Datenstrukturen und Algorithmen

Vorlesung 17: Kürzeste Pfadalgorithmen

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

http://www-i2.informatik.rwth-aachen.de/i2/dsal12/

22. Juni 2012

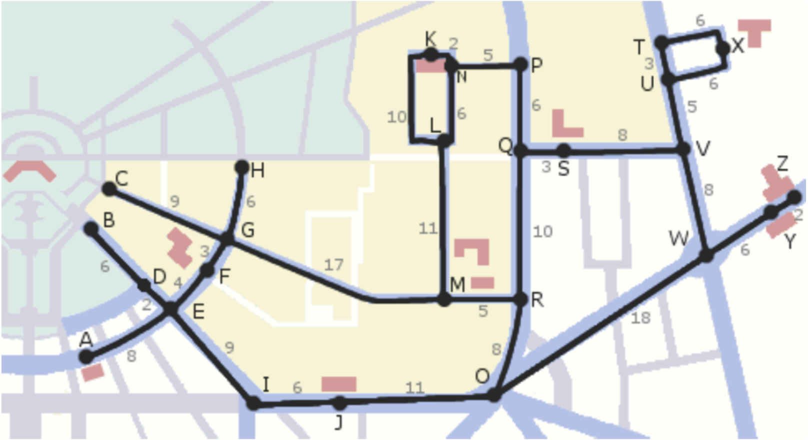
Übersicht

1. Kürzeste Pfade
2. Bellman-Ford
3. Dijkstra

Übersicht

Andere Rechenprobleme: kürzester Weg

Andere Rechenprobleme: kürzester Weg



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

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 (nächste Vorlesung).

Single-Source Shortest Paths

Gegeben sei ein gewichteter (gerichteter oder ungerichteter) Graph G.

Single-Source Shortest Paths

Gegeben sei ein gewichteter (gerichteter oder ungerichteter) Graph G.

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

Single-Source Shortest Paths

Gegeben sei ein gewichteter (gerichteter oder ungerichteter) Graph G.

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

Problem (Single-Source Shortest Path)

Finde den Weg mit minimalem Gewicht, von Knoten s (Quelle / source) aus zu jedem anderen Knoten aus G.

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 bestimmt der Algorithmus die kürzesten Pfade mit ihren Gewichten.

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 bestimmt der Algorithmus die kürzesten Pfade mit ihren Gewichten.
- Er berechnet (iterativ) Schätzungen `d[v]` für die Gewichte des kürzesten Pfades vom Startknoten nach v.

Bellman-Ford – Idee

- Wir wollen die Abstände aller Knoten $\mathtt{v} \in V$ zum Startknoten `start` bestimmen.

Bellman-Ford – Idee

- Wir wollen die Abstände aller Knoten `v` $\in V$ zum Startknoten `start` bestimmen.
- Dazu initialisieren wir `d[v]` mit ∞, sowie `d[start]=0`.

Bellman-Ford – Idee

- Wir wollen die Abstände aller Knoten `v` $\in V$ zum Startknoten `start` bestimmen.
- Dazu initialisieren wir `d[v]` mit ∞, sowie `d[start]=0`.
- Für alle Kanten $(v, w) \in E$ mit Gewicht $W(v, w)$:
  - Ist der bisher bekannte Abstand `d[w]` größer als `d[v]+W(v,w)`,

Bellman-Ford – Idee

- Wir wollen die Abstände aller Knoten `v` $\in V$ zum Startknoten `start` bestimmen.
- Dazu initialisieren wir `d[v]` mit ∞, sowie `d[start]=0`.
- Für alle Kanten $(v, w) \in E$ mit Gewicht $W(v, w)$:
  - Ist der bisher bekannte Abstand `d[w]` größer als `d[v]+W(v,w)`, dann verbessere `d[w]` auf diesen Wert (Relaxierung).

Bellman-Ford – Idee

- Wir wollen die Abstände aller Knoten `v` $\in V$ zum Startknoten `start` bestimmen.
- Dazu initialisieren wir `d[v]` mit ∞, sowie `d[start]=0`.
- Für alle Kanten $(v, w) \in E$ mit Gewicht $W(v, w)$:
  - Ist der bisