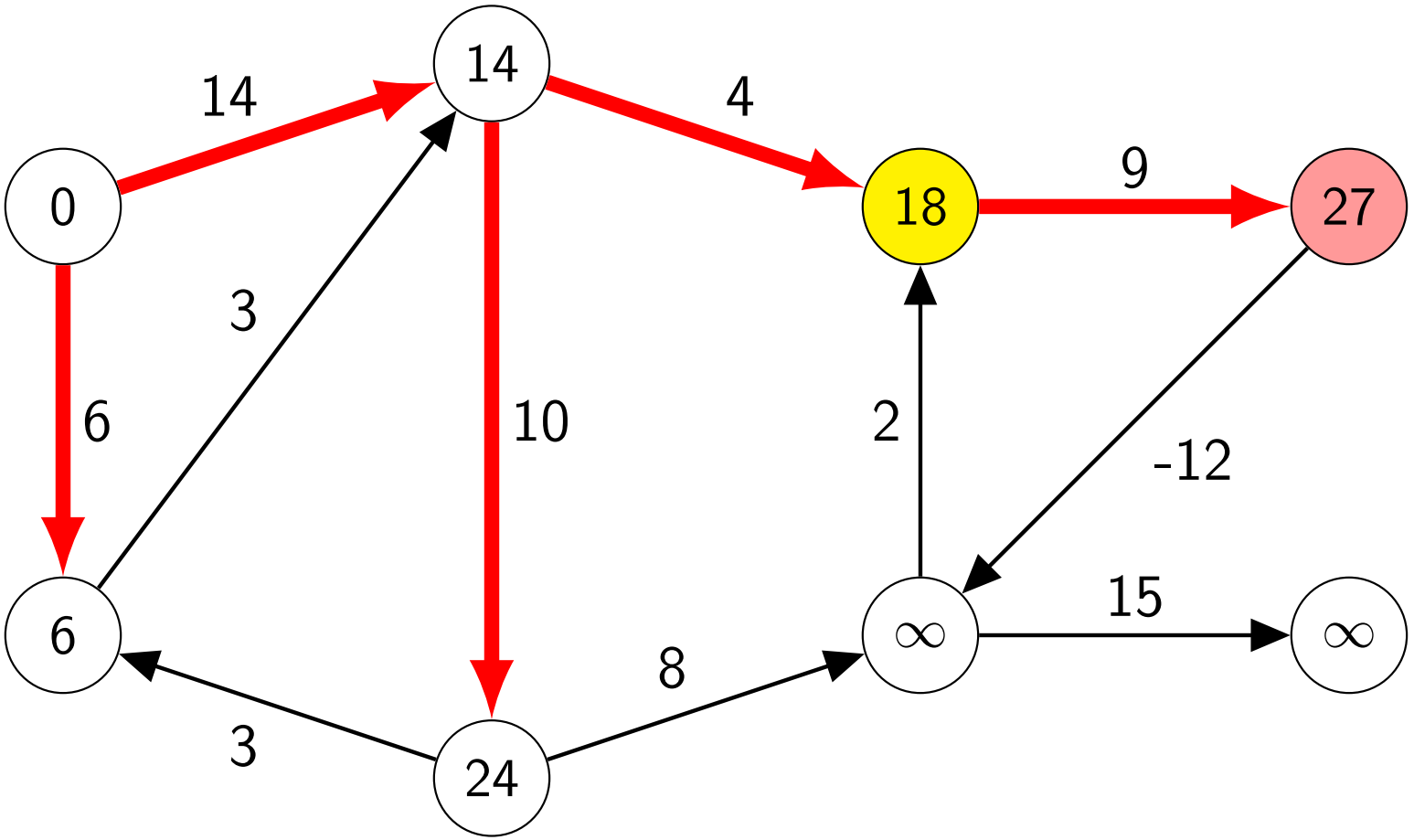
Bellman-Ford – Beispiel



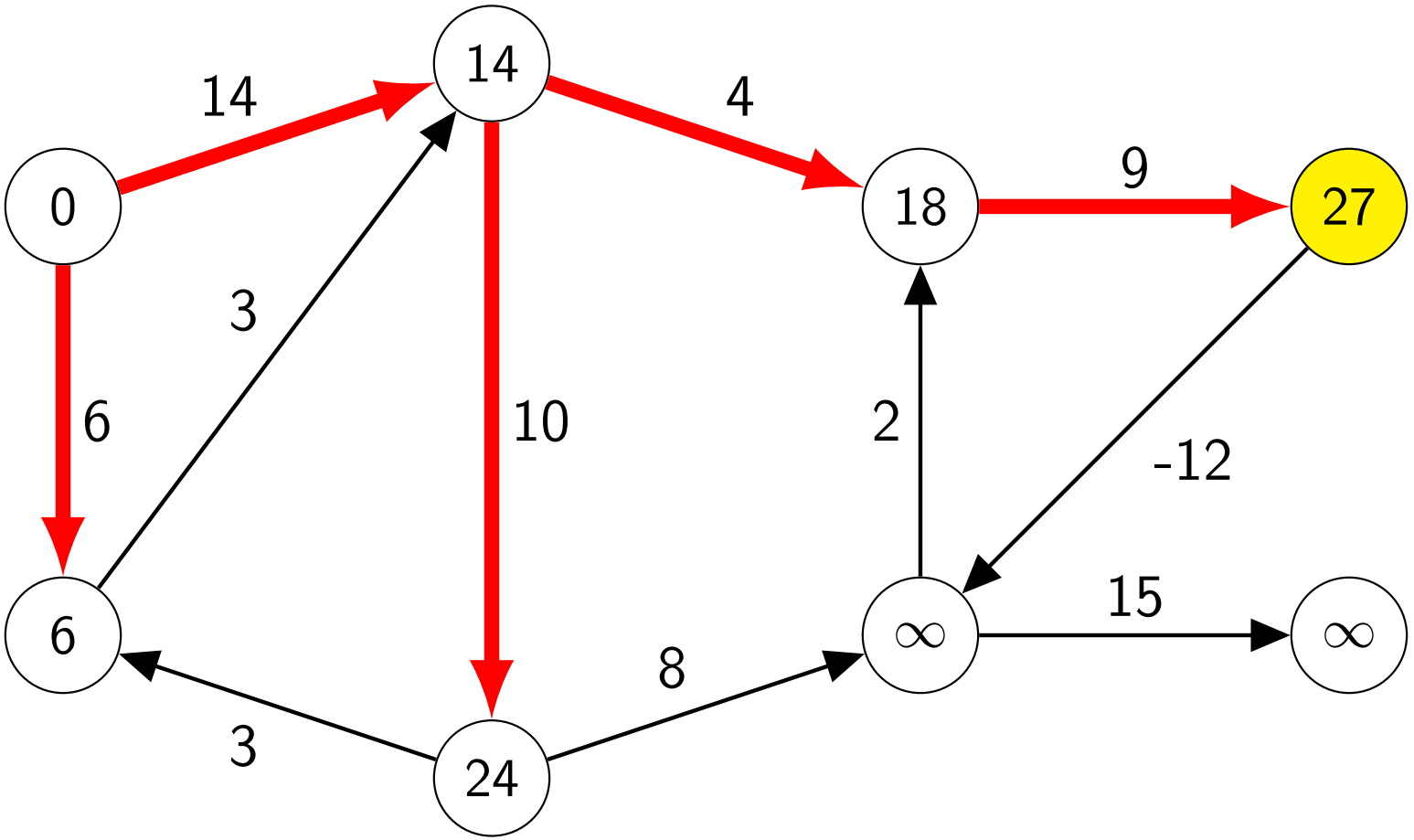
Bellman-Ford – Beispiel



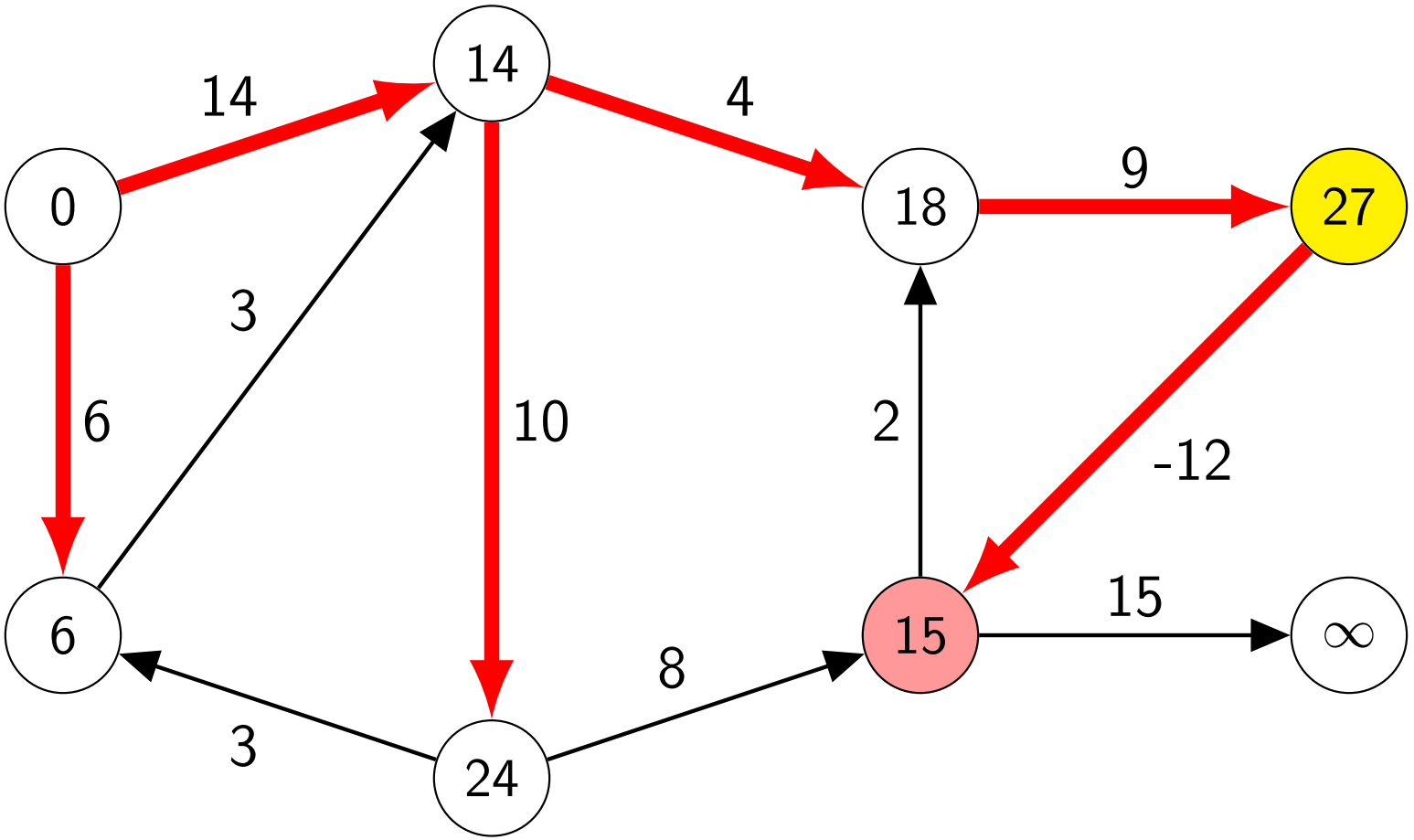
Bellman-Ford – Beispiel



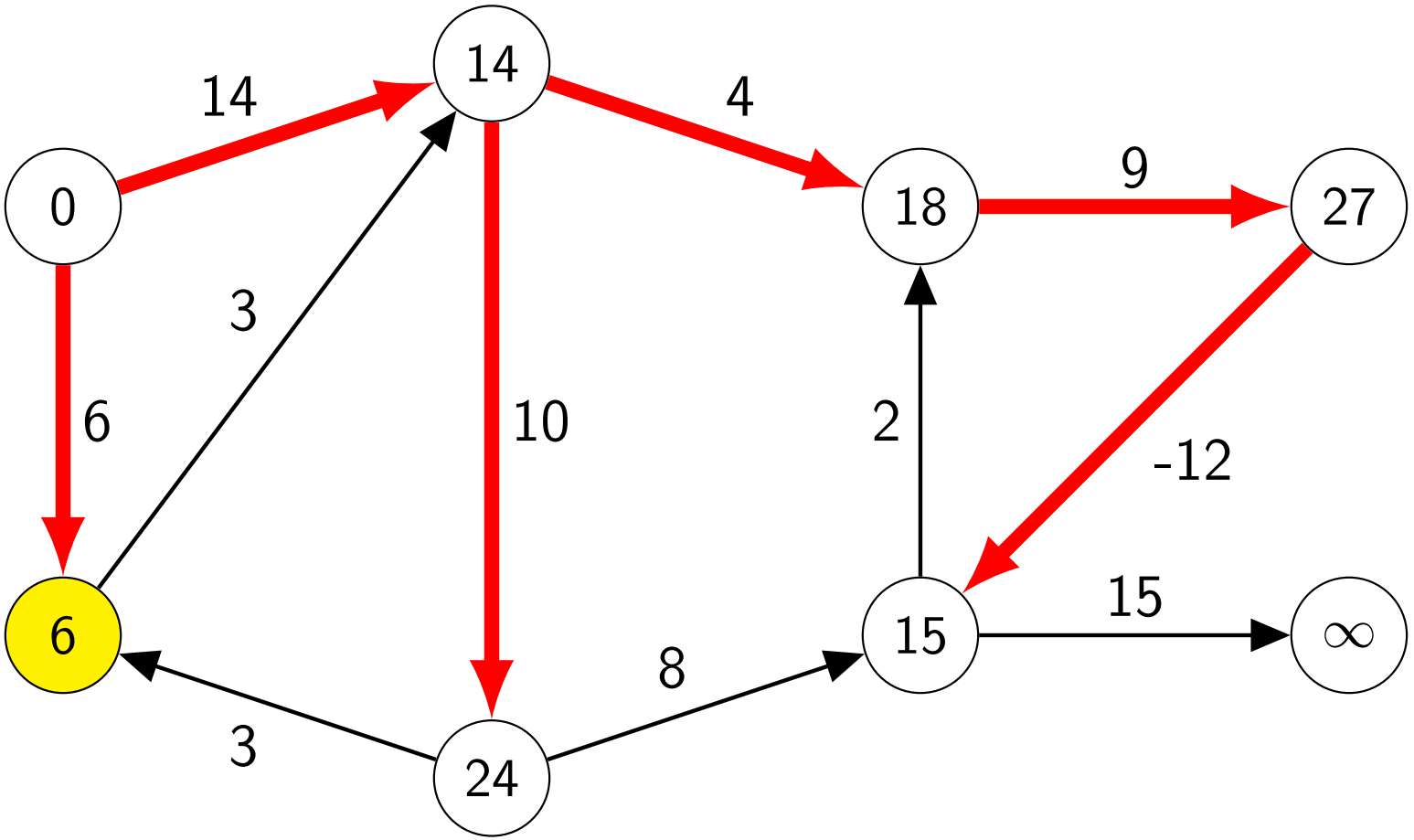
Bellman-Ford – Beispiel



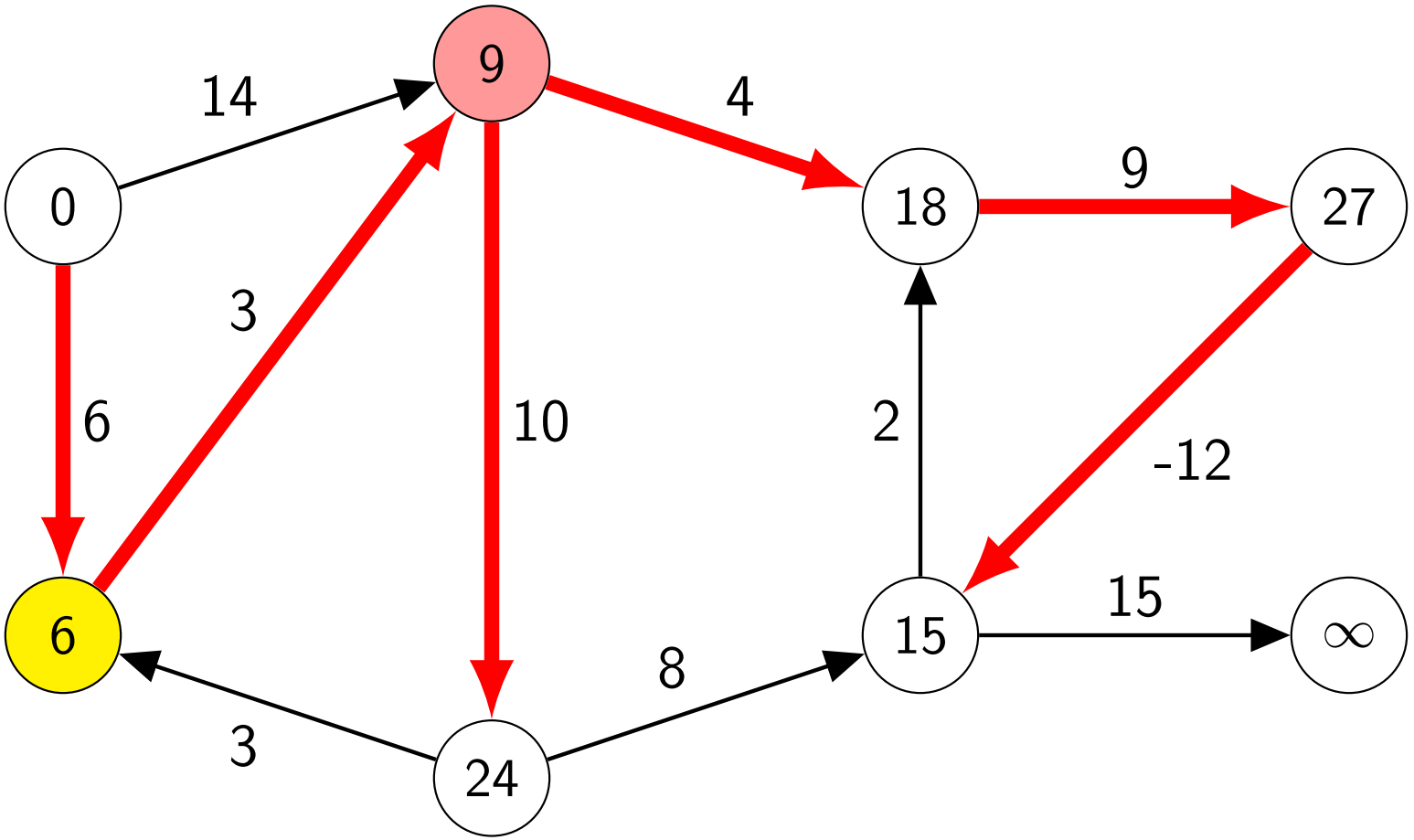
Bellman-Ford – Beispiel



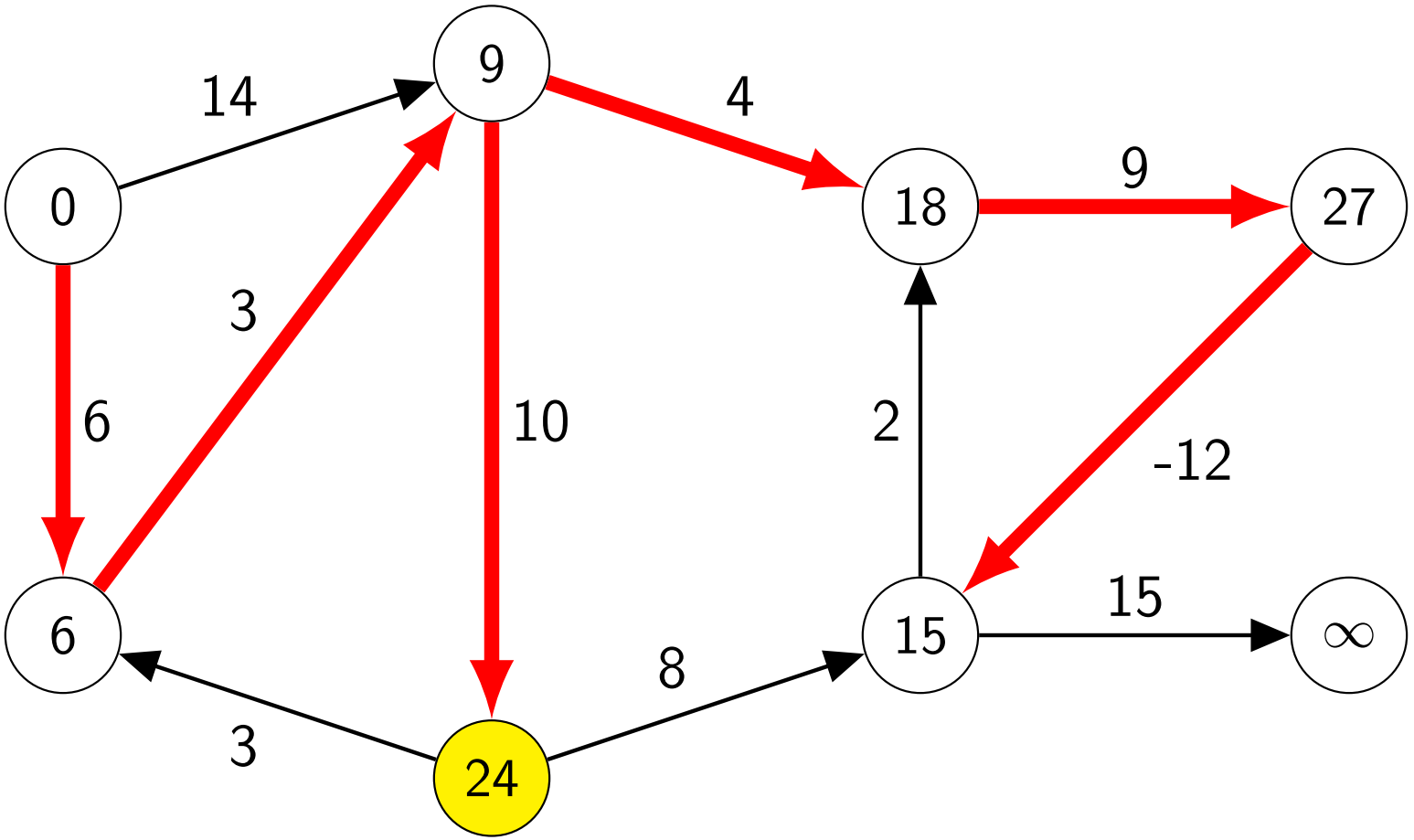
Bellman-Ford – Beispiel



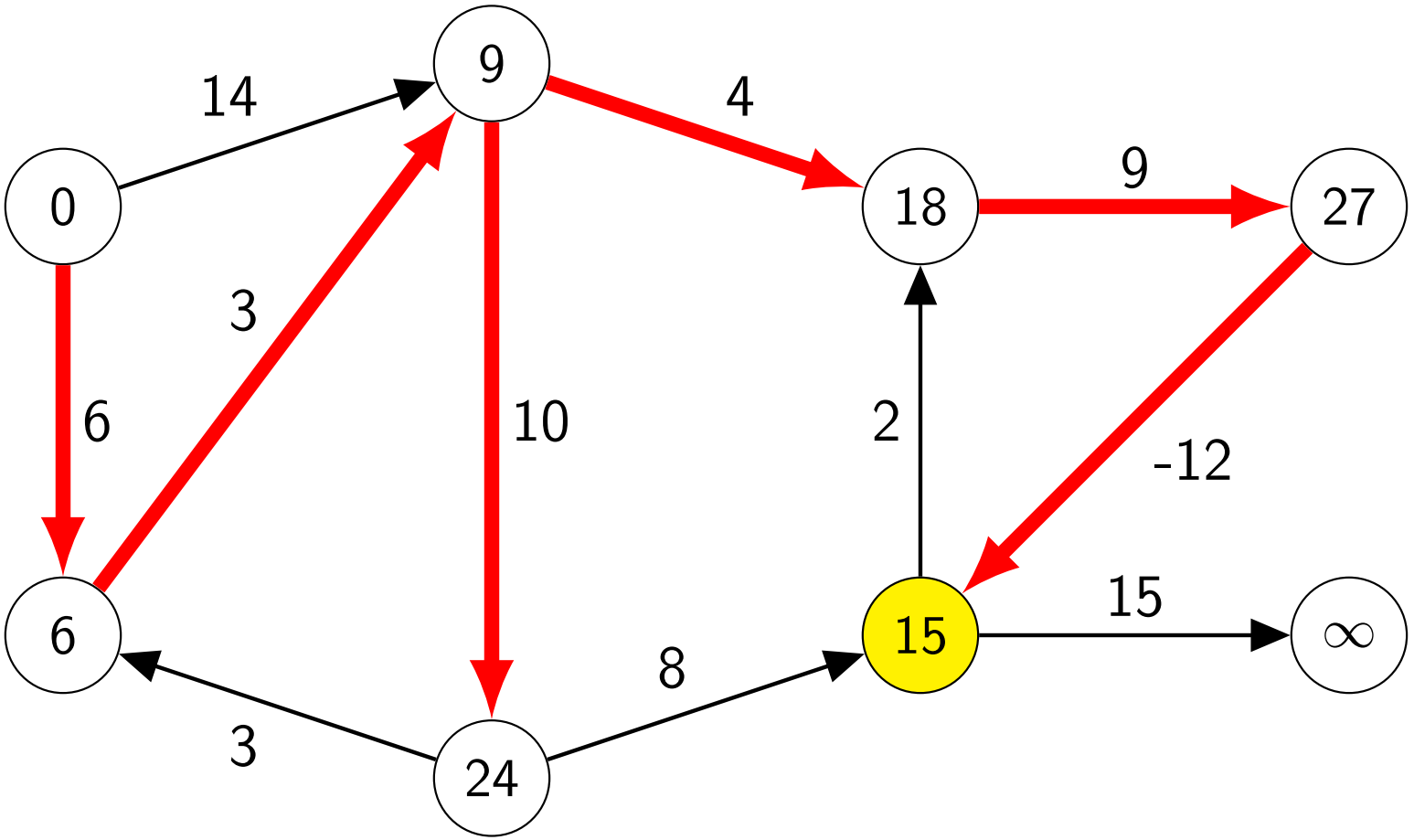
Bellman-Ford – Beispiel



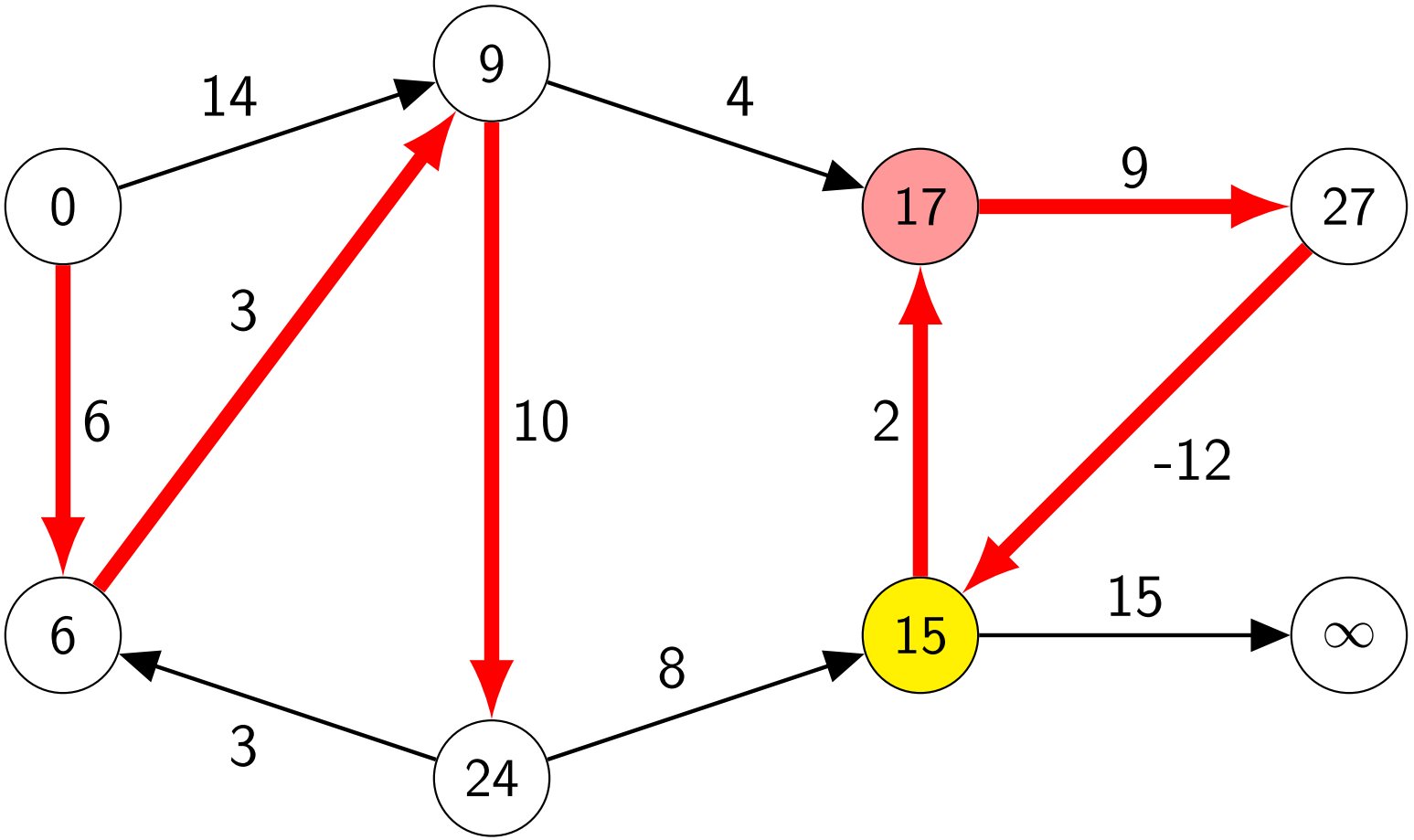
Bellman-Ford – Beispiel



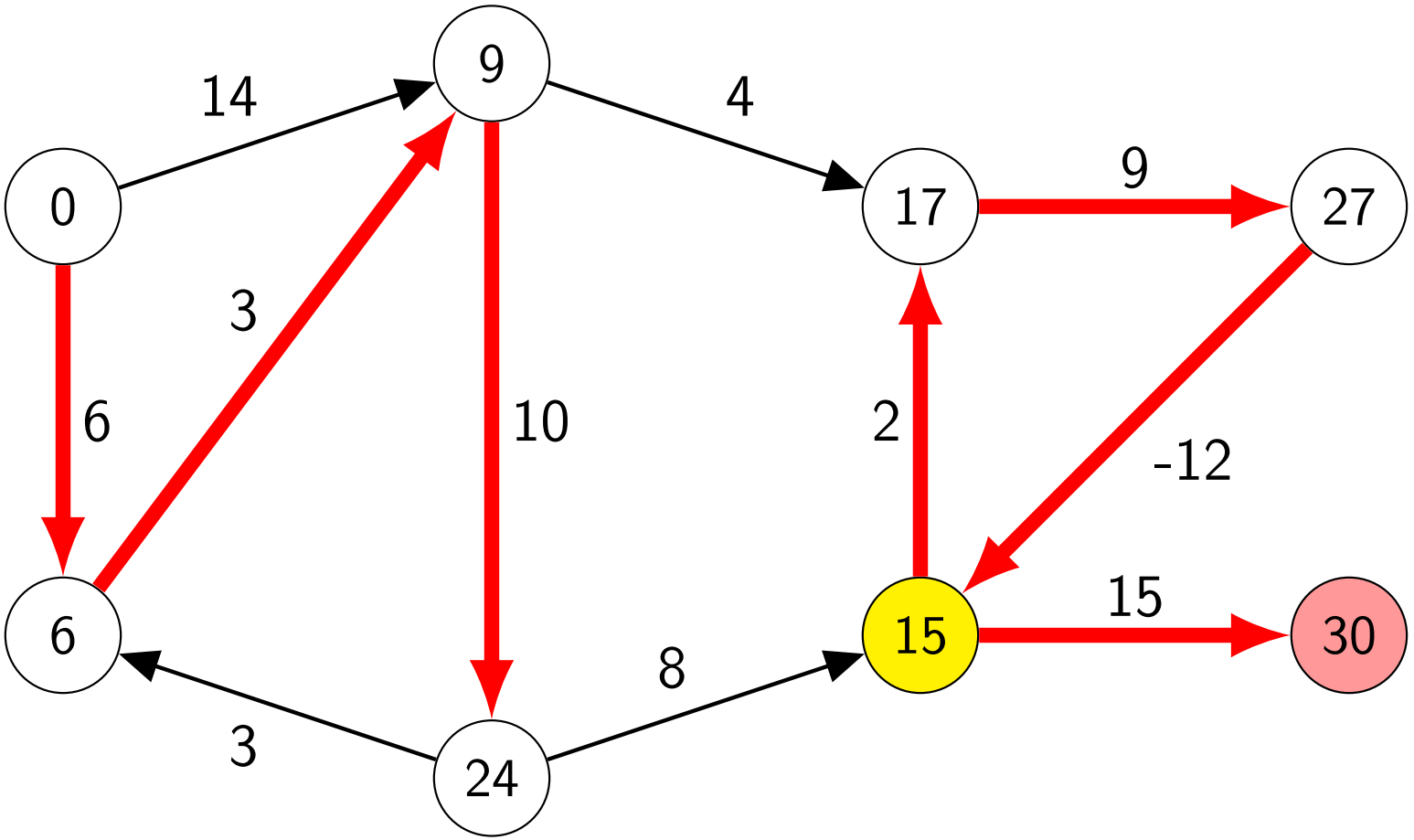
Bellman-Ford – Beispiel



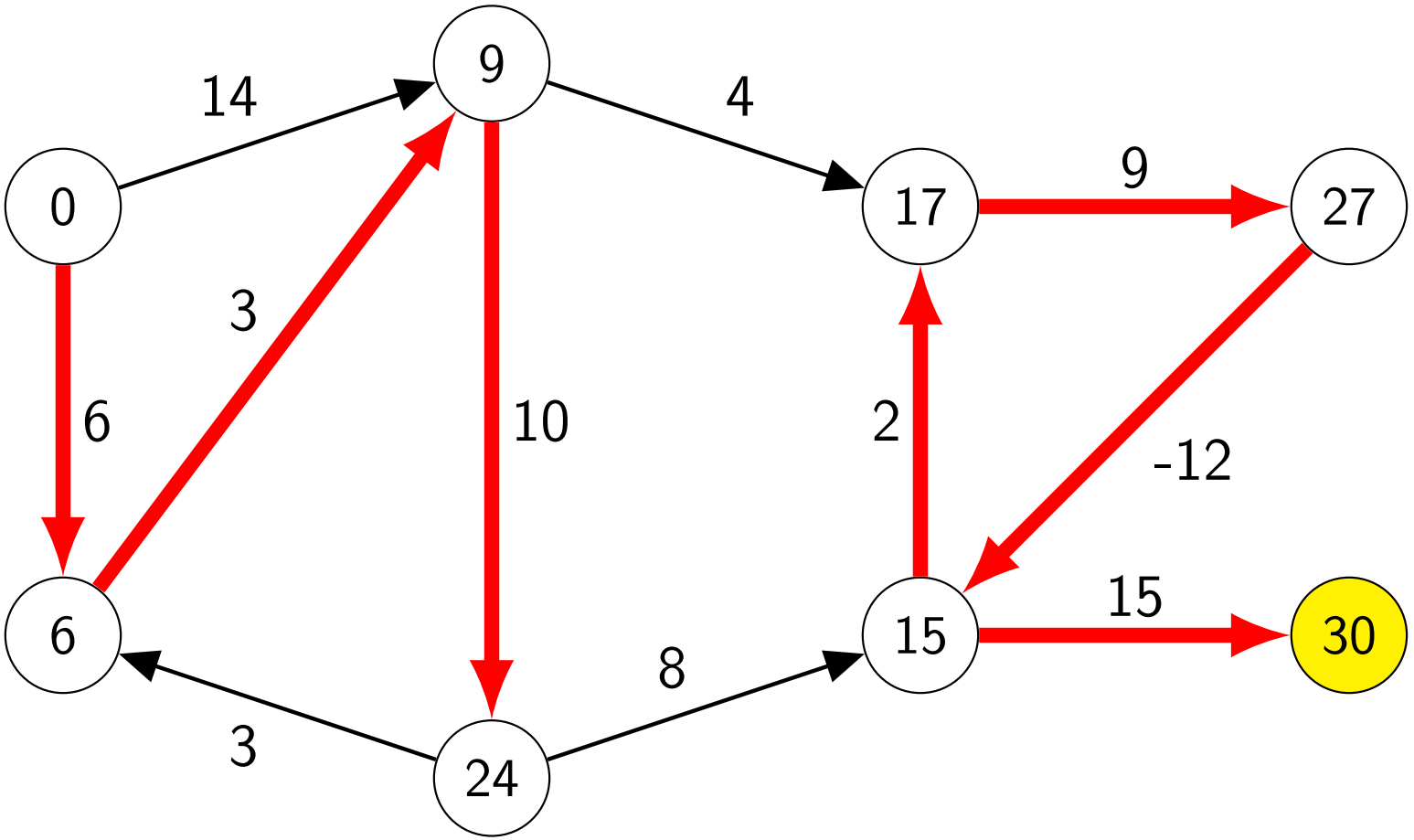
Bellman-Ford – Beispiel



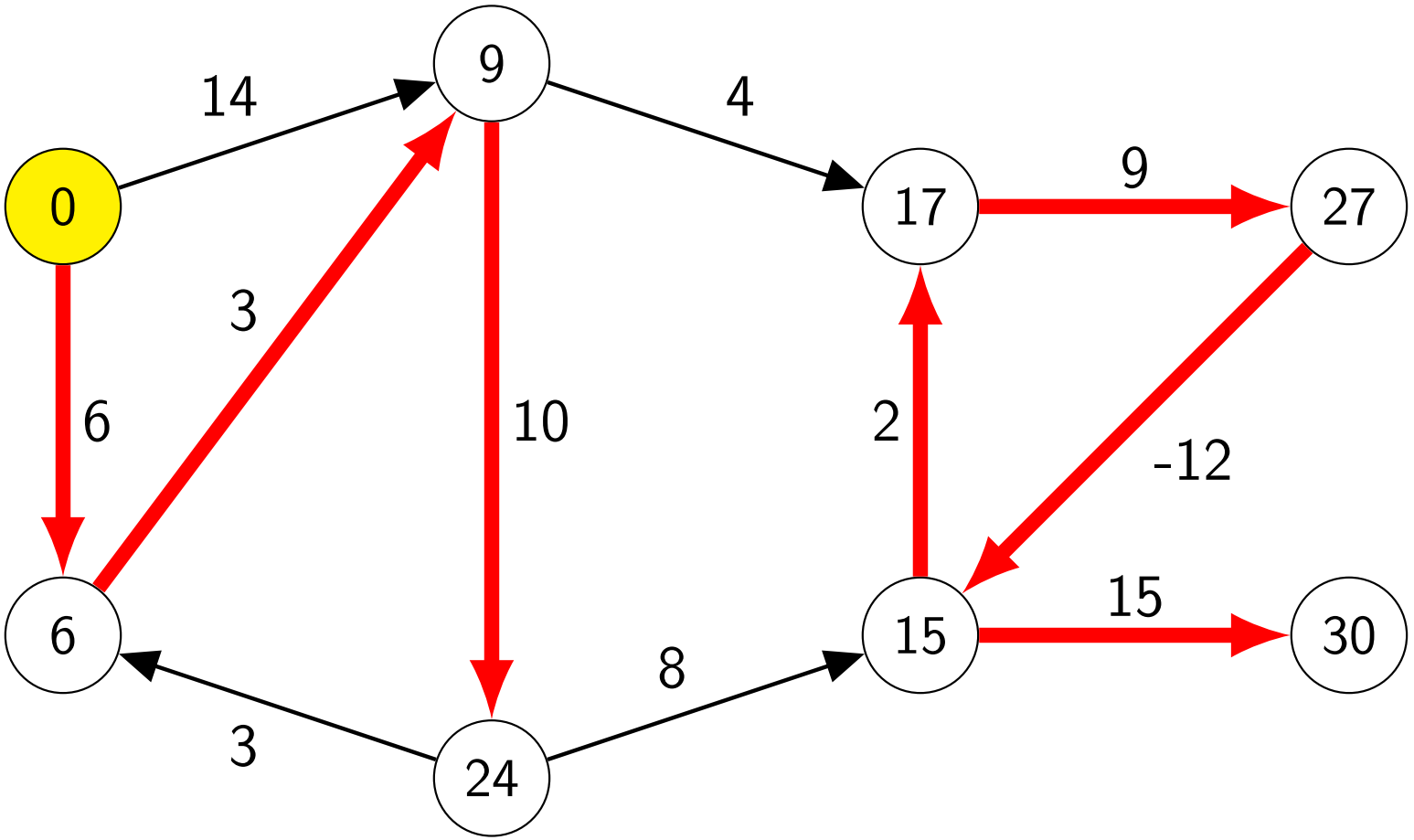
Bellman-Ford – Beispiel



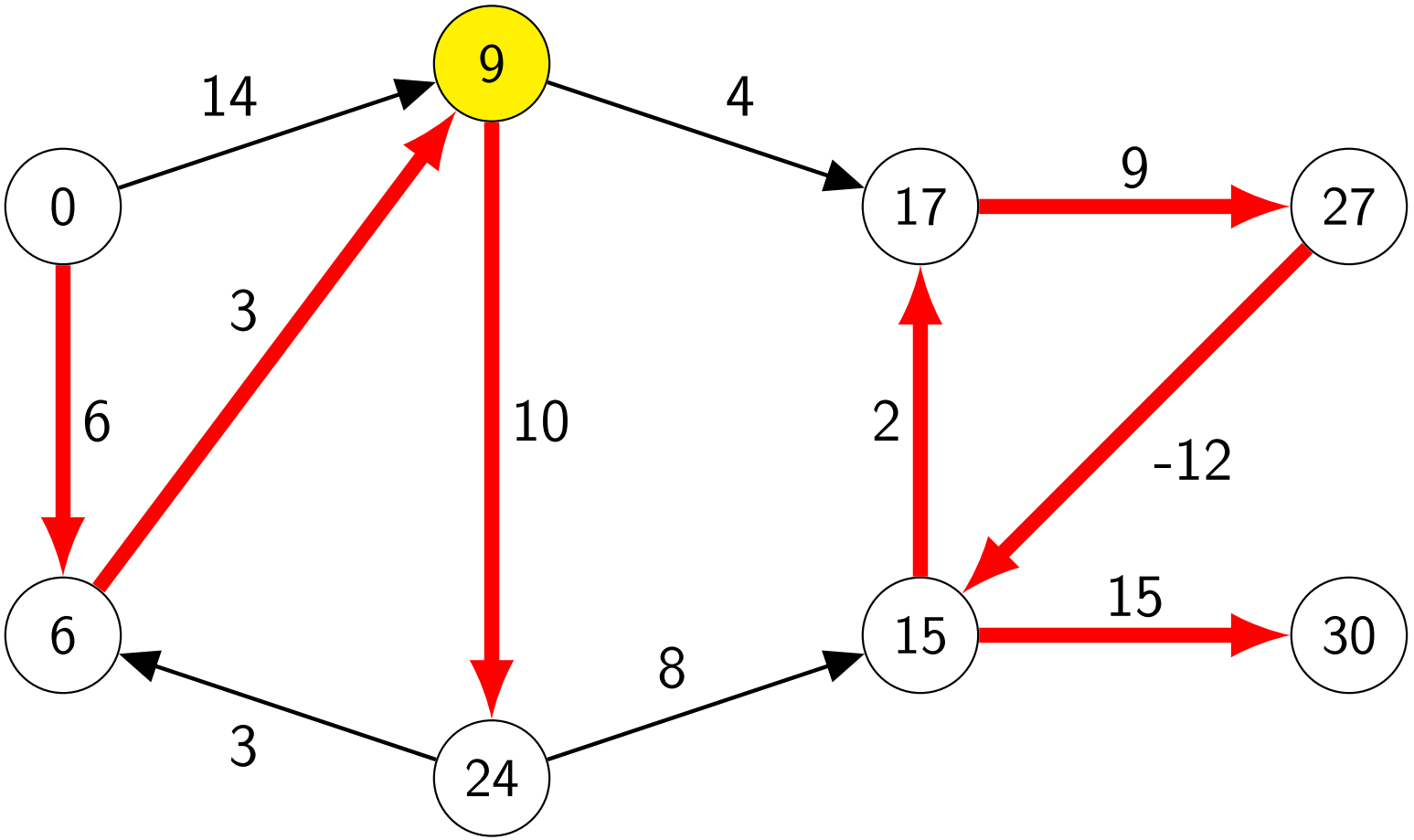
Bellman-Ford – Beispiel



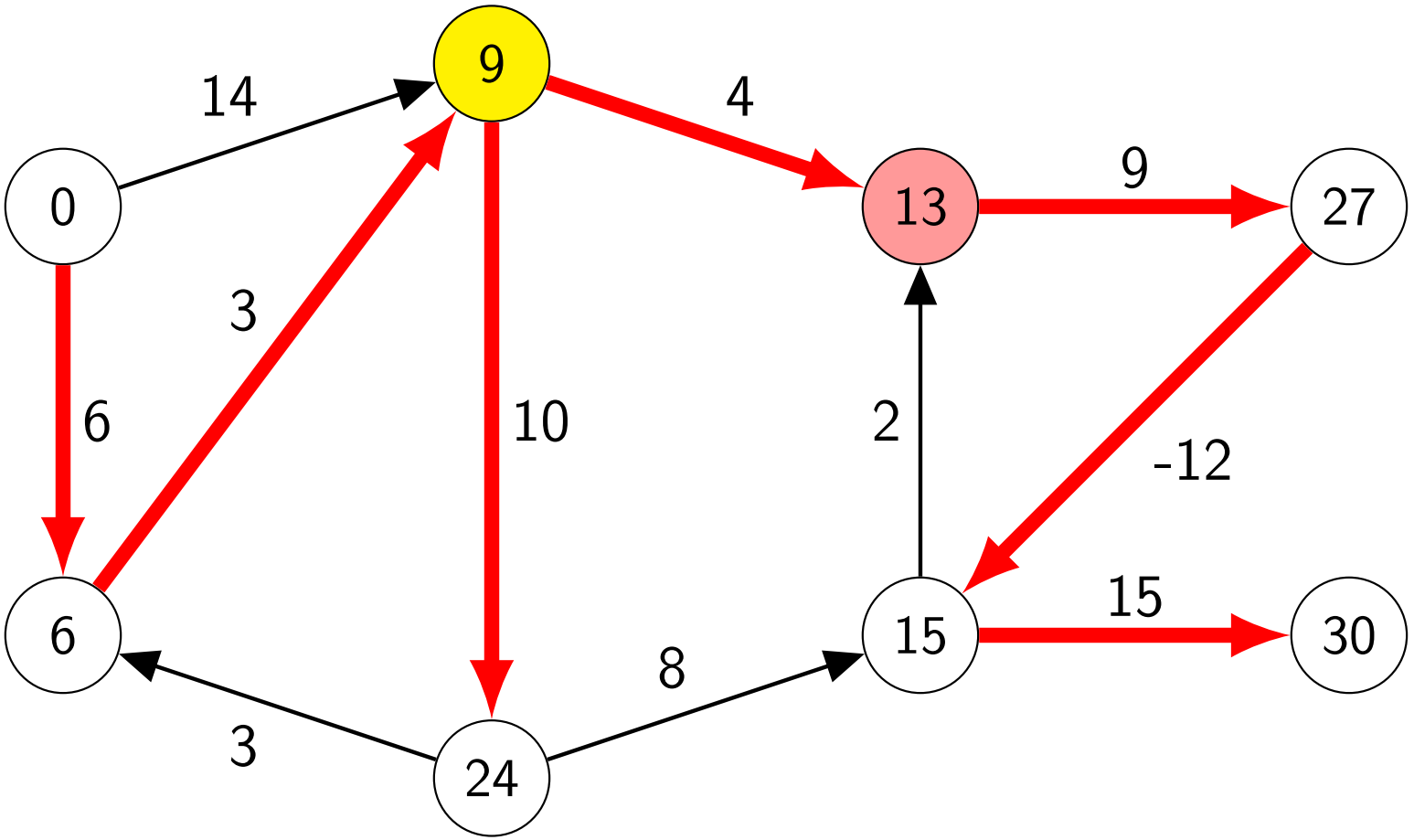
Bellman-Ford – Beispiel



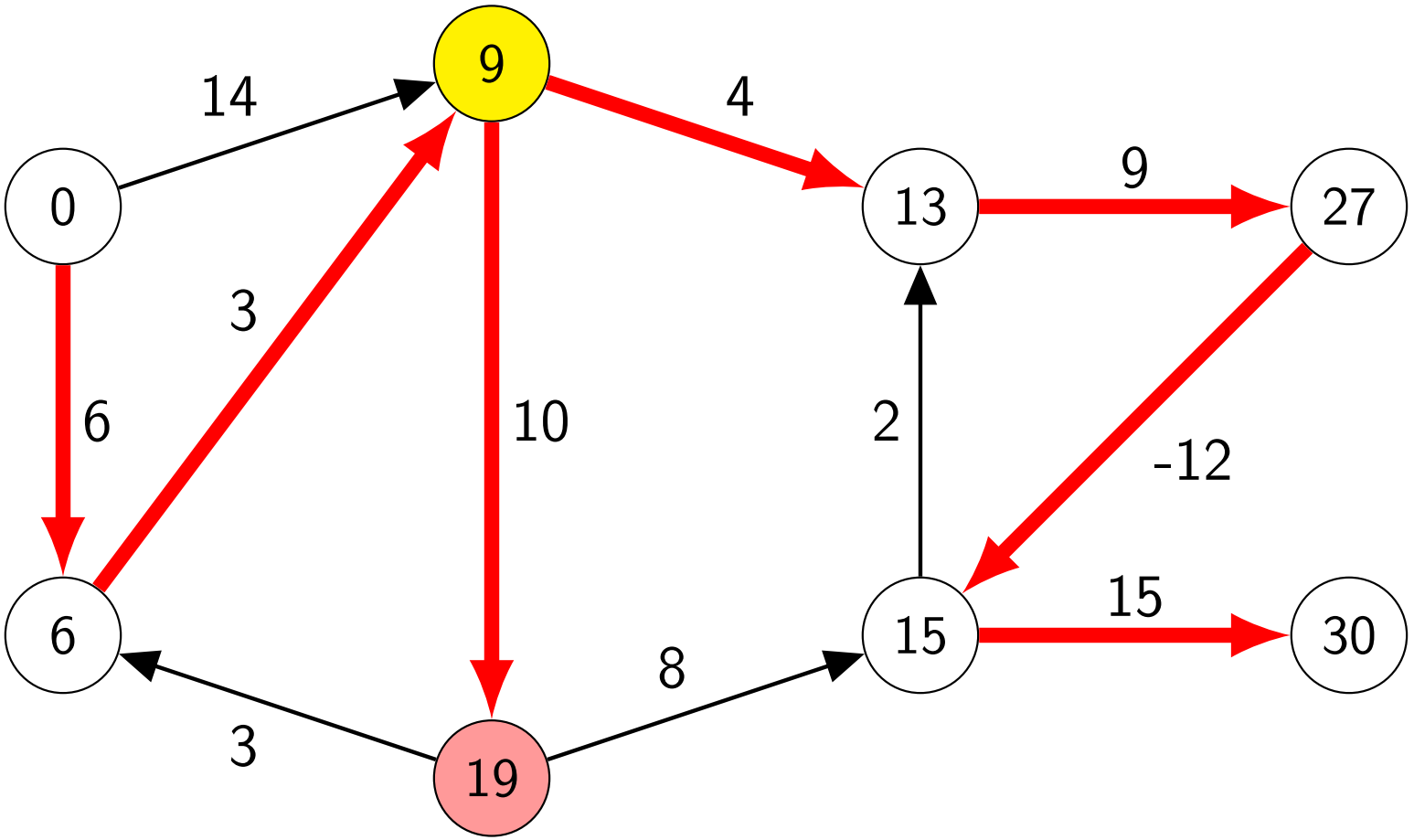
Bellman-Ford – Beispiel



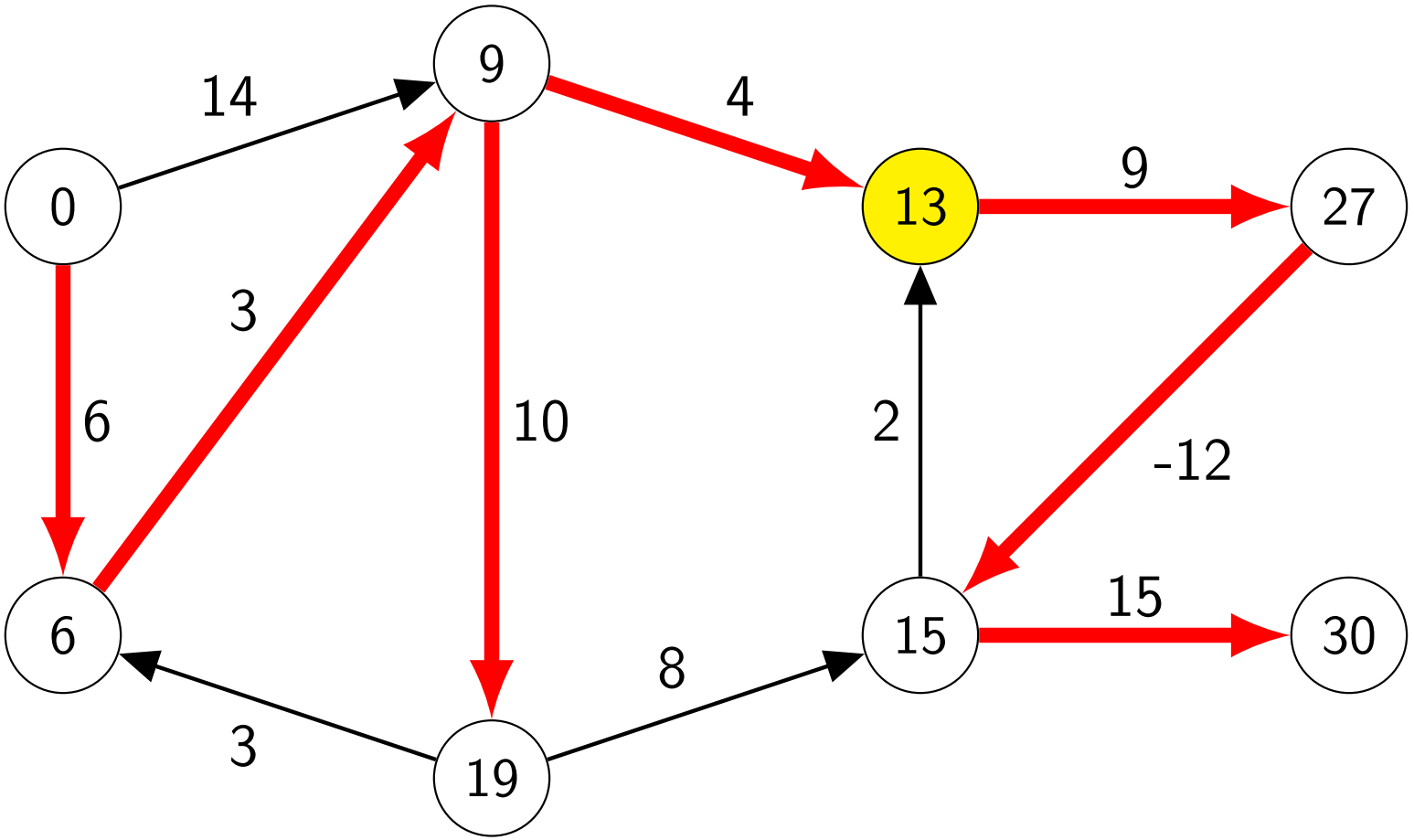
Bellman-Ford – Beispiel



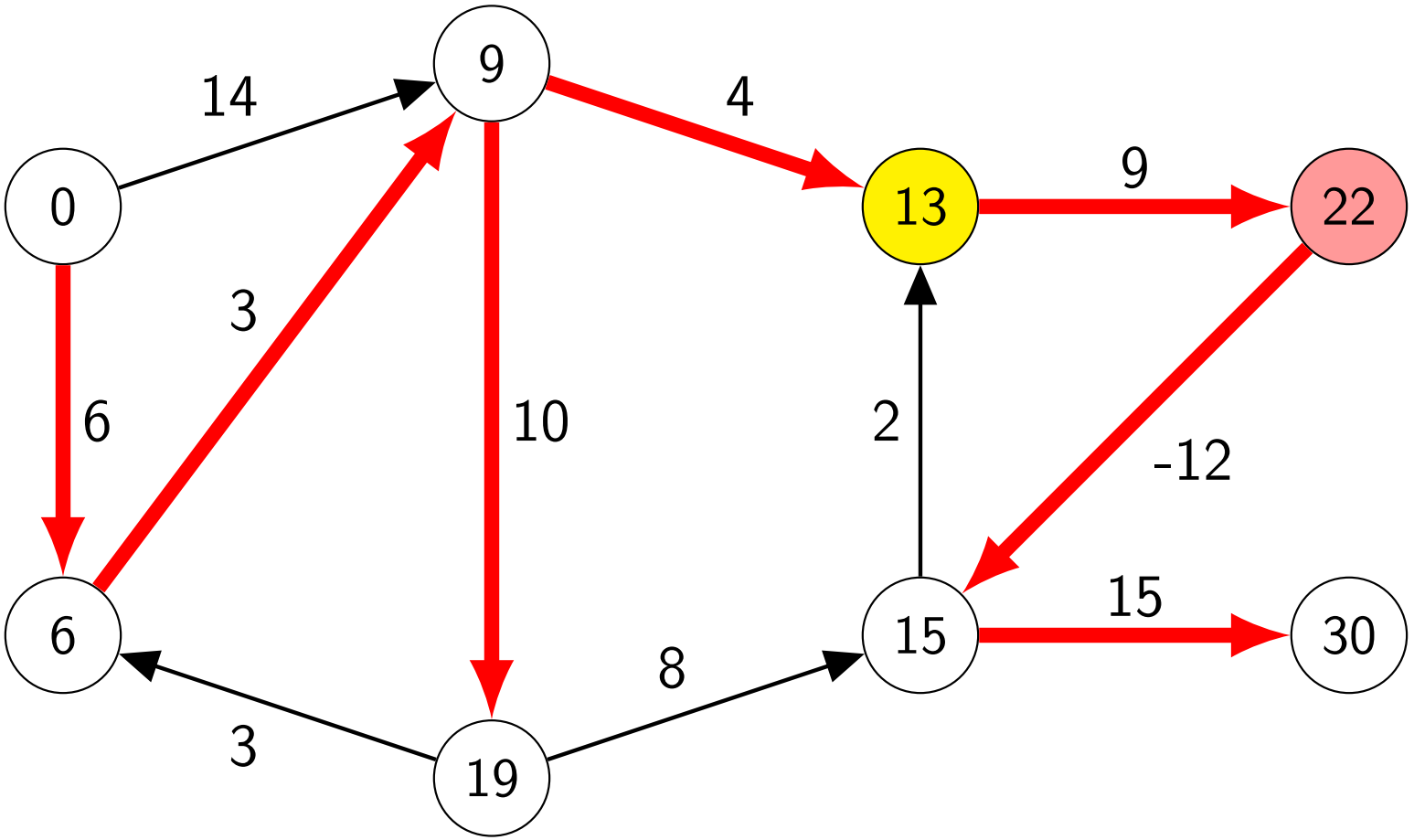
Bellman-Ford – Beispiel



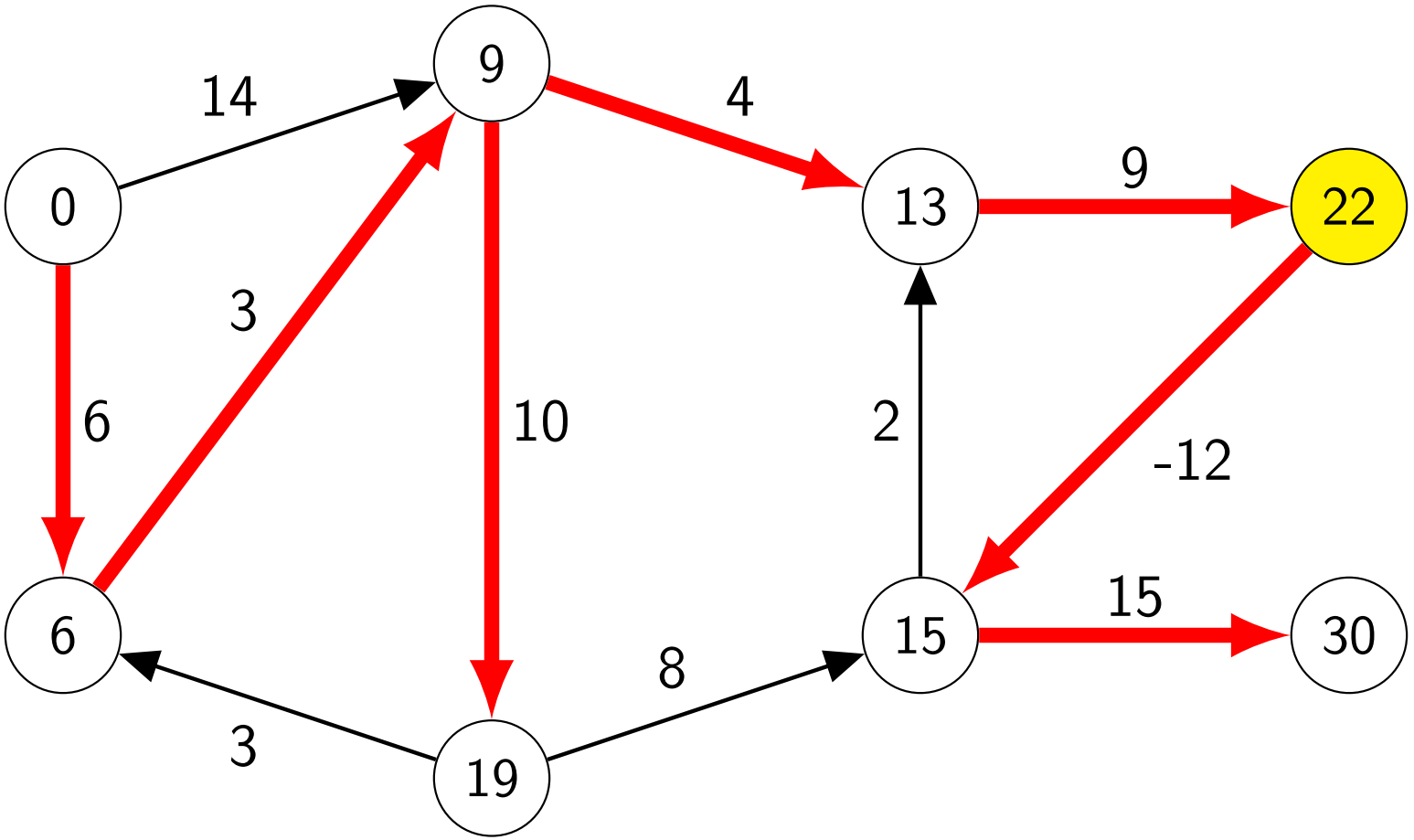
Bellman-Ford – Beispiel



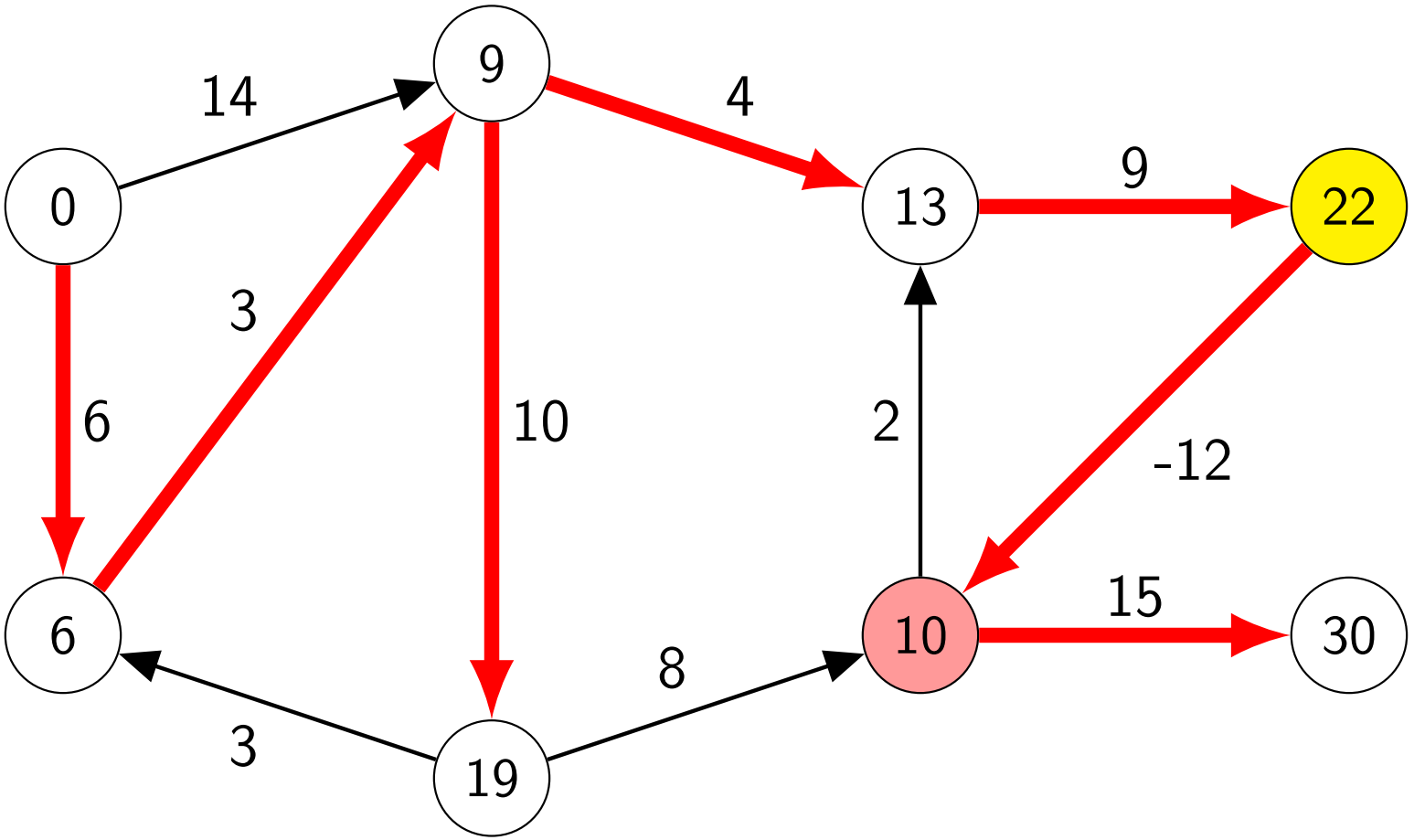
Bellman-Ford – Beispiel



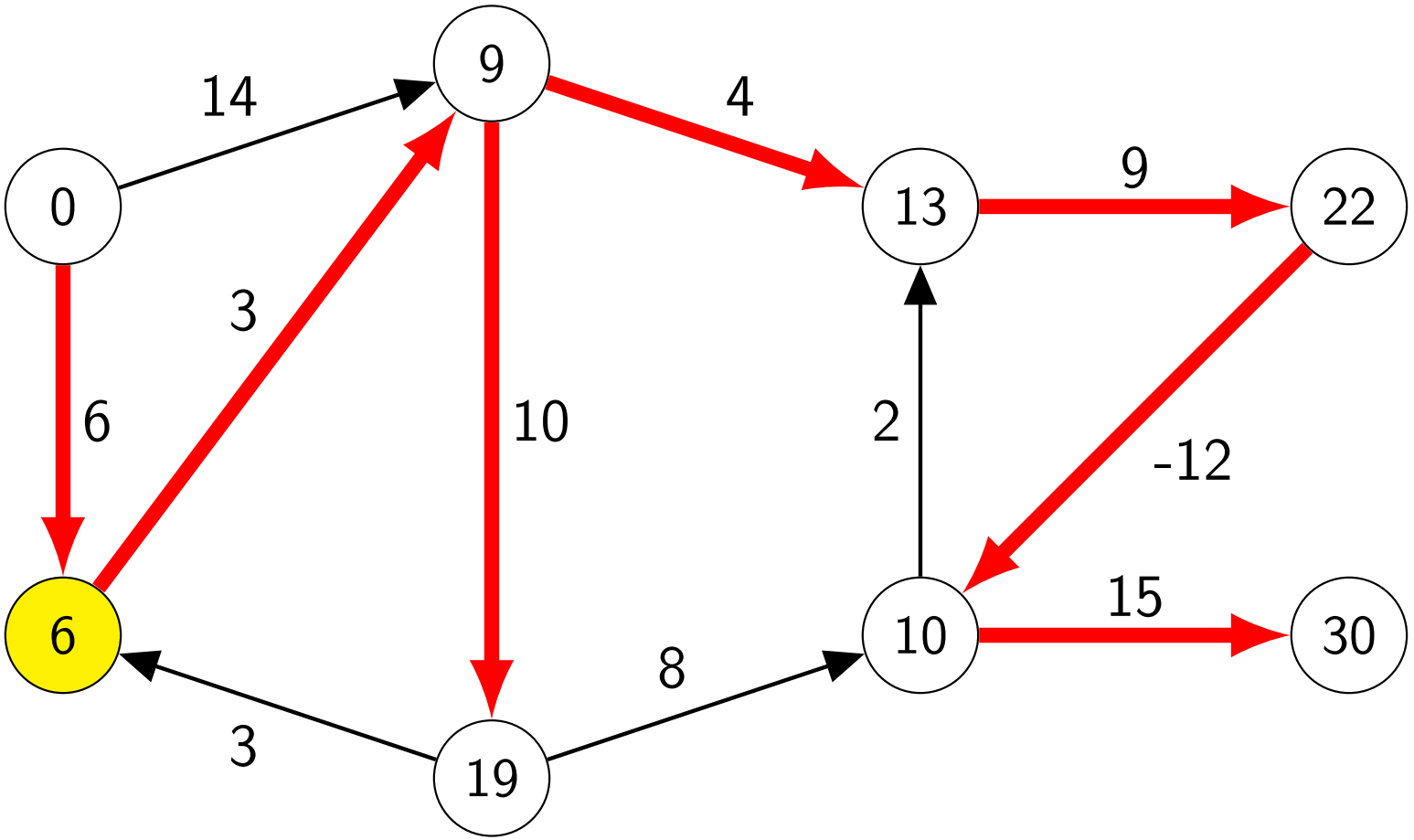
Bellman-Ford – Beispiel



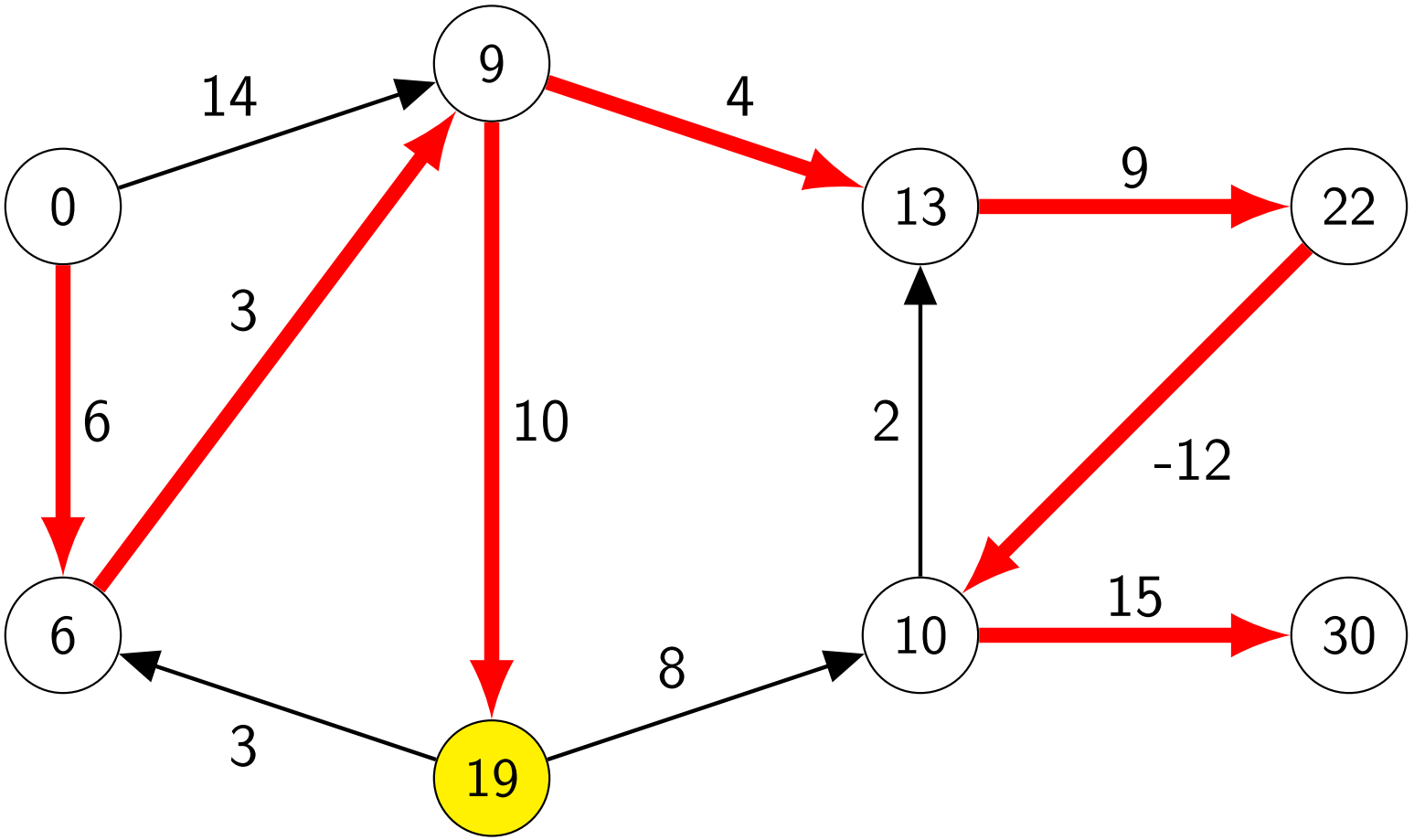
Bellman-Ford – Beispiel



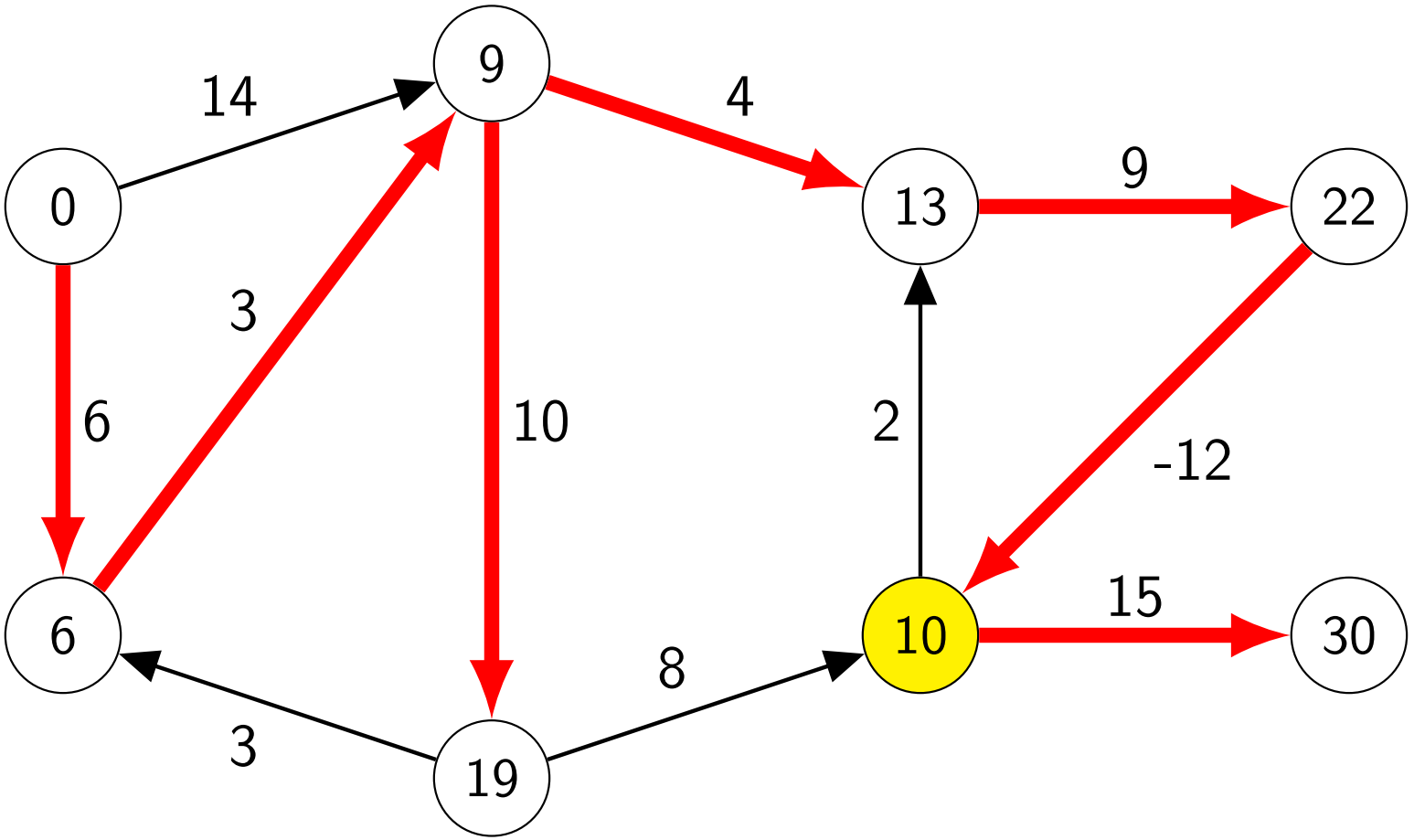
Bellman-Ford – Beispiel



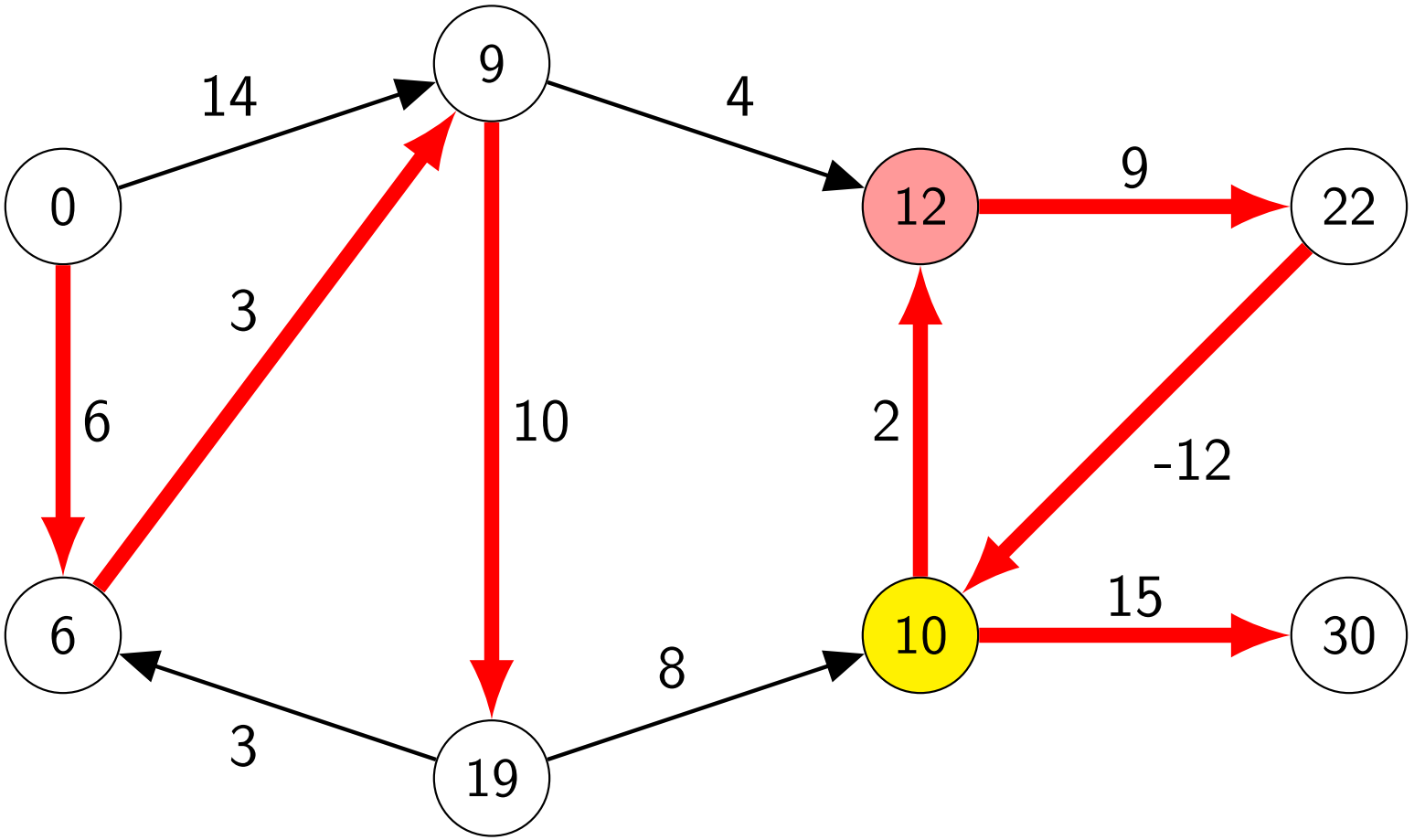
Bellman-Ford – Beispiel



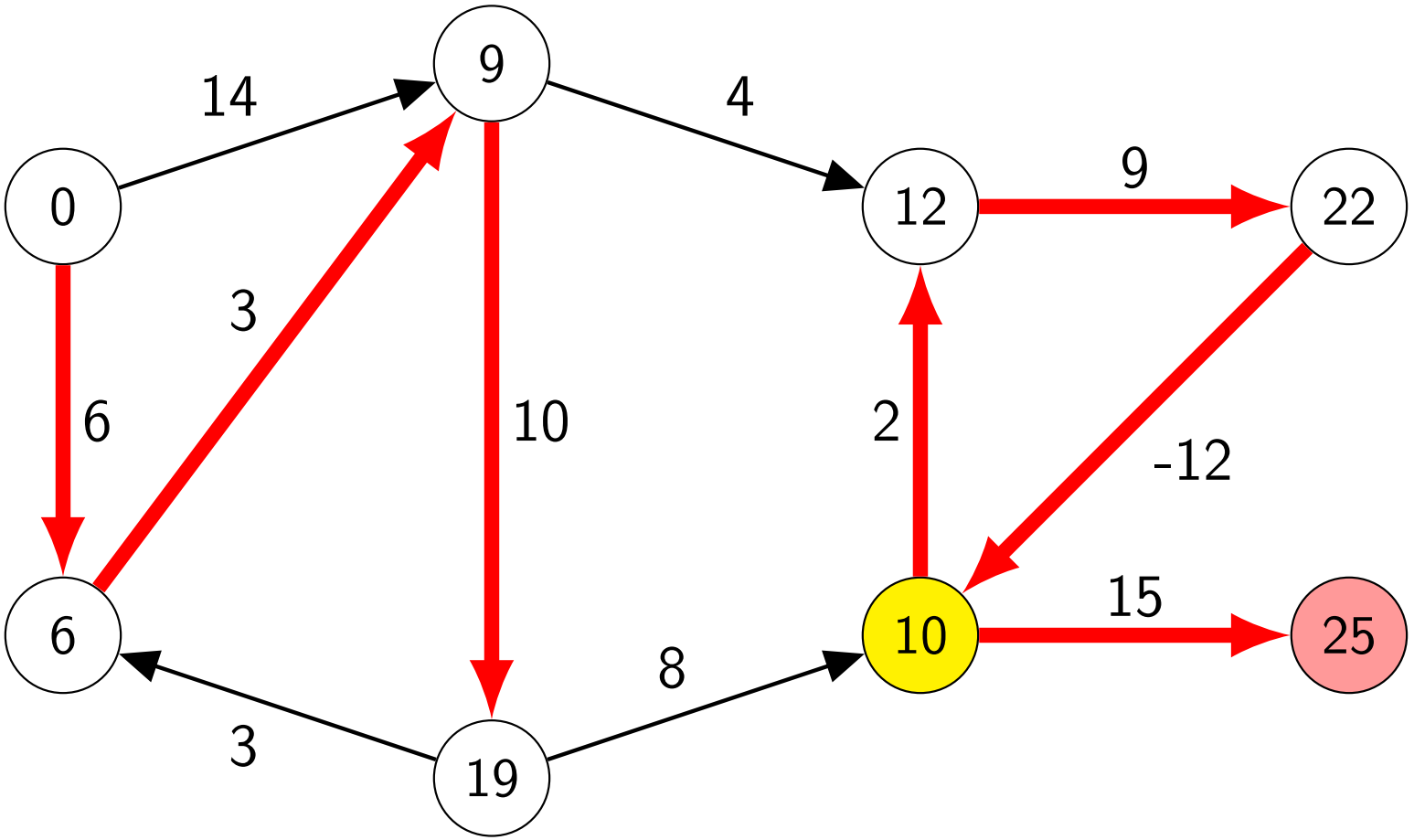
Bellman-Ford – Beispiel



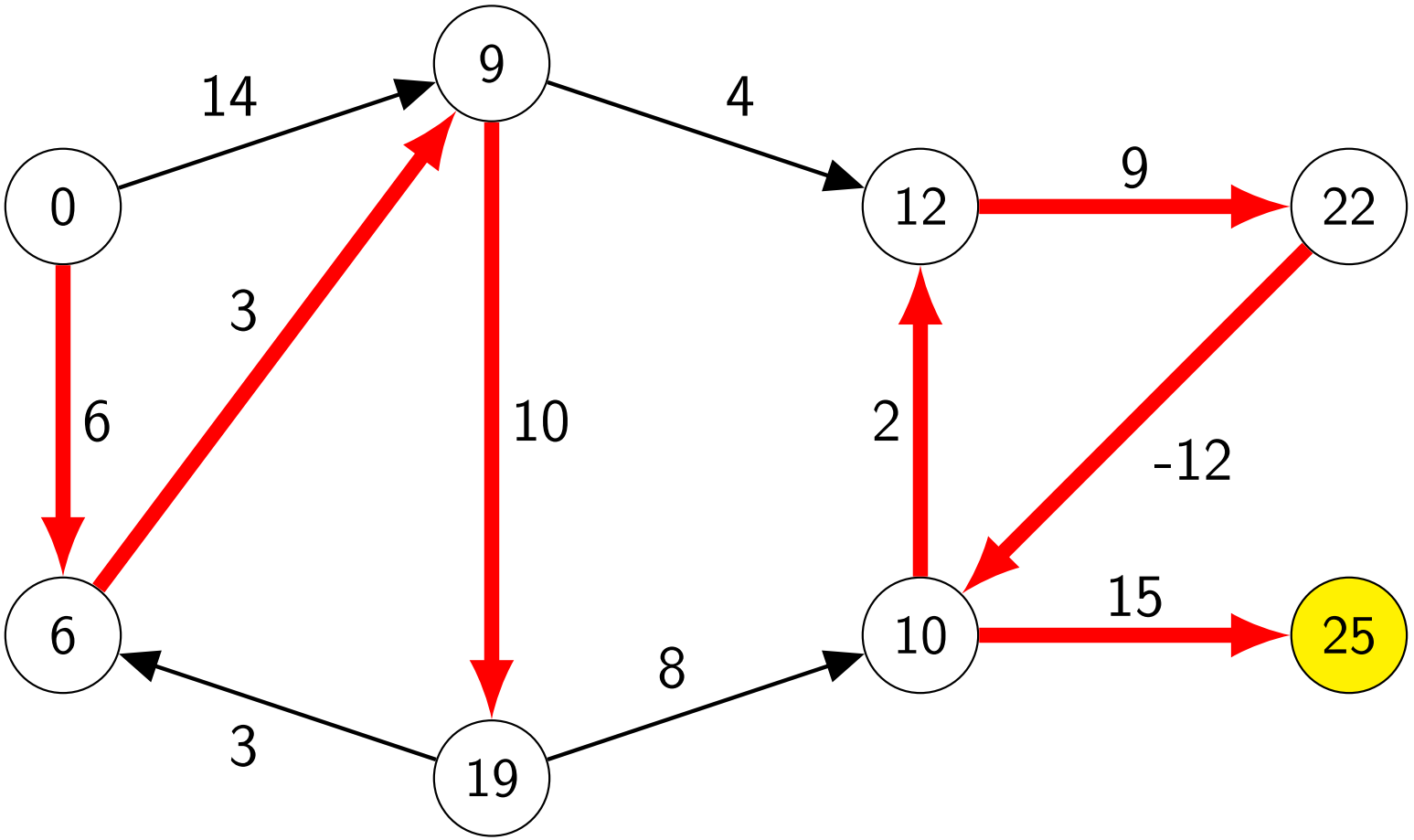
Bellman-Ford – Beispiel



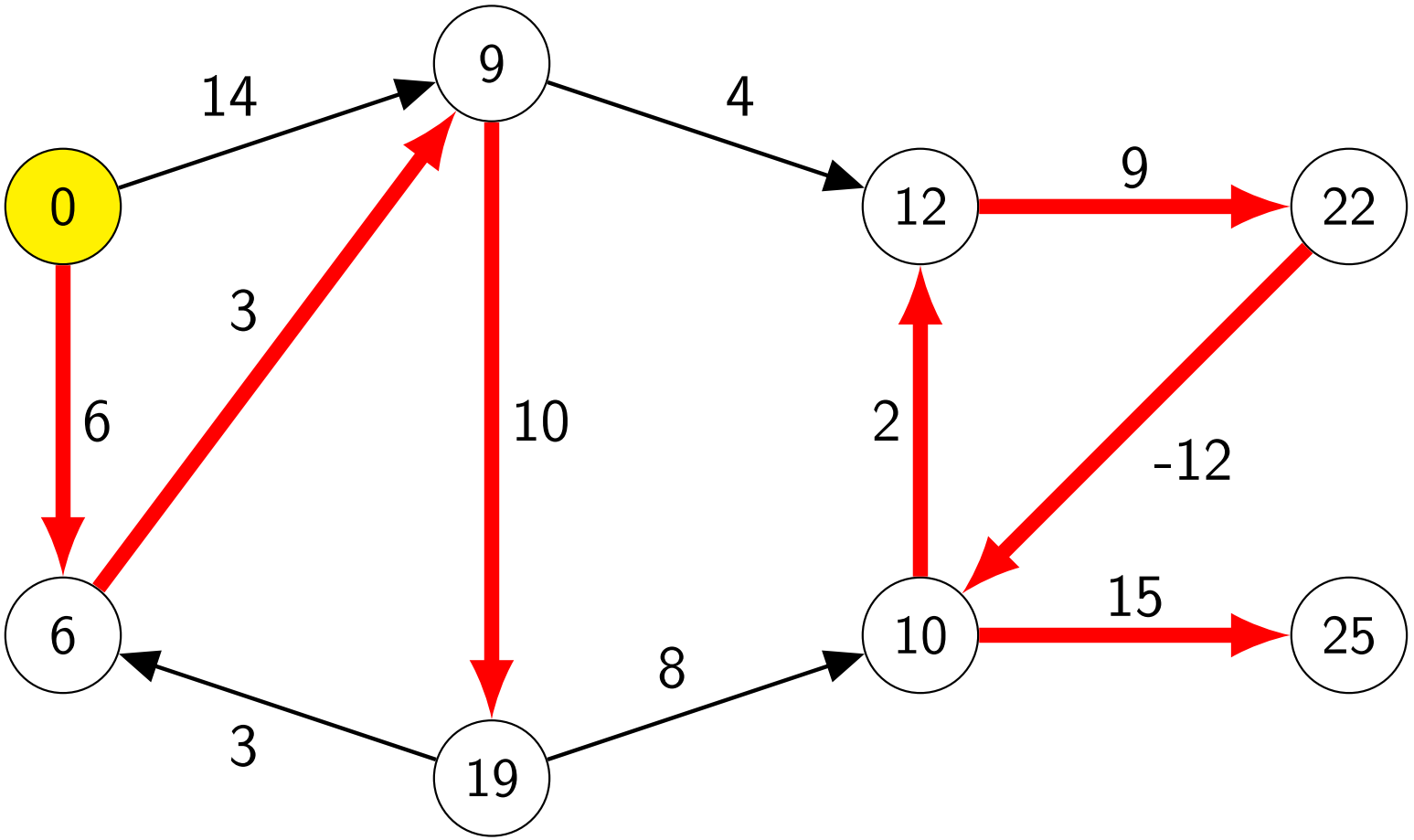
Bellman-Ford – Beispiel



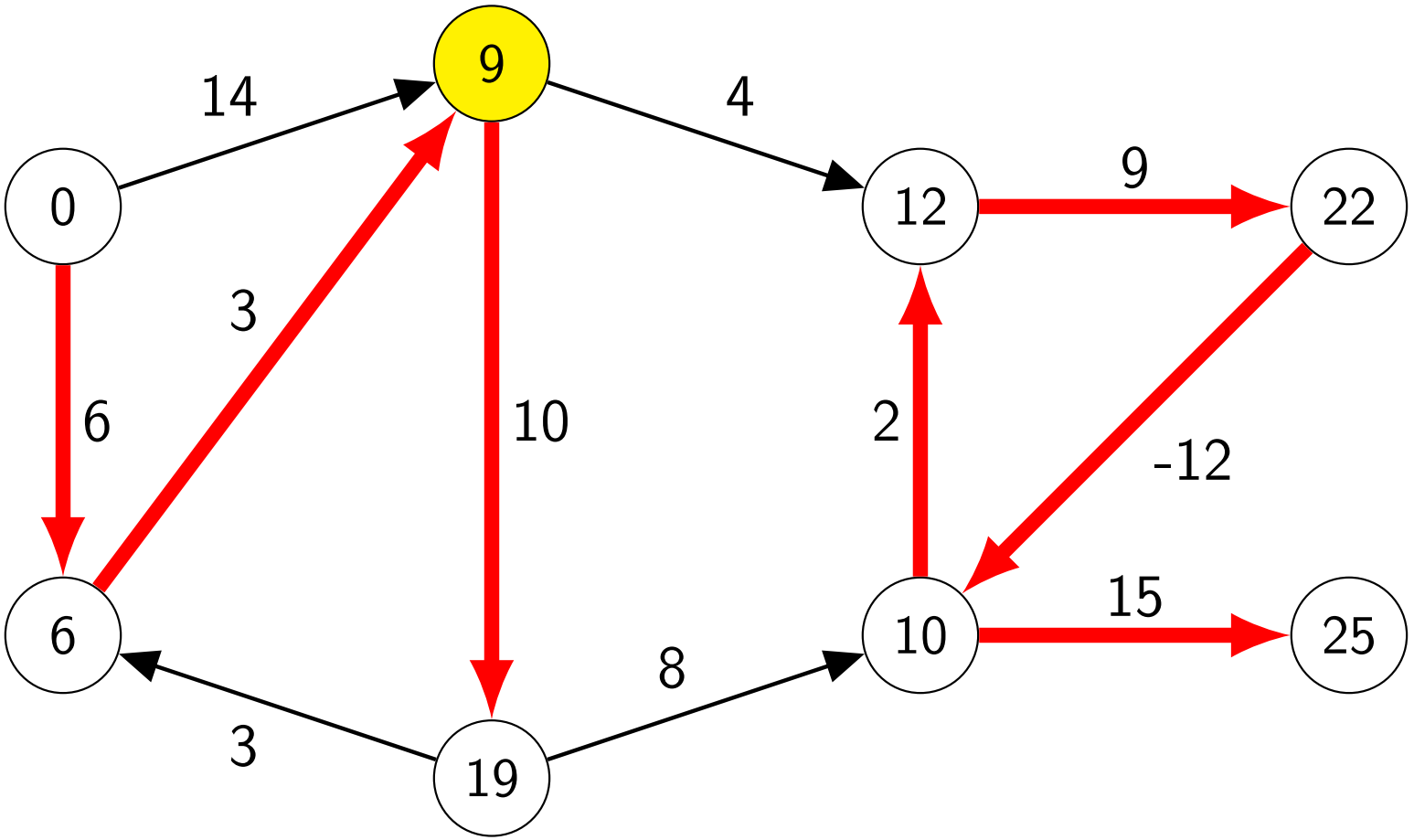
Bellman-Ford – Beispiel



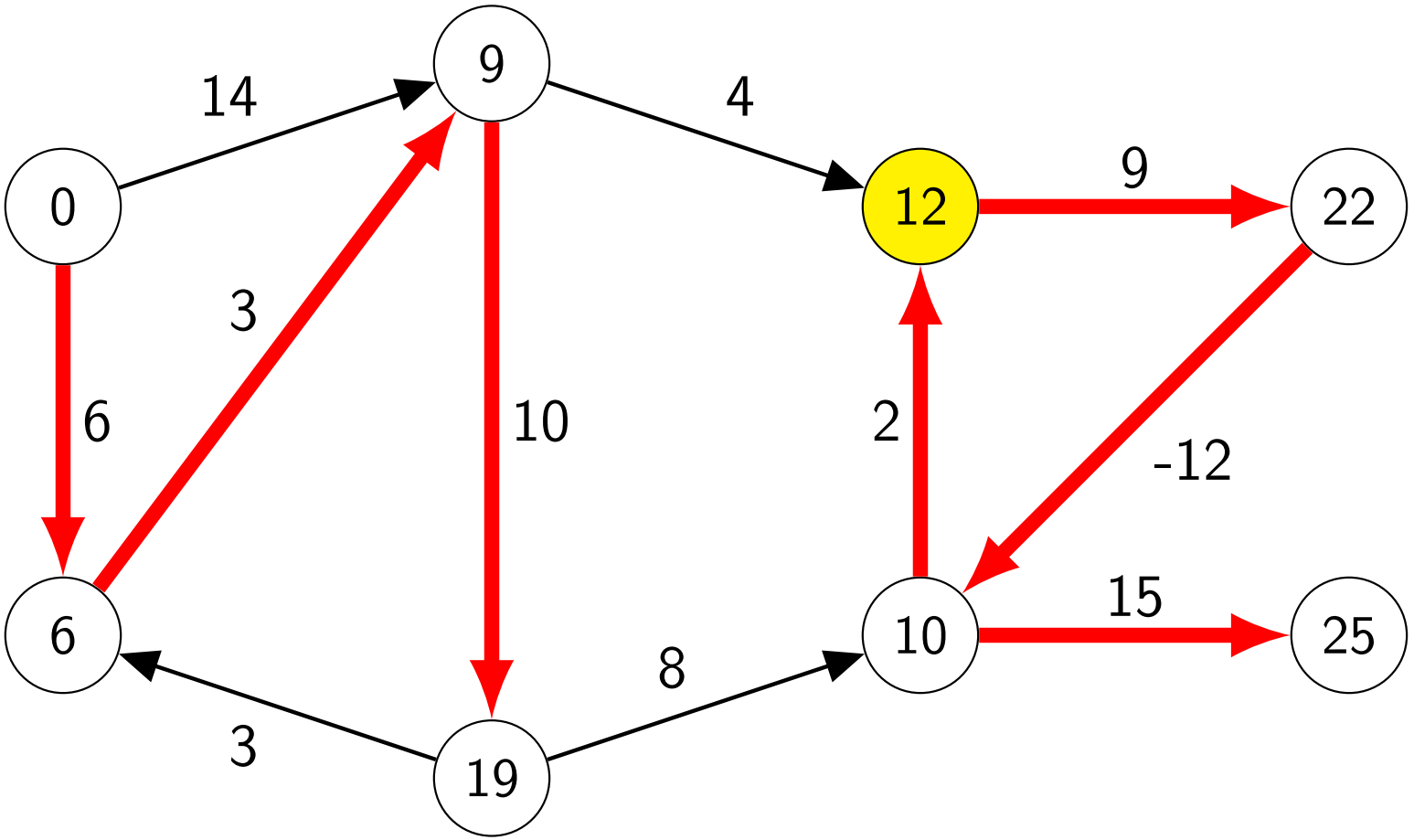
Bellman-Ford – Beispiel



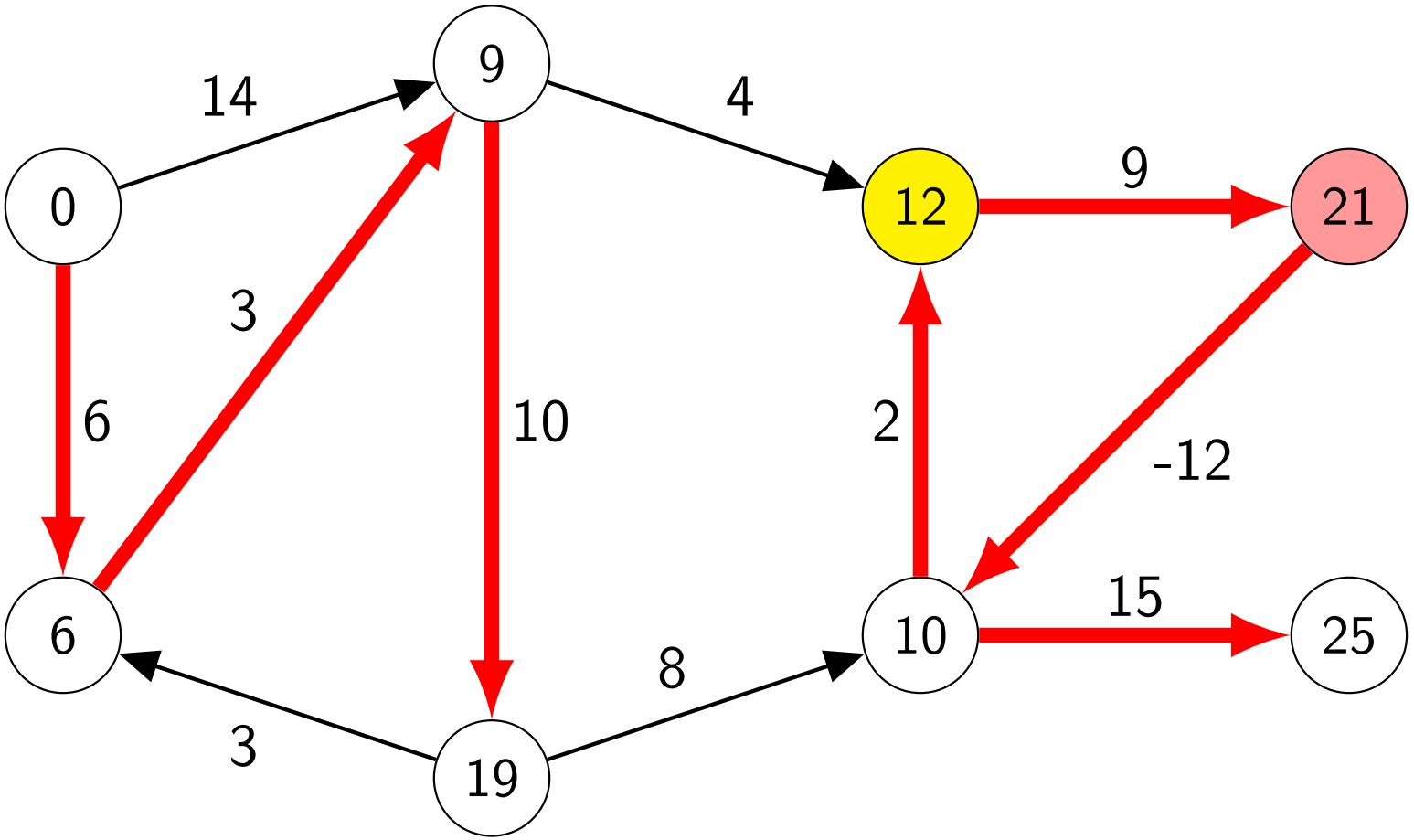
Bellman-Ford – Beispiel



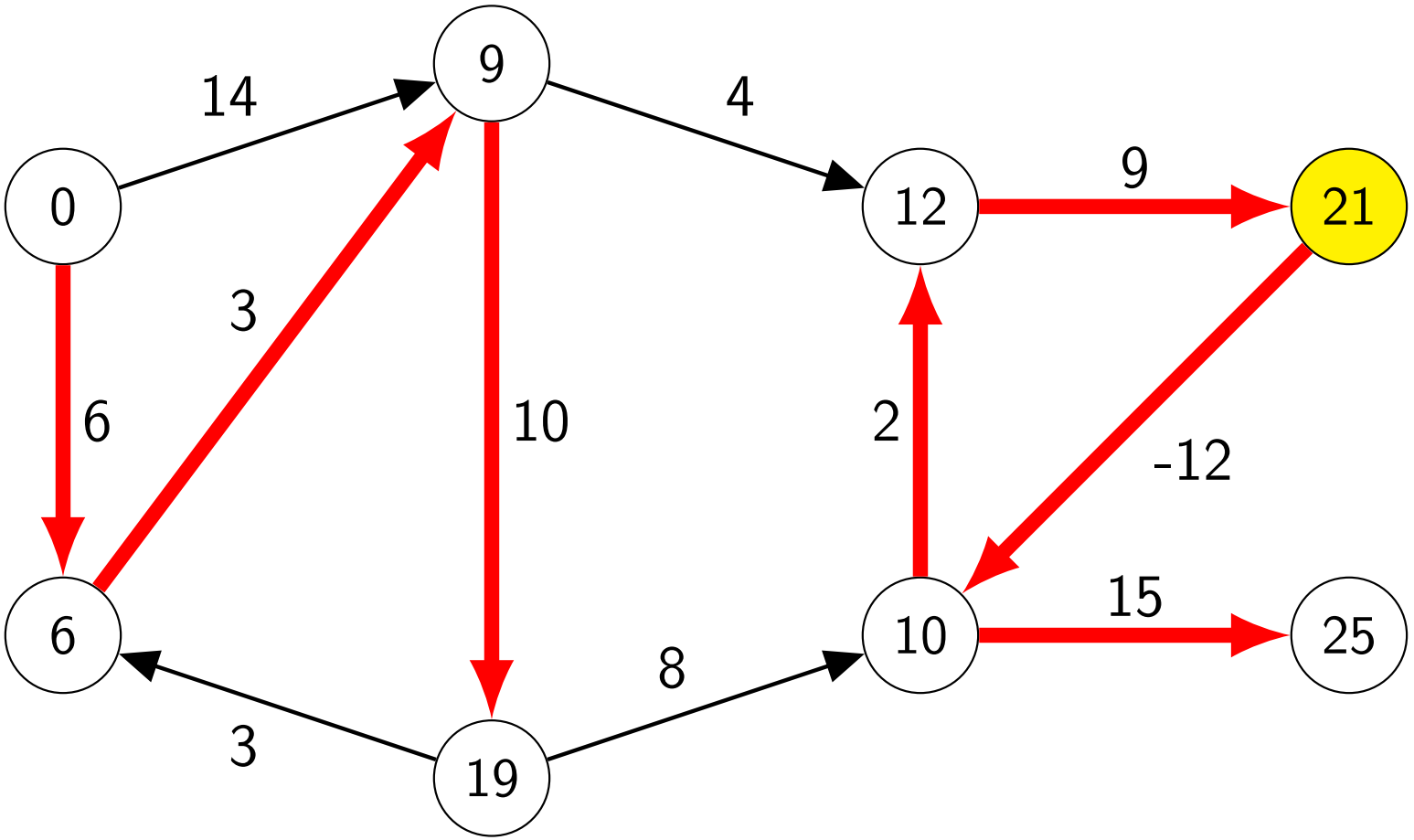
Bellman-Ford – Beispiel



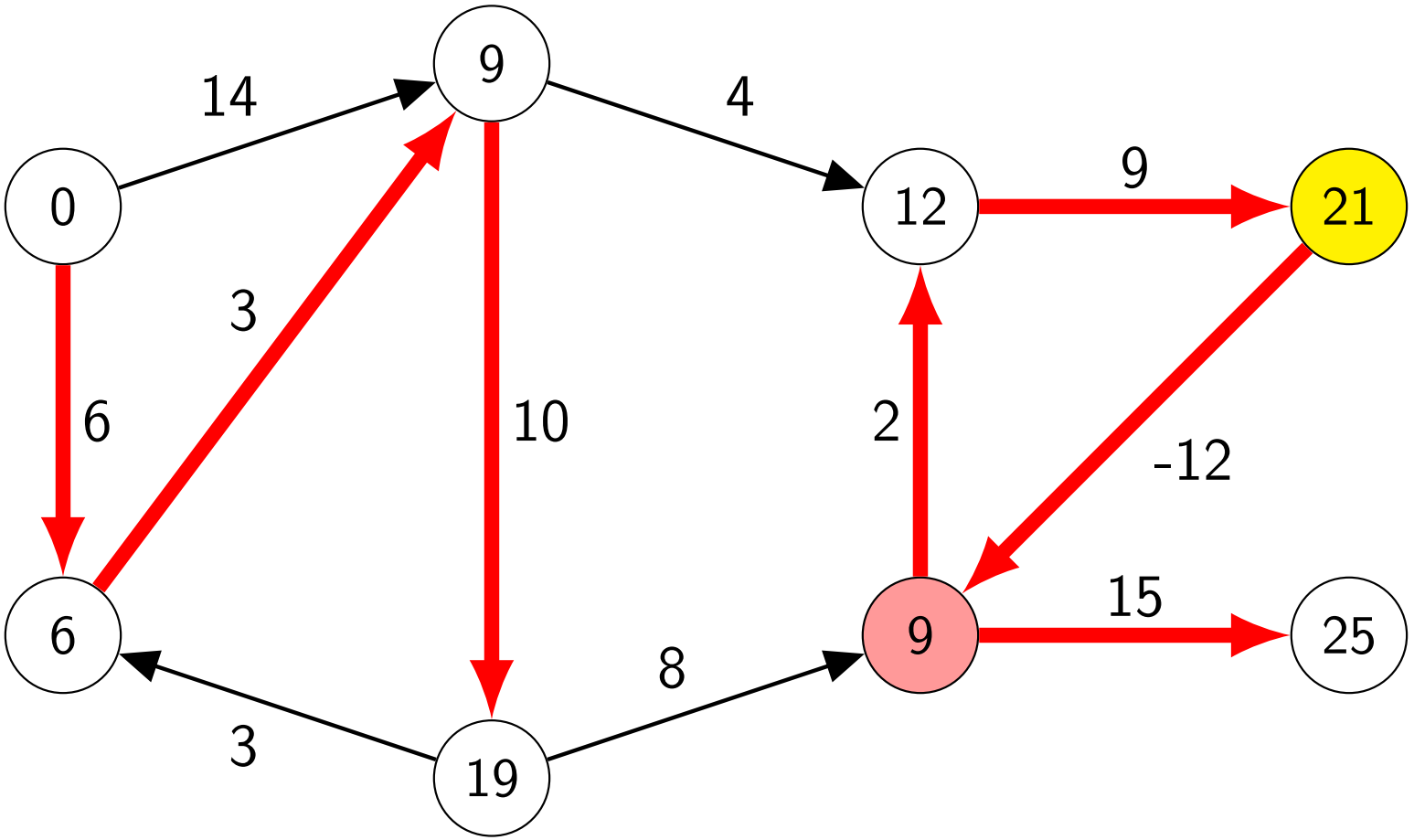
Bellman-Ford – Beispiel



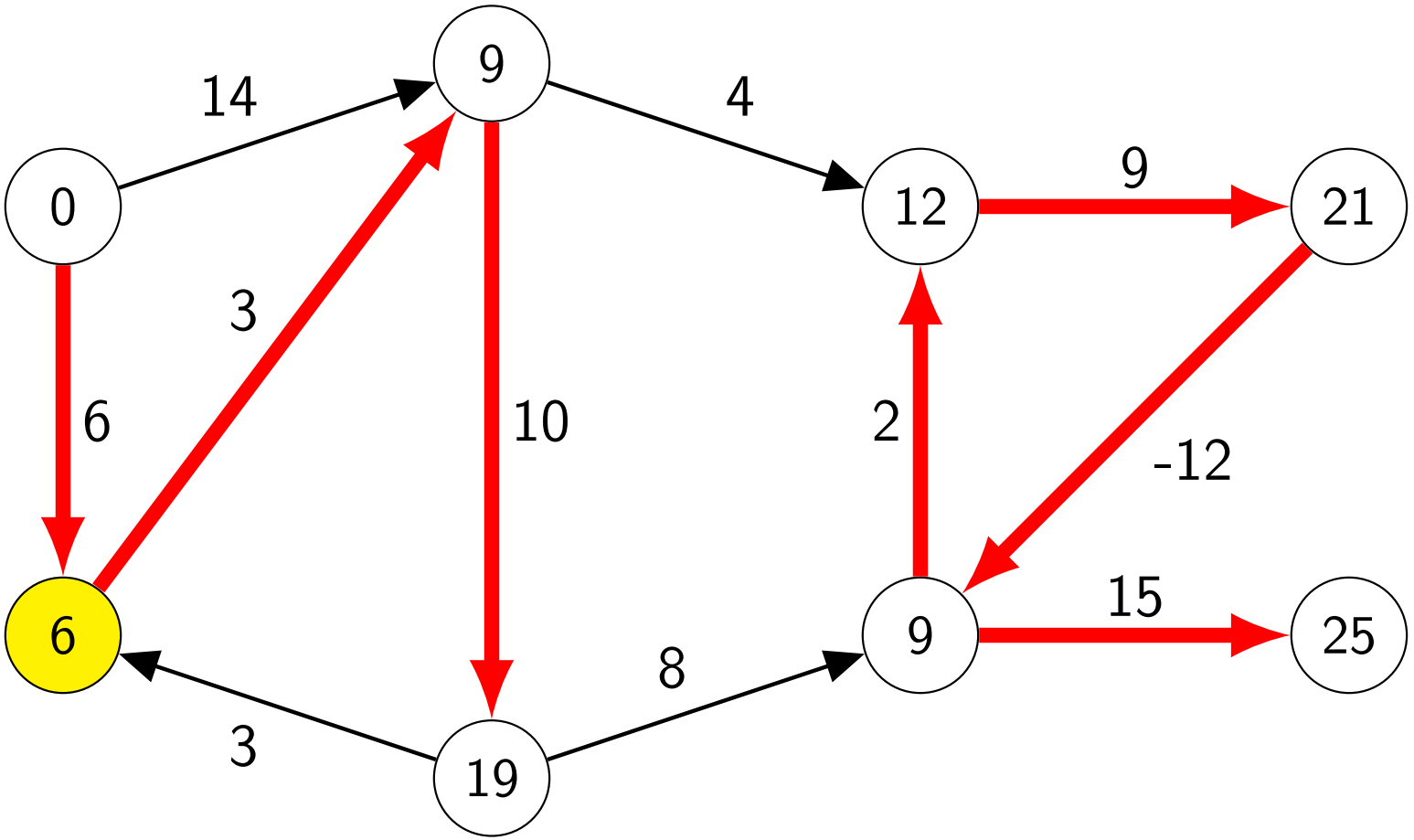
Bellman-Ford – Beispiel



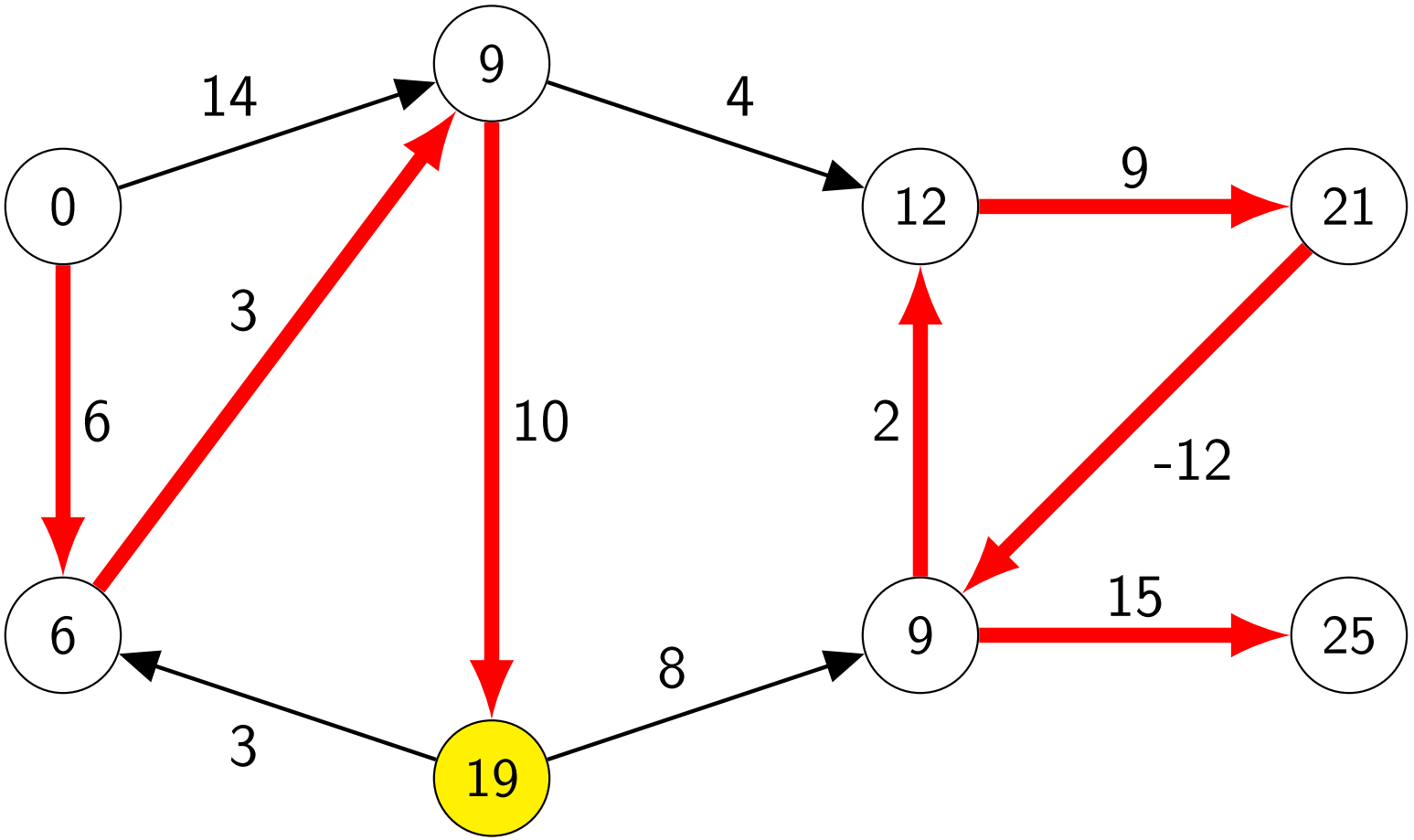
Bellman-Ford – Beispiel



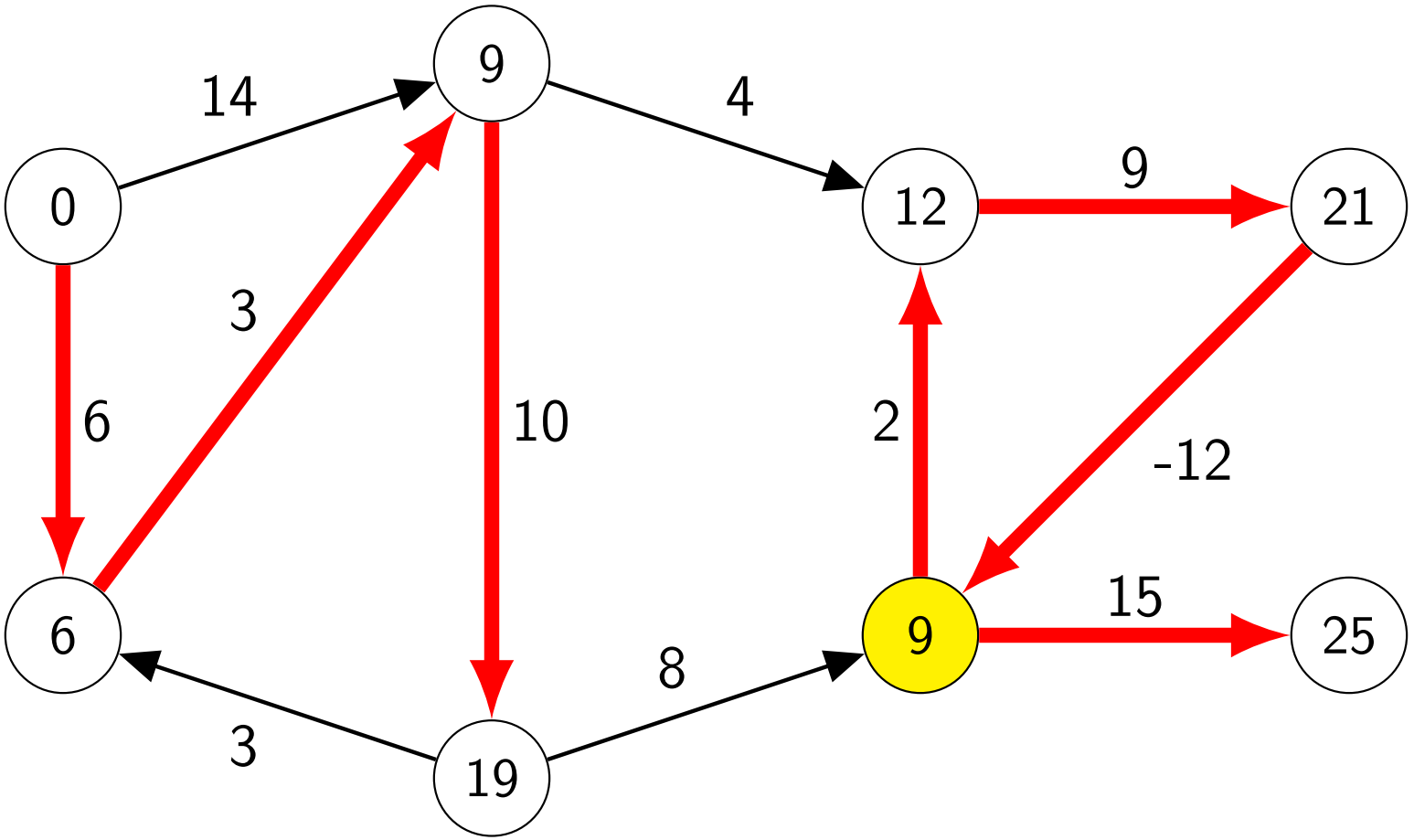
Bellman-Ford – Beispiel



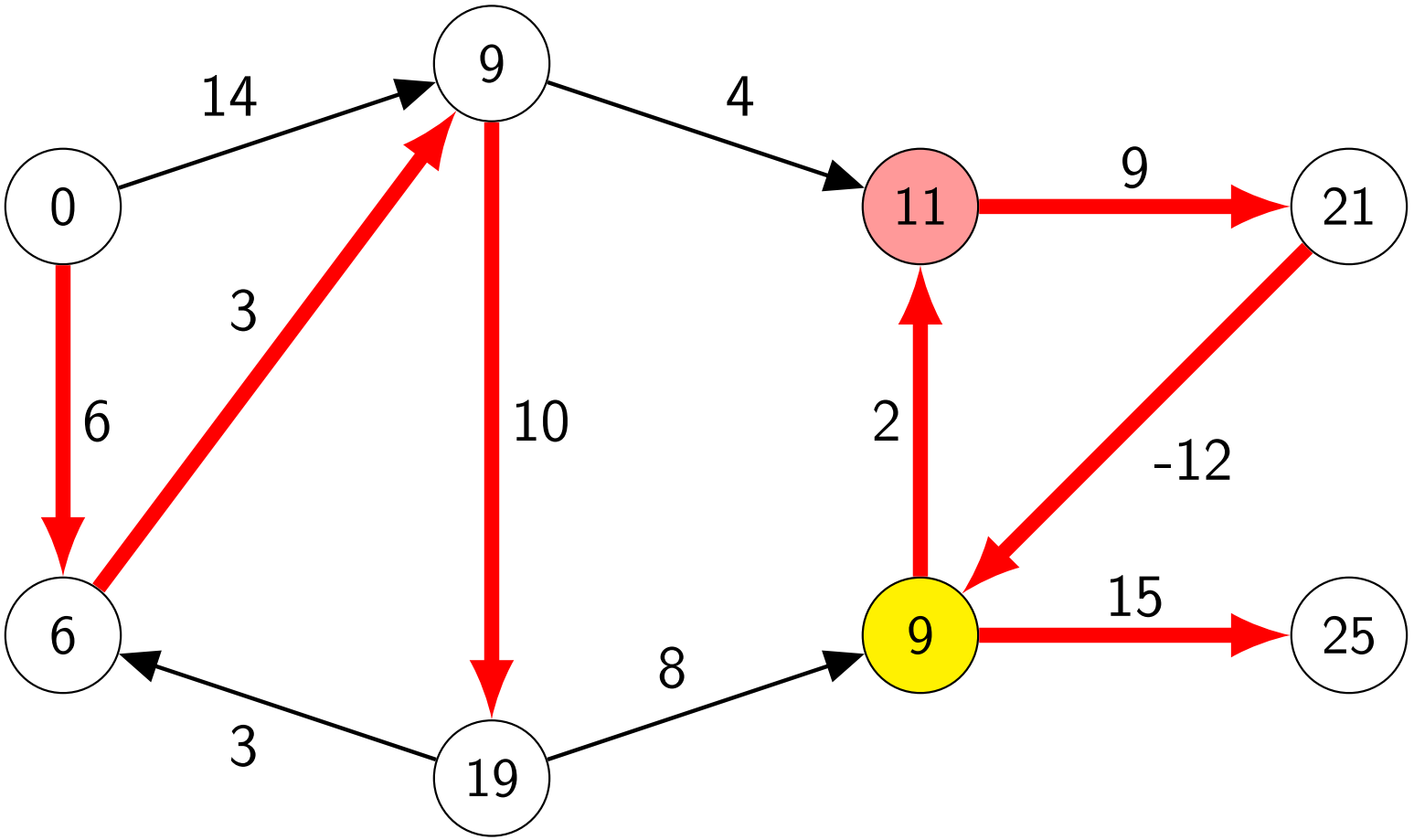
Bellman-Ford – Beispiel



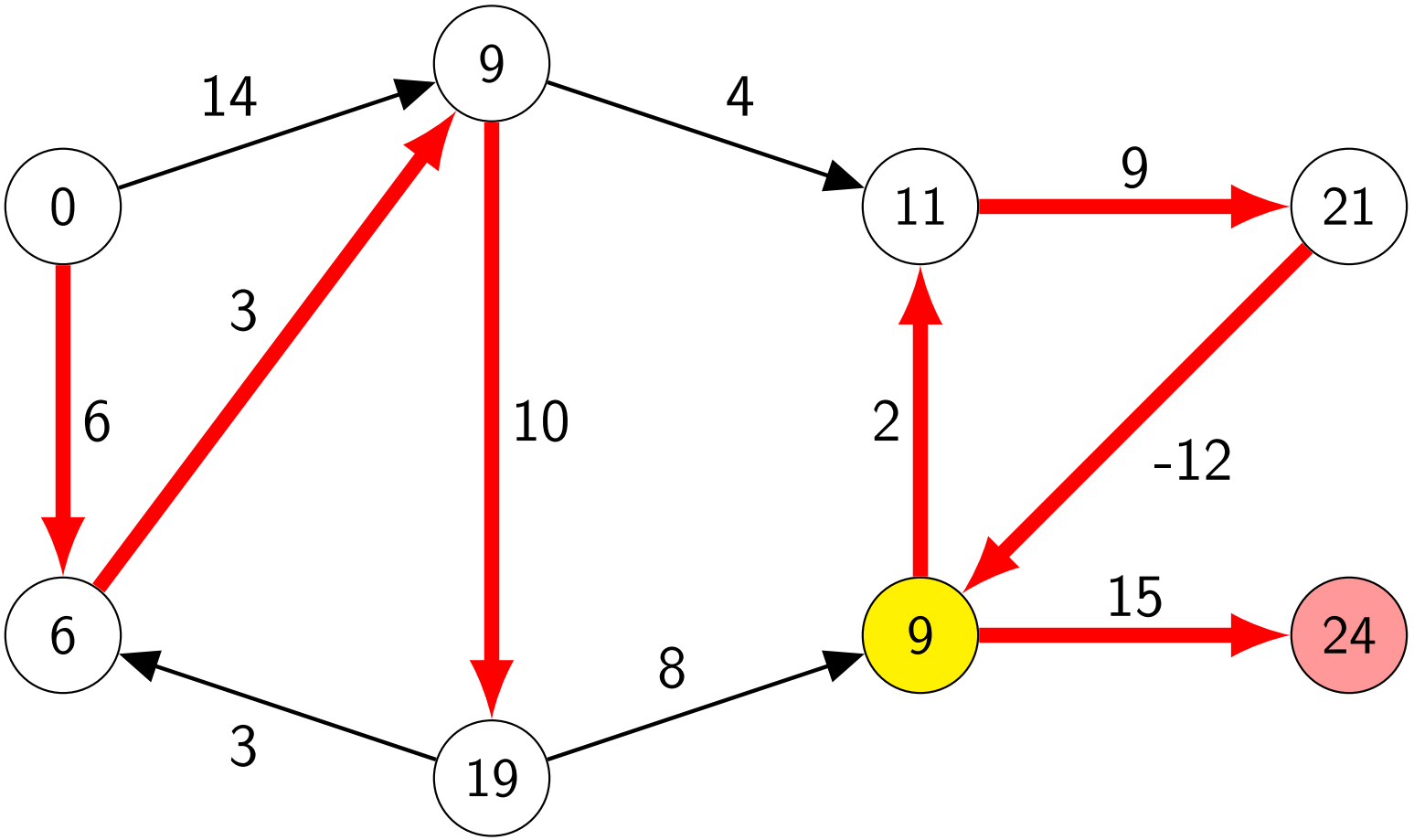
Bellman-Ford – Beispiel



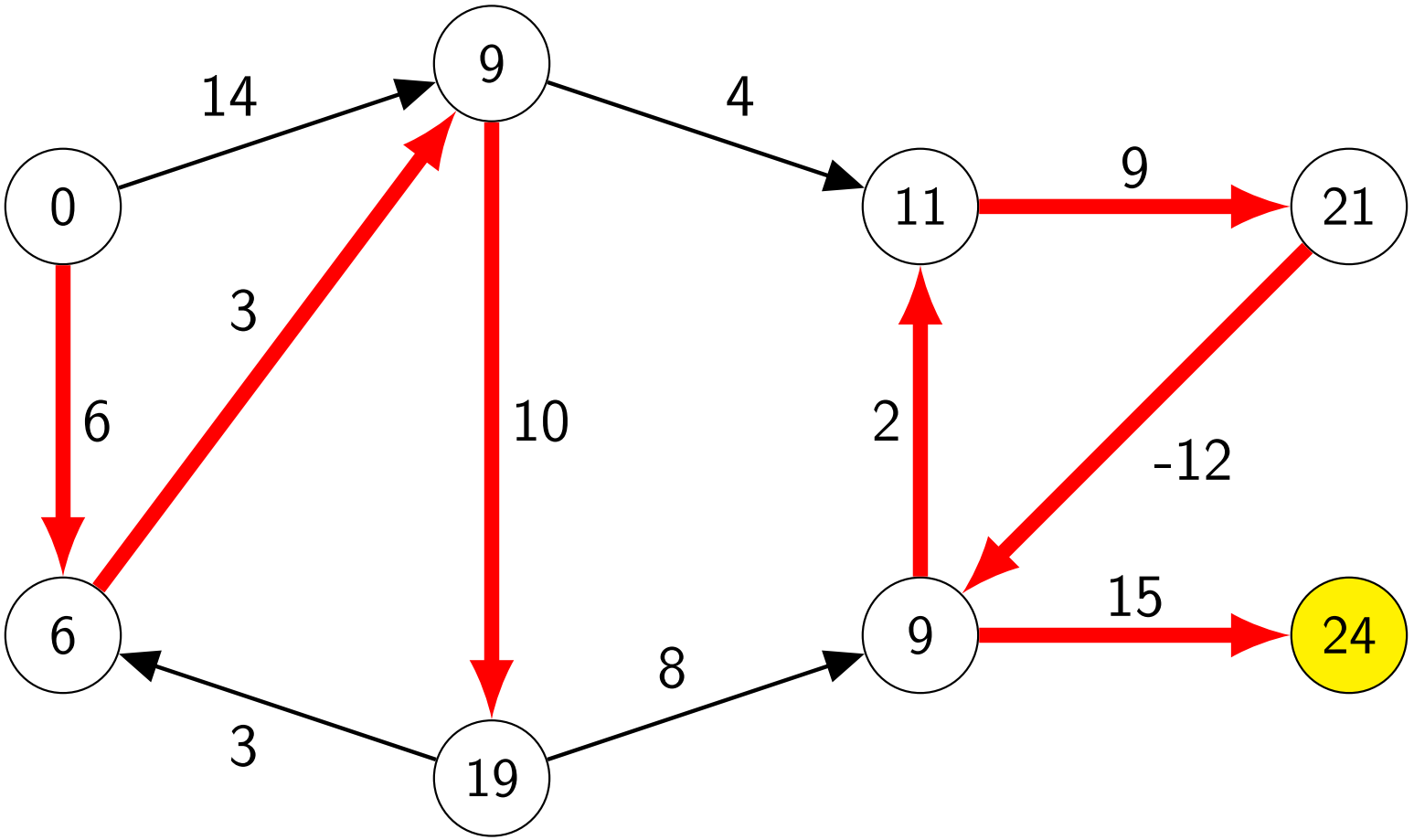
Bellman-Ford – Beispiel



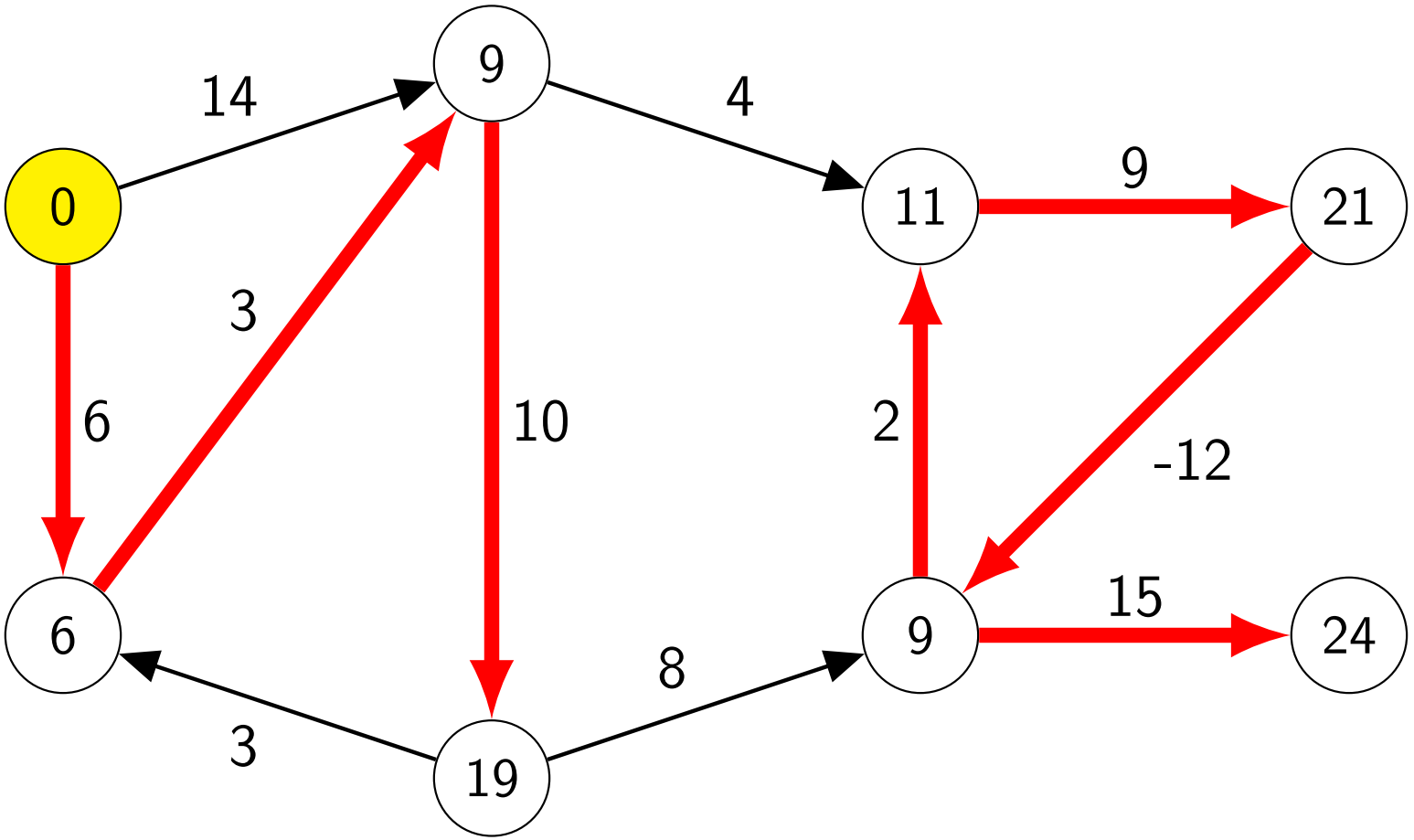
Bellman-Ford – Beispiel



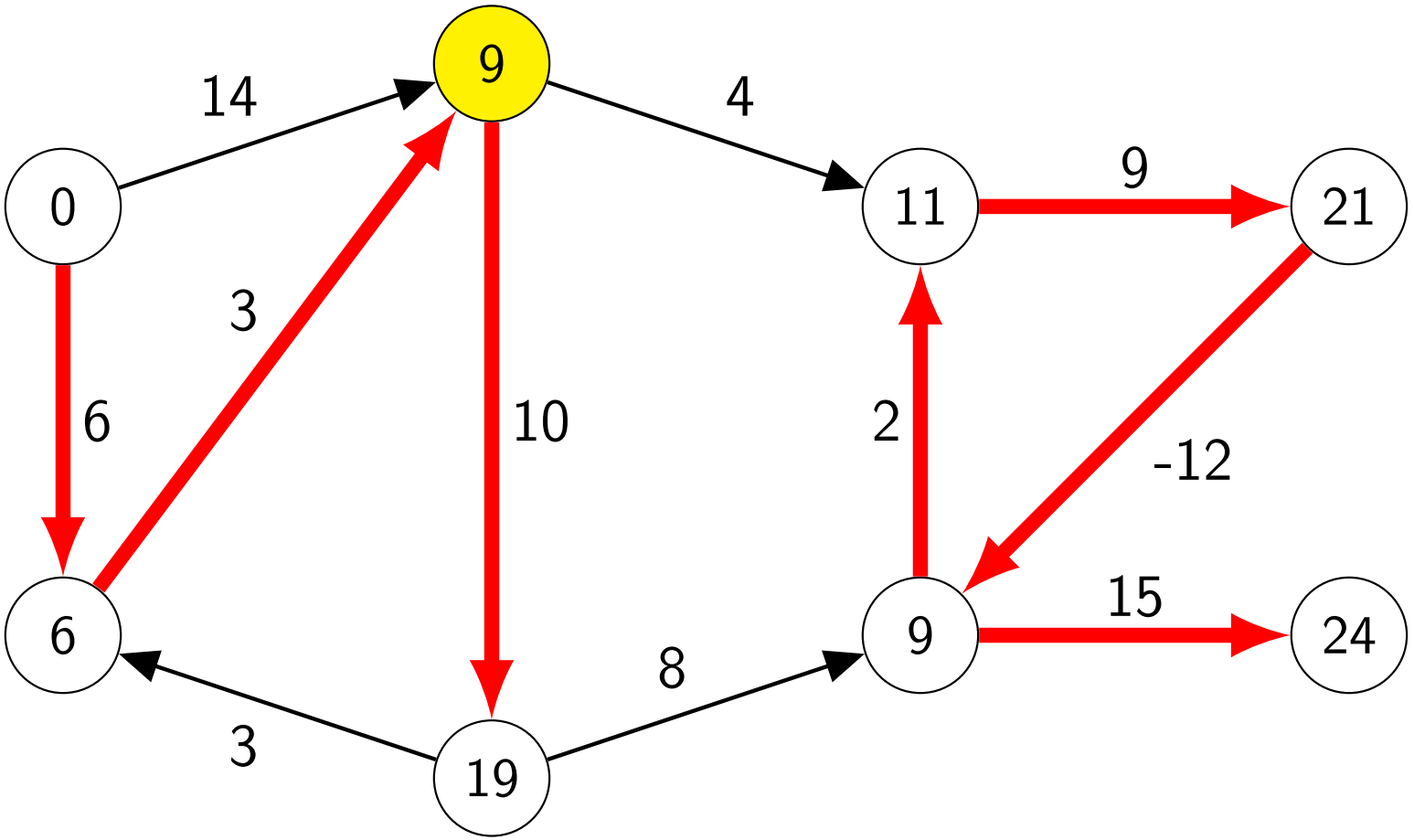
Bellman-Ford – Beispiel



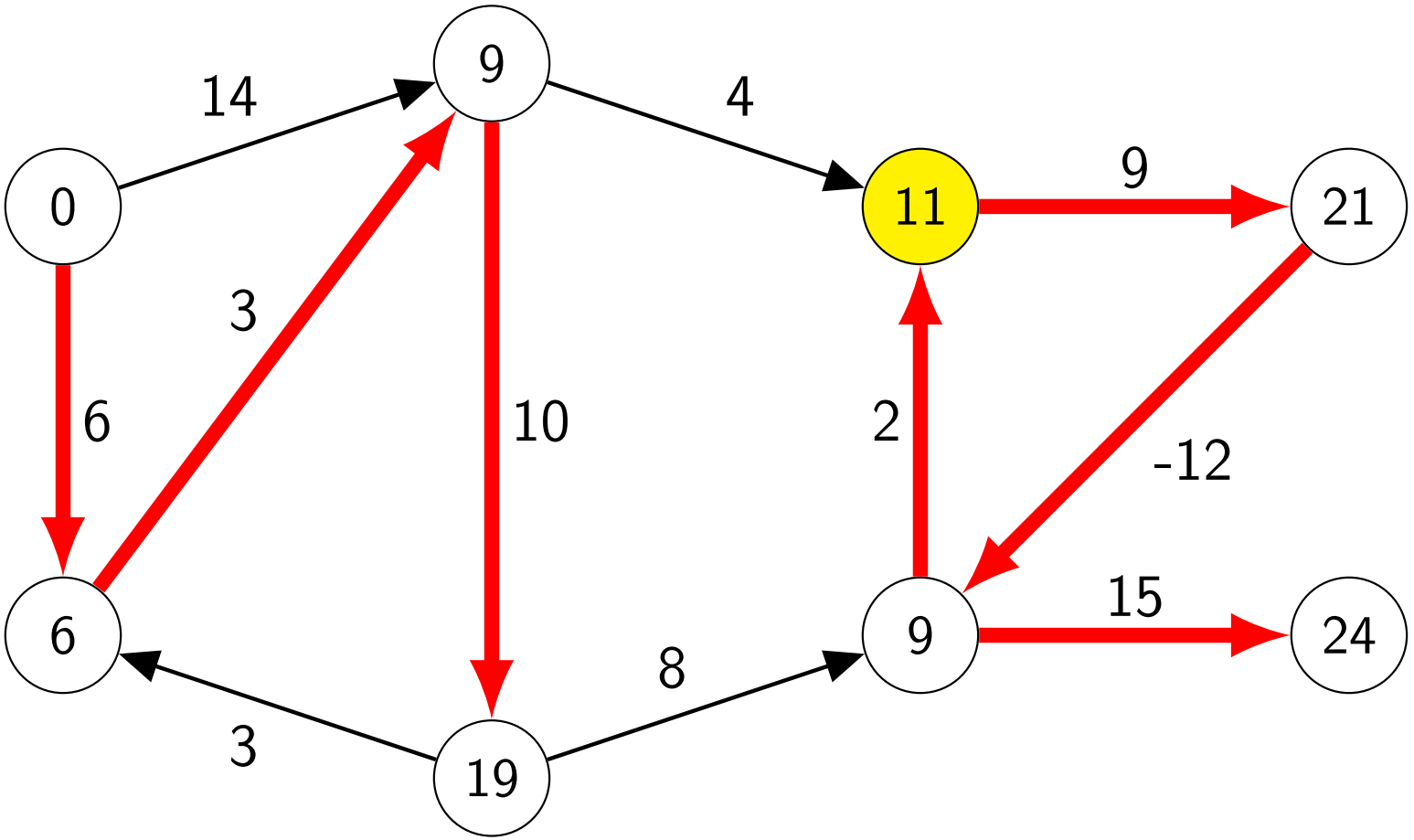
Bellman-Ford – Beispiel



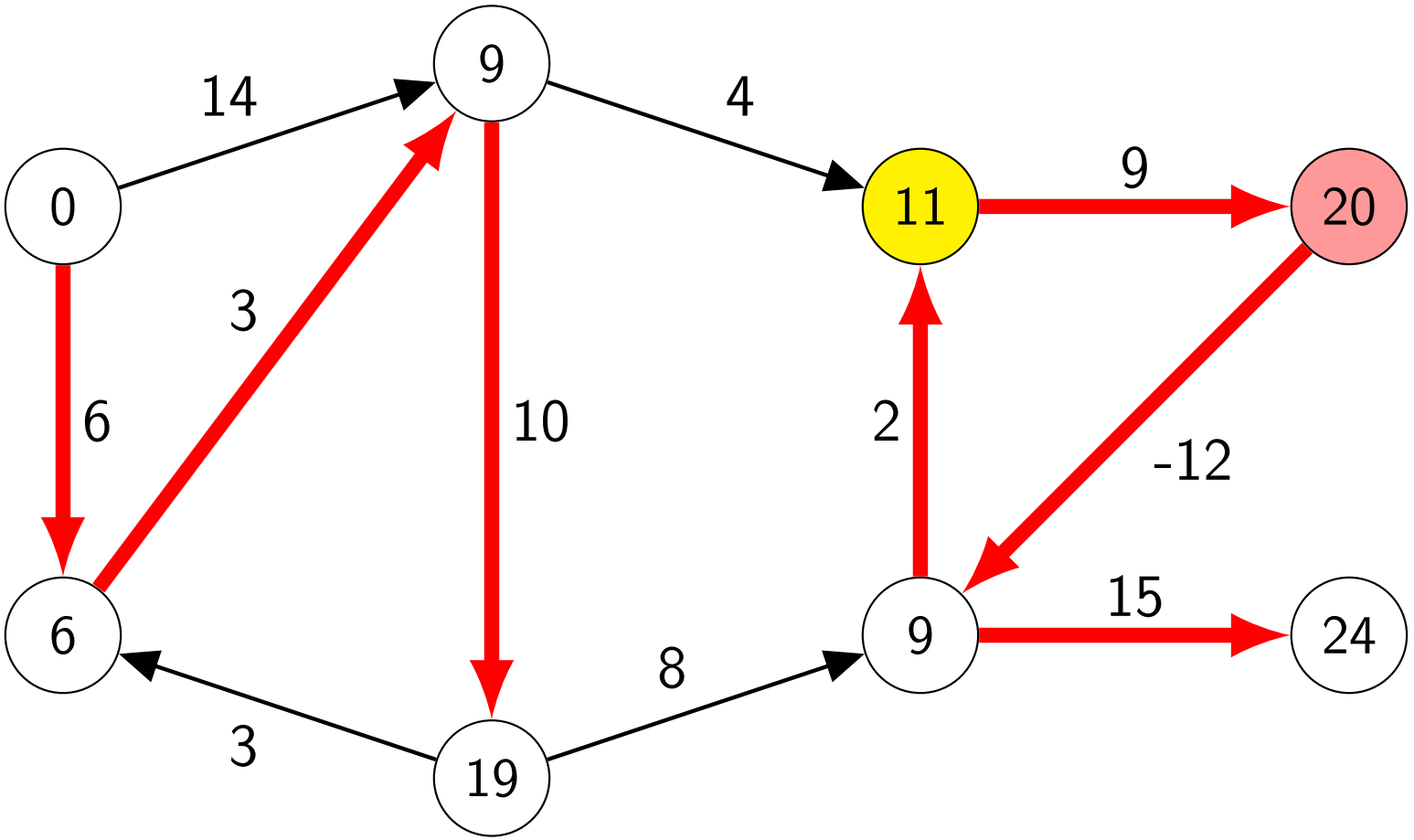
Bellman-Ford – Beispiel



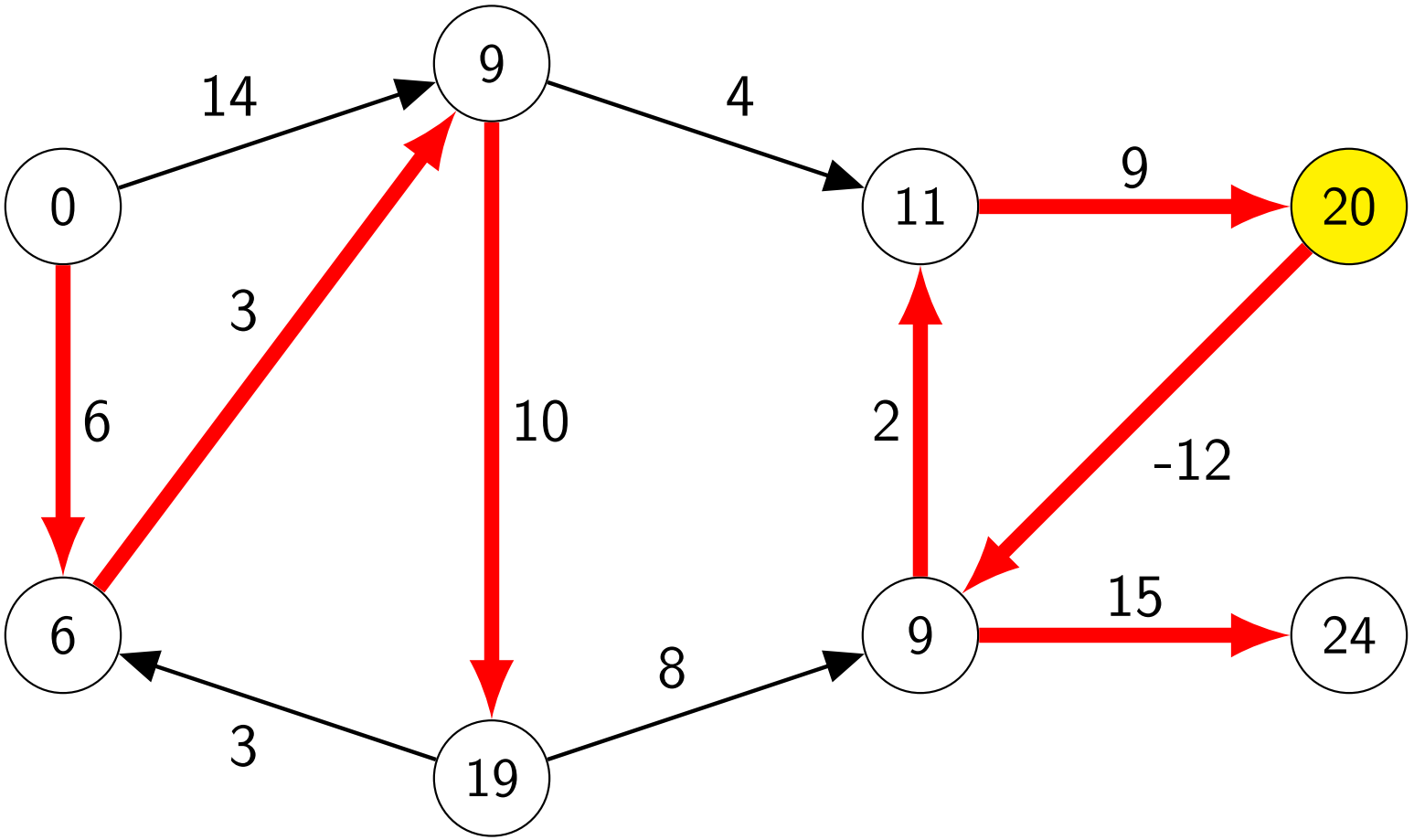
Bellman-Ford – Beispiel



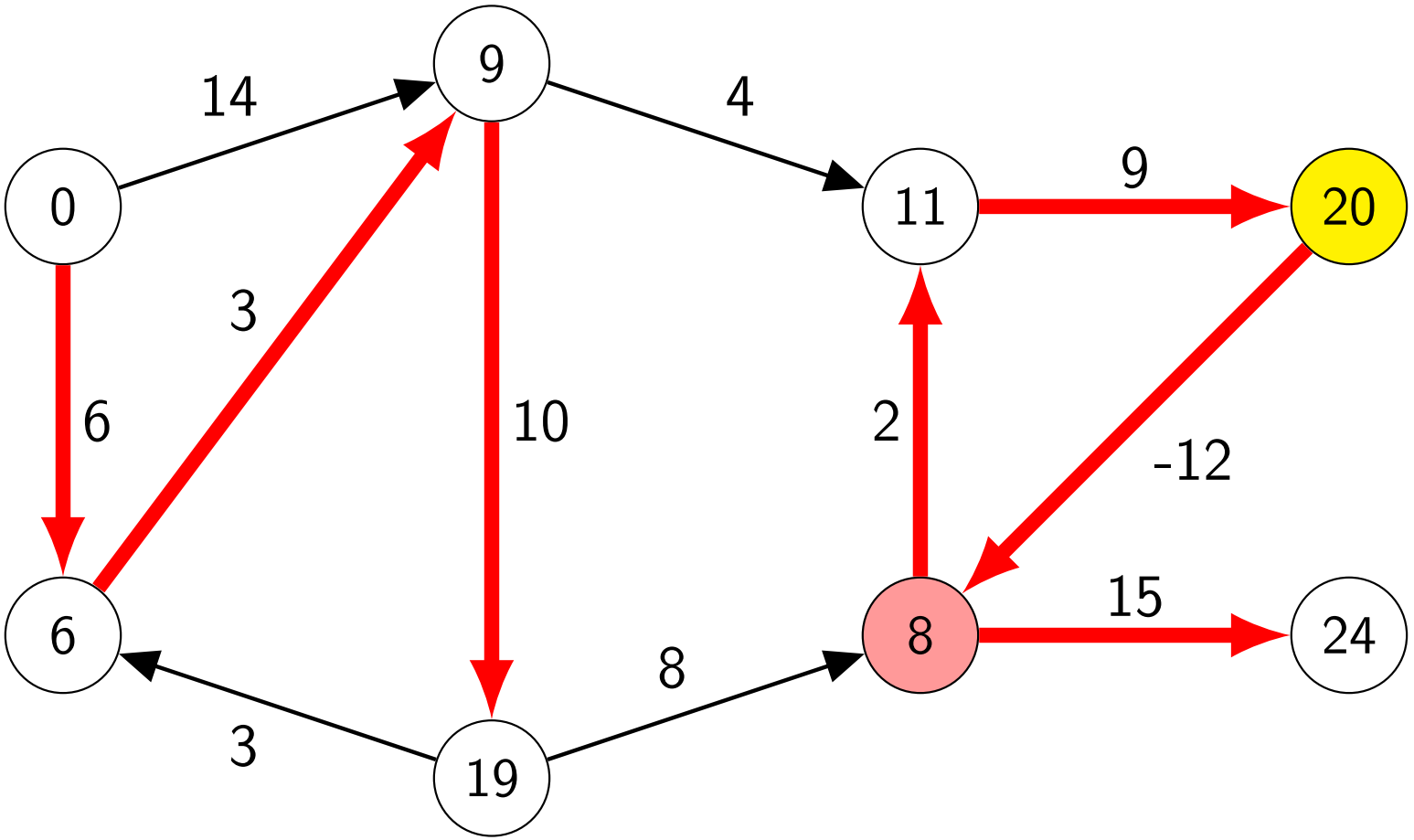
Bellman-Ford – Beispiel



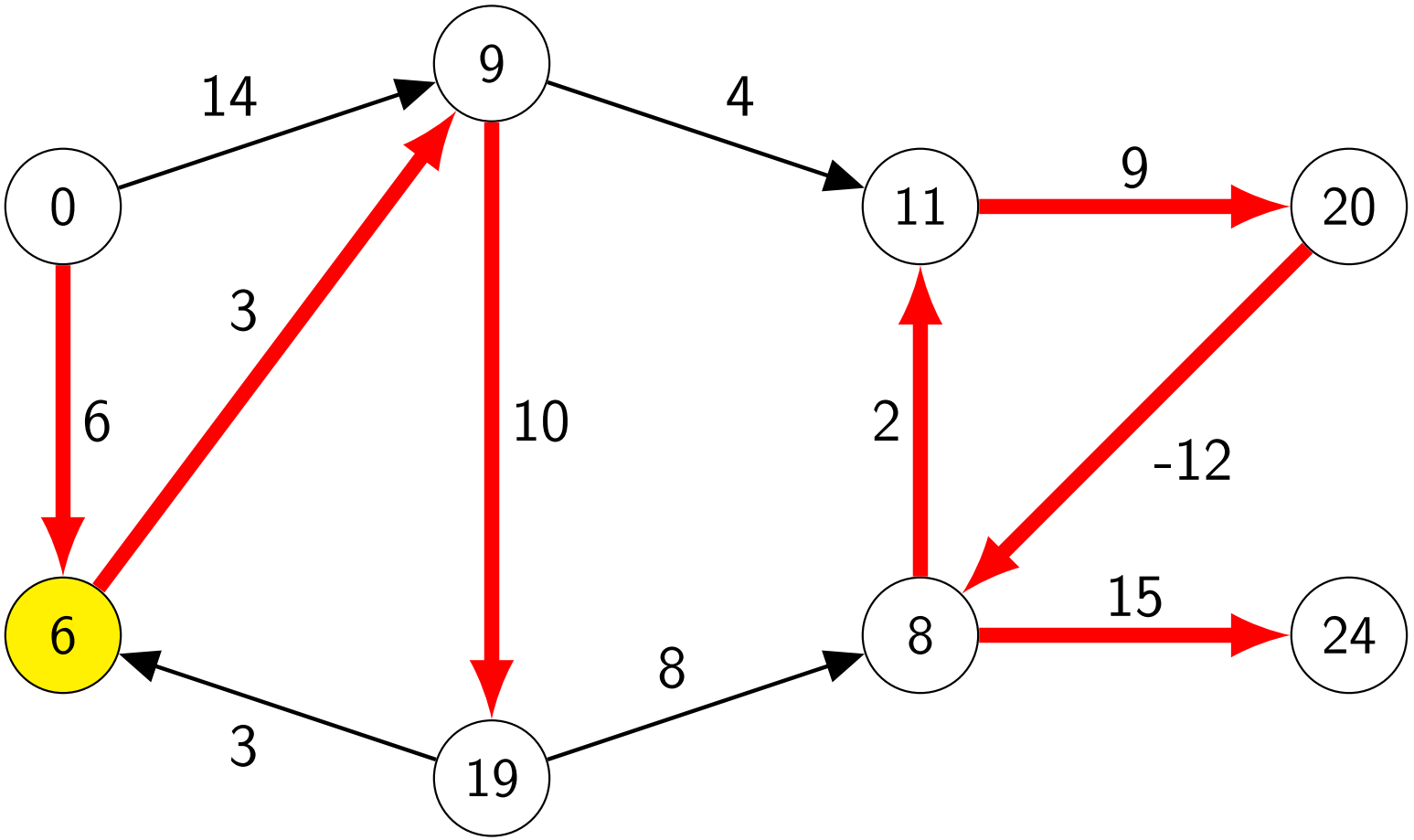
Bellman-Ford – Beispiel



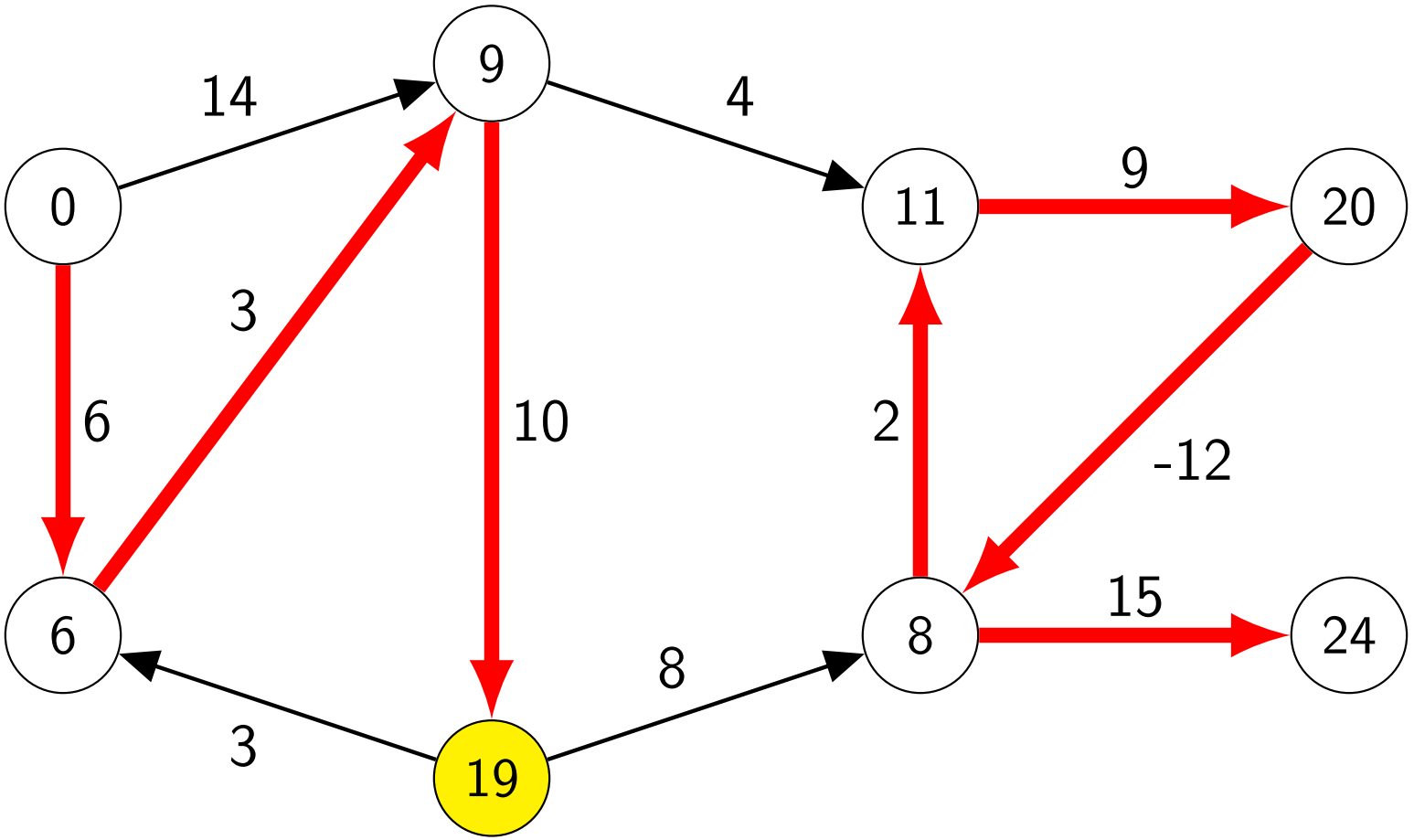
Bellman-Ford – Beispiel



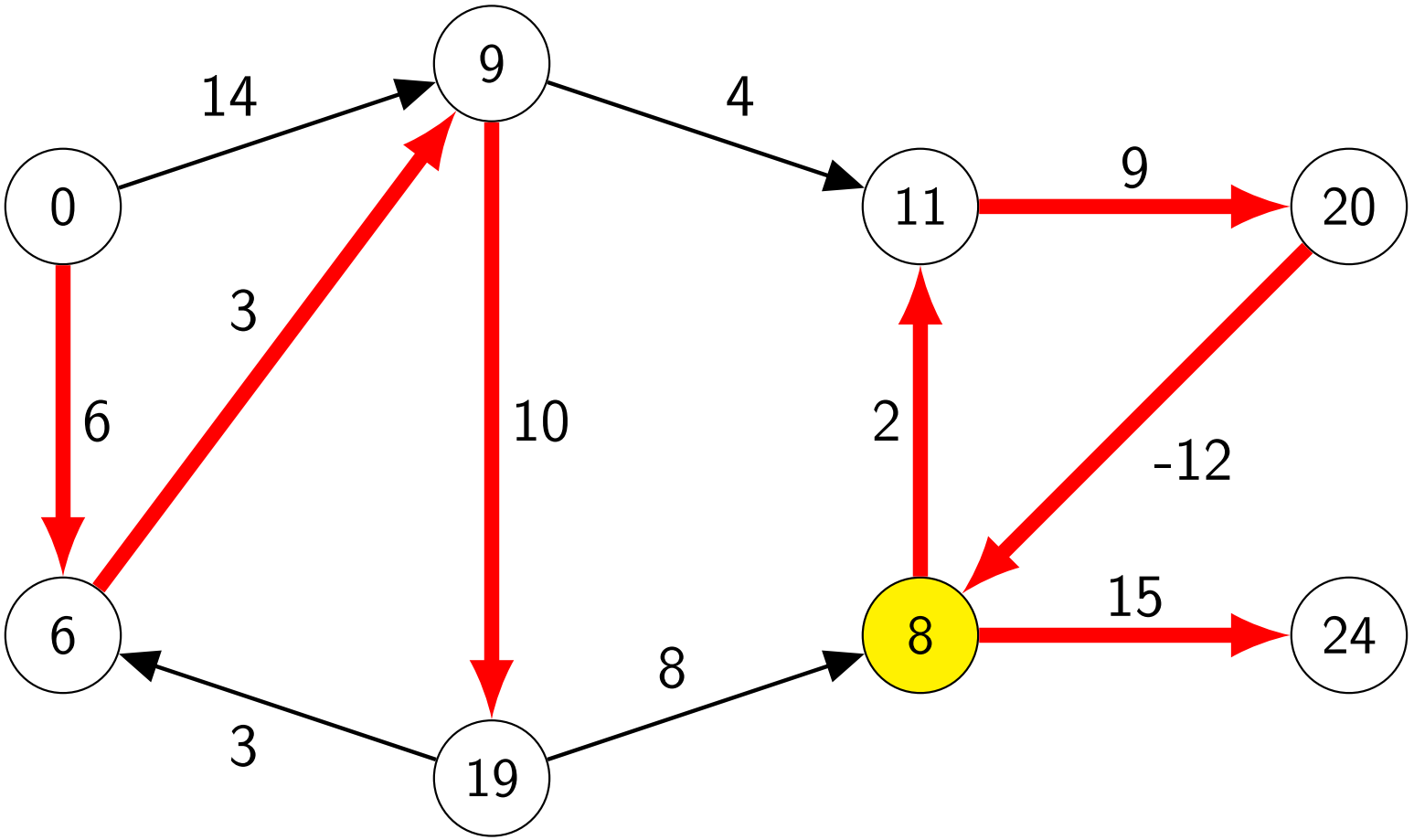
Bellman-Ford – Beispiel



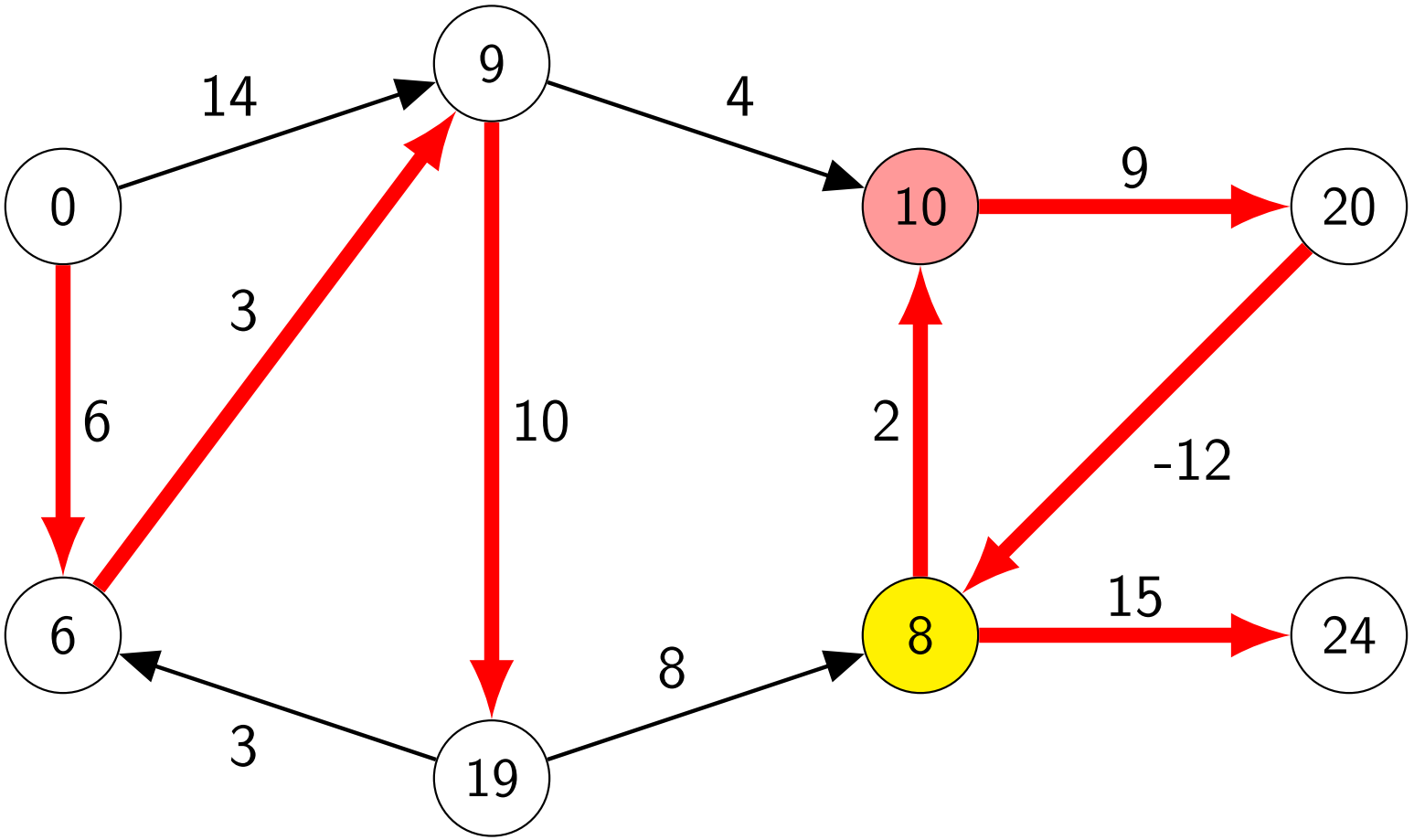
Bellman-Ford – Beispiel



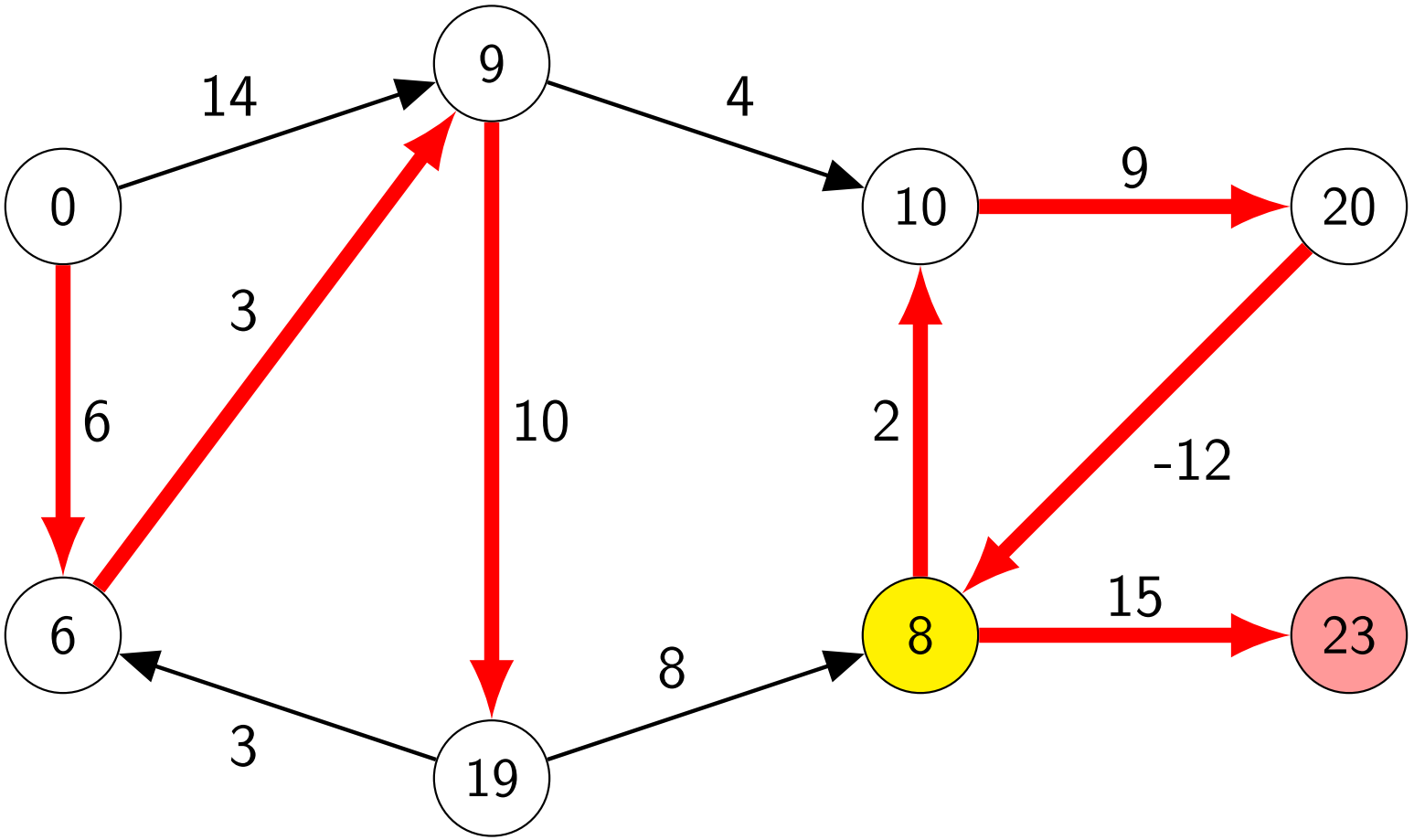
Bellman-Ford – Beispiel



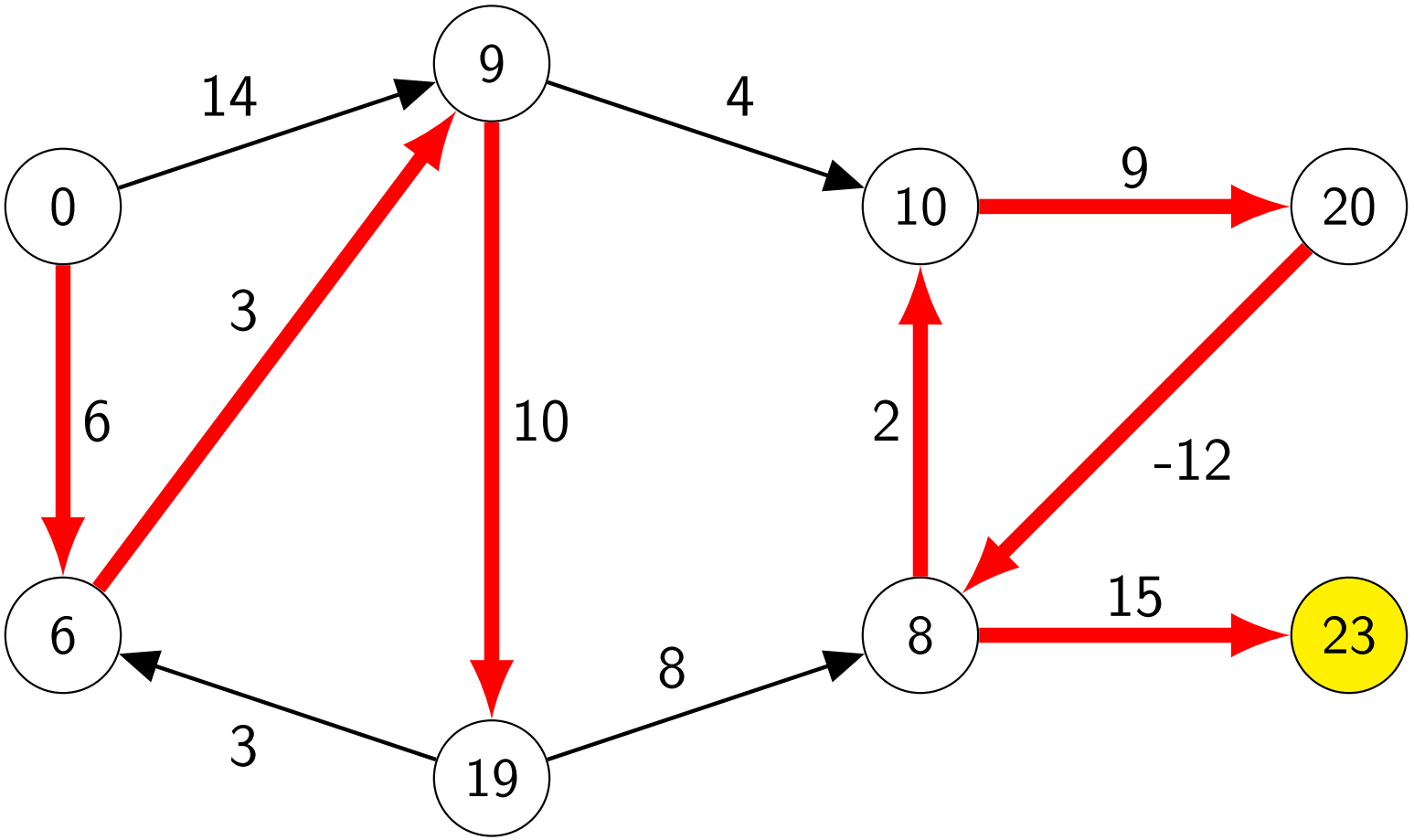
Bellman-Ford – Beispiel



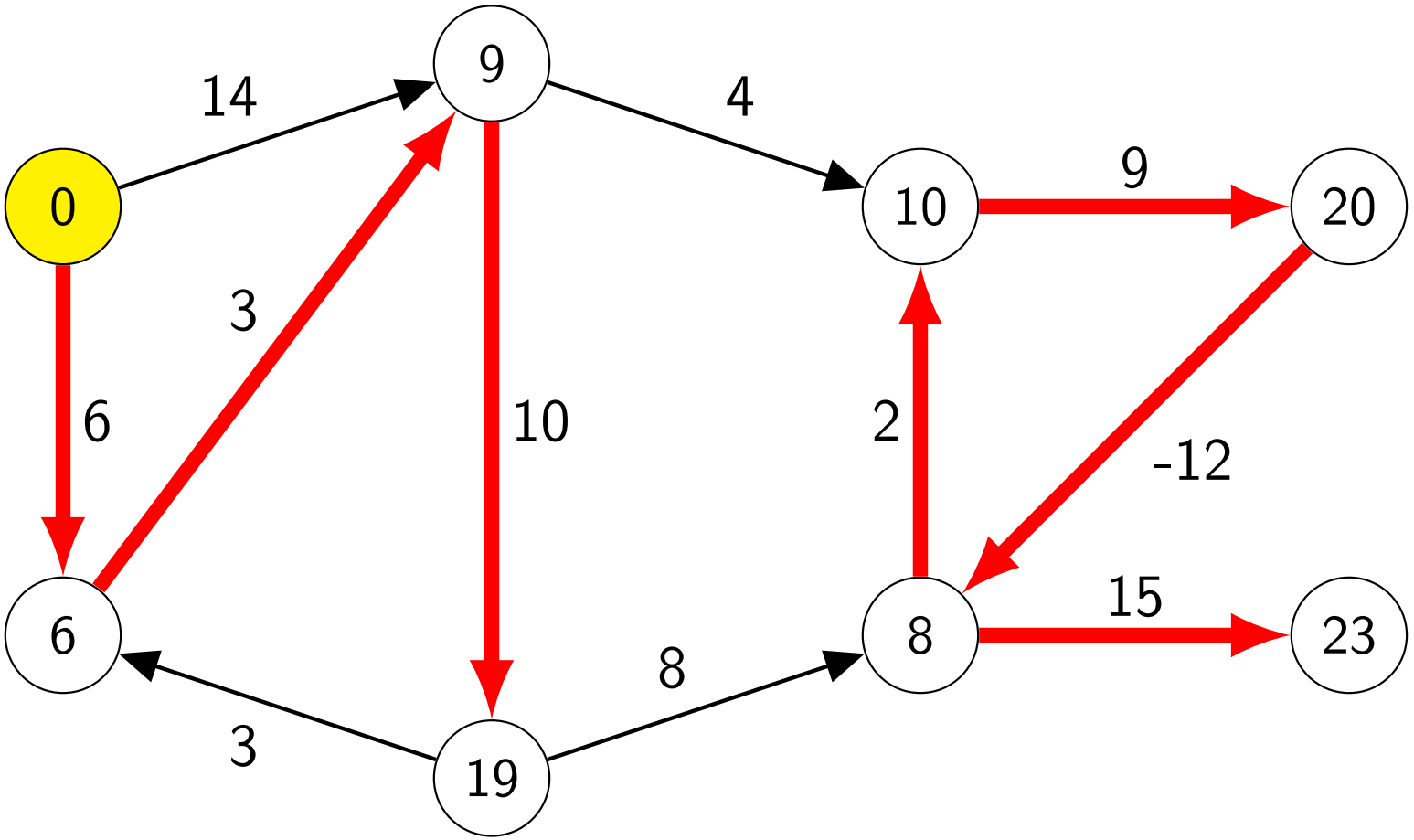
Bellman-Ford – Beispiel



Bellman-Ford – Beispiel



Bellman-Ford – Beispiel



Übersicht

Dijkstras Algorithmus

Annahme

Kantengewichte sind nicht-negativ, d. h. $W(v, w) \geqslant 0$ für all $(v, w) \in E$.

- Also: Kürzeste Pfade enthalten keine Zyklen.

Edsger Wybe Dijkstra (1930-2002), Turing Award 1972

Dijkstras Algorithmus: Übersicht

Wie beim Algorithmus von Prim ordnen wir die Knoten in drei Kategorien:

- Baumknoten: Knoten, die Teil vom bis jetzt konstruierten Baum sind.
- Randknoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.
- Ungesehene Knoten: Alle anderen Knoten.

Dijkstras Algorithmus: Übersicht

Wie beim Algorithmus von Prim ordnen wir die Knoten in drei Kategorien:

Baumknoten: Knoten, die Teil vom bis jetzt konstruierten Baum sind.

Randknoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Ungesehene Knoten: Alle anderen Knoten.

Grundkonzept:

- Kein Knoten außerhalb des Baumes hat einen kürzeren Pfad als die Knoten im Baum.

Dijkstras Algorithmus: Übersicht

Wie beim Algorithmus von Prim ordnen wir die Knoten in drei Kategorien:

Baumknoten: Knoten, die Teil vom bis jetzt konstruierten Baum sind.

Randknoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Ungesehene Knoten: Alle anderen Knoten.

Grundkonzept:

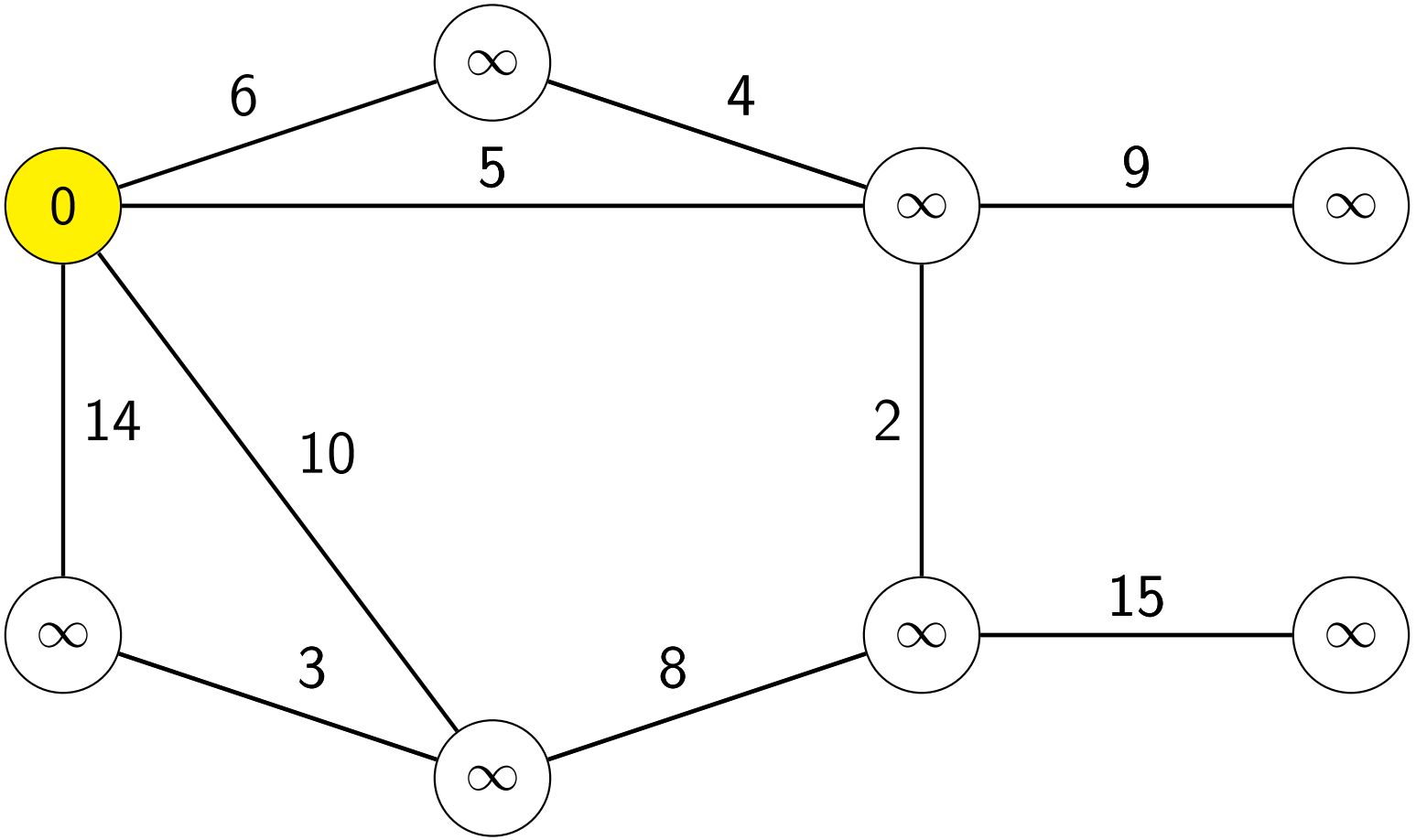
- Kein Knoten außerhalb des Baumes hat einen kürzeren Pfad als die Knoten im Baum.
- Jedem Knoten $\mathtt{v}$ ist ein Wert `dist[v]` zugeordnet. Dieser Wert ist:
  - für Baumknoten $\mathtt{dist}[\mathtt{v}] = \delta(\mathtt{s}, \mathtt{v})$;
  - für Randknoten das Minimum der Gewichte aller Pfade vom Startknoten zu $\mathtt{v}$, wobei die letzte Kante im Schnitt liegt;
  - für ungesehenen Knoten `+inf`.

Dijkstras Kürzeste-Wege-Algorithmus – Grundgerüst

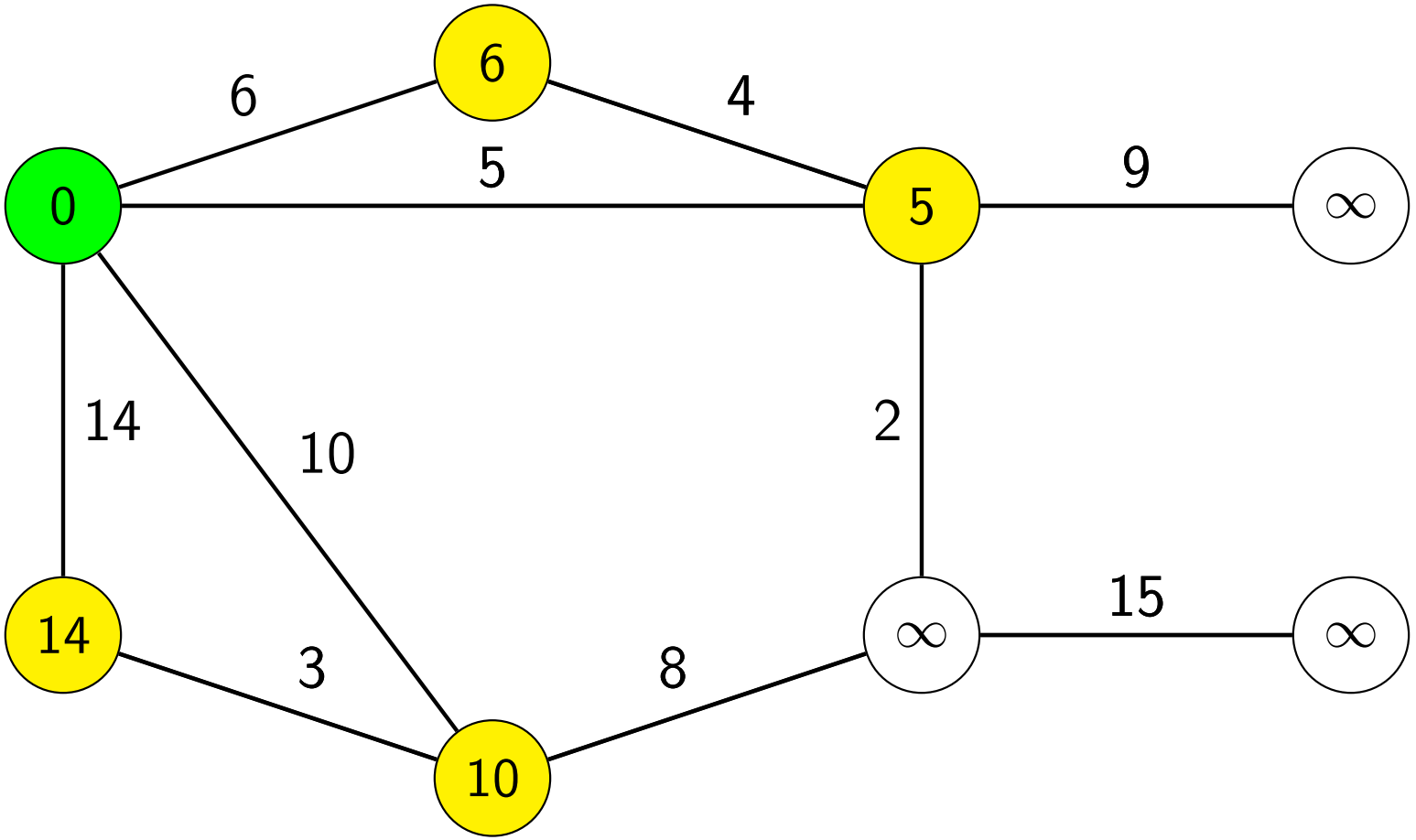
```
1  // ungerichteter Graph G mit n Knoten
2  void dijkstraSP(Graph G, int n) {
3    initialisiere alle Knoten als ungesehen (WHITE);
4    markiere s als Baum (BLACK) und setze d(s,s) = 0;
5    reklassifiziere alle zu s adjazenten Knoten als Rand (GRAY);
6    while (es gibt Randknoten) {
7      wähle von allen Kanten zwischen einem Baumknoten t und
8        einem Randknoten v mit minimalem d(s,t) + W(t,v);
9      reklassifiziere v als Baum (BLACK);
10     füge Kante (t,v) zum Baum hinzu;
11     setze d(s,v) = d(s,t) + W(t,v);
12     reklassifiziere alle zu v adjazenten ungesehenen Knoten
13       mit Rand (GRAY);
14   }
15 }
```

Unterschiede zu Prim's Algorithmus sind in rot gekennzeichnet.

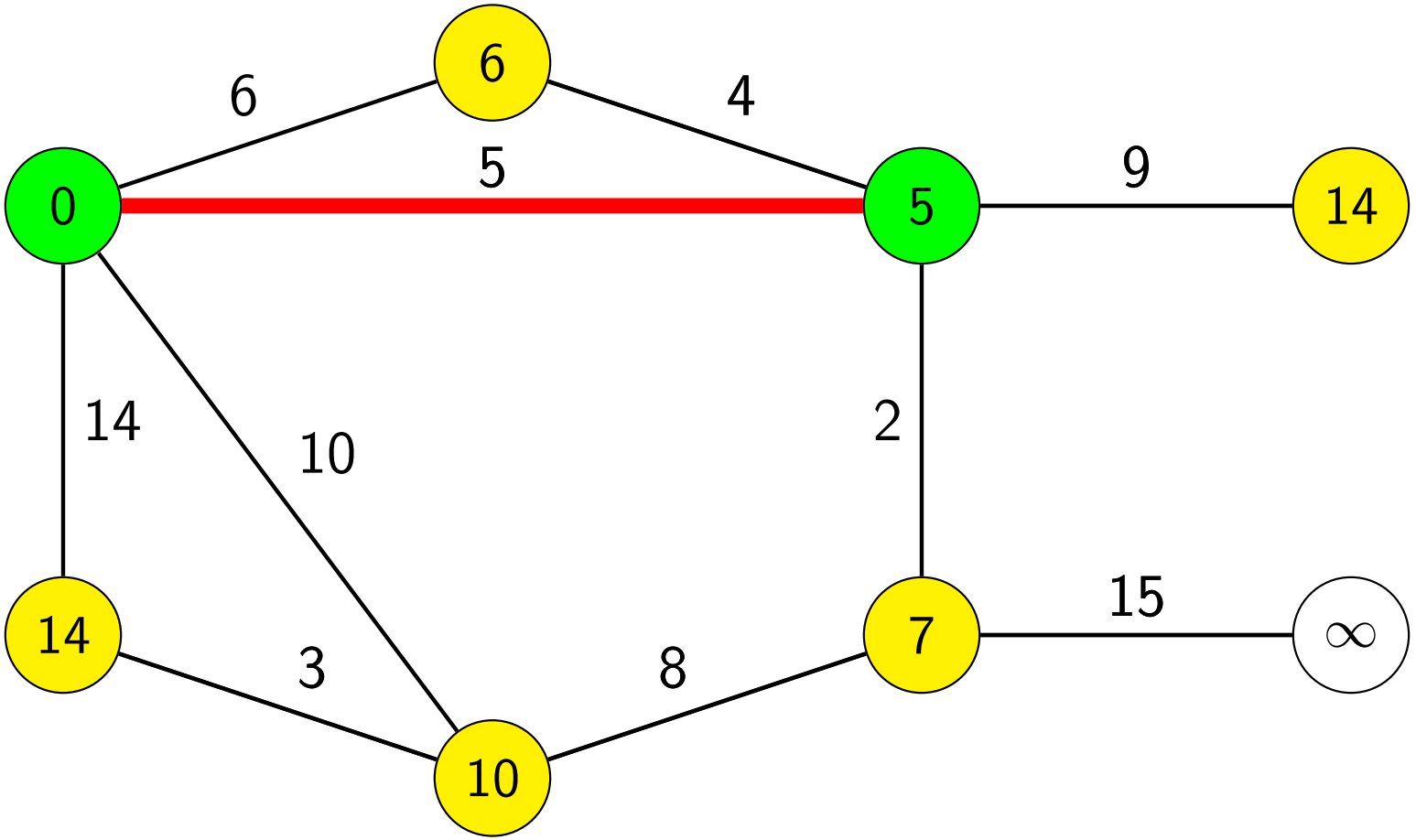
Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



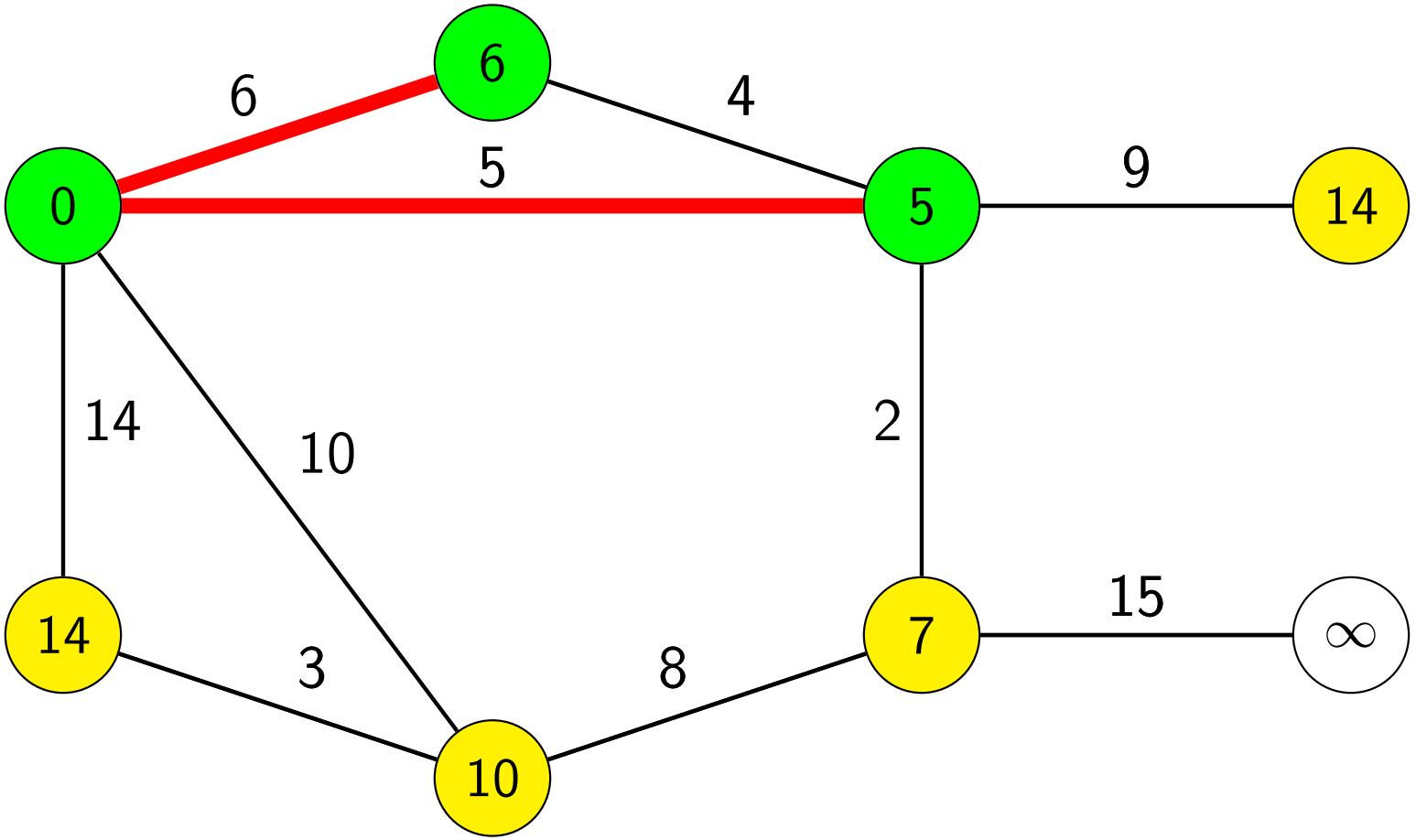
Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



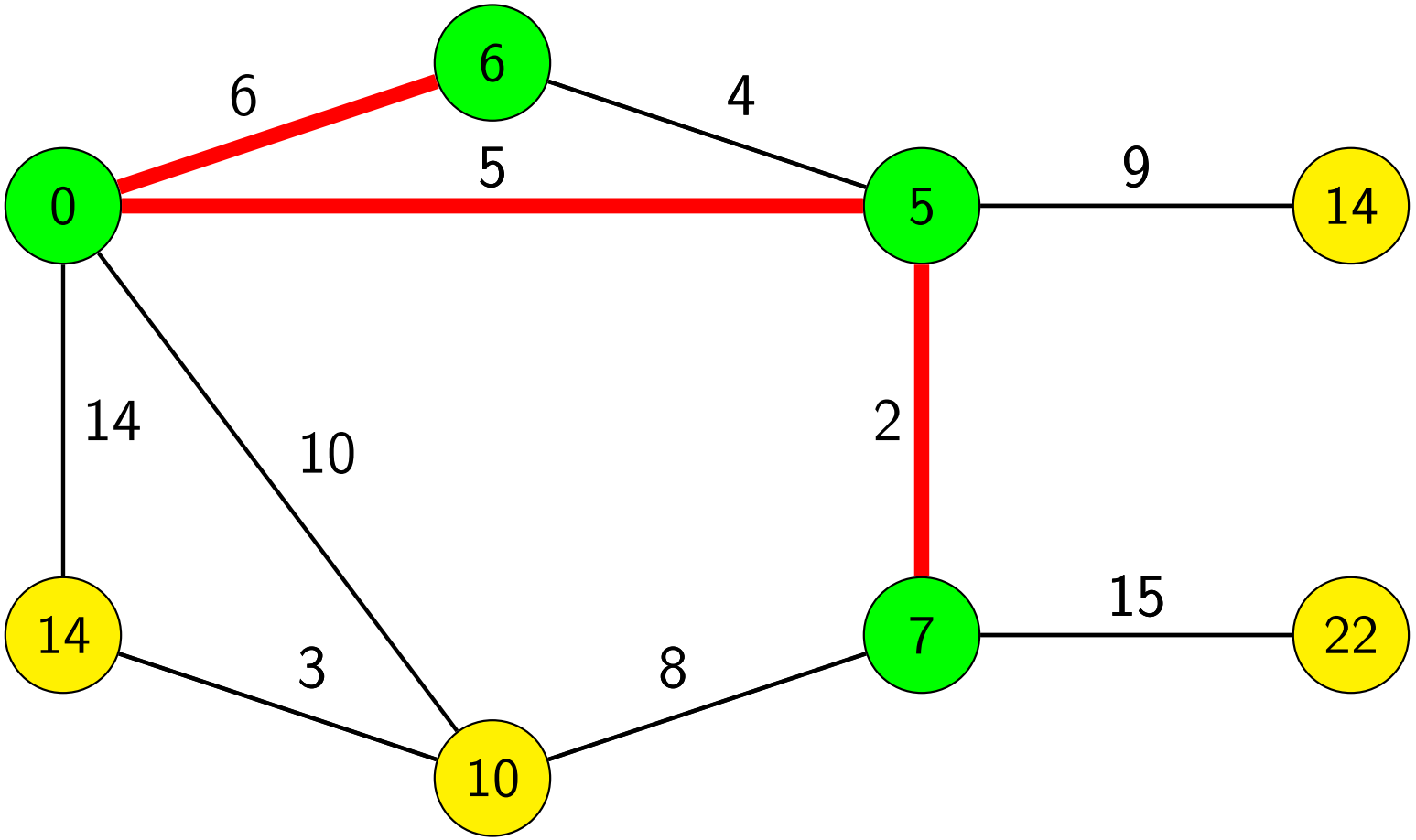
Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



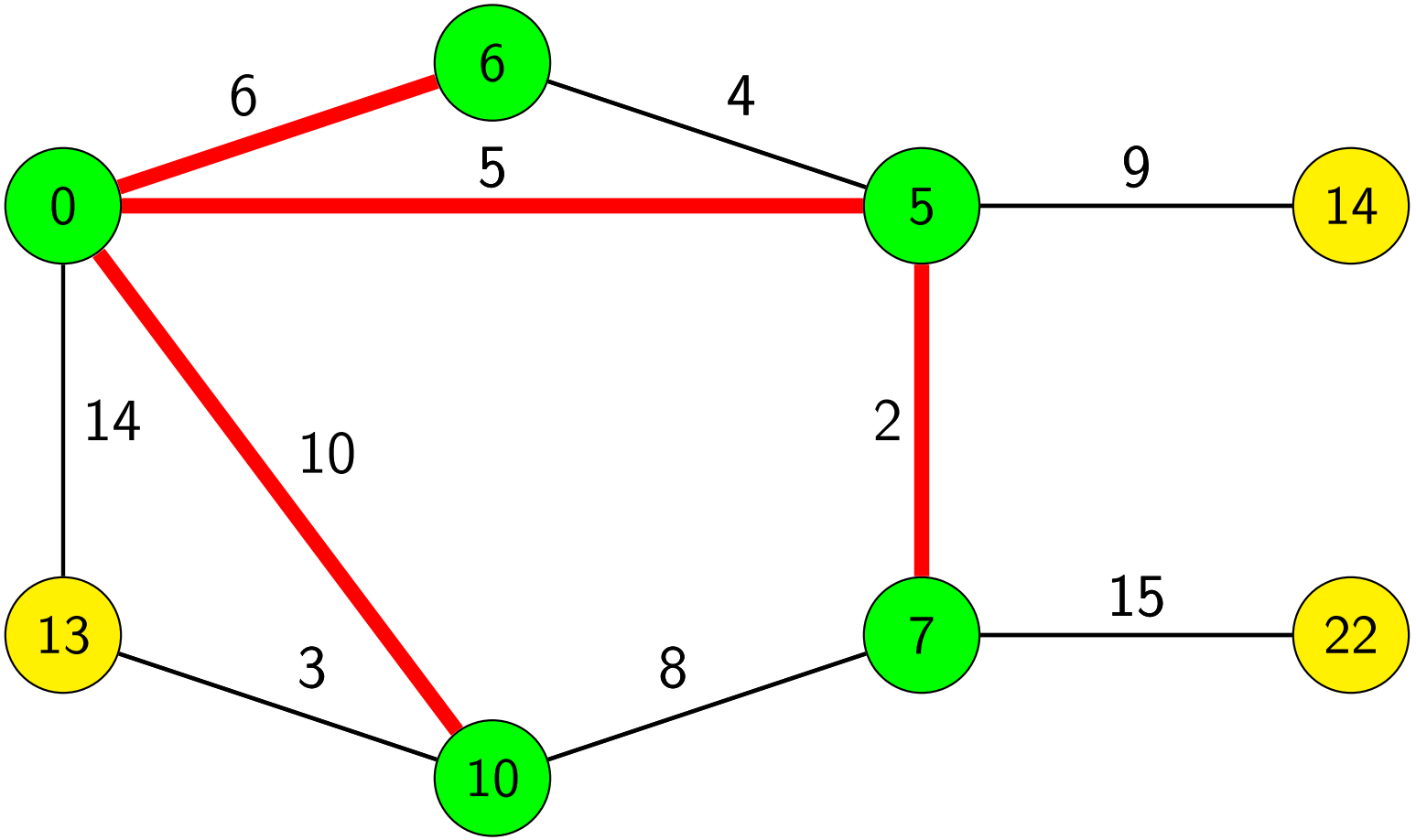
Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



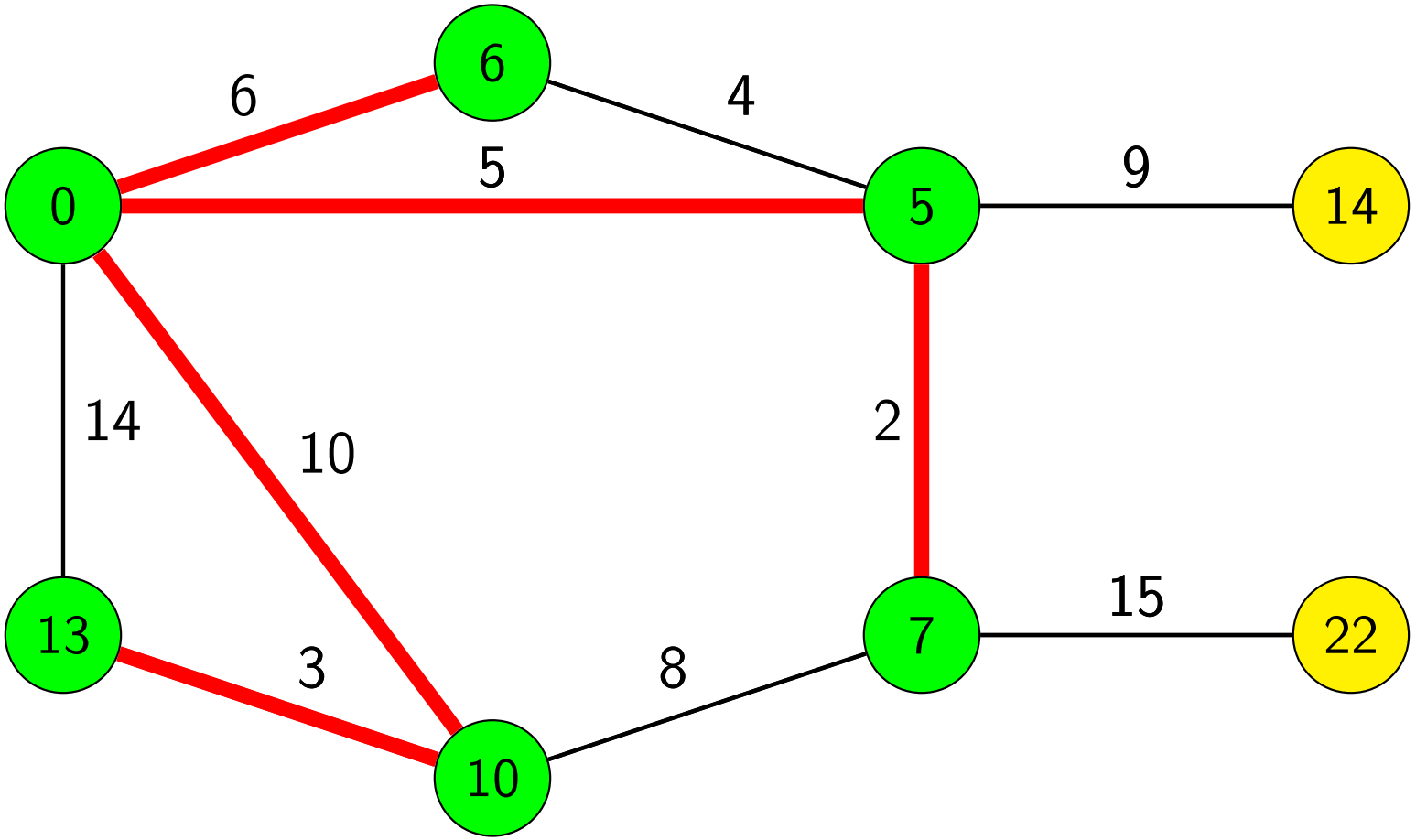
Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



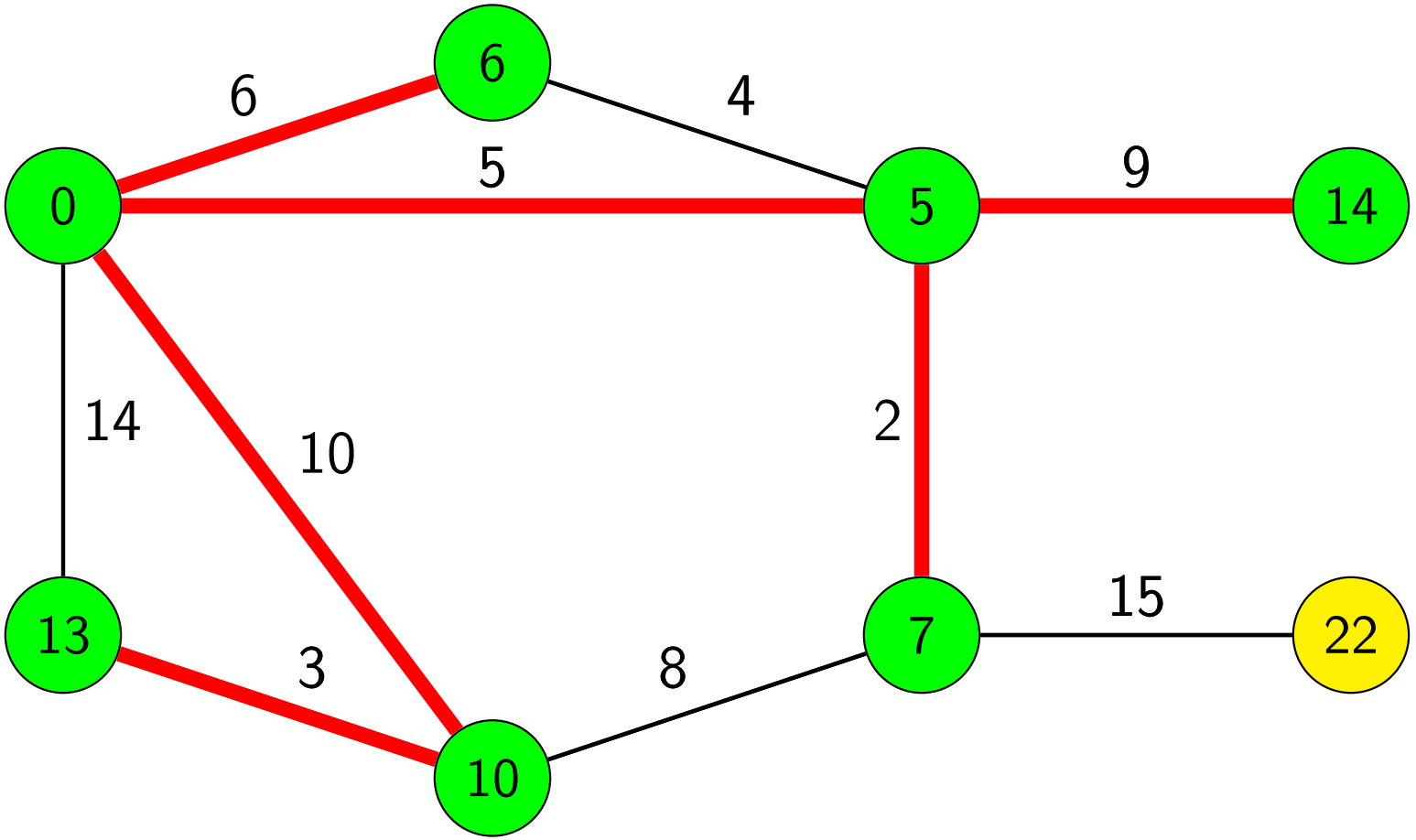
Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



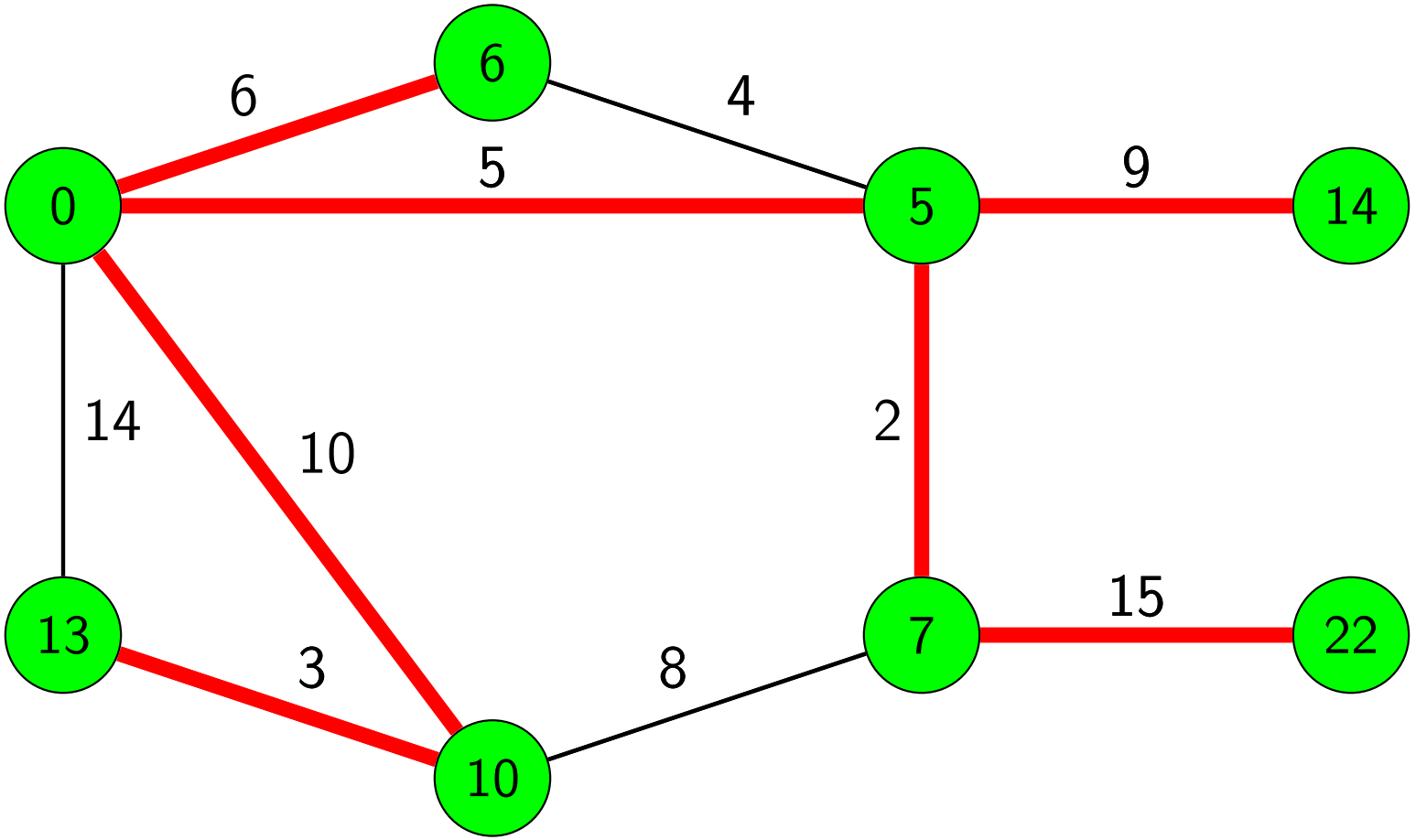
Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



Dijkstras Algorithmus: Korrektheit

Theorem

Sei B der schon konstruierte (schwarze) Teilbaum. Es gilt:

1. `dist[v]` $\geqslant \delta(s, v)$ für alle Knoten $v \in V$;

Dijkstras Algorithmus: Korrektheit

Theorem

Sei B der schon konstruierte (schwarze) Teilbaum. Es gilt:

1. `dist[v]` $\geqslant \delta(s, v)$ für alle Knoten $v \in V$;
2. `dist[v]` $= \delta(s, v)$ für alle Baumknoten $v \in B$;

Dijkstras Algorithmus: Korrektheit

Theorem

Sei B der schon konstruierte (schwarze) Teilbaum. Es gilt:

1. $\texttt{dist[v]} \geqslant \delta(s, v)$ für alle Knoten $v \in V$;
2. $\texttt{dist[v]} = \delta(s, v)$ für alle Baumknoten $v \in B$;
3. $\texttt{dist[v]} = \min(\{\infty\} \cup \{\delta(s, x) + W(x, v) \mid x \in B \textit{ und } (x, v) \in E\})$ für alle Nicht-Baumknoten $v \in V \setminus B$;

Dijkstras Algorithmus: Korrektheit

Theorem

Sei B der schon konstruierte (schwarze) Teilbaum. Es gilt:

1. $\texttt{dist[v]} \geqslant \delta(s, v)$ für alle Knoten $v \in V$;
2. $\texttt{dist[v]} = \delta(s, v)$ für alle Baumknoten $v \in B$;
3. $\texttt{dist[v]} = \min(\{\infty\} \cup \{\delta(s, x) + W(x, v) \mid x \in B \textit{ und } (x, v) \in E\})$ für alle Nicht-Baumknoten $v \in V \setminus B$;
4. $\delta(s, v) \leqslant \delta(s, u)$ für alle Baumknoten $v \in B$ und $u \in V \setminus B$.

Dijkstras Algorithmus: Korrektheit

1. $\texttt{dist}[\texttt{v}] \geqslant \delta(s, v)$ für alle Knoten $v \in V$;
2. $\texttt{dist}[\texttt{v}] = \delta(s, v)$ für alle Baumknoten $v \in B$;
3. $\texttt{dist}[\texttt{v}] = \min(\{\infty\} \cup \{\delta(s, x) + W(x, v) \mid x \in B \textit{ und } (x, v) \in E\})$ für $v \in V \setminus B$;
4. $\delta(s, v) \leqslant \delta(s, u)$ für alle Baumknoten $v \in B$ und $u \in V \setminus B$.

Beweis.

1., 3. und 4. sind (ziemlich) trivial. Betrachte 2.

Dijkstras Algorithmus: Korrektheit

1. $\texttt{dist}[\texttt{v}] \geqslant \delta(s, v)$ für alle Knoten $v \in V$;
2. $\texttt{dist}[\texttt{v}] = \delta(s, v)$ für alle Baumknoten $v \in B$;
3. $\texttt{dist}[\texttt{v}] = \min(\{\infty\} \cup \{\delta(s, x) + W(x, v) \mid x \in B \textit{ und } (x, v) \in E\})$ für $v \in V \setminus B$;
4. $\delta(s, v) \leqslant \delta(s, u)$ für alle Baumknoten $v \in B$ und $u \in V \setminus B$.

Beweis.

1., 3. und 4. sind (ziemlich) trivial. Betrachte 2. Per Induktion über die Anzahl der Baumknoten. Beweisskizze:

Dijkstras Algorithmus: Korrektheit

1. $\texttt{dist}[\texttt{v}] \geqslant \delta(s, v)$ für alle Knoten $v \in V$;
2. $\texttt{dist}[\texttt{v}] = \delta(s, v)$ für alle Baumknoten $v \in B$;
3. $\texttt{dist}[\texttt{v}] = \min(\{\infty\} \cup \{\delta(s, x) + W(x, v) \mid x \in B \textit{ und } (x, v) \in E\})$ für $v \in V \setminus B$;
4. $\delta(s, v) \leqslant \delta(s, u)$ für alle Baumknoten $v \in B$ und $u \in V \setminus B$.

Beweis.

1., 3. und 4. sind (ziemlich) trivial. Betrachte 2. Per Induktion über die Anzahl der Baumknoten. Beweisskizze:

- Basis: B ist leer und $B = \{s\}$ (nach der ersten Iteration): trivial.

Dijkstras Algorithmus: Korrektheit

1. $\texttt{dist}[\texttt{v}] \geqslant \delta(s, v)$ für alle Knoten $v \in V$;
2. $\texttt{dist}[\texttt{v}] = \delta(s, v)$ für alle Baumknoten $v \in B$;
3. $\texttt{dist}[\texttt{v}] = \min(\{\infty\} \cup \{\delta(s, x) + W(x, v) \mid x \in B \textit{ und } (x, v) \in E\})$ für $v \in V \setminus B$;
4. $\delta(s, v) \leqslant \delta(s, u)$ für alle Baumknoten $v \in B$ und $u \in V \setminus B$.

Beweis.

1., 3. und 4. sind (ziemlich) trivial. Betrachte 2. Per Induktion über die Anzahl der Baumknoten. Beweisskizze:

- Basis: B ist leer und $B = \{s\}$ (nach der ersten Iteration): trivial.
- Ind. Schritt: nehme an, das Theorem gilt für $|B| = k > 0$.

Dijkstras Algorithmus: Korrektheit

1. $\texttt{dist}[\texttt{v}] \geqslant \delta(s, v)$ für alle Knoten $v \in V$;
2. $\texttt{dist}[\texttt{v}] = \delta(s, v)$ für alle Baumknoten $v \in B$;
3. $\texttt{dist}[\texttt{v}] = \min(\{\infty\} \cup \{\delta(s, x) + W(x, v) \mid x \in B \textit{ und } (x, v) \in E\})$ für $v \in V \setminus B$;
4. $\delta(s, v) \leqslant \delta(s, u)$ für alle Baumknoten $v \in B$ und $u \in V \setminus B$.

Beweis.

1., 3. und 4. sind (ziemlich) trivial. Betrachte 2. Per Induktion über die Anzahl der Baumknoten. Beweisskizze:

- Basis: B ist leer und $B = \{s\}$ (nach der ersten Iteration): trivial.
- Ind. Schritt: nehme an, das Theorem gilt für $|B| = k > 0$. Sei v der Knoten der als nächste zum Baum B hinzugefügt wird.

Dijkstras Algorithmus: Korrektheit

1. $\texttt{dist[v]} \geqslant \delta(s, v)$ für alle Knoten $v \in V$;
2. $\texttt{dist[v]} = \delta(s, v)$ für alle Baumknoten $v \in B$;
3. $\texttt{dist[v]} = \min(\{\infty\} \cup \{\delta(s, x) + W(x, v) \mid x \in B \text{ und } (x, v) \in E\})$ für $v \in V \setminus B$;
4. $\delta(s, v) \leqslant \delta(s, u)$ für alle Baumknoten $v \in B$ und $u \in V \setminus B$.

Beweis.

1., 3. und 4. sind (ziemlich) trivial. Betrachte 2. Per Induktion über die Anzahl der Baumknoten. Beweisskizze:

- Basis: B ist leer und $B = \{s\}$ (nach der ersten Iteration): trivial.
- Ind. Schritt: nehme an, das Theorem gilt für $|B| = k > 0$. Sei v der Knoten der als nächste zum Baum B hinzugefügt wird. Der Fall $\texttt{dist[v]} = +\texttt{inf}$ ist trivial.

Dijkstras Algorithmus: Korrektheit

1. $\texttt{dist[v]} \geqslant \delta(s, v)$ für alle Knoten $v \in V$;
2. $\texttt{dist[v]} = \delta(s, v)$ für alle Baumknoten $v \in B$;
3. $\texttt{dist[v]} = \min(\{\infty\} \cup \{\delta(s, x) + W(x, v) \mid x \in B \text{ und } (x, v) \in E\})$ für $v \in V \setminus B$;
4. $\delta(s, v) \leqslant \delta(s, u)$ für alle Baumknoten $v \in B$ und $u \in V \setminus B$.

Beweis.

1., 3. und 4. sind (ziemlich) trivial. Betrachte 2. Per Induktion über die Anzahl der Baumknoten. Beweisskizze:

- Basis: B ist leer und $B = \{s\}$ (nach der ersten Iteration): trivial.
- Ind. Schritt: nehme an, das Theorem gilt für $|B| = k > 0$. Sei v der Knoten der als nächste zum Baum B hinzugefügt wird. Der Fall $\texttt{dist[v]} = +\texttt{inf}$ ist trivial. Betrachte $\texttt{dist[v]} \neq +\texttt{inf}$.

Dijkstras Algorithmus: Korrektheit

1. $\texttt{dist[v]} \geqslant \delta(s, v)$ für alle Knoten $v \in V$;
2. $\texttt{dist[v]} = \delta(s, v)$ für alle Baumknoten $v \in B$;
3. $\texttt{dist[v]} = \min(\{\infty\} \cup \{\delta(s, x) + W(x, v) \mid x \in B \text{ und } (x, v) \in E\})$ für $v \in V \setminus B$;
4. $\delta(s, v) \leqslant \delta(s, u)$ für alle Baumknoten $v \in B$ und $u \in V \setminus B$.

Beweis.

1., 3. und 4. sind (ziemlich) trivial. Betrachte 2. Per Induktion über die Anzahl der Baumknoten. Beweisskizze:

- Basis: B ist leer und $B = \{s\}$ (nach der ersten Iteration): trivial.
- Ind. Schritt: nehme an, das Theorem gilt für $|B| = k > 0$. Sei v der Knoten der als nächste zum Baum B hinzugefügt wird. Der Fall $\texttt{dist[v]} = +\texttt{inf}$ ist trivial. Betrachte $\texttt{dist[v]} \neq +\texttt{inf}$. Dann gibt es eine Kante vom Baumknoten $x = \texttt{prev[v]} \in B$ zu $v \notin B$.

Dijkstras Algorithmus: Korrektheit

1. $\texttt{dist[v]} \geqslant \delta(s, v)$ für alle Knoten $v \in V$;
2. $\texttt{dist[v]} = \delta(s, v)$ für alle Baumknoten $v \in B$;
3. $\texttt{dist[v]} = \min(\{\infty\} \cup \{\delta(s, x) + W(x, v) \mid x \in B \text{ und } (x, v) \in E\})$ für $v \in V \setminus B$;
4. $\delta(s, v) \leqslant \delta(s, u)$ für alle Baumknoten $v \in B$ und $u \in V \setminus B$.

Beweis.

1., 3. und 4. sind (ziemlich) trivial. Betrachte 2. Per Induktion über die Anzahl der Baumknoten. Beweisskizze:

- Basis: B ist leer und $B = \{s\}$ (nach der ersten Iteration): trivial.
- Ind. Schritt: nehme an, das Theorem gilt für $|B| = k > 0$. Sei v der Knoten der als nächste zum Baum B hinzugefügt wird. Der Fall $\texttt{dist[v]} = +\texttt{inf}$ ist trivial. Betrachte $\texttt{dist[v]} \neq +\texttt{inf}$. Dann gibt es eine Kante vom Baumknoten $x = \texttt{prev[v]} \in B$ zu $v \notin B$. Aus 3. folgt, dass $\texttt{dist[v]}$ das Gewicht des kürzesten Pfades von s nach v über Baumknoten ist.

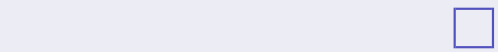
Dijkstras Algorithmus: Korrektheit

1. $\texttt{dist[v]} \geqslant \delta(s, v)$ für alle Knoten $v \in V$;
2. $\texttt{dist[v]} = \delta(s, v)$ für alle Baumknoten $v \in B$;
3. $\texttt{dist[v]} = \min(\{\infty\} \cup \{\delta(s, x) + W(x, v) \mid x \in B \text{ und } (x, v) \in E\})$ für $v \in V \setminus B$;
4. $\delta(s, v) \leqslant \delta(s, u)$ für alle Baumknoten $v \in B$ und $u \in V \setminus B$.

Beweis.

1., 3. und 4. sind (ziemlich) trivial. Betrachte 2. Per Induktion über die Anzahl der Baumknoten. Beweisskizze:

- Basis: B ist leer und $B = \{s\}$ (nach der ersten Iteration): trivial.
- Ind. Schritt: nehme an, das Theorem gilt für $|B| = k > 0$. Sei v der Knoten der als nächste zum Baum B hinzugefügt wird. Der Fall $\texttt{dist[v]} = +\texttt{inf}$ ist trivial. Betrachte $\texttt{dist[v]} \neq +\texttt{inf}$. Dann gibt es eine Kante vom Baumknoten $x = \texttt{prev[v]} \in B$ zu $v \notin B$. Aus 3. folgt, dass $\texttt{dist[v]}$ das Gewicht des kürzesten Pfades von s nach v über Baumknoten ist. Aus 4. und der Wahl von v folgt, dass es keinen kürzeren Pfad gibt, d.h., $\texttt{dist[v]} = \delta(s, v)$.



Korrektheit

Theorem (Korrektheit)

Dijkstras Algorithmus berechnet den kürzesten Abstand von Knoten s zu jedem von s erreichbaren Knoten in G.

Dijkstras Algorithmus – Implementierung

```
1  //Input: gewichteter Graph mit n Knoten, Startknoten
2  void dijkstra(int adj[n], int n, int start, int &dist[n],
3                int &prev[n]) {
4    for (int i = 0; i < n; i++) {
5       dist[i] = inf; prev[i] = -1;
6    }
7    dist[start] = 0; //start ist einziger Randknoten mit Kosten 0
8    Q = 0,...,n-1; //Q enthält alle ungesehenen Knoten und
9                   //alle Randknoten
10   while (Q not empty) {
11     //verschiebe den billigsten Randknoten v in den Baum:
12     //extractMin(Q) bestimmt ein Element e aus Q mit minimalem dist[e],
           entfernt e aus Q und gibt e zurück
13     v = extractMin(Q);
14     for each (edge in adj[v]) //aktualisiere Kosten für Randknoten
15       if (edge.target in Q and
16         dist[v]+edge.weight < dist[edge.target]) {
17         dist[edge.target] = dist[v] + edge.weight;
18         prev[edge.target] = v;
19       }
20   }
21 }
```

Eigenschaften von Dijkstras Algorithmus

- Kürzeste Wege werden mit zunehmendem Abstand zur Quelle s gefunden.

Eigenschaften von Dijkstras Algorithmus

- Kürzeste Wege werden mit zunehmendem Abstand zur Quelle s gefunden.
- Implementierung: Ähnlich dem Algorithmus von Prim.

Eigenschaften von Dijkstras Algorithmus

- Kürzeste Wege werden mit zunehmendem Abstand zur Quelle s gefunden.
- Implementierung: Ähnlich dem Algorithmus von Prim.
- Zeitkomplexität im Worst-Case: $\Theta(|V|^2)$.

Eigenschaften von Dijkstras Algorithmus

- Kürzeste Wege werden mit zunehmendem Abstand zur Quelle s gefunden.
- Implementierung: Ähnlich dem Algorithmus von Prim.
- Zeitkomplexität im Worst-Case: $\Theta(|V|^2)$.
- Untere Schranke der Komplexität: $\Omega(|E|)$.
  - da im schlimmsten Fall alle Kanten überprüft werden müssen.

Eigenschaften von Dijkstras Algorithmus

- Kürzeste Wege werden mit zunehmendem Abstand zur Quelle s gefunden.
- Implementierung: Ähnlich dem Algorithmus von Prim.
- Zeitkomplexität im Worst-Case: $\Theta(|V|^2)$.
- Untere Schranke der Komplexität: $\Omega(|E|)$.
  - da im schlimmsten Fall alle Kanten überprüft werden müssen.
- Platzkomplexität: $O(|V|)$.
- Erlaubt keine negative Kosten. Warum?

Übersicht

All-Pairs Shortest Paths

Wir betrachten gewichtete gerichtete Graphen $G = (V, E, W)$.

- Negative Gewichte sind zugelassen, aber keine Zyklen mit negativem Gewicht.
- Nicht vorhandene Kanten haben Gewicht $W(\cdot, \cdot) = +\texttt{inf}$.

All-Pairs Shortest Paths

Wir betrachten gewichtete gerichtete Graphen $G = (V, E, W)$.

- Negative Gewichte sind zugelassen, aber keine Zyklen mit negativem Gewicht.
- Nicht vorhandene Kanten haben Gewicht $W(\cdot, \cdot) = +\texttt{inf}$.

Problem (All-Pairs Shortest Path)

Berechne für jedes Paar `i, j` *das Gewicht* `D[i,j]` *des kürzesten Pfades.*

All-Pairs Shortest Paths

Wir betrachten gewichtete gerichtete Graphen $G = (V, E, W)$.

- Negative Gewichte sind zugelassen, aber keine Zyklen mit negativem Gewicht.
- Nicht vorhandene Kanten haben Gewicht $W(\cdot, \cdot) = +\texttt{inf}$.

Problem (All-Pairs Shortest Path)

Berechne für jedes Paar `i, j` *das Gewicht* `D[i,j]` *des kürzesten Pfades.*

Naive Lösung:
wende ein SSSP-Algorithmus (z.B. Bellman-Ford) $|V|$ mal an.

All-Pairs Shortest Paths

Wir betrachten gewichtete gerichtete Graphen $G = (V, E, W)$.

- Negative Gewichte sind zugelassen, aber keine Zyklen mit negativem Gewicht.
- Nicht vorhandene Kanten haben Gewicht $W(\cdot, \cdot) = +\texttt{inf}$.

Problem (All-Pairs Shortest Path)

Berechne für jedes Paar `i, j` *das Gewicht* `D[i,j]` *des kürzesten Pfades.*

Naive Lösung:
wende ein SSSP-Algorithmus (z.B. Bellman-Ford) $|V|$ mal an.
Dies führt zu einer Worst-Case Zeitkomplexität $\mathcal{O}(|V|^4)$.

Effizientere Version: Floyd's Algorithmus.

Binäre Relationen

Binäre Relation

Eine (binäre) Relation über einer Menge S ist eine Teilmenge von $R \subseteq S \times S = S^2$.

Binäre Relationen

Binäre Relation

Eine (binäre) Relation über einer Menge S ist eine Teilmenge von $R \subseteq S \times S = S^2$.

Reflexivität, Transitivität

Eine Relation R ist reflexiv, wenn $(u, u) \in R$ für alle $u \in S$.
Sie heißt transitiv, wenn aus $(u, v) \in R$ und $(v, w) \in R$ folgt $(u, w) \in R$.

Binäre Relationen

Binäre Relation

Eine (binäre) Relation über einer Menge S ist eine Teilmenge von $R \subseteq S \times S = S^2$.

Reflexivität, Transitivität

Eine Relation R ist reflexiv, wenn $(u, u) \in R$ für alle $u \in S$.
Sie heißt transitiv, wenn aus $(u, v) \in R$ und $(v, w) \in R$ folgt $(u, w) \in R$.

Transitive Hülle

Die transitive Hülle R^* einer Relation R ist die kleinste Erweiterung (Obermenge) $R \subseteq R^* \subseteq S^2$, so dass R^* reflexiv und transitiv ist.

Binäre Relationen

Binäre Relation

Eine (binäre) Relation über einer Menge S ist eine Teilmenge von $R \subseteq S \times S = S^2$.

Reflexivität, Transitivität

Eine Relation R ist reflexiv, wenn $(u, u) \in R$ für alle $u \in S$.
Sie heißt transitiv, wenn aus $(u, v) \in R$ und $(v, w) \in R$ folgt $(u, w) \in R$.

Transitive Hülle

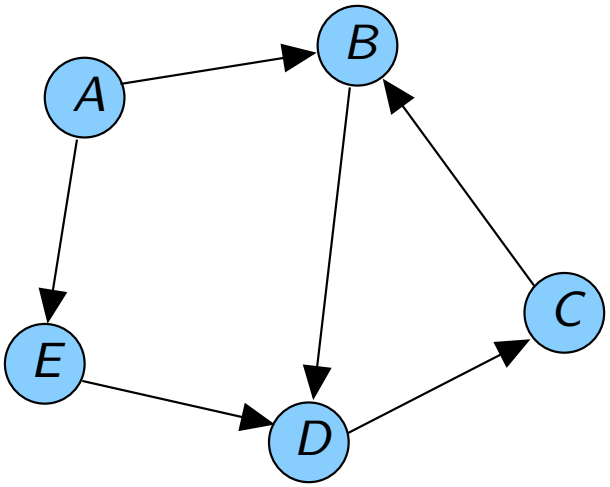
Die transitive Hülle R^* einer Relation R ist die kleinste Erweiterung (Obermenge) $R \subseteq R^* \subseteq S^2$, so dass R^* reflexiv und transitiv ist.

Transitive Hülle eines Graphen

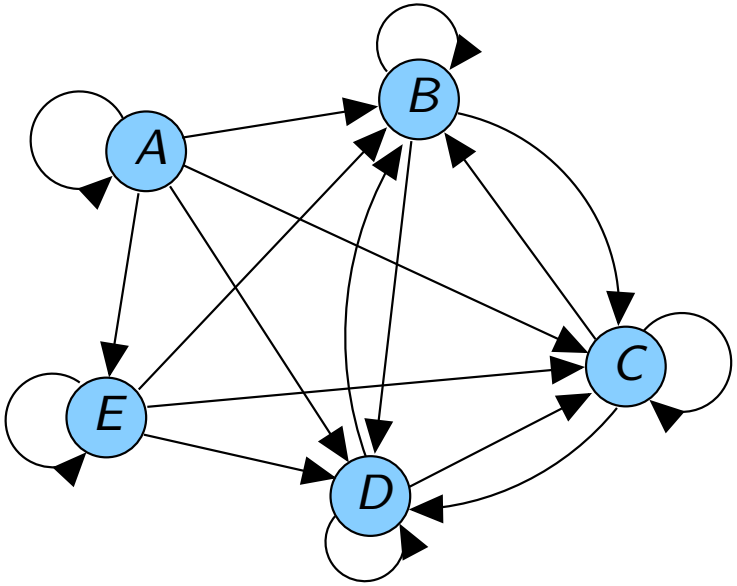
Für Graphen mit $S = V$ und $R = E$ gilt:
$(u, v) \in R^*$ gdw. es gibt einen Pfad von u nach v.

Transitive Hülle: Beispiel

$$R = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$ und die transitive Hülle $R^* = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$



Binäre Relation R

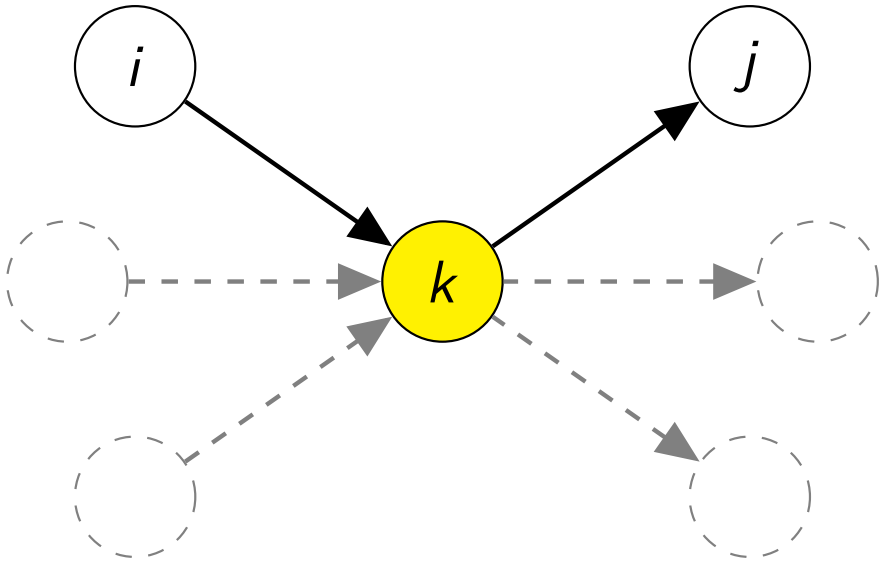


Transitive Hülle R^*

Übersicht

Algorithmus von Warshall: Idee

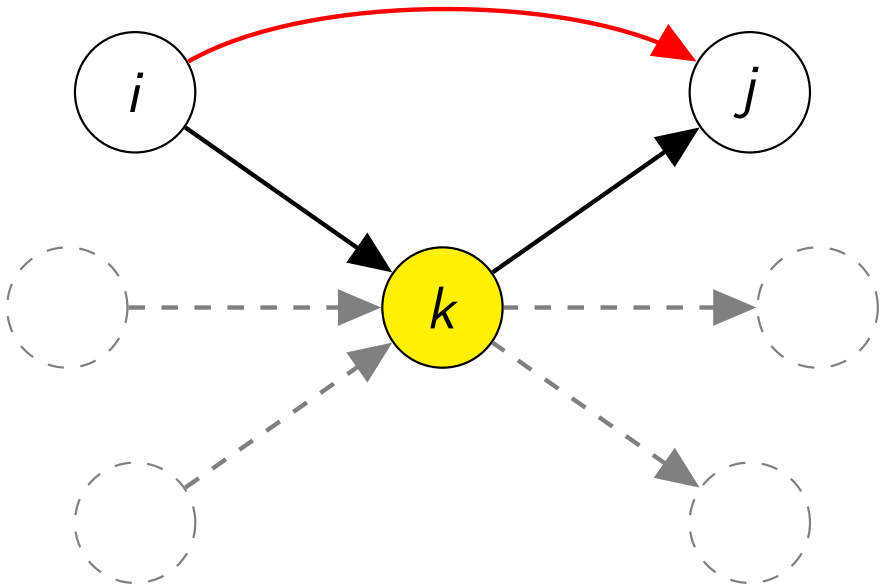
- Aus `R[i,k]` und `R[k,j]` folgt die Erreichbarkeit `R[i,j] = true`.




```
1 foreach (k ∈ V)
2   foreach (eingehende Kante (i, k) ∈ E)
3     foreach (ausgehende Kante (k, j) ∈ E)
4       Füge (i, j) zu E hinzu.
```

Algorithmus von Warshall: Idee

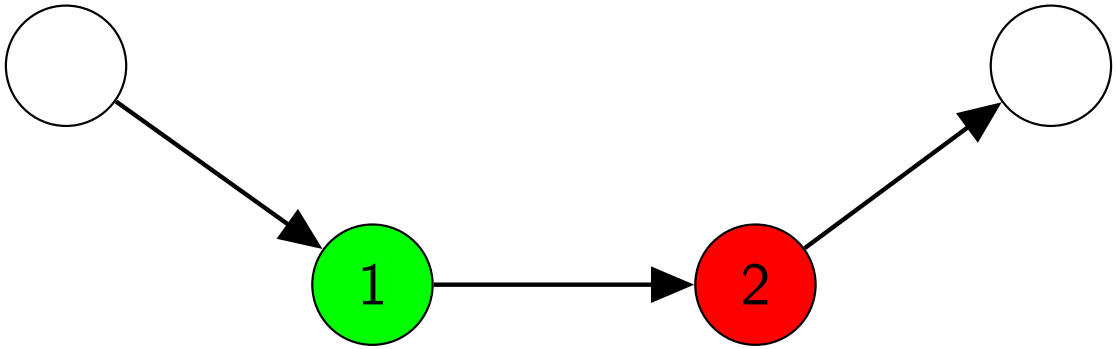
- Aus `R[i,k]` und `R[k,j]` folgt die Erreichbarkeit `R[i,j] = true`.




```
1 foreach (k ∈ V)
2   foreach (eingehende Kante (i, k) ∈ E)
3     foreach (ausgehende Kante (k, j) ∈ E)
4       Füge (i, j) zu E hinzu.
```

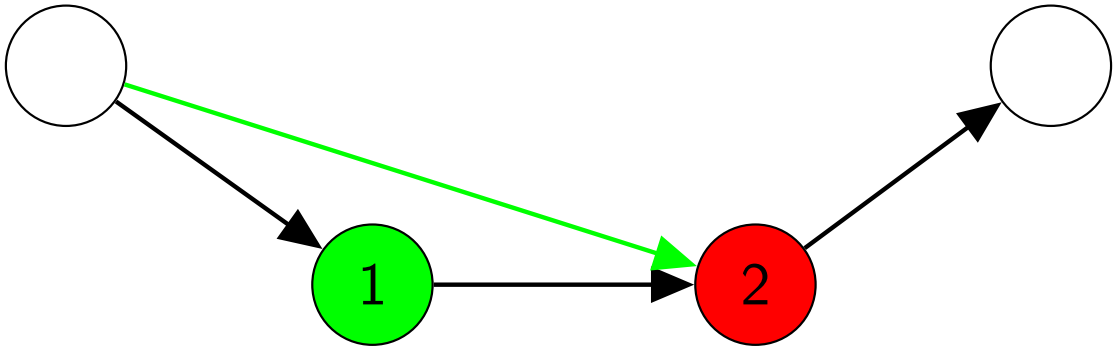
Algorithmus von Warshall: Idee

Das reicht bereits aus, um längere Pfade zu berücksichtigen:



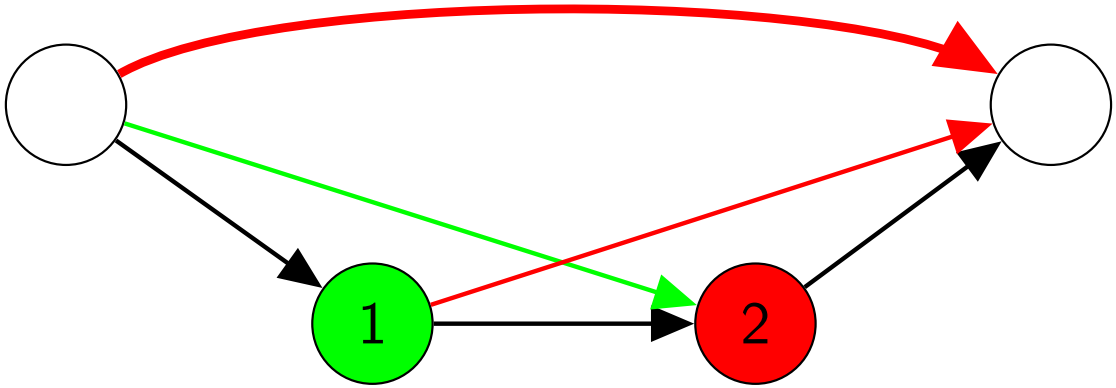
Algorithmus von Warshall: Idee

Das reicht bereits aus, um längere Pfade zu berücksichtigen:



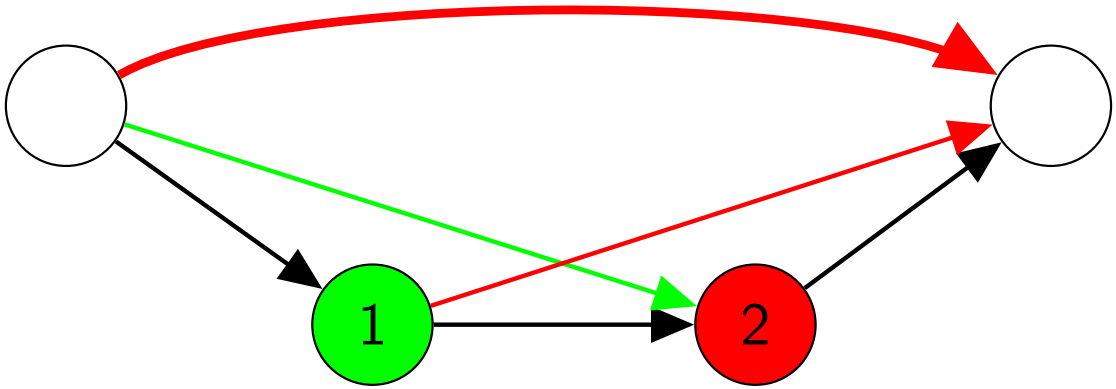
Algorithmus von Warshall: Idee

Das reicht bereits aus, um längere Pfade zu berücksichtigen:

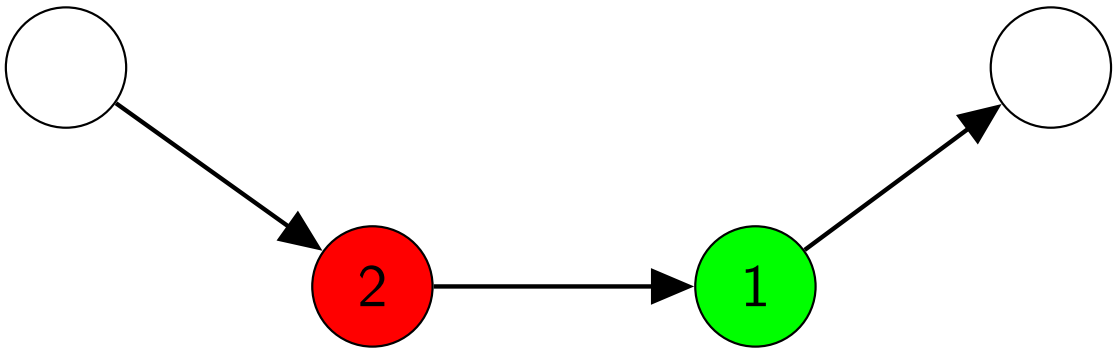


Algorithmus von Warshall: Idee

Das reicht bereits aus, um längere Pfade zu berücksichtigen:

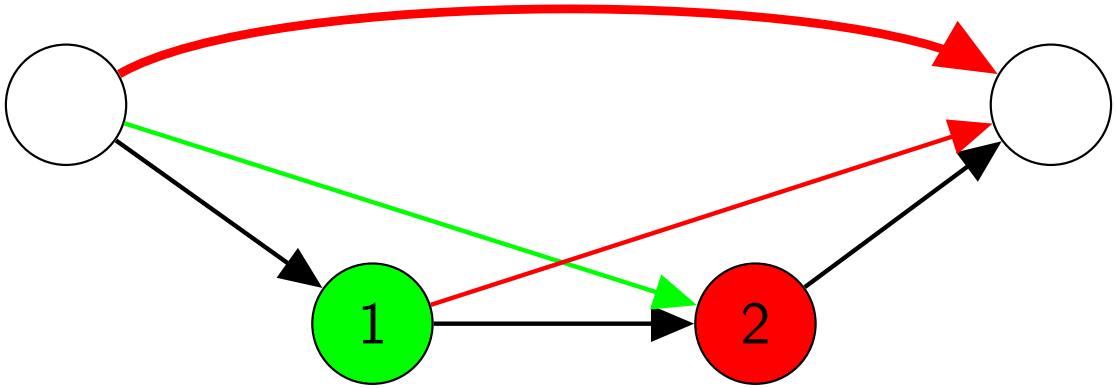


- Die Reihenfolge spielt dabei keine Rolle:

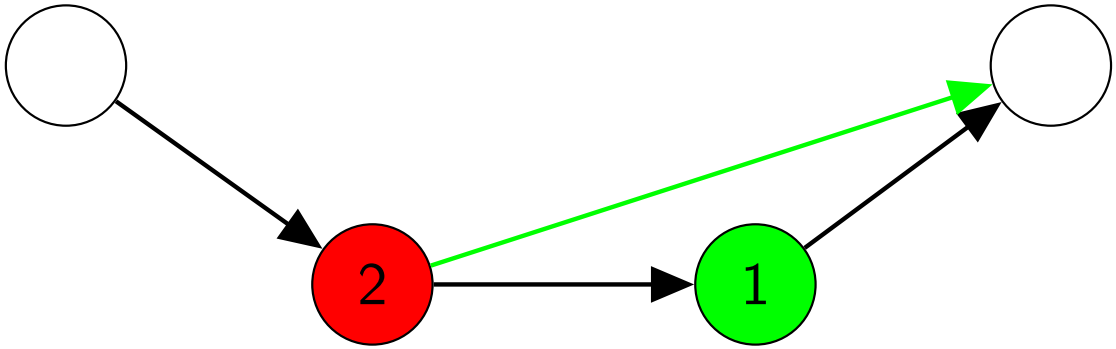


Algorithmus von Warshall: Idee

Das reicht bereits aus, um längere Pfade zu berücksichtigen:

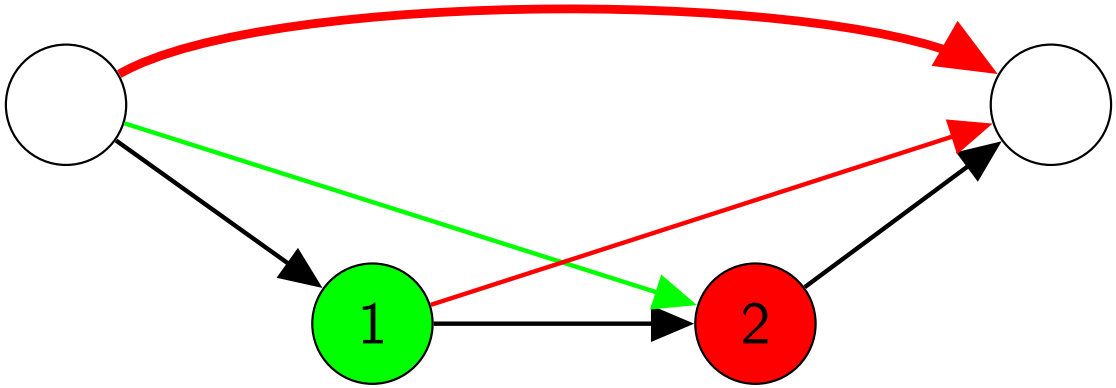


- Die Reihenfolge spielt dabei keine Rolle:

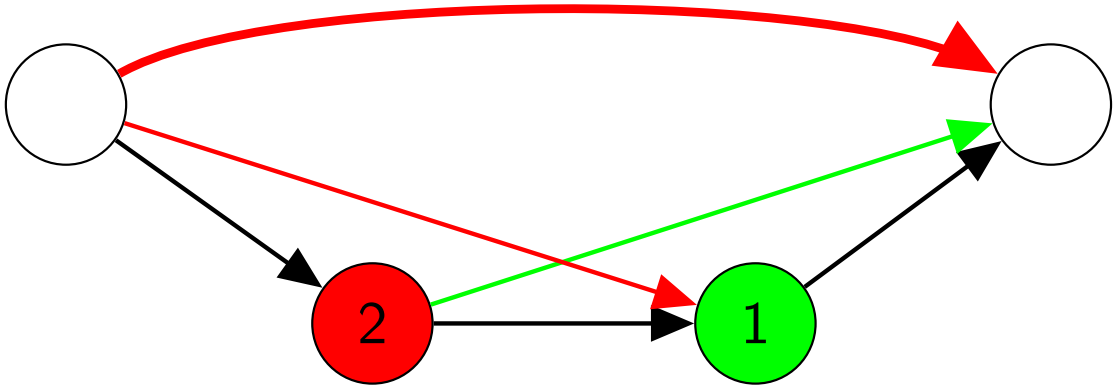


Algorithmus von Warshall: Idee

Das reicht bereits aus, um längere Pfade zu berücksichtigen:

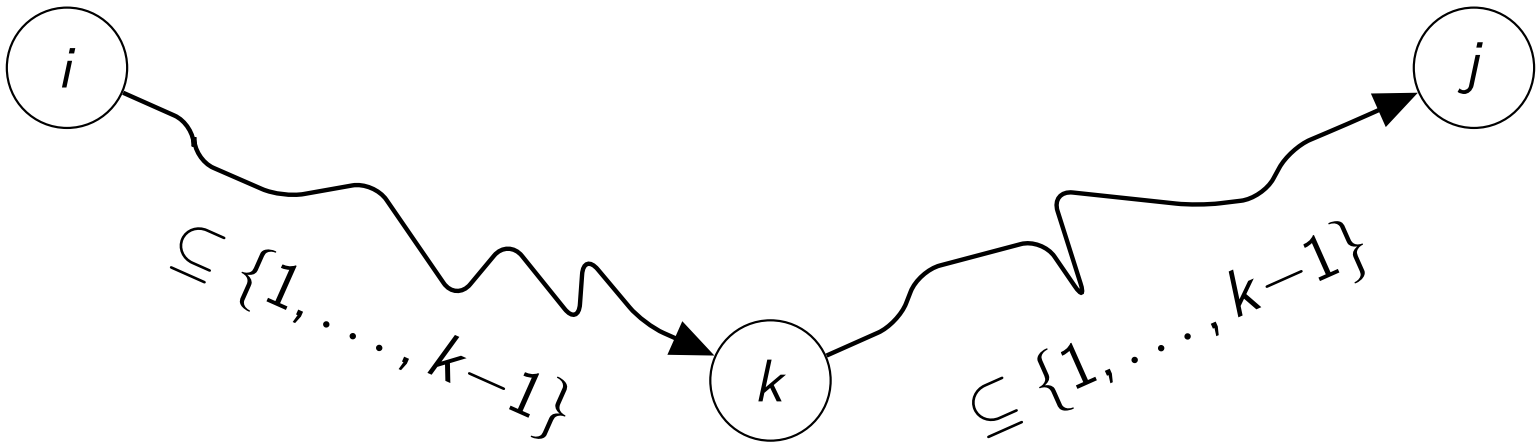


- Die Reihenfolge spielt dabei keine Rolle:



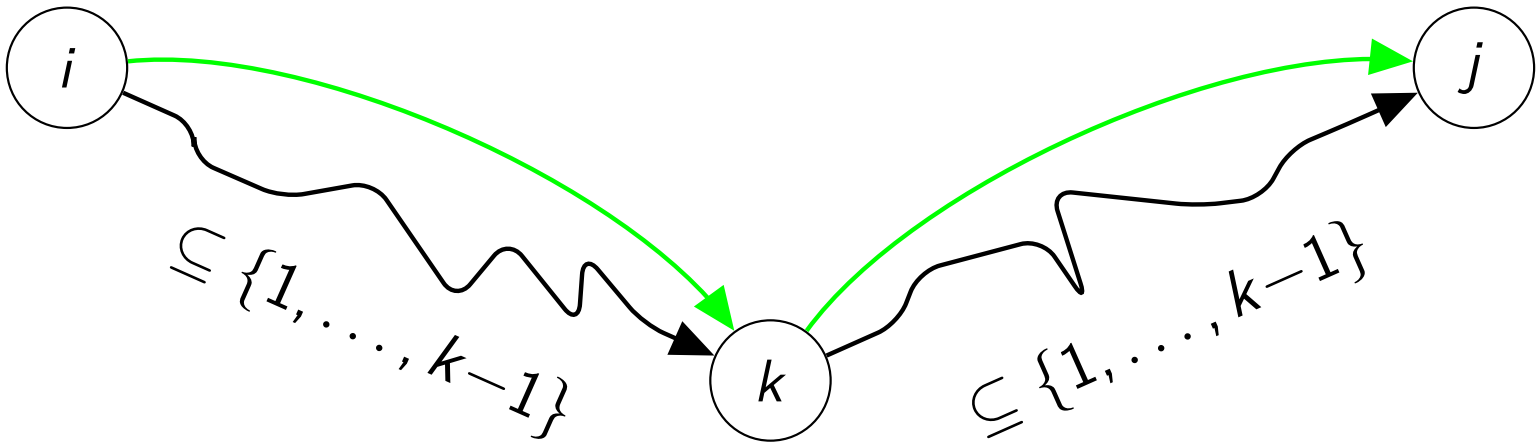
Algorithmus von Warshall: Idee

Allgemeiner Fall:



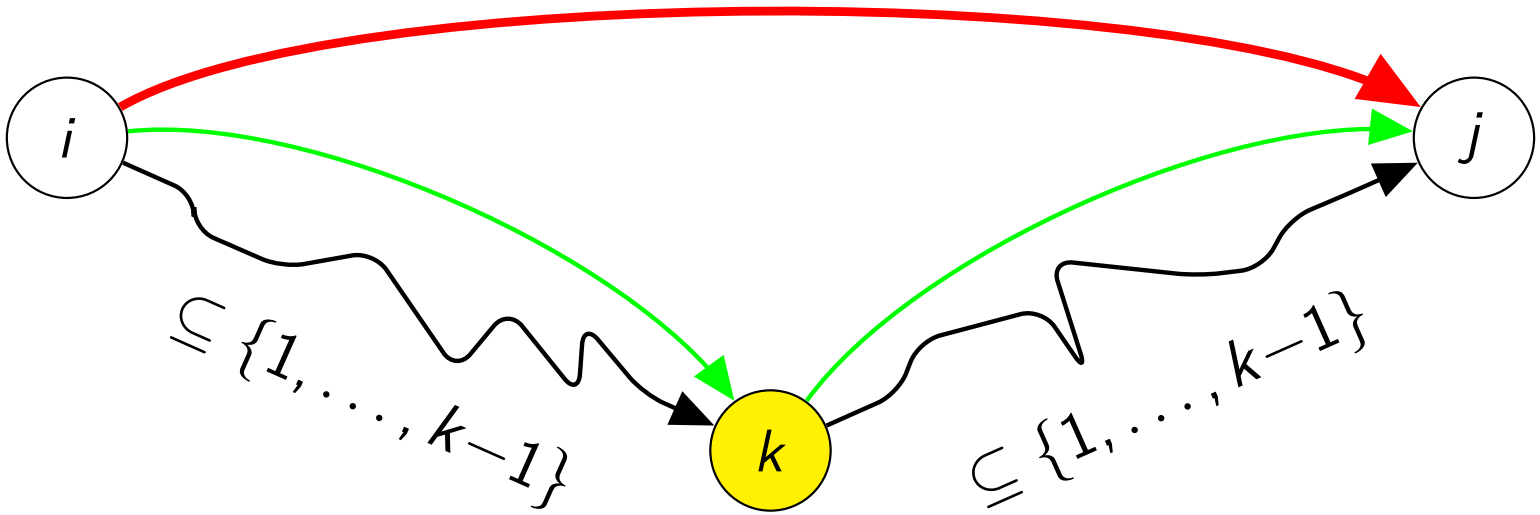
Algorithmus von Warshall: Idee

Allgemeiner Fall:



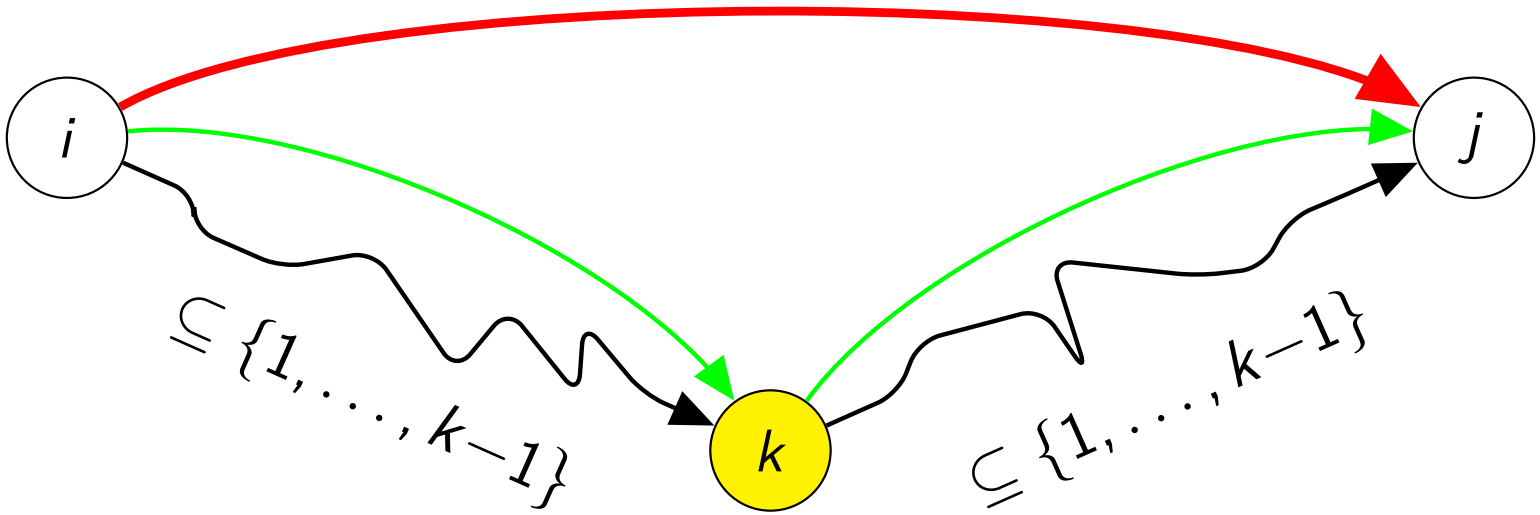
Algorithmus von Warshall: Idee

Allgemeiner Fall:



Algorithmus von Warshall: Idee

Allgemeiner Fall:



- Das lässt sich als Rekursionsgleichung schreiben, wobei $t_{ij}^{(k)} = \texttt{true}$ besagt, dass nach Berücksichtigung der Zwischenknoten $\{1, \ldots, k\}$ der Knoten j von i aus erreichbar ist:

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right)$$

Algorithmus von Warshall: Idee

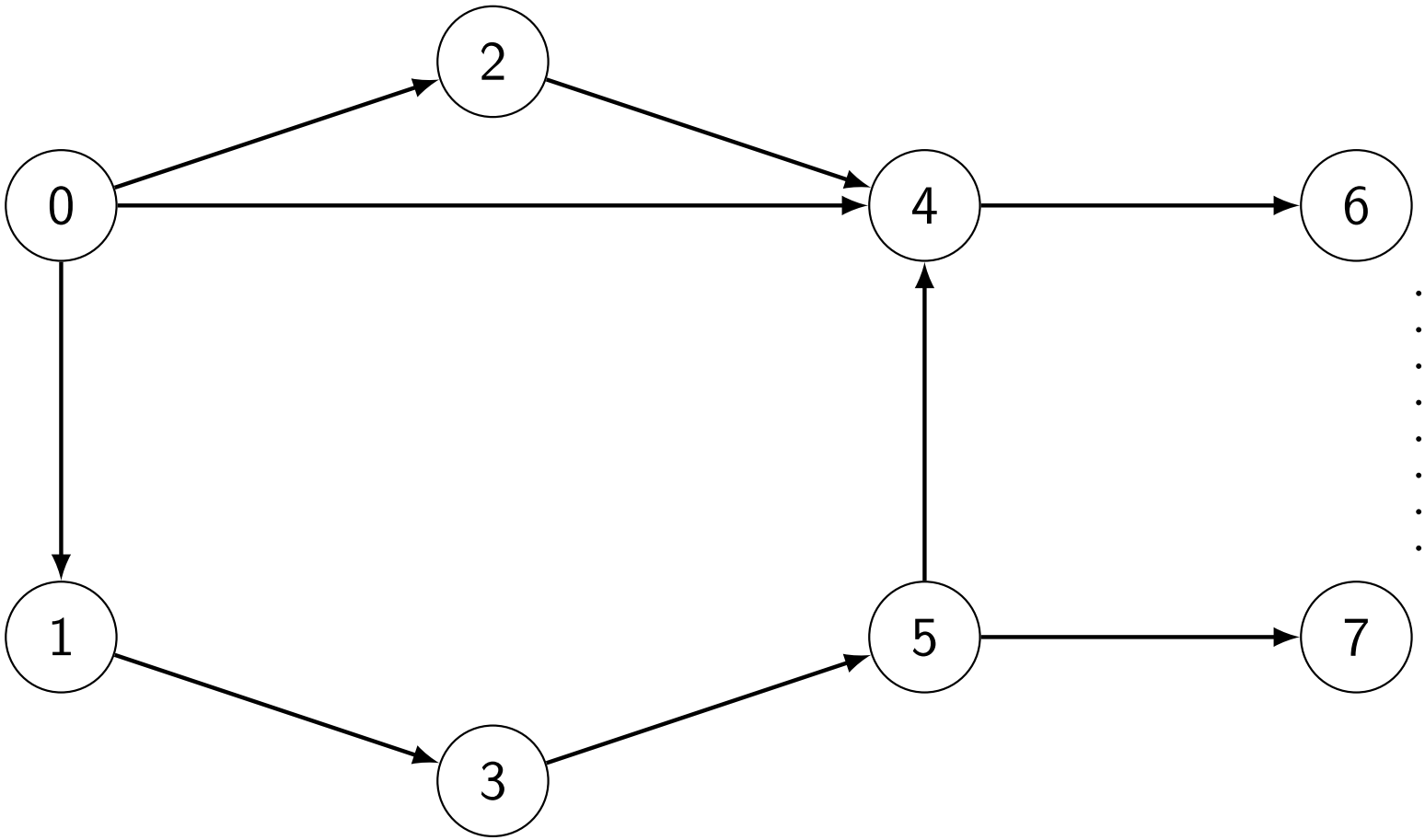
$$t_{ij}^{(k)} = \begin{cases} \texttt{false} & \text{für } k = 0, \text{ falls } (i,j) \notin E \\ \texttt{true} & \text{für } k = 0, \text{ falls } (i,j) \in E \\ t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right) & \text{für } k > 0 \end{cases}$$

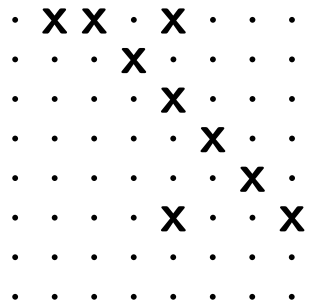
Algorithmus von Warshall: Idee

$$t_{ij}^{(k)} = \begin{cases} \texttt{false} & \text{für } k = 0, \text{ falls } (i,j) \notin E \\ \texttt{true} & \text{für } k = 0, \text{ falls } (i,j) \in E \\ t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right) & \text{für } k > 0 \end{cases}$$

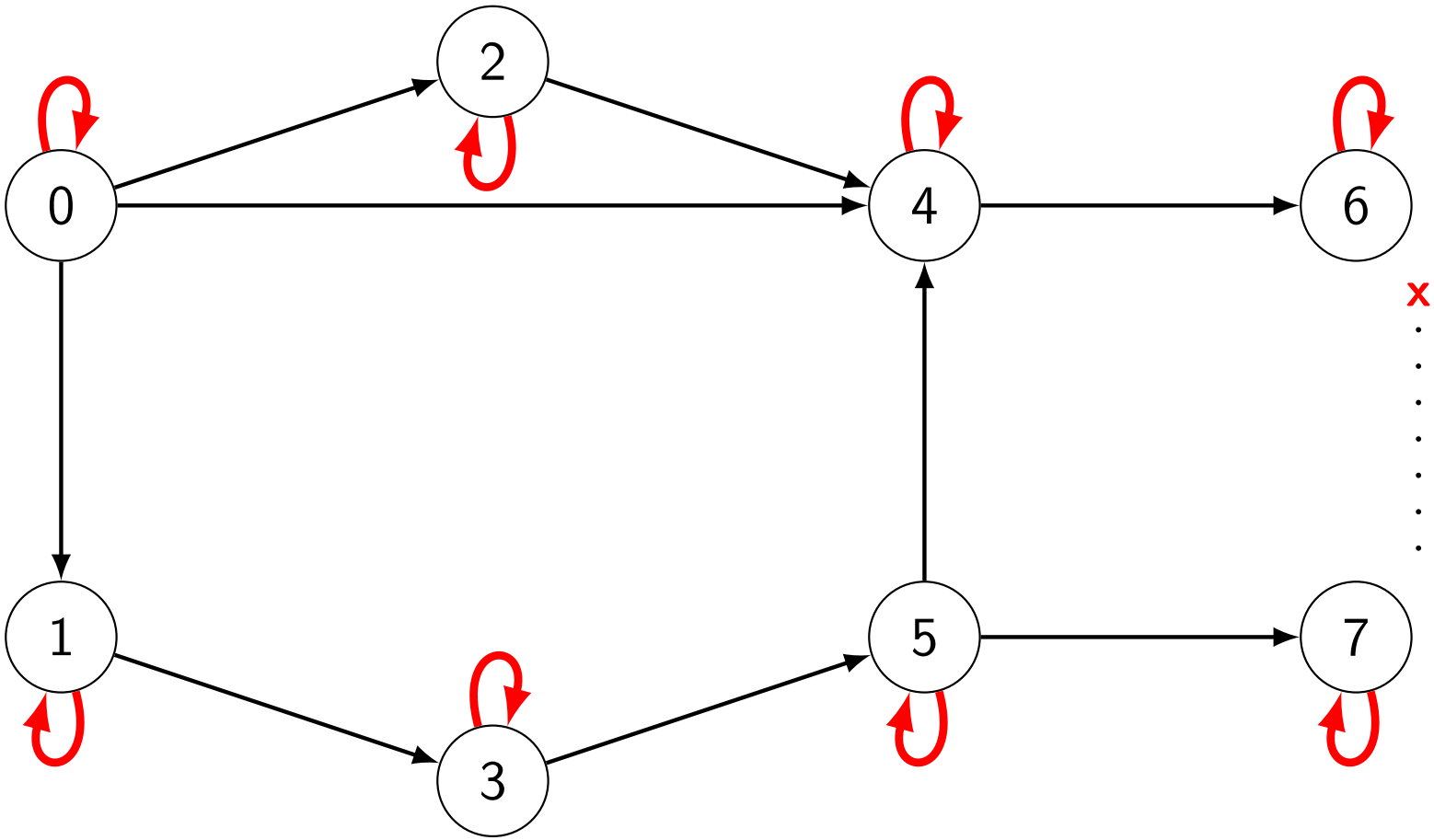
- Da zur Berechnung von $t_{ij}^{(k)}$ nur $t_{ij}^{(k-1)}$ – und keine ältere Werte $t_{ij}^{(n)}$ mit $n < k-1$ – gebraucht wird, kann die Berechnung direkt im Ausgabearray (in-place) erfolgen: `R[i,j]` = $t_{ij}^{(\cdot)}$.

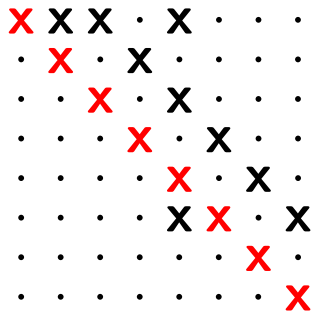
Algorithmus von Warshall: Beispiel



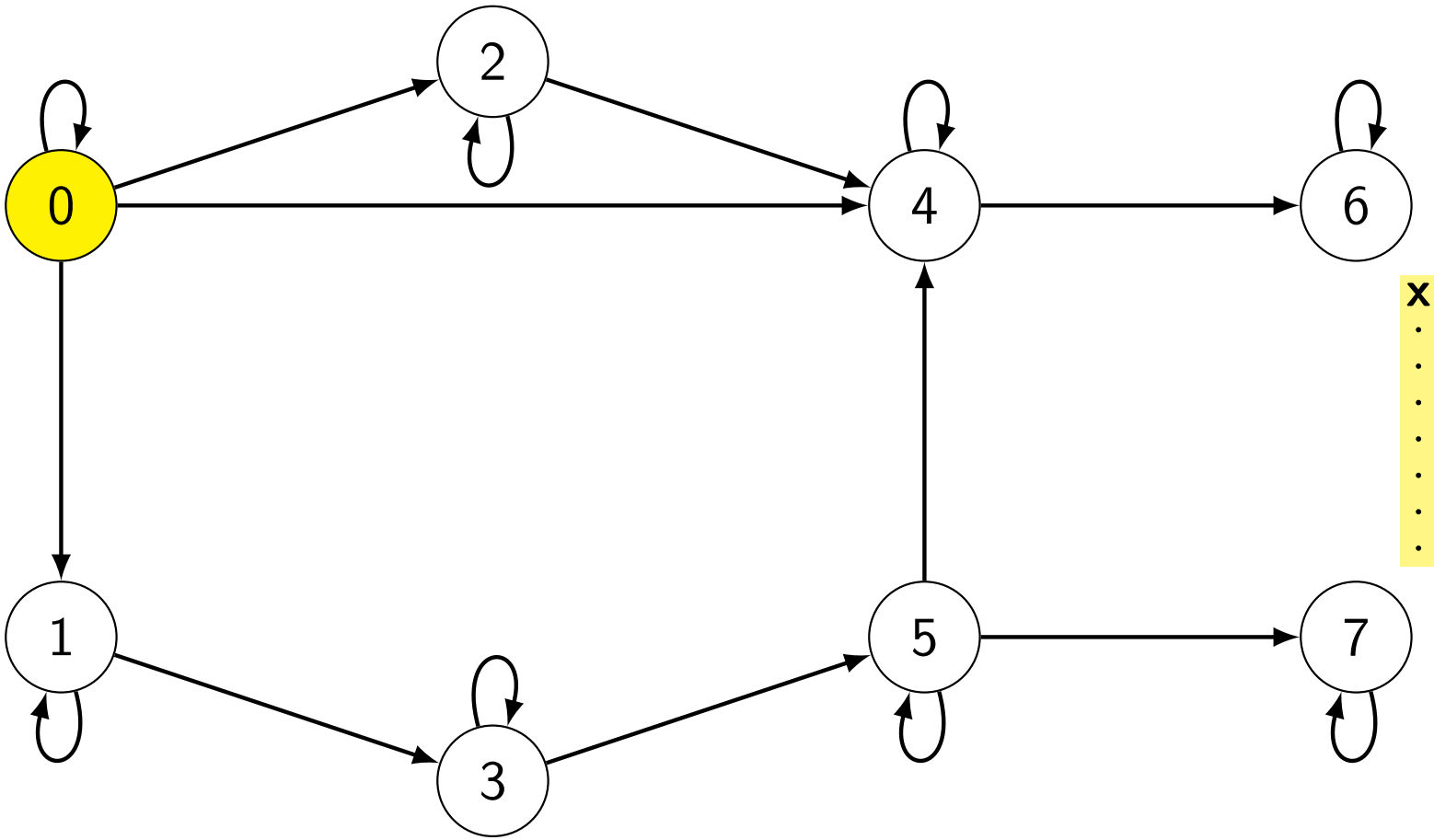


Algorithmus von Warshall: Beispiel



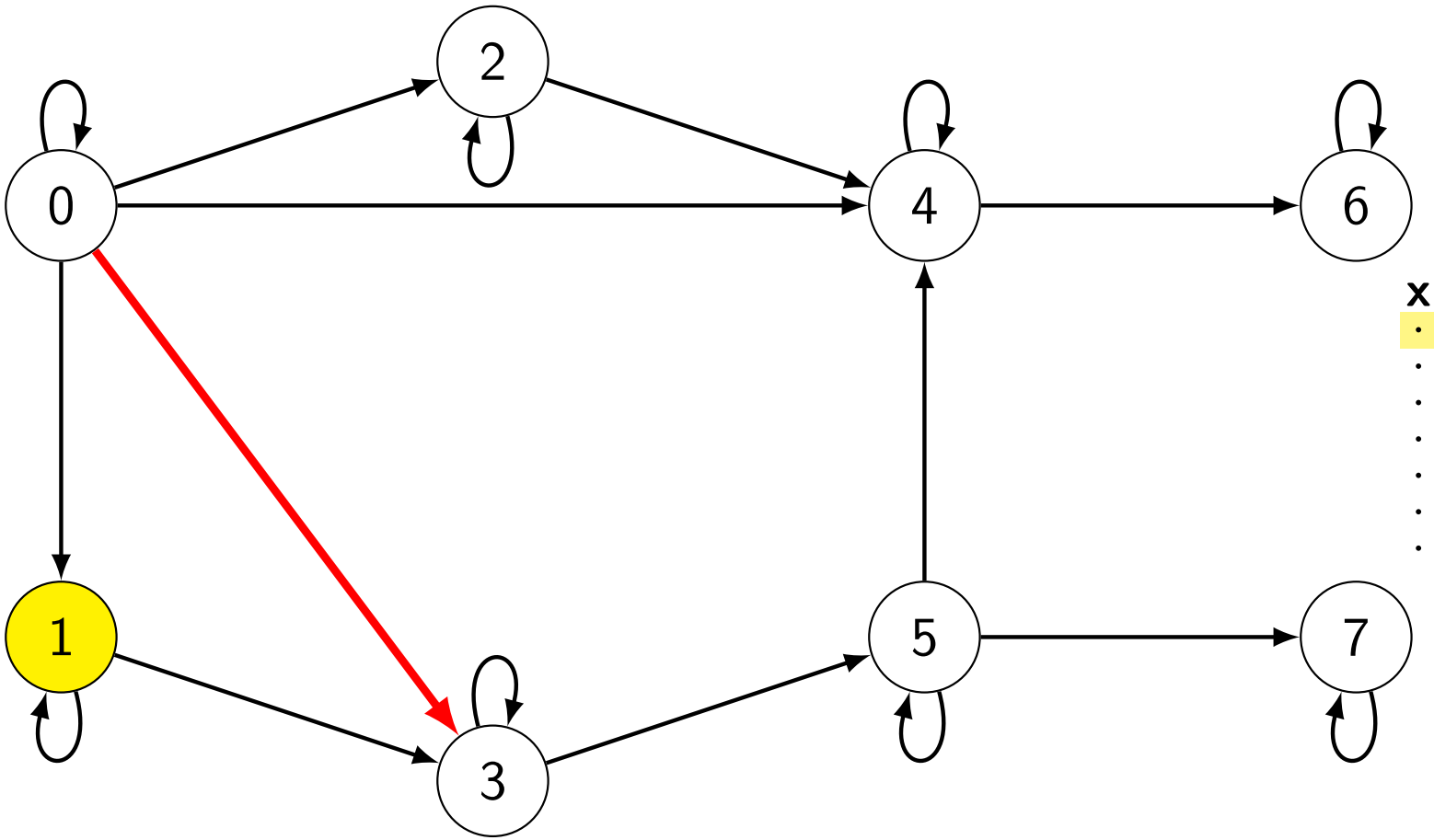


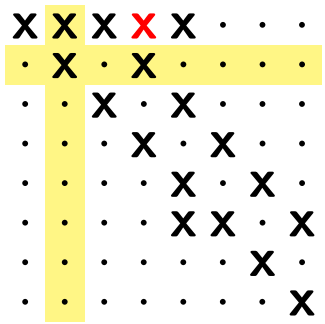
Algorithmus von Warshall: Beispiel



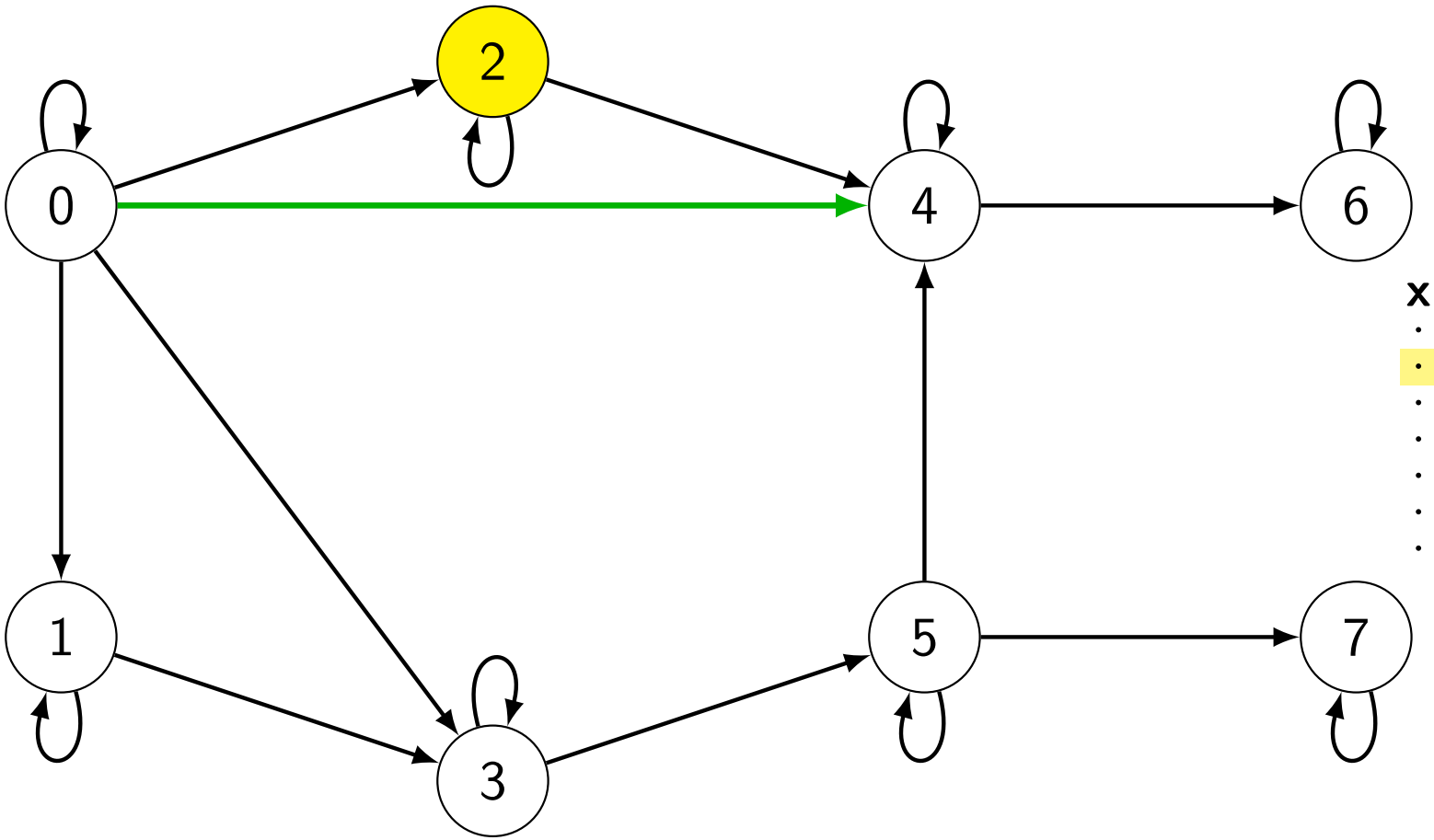
X	X	X	·	X	·	·	·
·	X	·	X	·	·	·	·
·	·	X	·	X	·	·	·
·	·	·	X	·	X	·	·
·	·	·	·	X	·	X	·
·	·	·	·	X	X	·	X
·	·	·	·	·	·	X	·
·	·	·	·	·	·	·	X

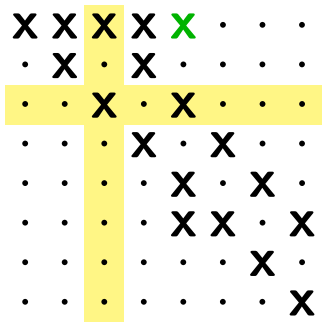
Algorithmus von Warshall: Beispiel



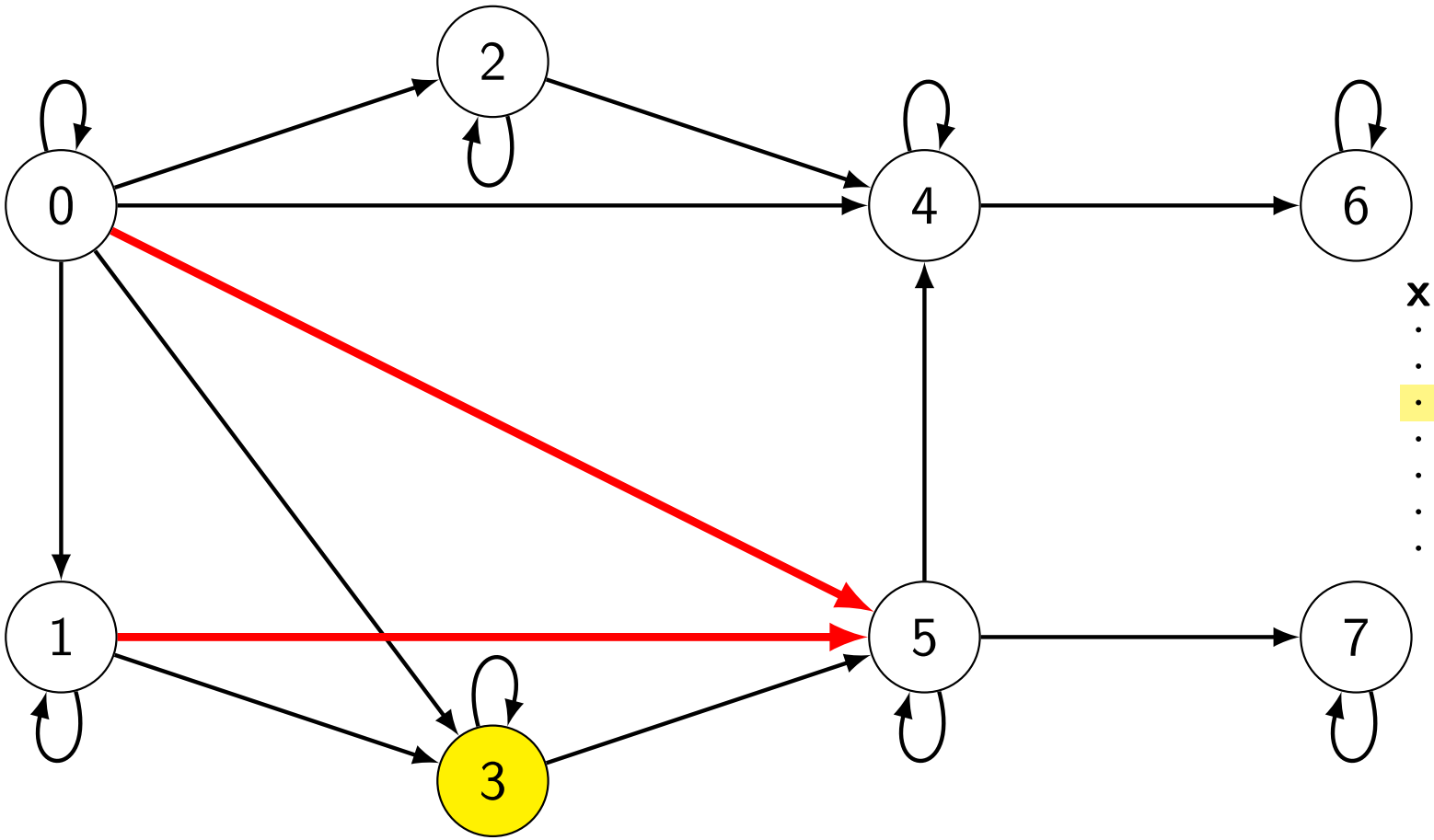


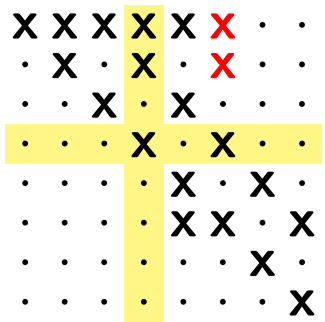
Algorithmus von Warshall: Beispiel



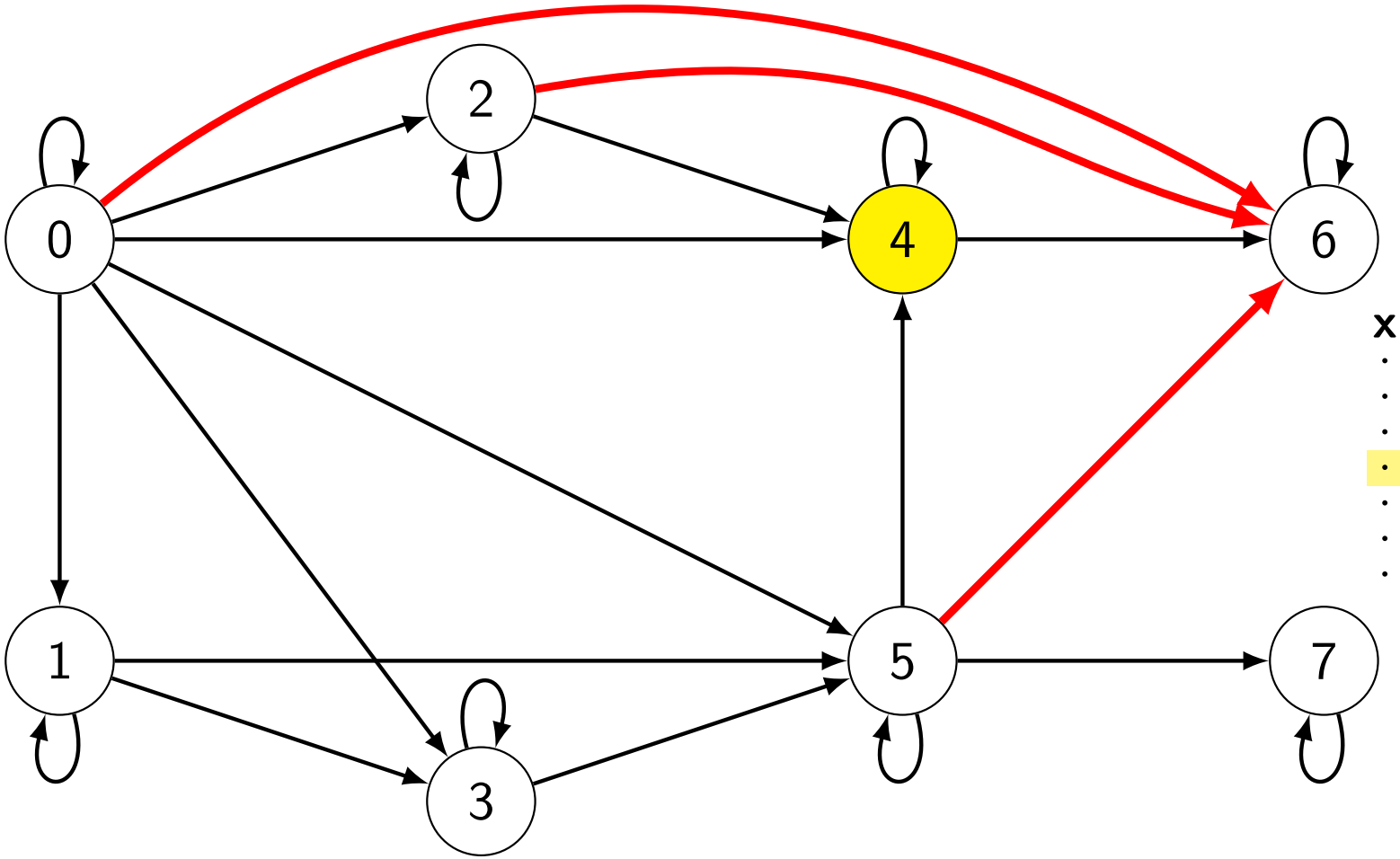


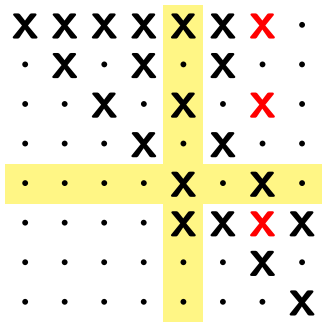
Algorithmus von Warshall: Beispiel



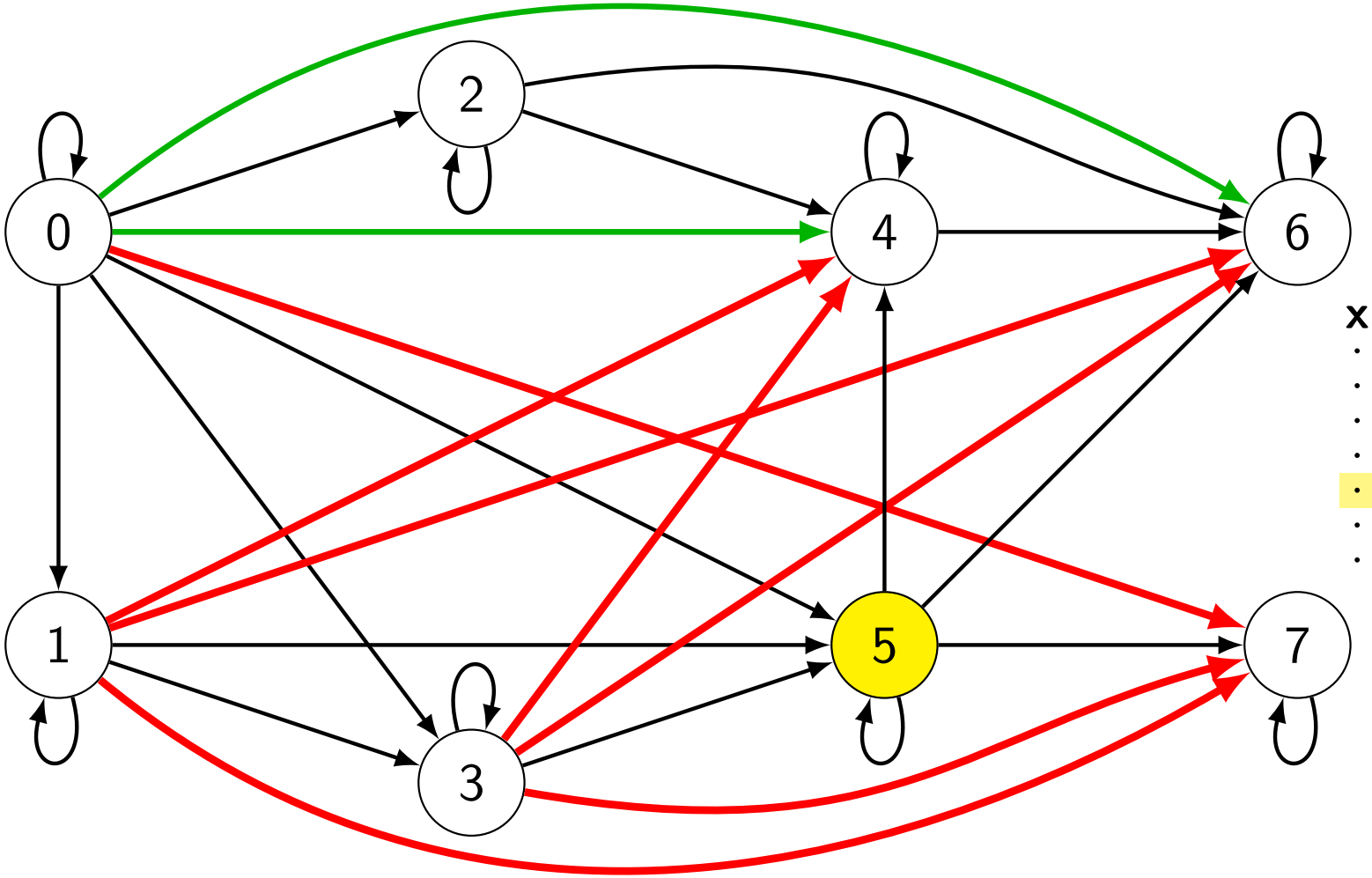


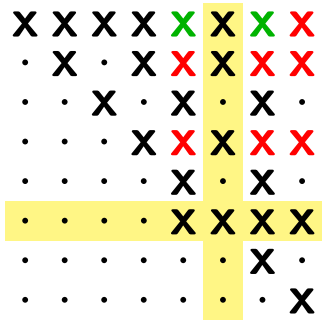
Algorithmus von Warshall: Beispiel



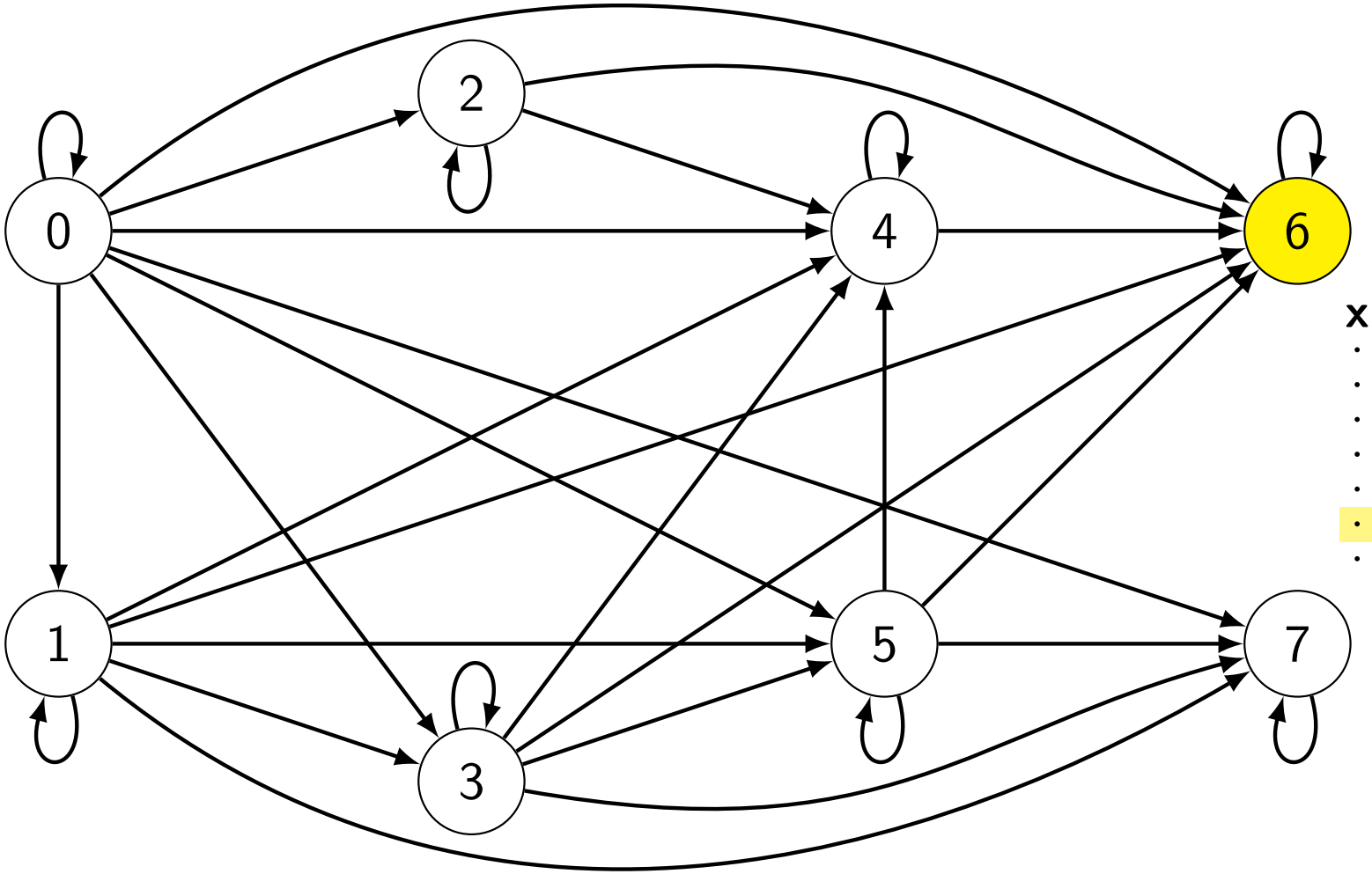


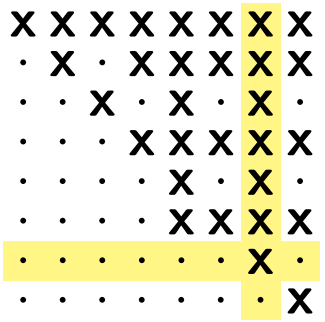
Algorithmus von Warshall: Beispiel



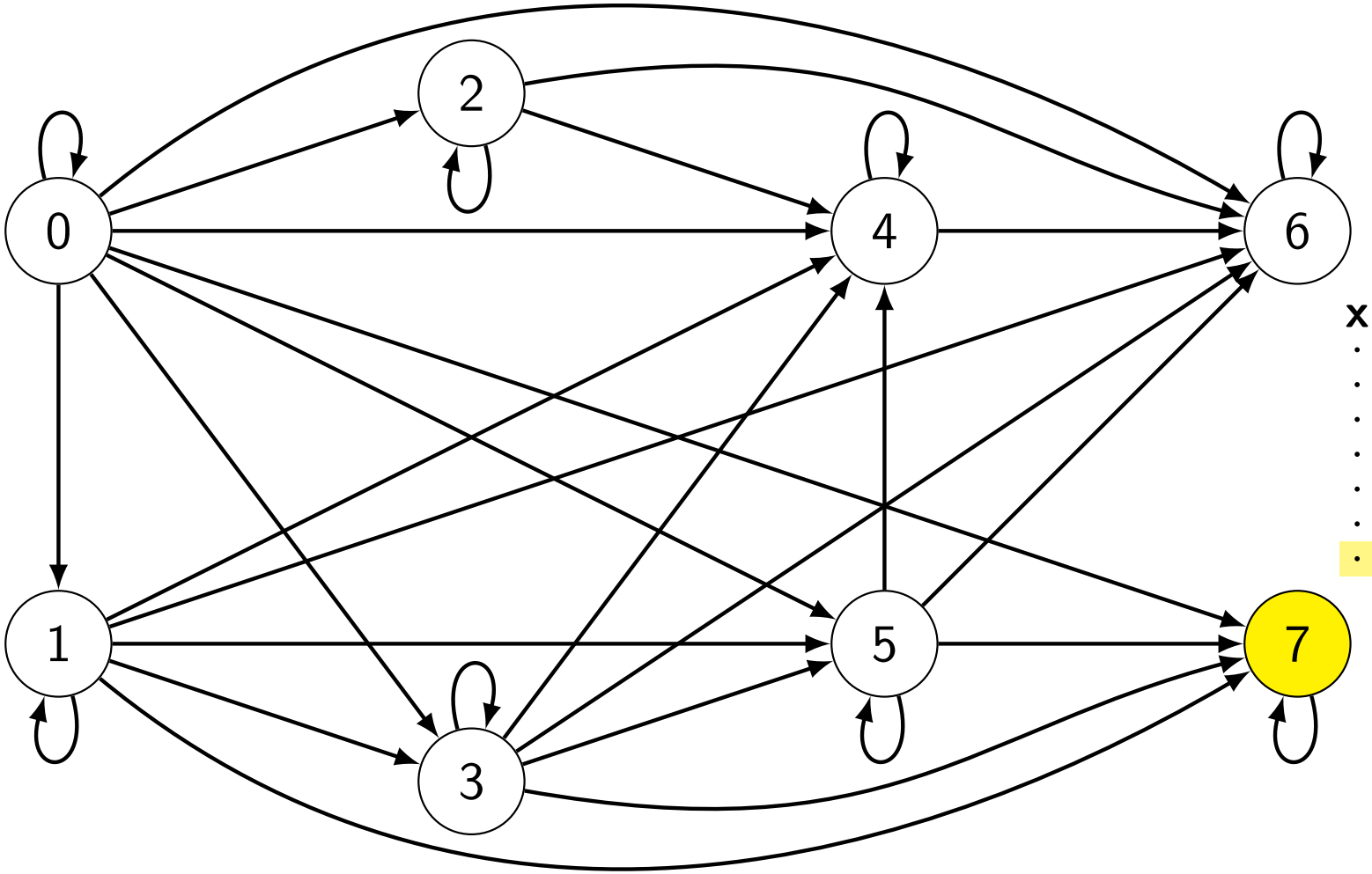


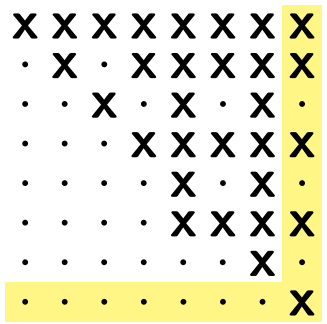
Algorithmus von Warshall: Beispiel





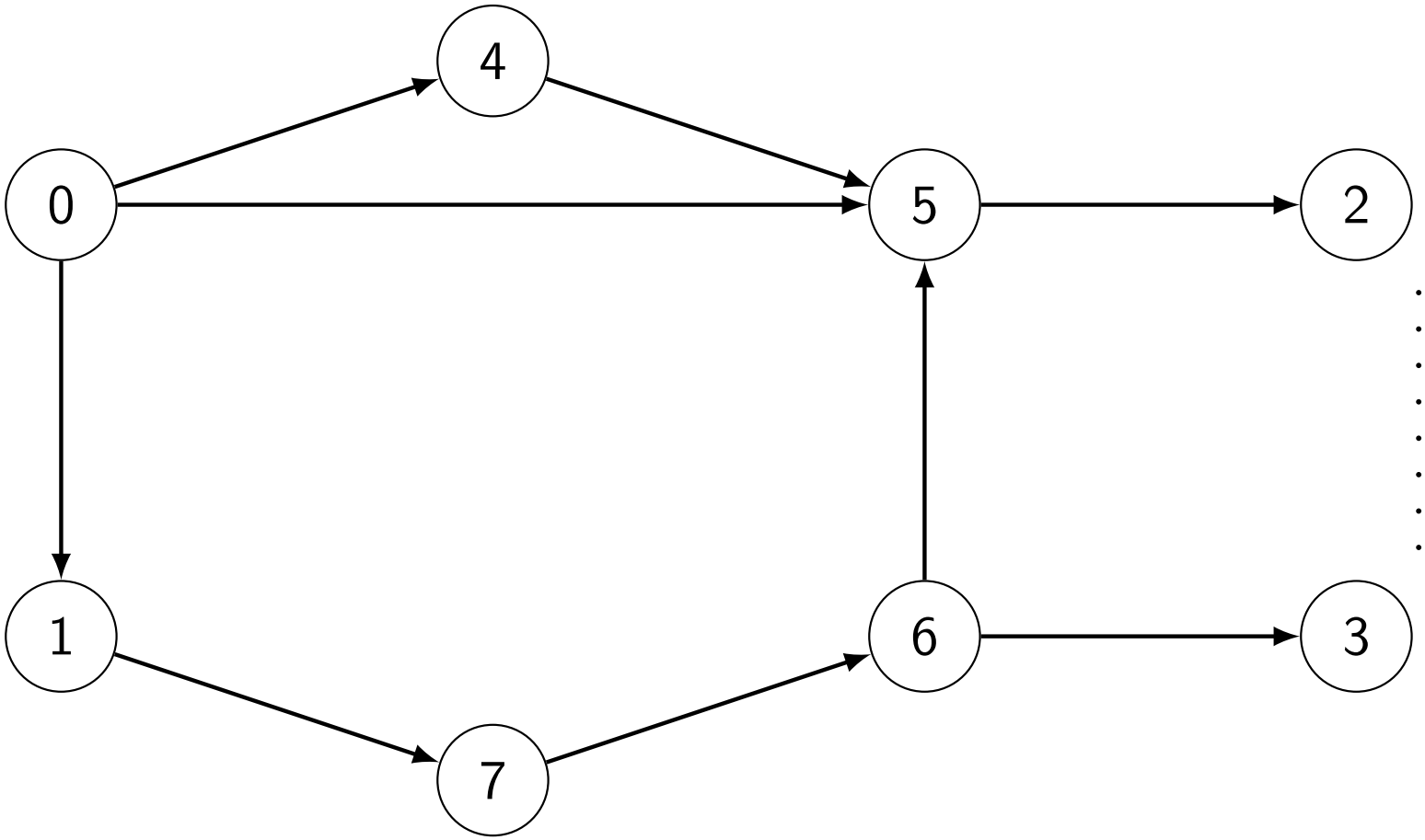
Algorithmus von Warshall: Beispiel

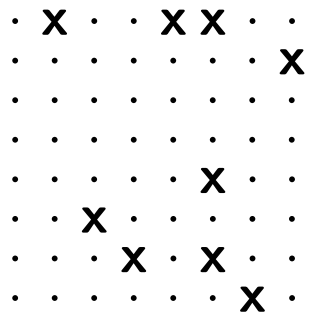




Algorithmus von Warshall: Beispiel

Andere Reihenfolge der Knoten:

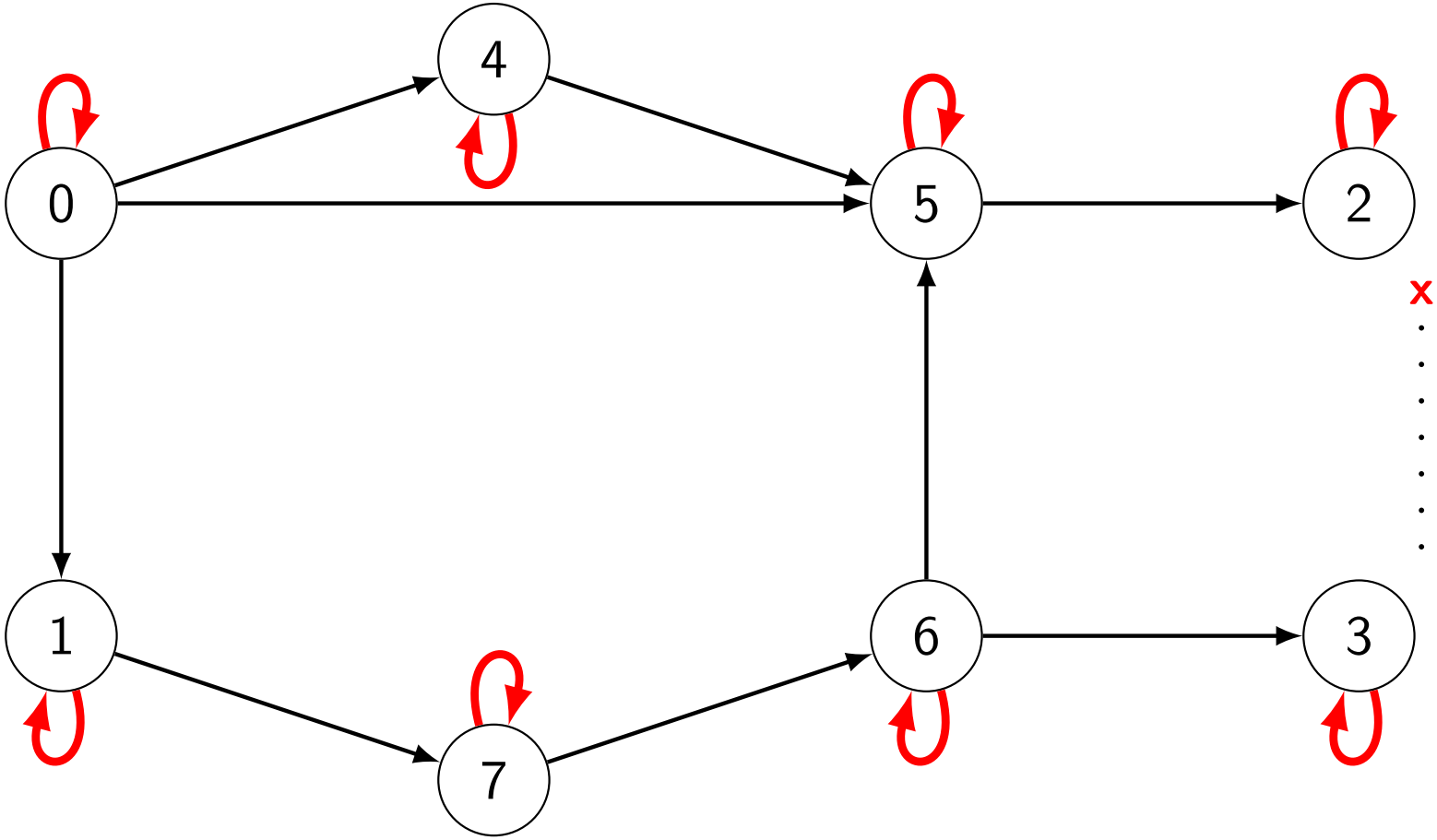


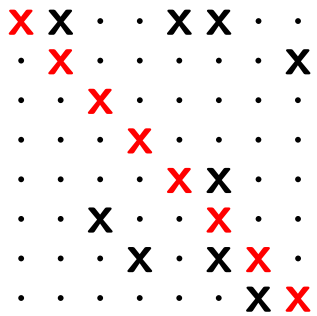


Durch Permutierung der Knoten erhält man die gleiche Matrixdarstellung von R^* wie vorher.

Algorithmus von Warshall: Beispiel

Andere Reihenfolge der Knoten:

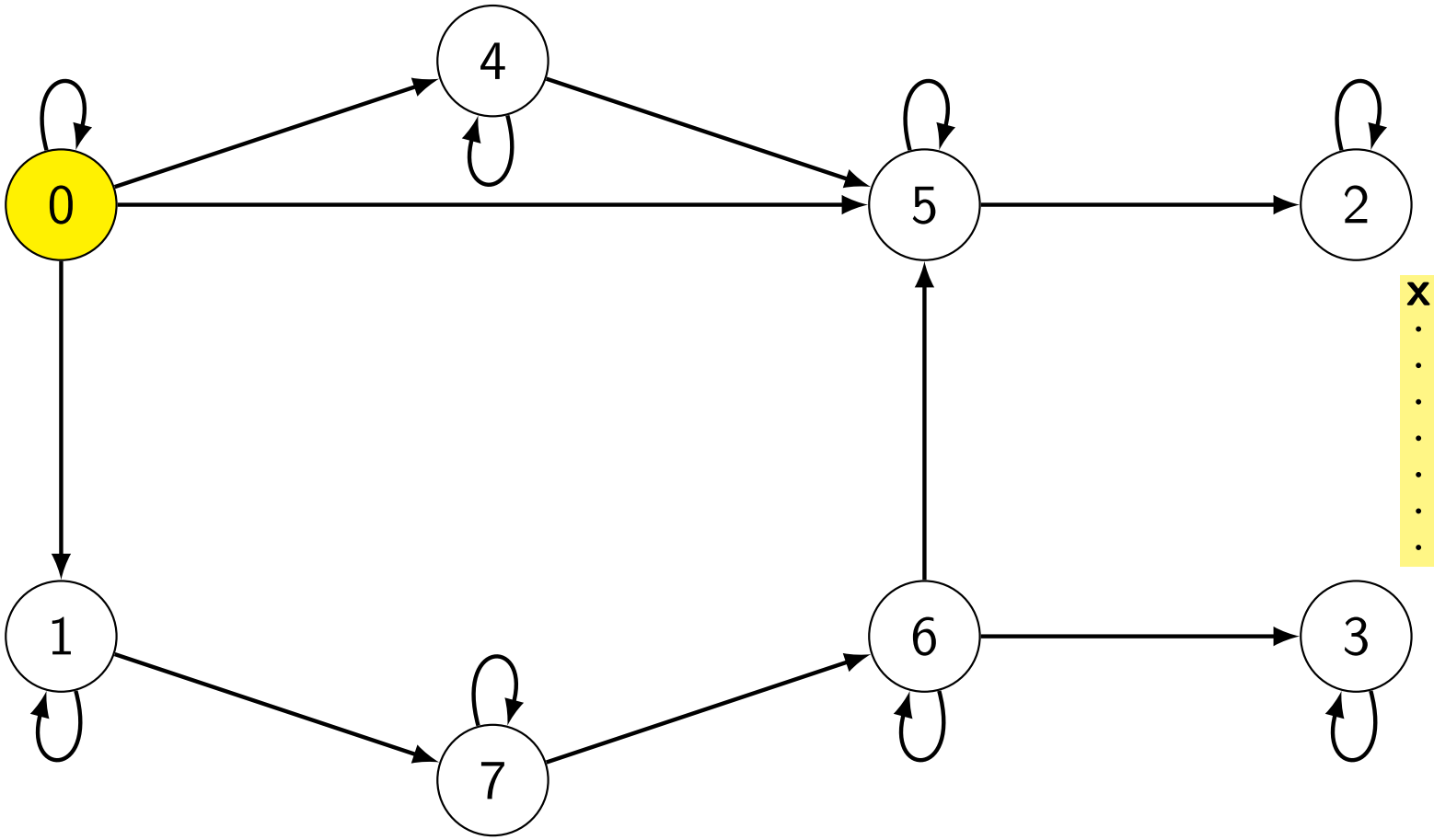




Durch Permutierung der Knoten erhält man die gleiche Matrixdarstellung von R^* wie vorher.

Algorithmus von Warshall: Beispiel

Andere Reihenfolge der Knoten:

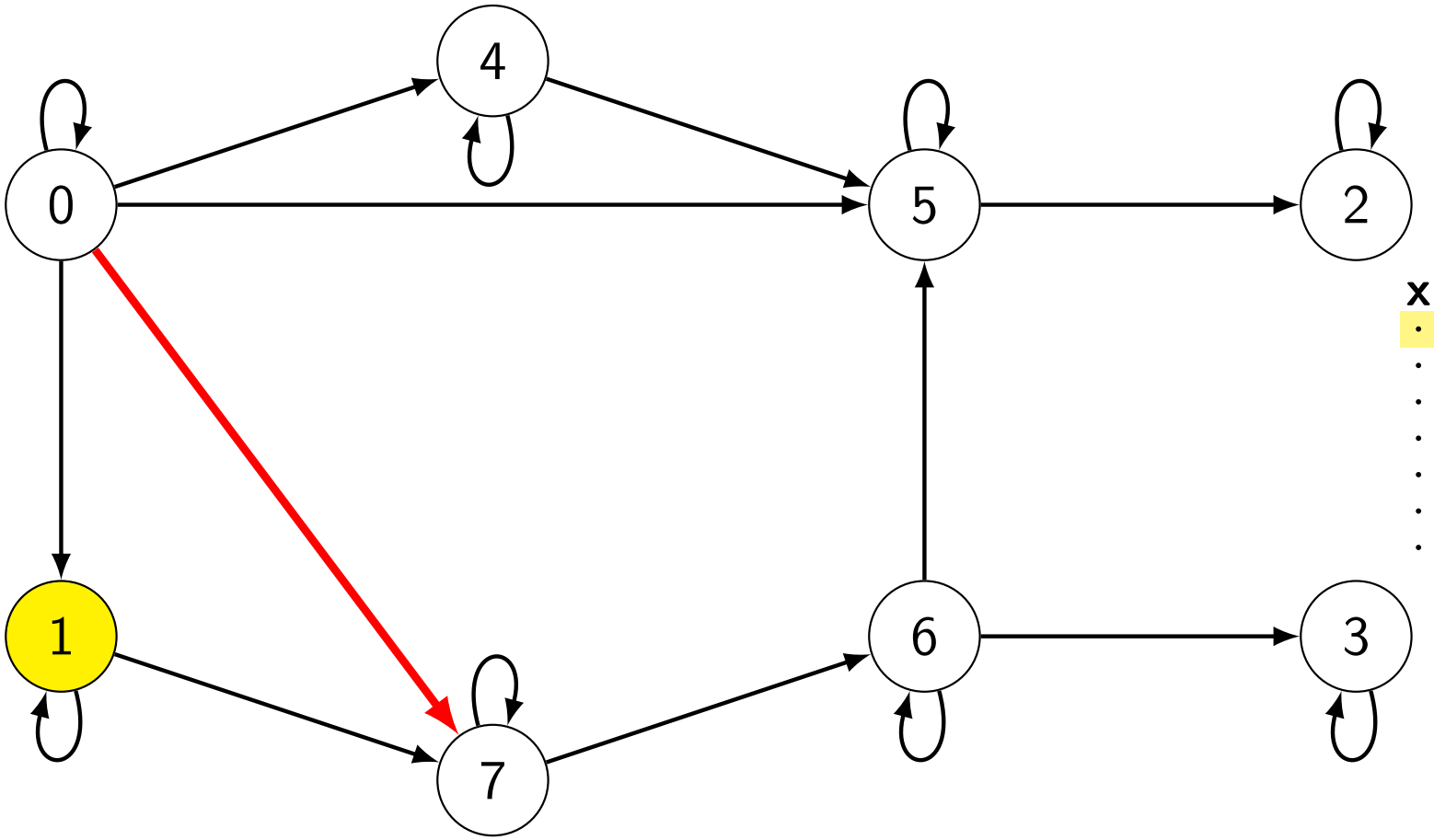


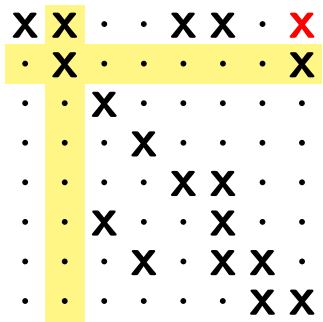
X	X	·	·	X	X	·	·
·	X	·	·	·	·	·	X
·	·	X	·	·	·	·	·
·	·	·	X	·	·	·	·
·	·	·	·	X	X	·	·
·	·	X	·	·	X	·	·
·	·	·	X	·	X	X	·
·	·	·	·	·	·	X	X

Durch Permutierung der Knoten erhält man die gleiche Matrixdarstellung von R^* wie vorher.

Algorithmus von Warshall: Beispiel

Andere Reihenfolge der Knoten:

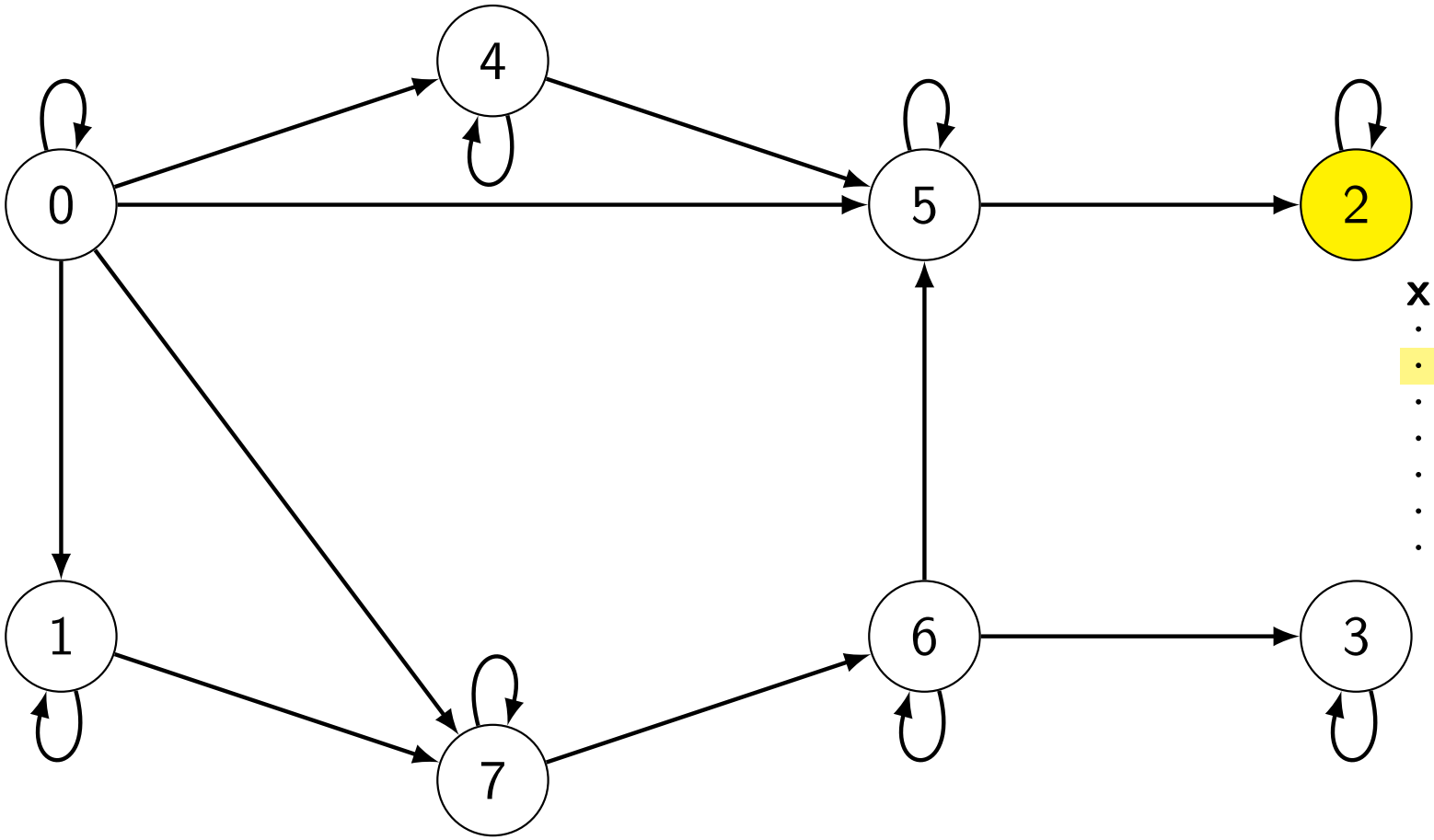


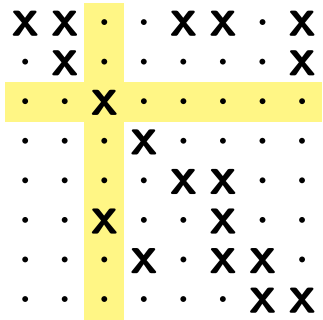


Durch Permutierung der Knoten erhält man die gleiche Matrixdarstellung von R^* wie vorher.

Algorithmus von Warshall: Beispiel

Andere Reihenfolge der Knoten:

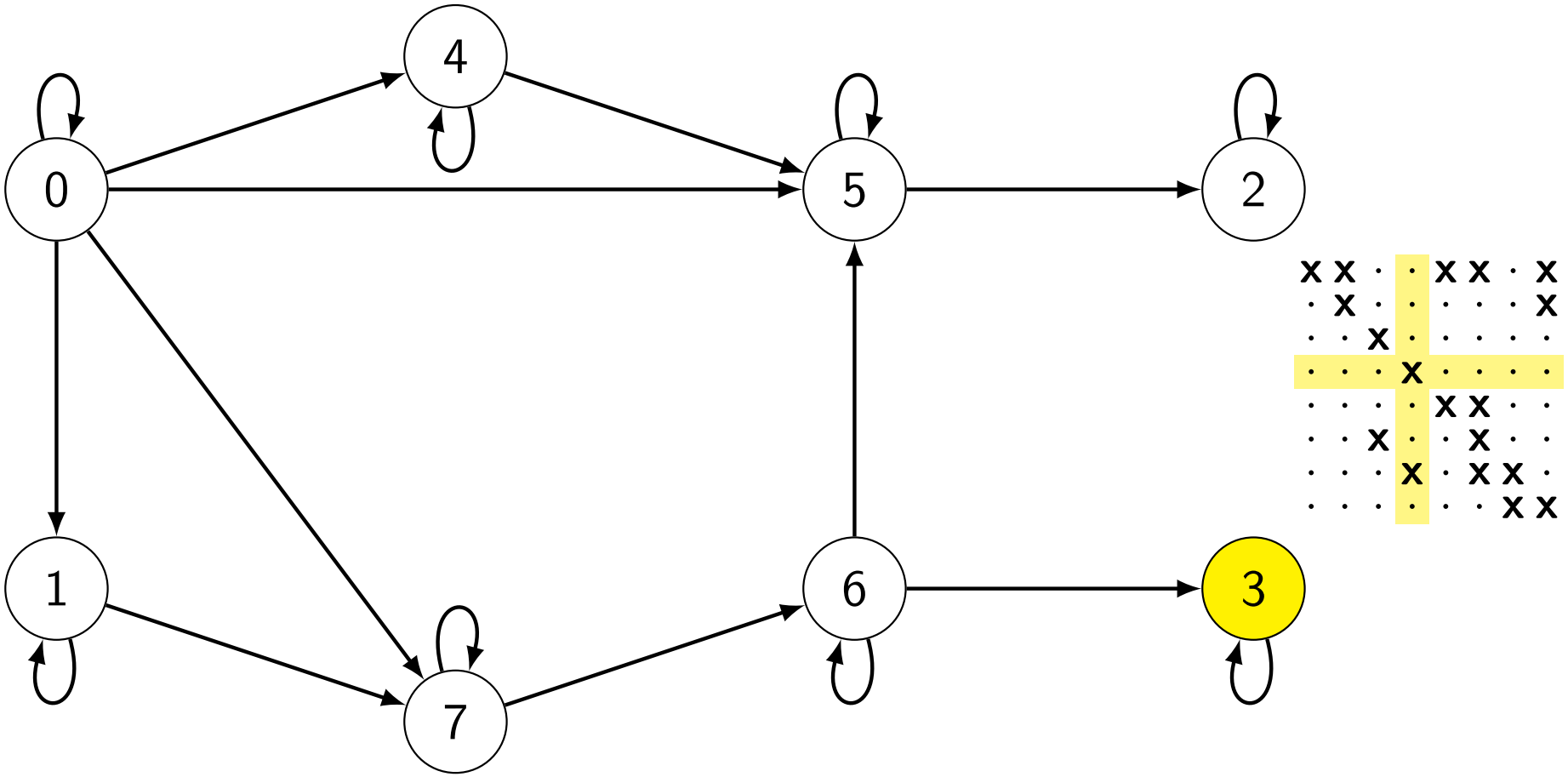




Durch Permutierung der Knoten erhält man die gleiche Matrixdarstellung von R^* wie vorher.

Algorithmus von Warshall: Beispiel

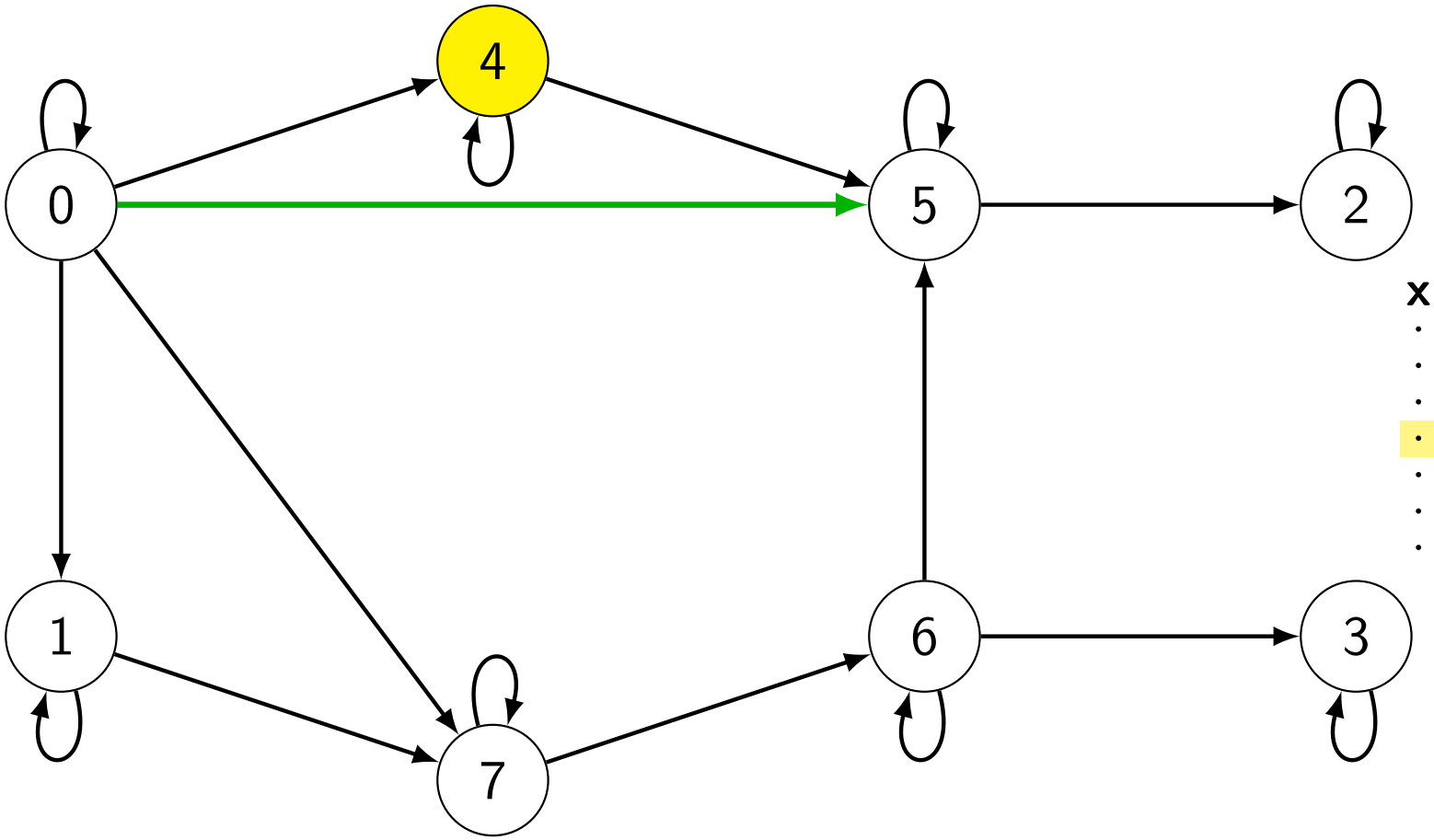
Andere Reihenfolge der Knoten:

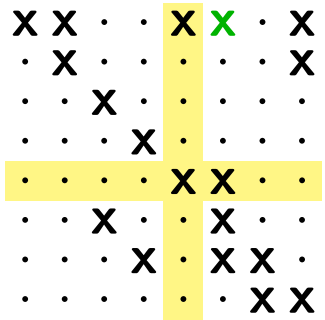


Durch Permutierung der Knoten erhält man die gleiche Matrixdarstellung von R^* wie vorher.

Algorithmus von Warshall: Beispiel

Andere Reihenfolge der Knoten:

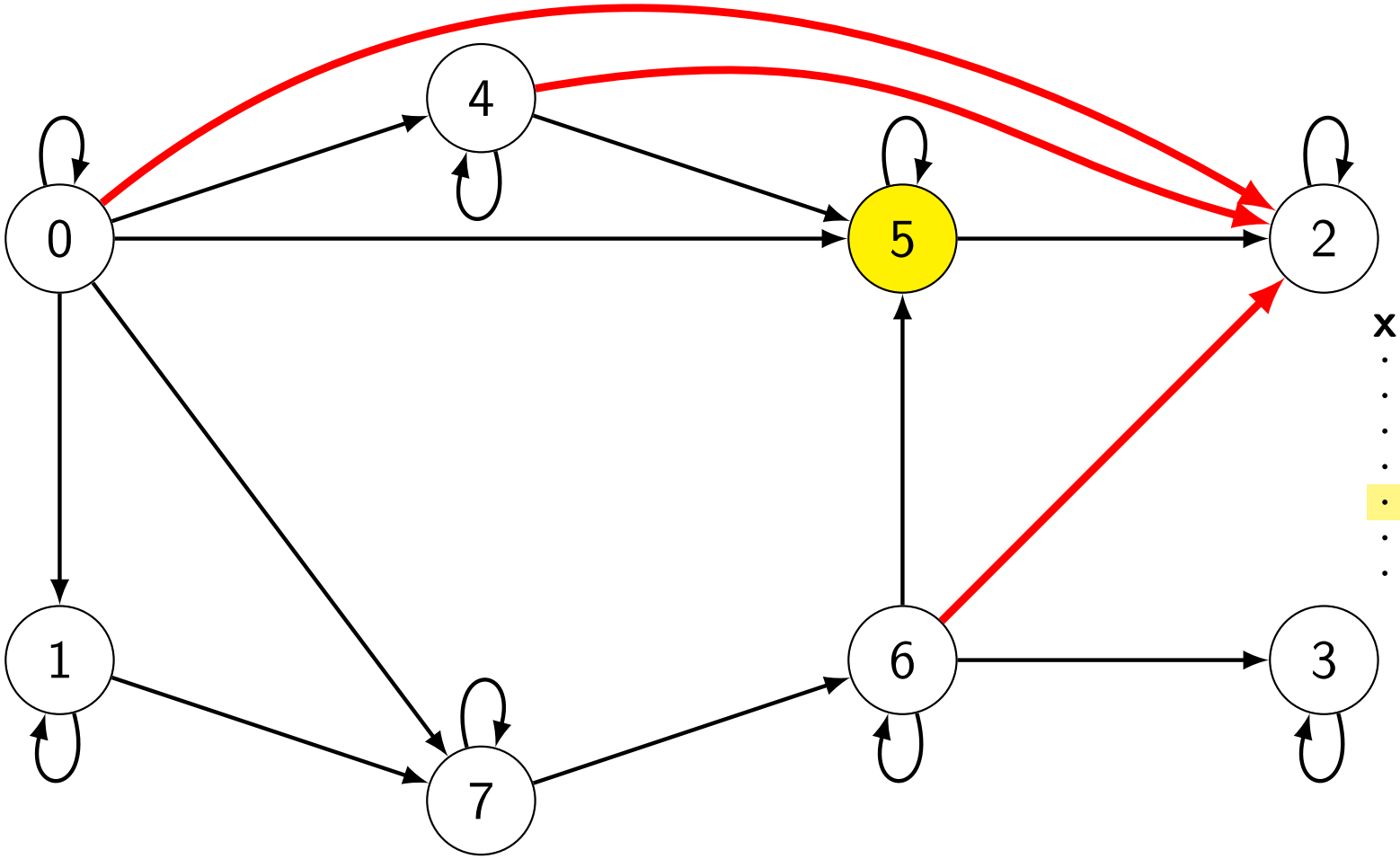


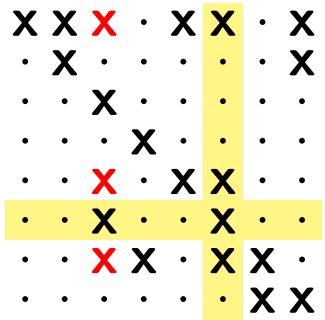


Durch Permutierung der Knoten erhält man die gleiche Matrixdarstellung von R^* wie vorher.

Algorithmus von Warshall: Beispiel

Andere Reihenfolge der Knoten:

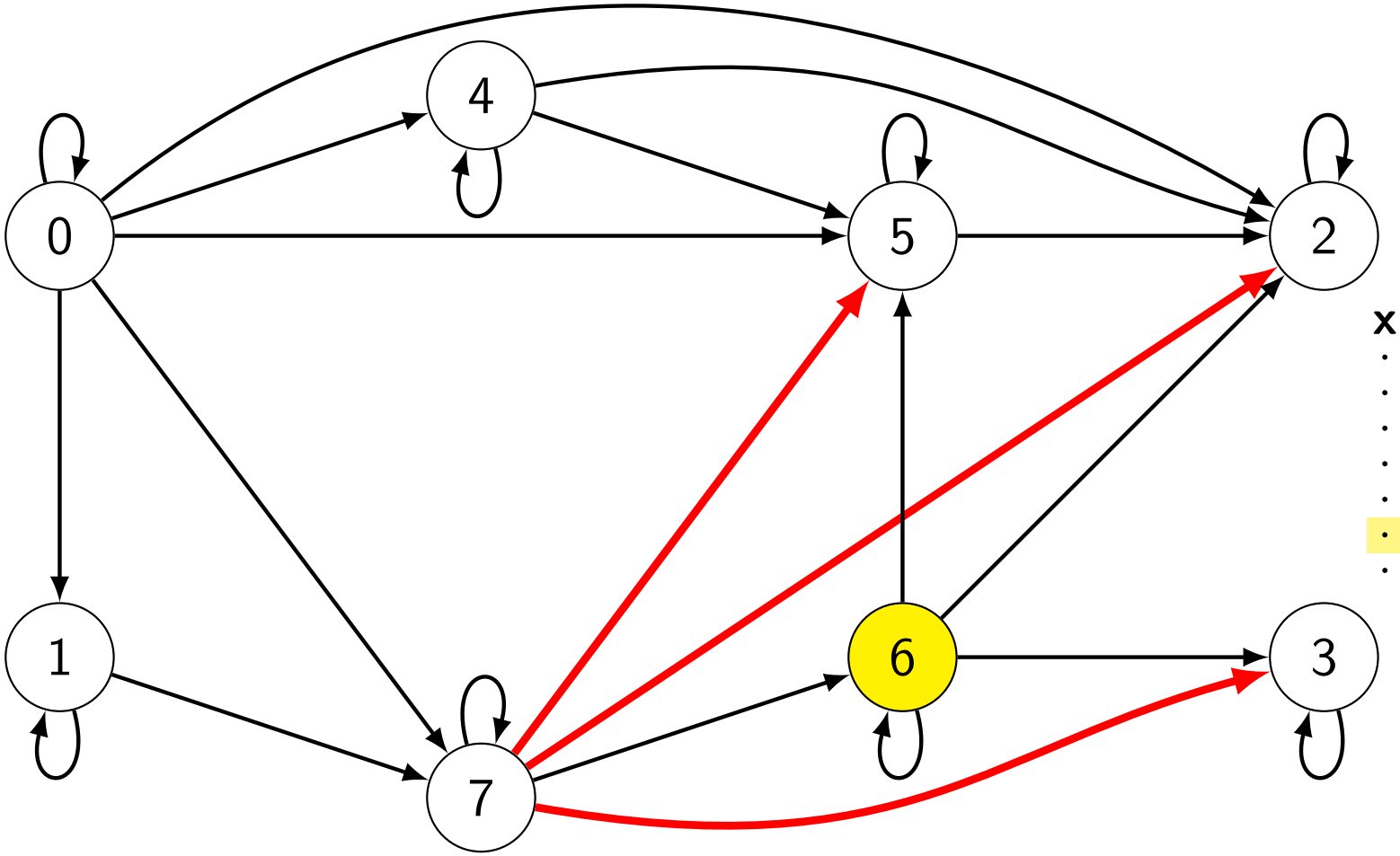


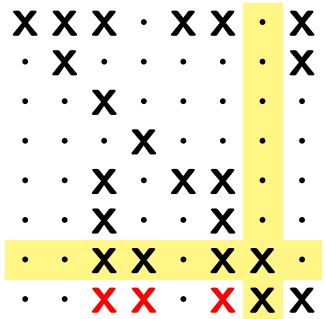


Durch Permutierung der Knoten erhält man die gleiche Matrixdarstellung von R^* wie vorher.

Algorithmus von Warshall: Beispiel

Andere Reihenfolge der Knoten:

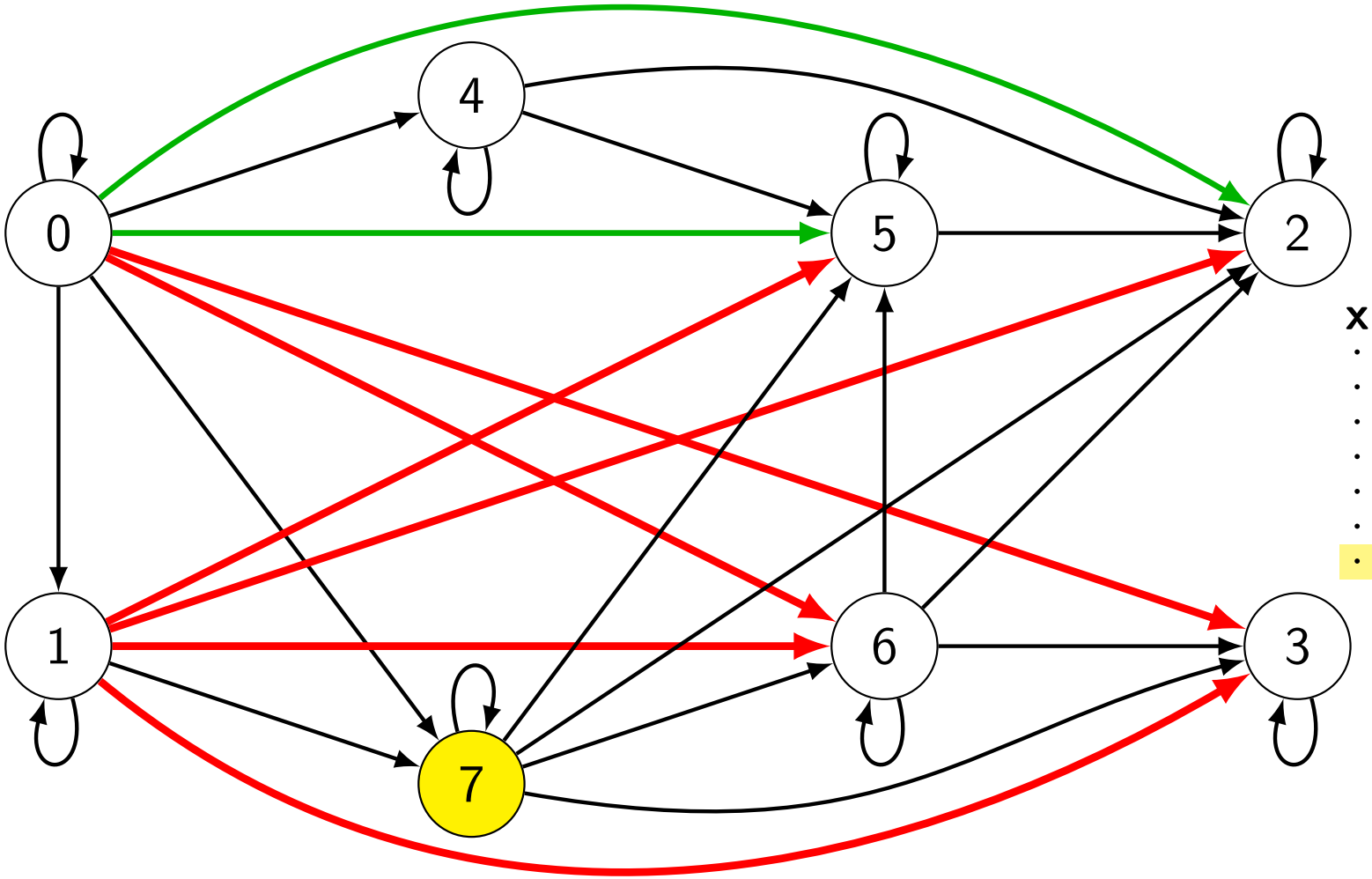


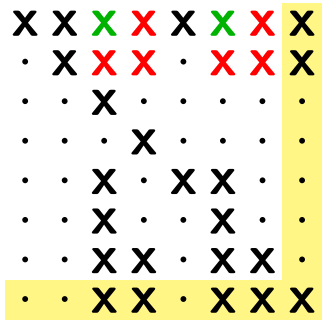


Durch Permutierung der Knoten erhält man die gleiche Matrixdarstellung von R^* wie vorher.

Algorithmus von Warshall: Beispiel

Andere Reihenfolge der Knoten:





Durch Permutierung der Knoten erhält man die gleiche Matrixdarstellung von R^* wie vorher.

Algorithmus von Warshall

```
1 foreach (k ∈ V)
2   foreach (eingehende Kante (i, k) ∈ E)
3     foreach (ausgehende Kante (k, j) ∈ E)
4       Füge (i, j) zu E hinzu.
```

Algorithmus von Warshall

```
1 foreach (k ∈ V)
2   foreach (eingehende Kante (i, k) ∈ E)
3     foreach (ausgehende Kante (k, j) ∈ E)
4       Füge (i, j) zu E hinzu.
```

```
1  void transClos(bool A[n][n], int n, bool &R[n][n]) {
2    for (int i = 0; i < n; i++)
3      for (int j = 0; j < n; j++)
4        R[i,j] = A[i,j]; // Kopiere A nach R
5
6    for (int i = 0; i < n; i++)
7      R[i,i] = true; // reflexive Hülle / reflexiver Abschluss
8
9    for (int k = 0; k < n; k++)
10     for (int i = 0; i < n; i++)
11       for (int j = 0; j < n; j++)
12         R[i,j] = R[i,j] || (R[i,k] && R[k,j]);
13 }
```

Algorithmus von Warshall

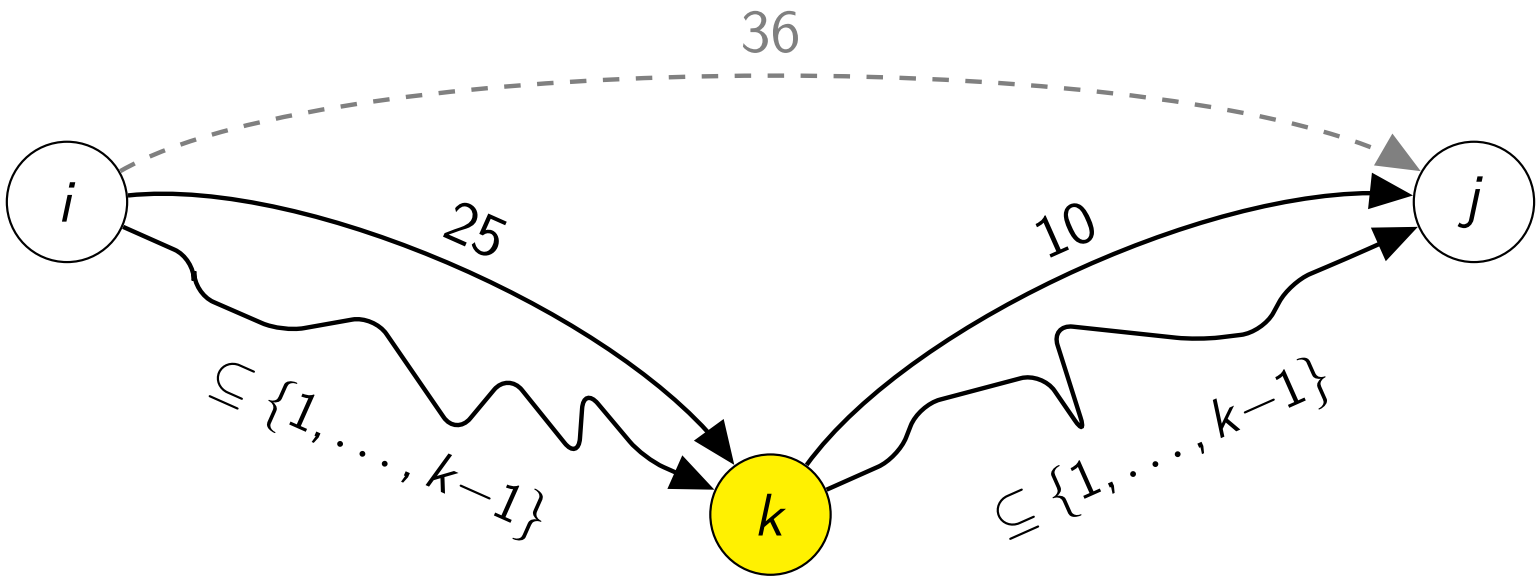
```
1 foreach (k ∈ V)
2   foreach (eingehende Kante (i, k) ∈ E)
3     foreach (ausgehende Kante (k, j) ∈ E)
4       Füge (i, j) zu E hinzu.
```

```
1  void transClos(bool A[n][n], int n, bool &R[n][n]) {
2    for (int i = 0; i < n; i++)
3      for (int j = 0; j < n; j++)
4        R[i,j] = A[i,j]; // Kopiere A nach R
5
6    for (int i = 0; i < n; i++)
7      R[i,i] = true; // reflexive Hülle / reflexiver Abschluss
8
9    for (int k = 0; k < n; k++)
10     for (int i = 0; i < n; i++)
11       for (int j = 0; j < n; j++)
12         R[i,j] = R[i,j] || (R[i,k] && R[k,j]);
13 }
```

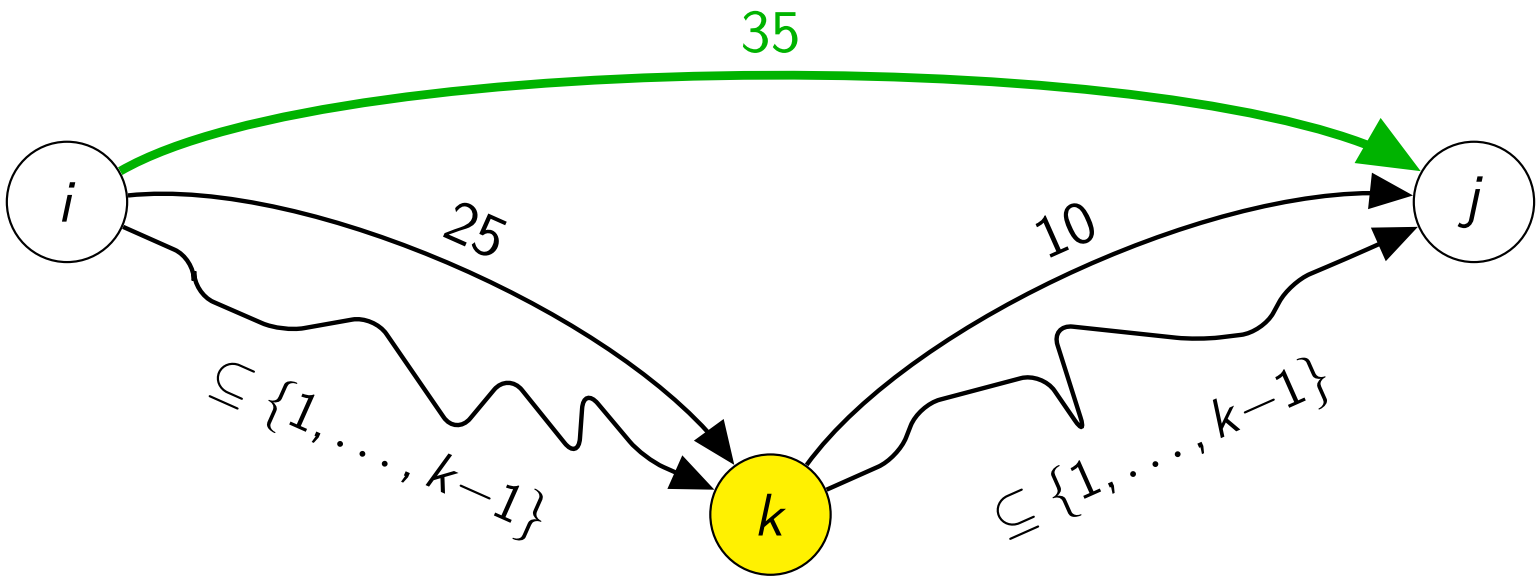
- Zeitkomplexität: $\Theta(|V|^3)$, Platzkomplexität: $\Theta(|V|^2)$.

Übersicht

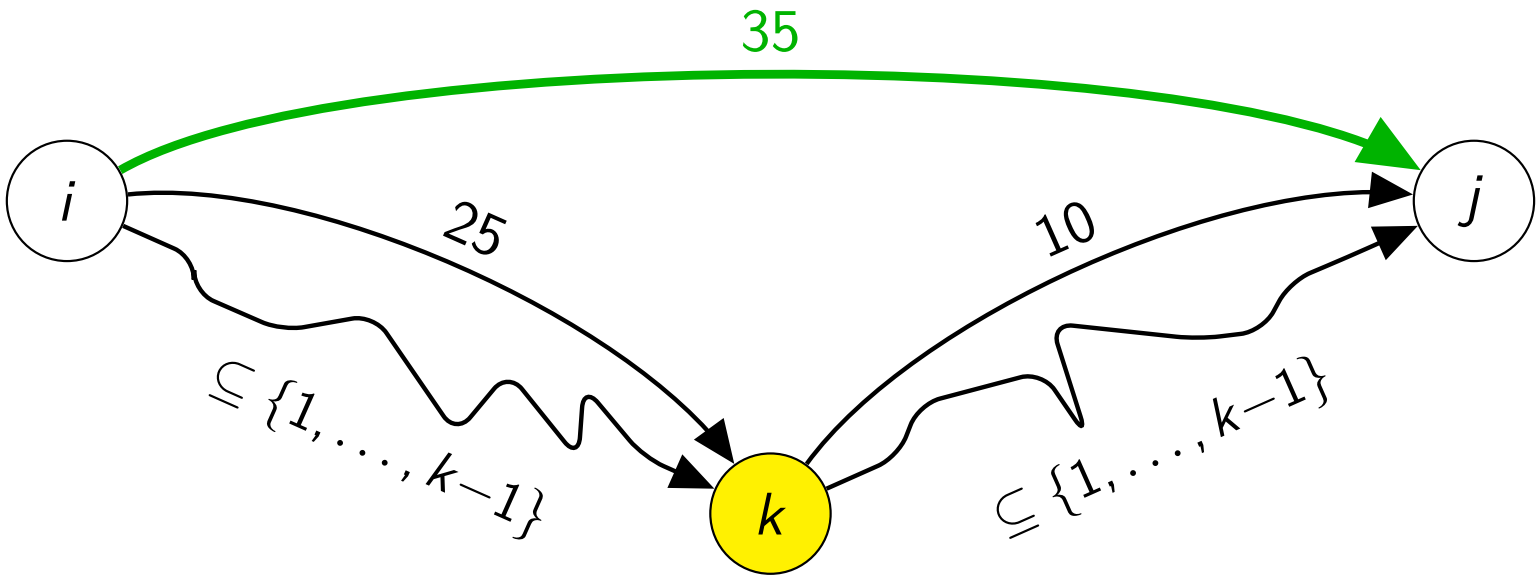
Der Algorithmus von Floyd: Idee



Der Algorithmus von Floyd: Idee

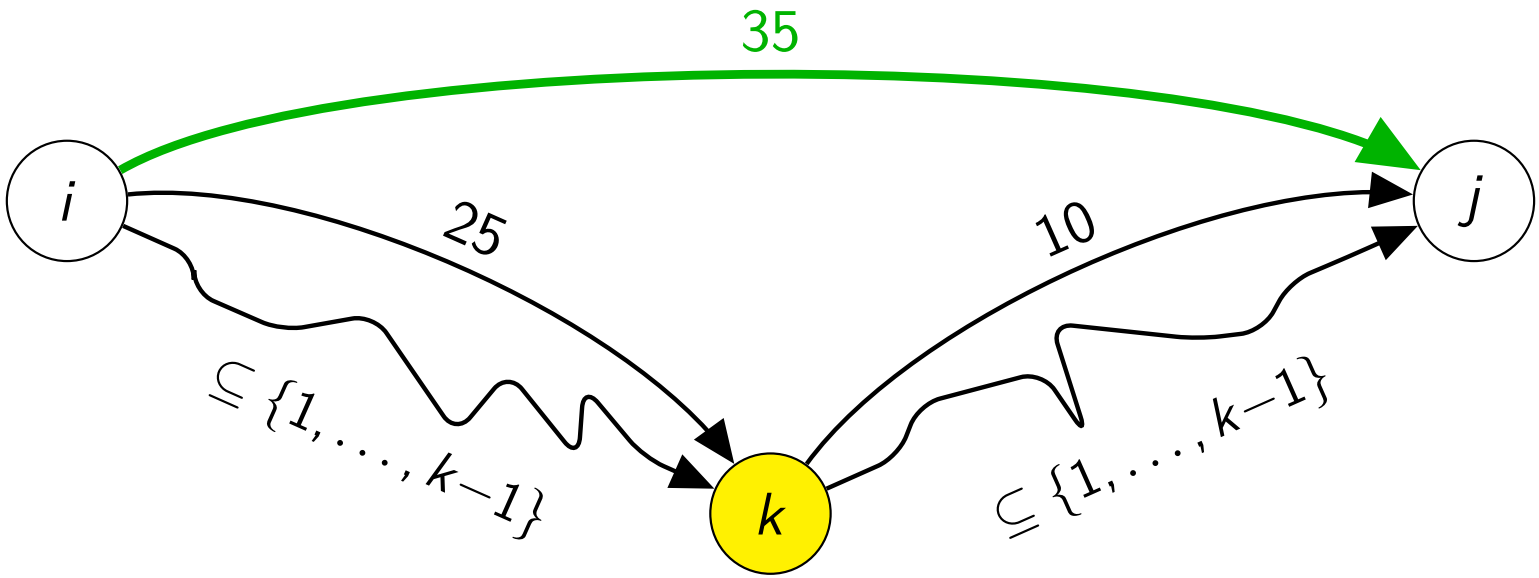


Der Algorithmus von Floyd: Idee



- Vorgehen wie bei Warshall, jedoch mit folgender Rekursionsgleichung:

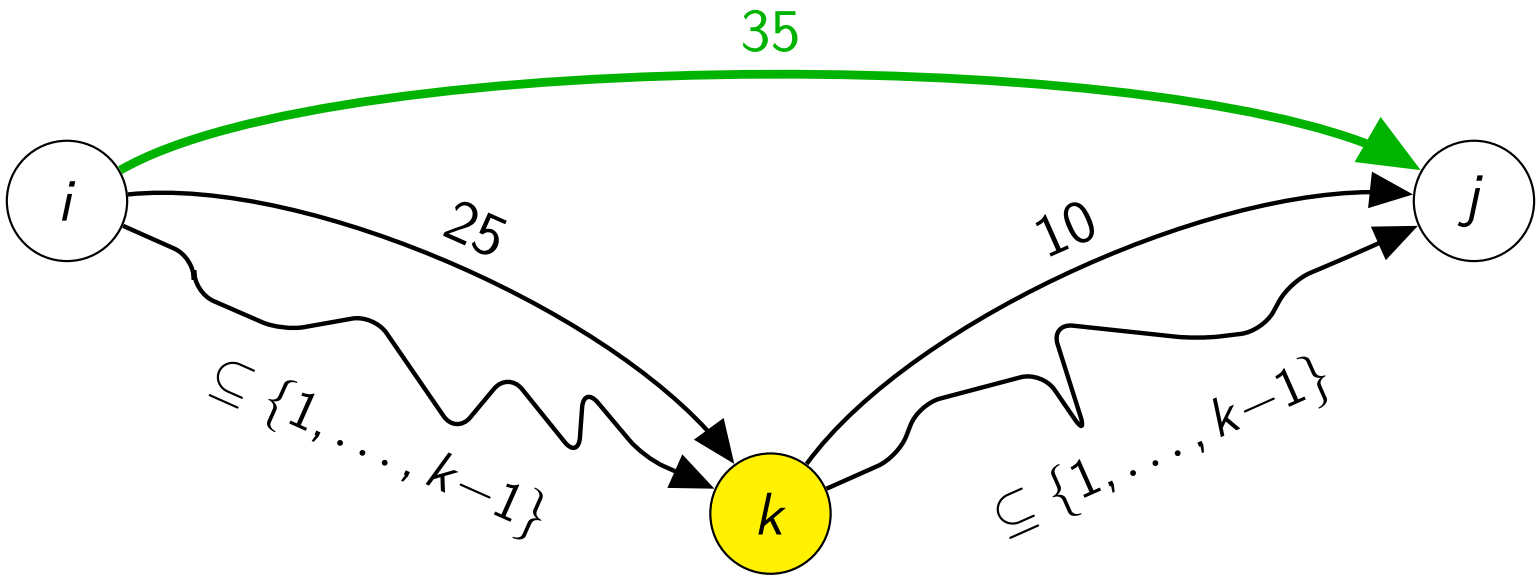
Der Algorithmus von Floyd: Idee



- Vorgehen wie bei Warshall, jedoch mit folgender Rekursionsgleichung:

$$d_{ij}^{(k)} = \begin{cases} W(i,j) & \text{für } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{für } k > 0 \end{cases}$$

Der Algorithmus von Floyd: Idee

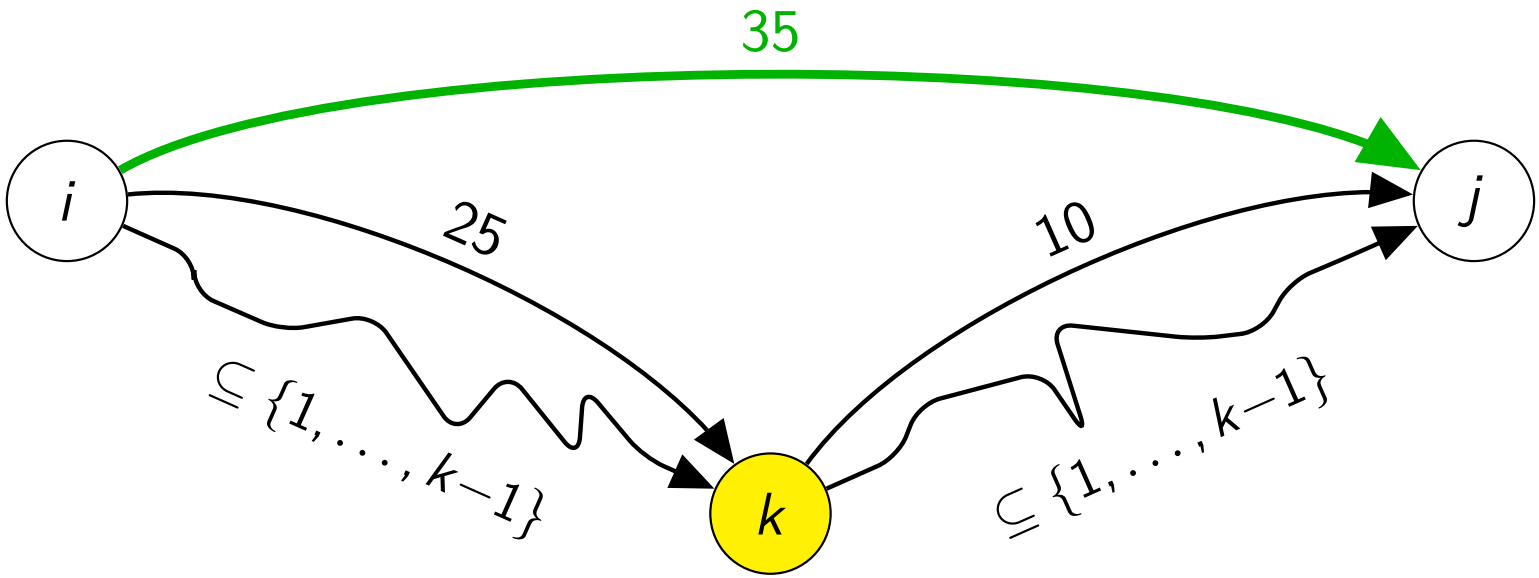


- Vorgehen wie bei Warshall, jedoch mit folgender Rekursionsgleichung:

$$d_{ij}^{(k)} = \begin{cases} W(i,j) & \text{für } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{für } k > 0 \end{cases}$$

$$\text{(statt: } t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)\text{)}$$

Der Algorithmus von Floyd: Idee



- Vorgehen wie bei Warshall, jedoch mit folgender Rekursionsgleichung:

$$d_{ij}^{(k)} = \begin{cases} W(i,j) & \text{für } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{für } k > 0 \end{cases}$$

$$\text{(statt: } t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)\text{)}$$

- Auch hier arbeiten wir direkt im Ausgabearray: `D[i,j]` = $d_{ij}^{(\cdot)}$.

Der Algorithmus von Floyd: Beispiel

https://www.cs.usfca.edu/~galles/visualization/Floyd.html

Der Algorithmus von Floyd: Implementierung

```
1  void floydSP(double W[n][n], int n, double &D[n][n]) {
2    for (int i = 0; i < n; i++)
3      for (int j = 0; j < n; j++)
4        D[i,j] = W[i,j]; // Kopiere W nach D
5
6    for (int i = 0; i < n; i++) // reflexive Hülle
7      D[i,i] = 0;
8
9    for (int k = 0; k < n; k++)
10     for (int i = 0; i < n; i++)
11       for (int j = 0; j < n; j++)
12         D[i,j] = min(D[i,j], D[i,k] + D[k,j]);
13 }
```

Der Algorithmus von Floyd: Implementierung

```
1  void floydSP(double W[n][n], int n, double &D[n][n]) {
2    for (int i = 0; i < n; i++)
3      for (int j = 0; j < n; j++)
4        D[i,j] = W[i,j]; // Kopiere W nach D
5
6    for (int i = 0; i < n; i++) // reflexive Hülle
7      D[i,i] = 0;
8
9    for (int k = 0; k < n; k++)
10     for (int i = 0; i < n; i++)
11       for (int j = 0; j < n; j++)
12         D[i,j] = min(D[i,j], D[i,k] + D[k,j]);
13 }
```

- Zeitkomplexität: $\Theta(|V|^3)$, Platzkomplexität: $\Theta(|V|^2)$.

Der Algorithmus von Floyd: Implementierung

```
1  void floydSP(double W[n][n], int n, double &D[n][n]) {
2    for (int i = 0; i < n; i++)
3      for (int j = 0; j < n; j++)
4        D[i,j] = W[i,j]; // Kopiere W nach D
5
6    for (int i = 0; i < n; i++) // reflexive Hülle
7      D[i,i] = 0;
8
9    for (int k = 0; k < n; k++)
10     for (int i = 0; i < n; i++)
11       for (int j = 0; j < n; j++)
12         D[i,j] = min(D[i,j], D[i,k] + D[k,j]);
13 }
```

- Zeitkomplexität: $\Theta(|V|^3)$, Platzkomplexität: $\Theta(|V|^2)$.
- Hier nicht behandelt: Der Algorithmus kann auch mit negativen Zyklen umgehen.

Der Algorithmus von Floyd: Erweiterung

- Der Algorithmus lässt sich einfach auf die Rückgabe von Pfaden erweitern (z. B. Routingtabellen).

Der Algorithmus von Floyd: Erweiterung

- Der Algorithmus lässt sich einfach auf die Rückgabe von Pfaden erweitern (z. B. Routingtabellen).
- Dazu speichere zu jedem Paar i, j jeweils den letzten Zwischenknoten des kürzesten Pfades in π_{ij} (den Vorgänger von j).

Der Algorithmus von Floyd: Erweiterung

- Der Algorithmus lässt sich einfach auf die Rückgabe von Pfaden erweitern (z. B. Routingtabellen).
- Dazu speichere zu jedem Paar i, j jeweils den letzten Zwischenknoten des kürzesten Pfades in π_{ij} (den Vorgänger von j).

$$\pi_{ij}^{(k)} = \begin{cases} i & \text{für } k = 0, i \neq j, \text{ falls } W(i,j) \neq +\texttt{inf} \end{cases}$$

Der Algorithmus von Floyd: Erweiterung

- Der Algorithmus lässt sich einfach auf die Rückgabe von Pfaden erweitern (z. B. Routingtabellen).
- Dazu speichere zu jedem Paar i, j jeweils den letzten Zwischenknoten des kürzesten Pfades in π_{ij} (den Vorgänger von j).

$$\pi_{ij}^{(k)} = \begin{cases} i & \text{für } k = 0, i \neq j, \text{ falls } W(i,j) \neq +\texttt{inf} \\ \texttt{null} & \text{für } k = 0, \text{ sonst} \end{cases}$$

Der Algorithmus von Floyd: Erweiterung

- Der Algorithmus lässt sich einfach auf die Rückgabe von Pfaden erweitern (z. B. Routingtabellen).
- Dazu speichere zu jedem Paar i, j jeweils den letzten Zwischenknoten des kürzesten Pfades in π_{ij} (den Vorgänger von j).

$$\pi_{ij}^{(k)} = \begin{cases} i & \text{für } k = 0, i \neq j, \text{ falls } W(i,j) \neq +\texttt{inf} \\ \texttt{null} & \text{für } k = 0, \text{ sonst} \\ \pi_{kj}^{(k-1)} & \text{für } k > 0, \text{ falls } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Der Algorithmus von Floyd: Erweiterung

- Der Algorithmus lässt sich einfach auf die Rückgabe von Pfaden erweitern (z. B. Routingtabellen).
- Dazu speichere zu jedem Paar i, j jeweils den letzten Zwischenknoten des kürzesten Pfades in π_{ij} (den Vorgänger von j).

$$\pi_{ij}^{(k)} = \begin{cases} i & \text{für } k = 0, i \neq j, \text{ falls } W(i,j) \neq +\texttt{inf} \\ \texttt{null} & \text{für } k = 0, \text{ sonst} \\ \pi_{kj}^{(k-1)} & \text{für } k > 0, \text{ falls } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{ij}^{(k-1)} & \text{sonst} \end{cases}$$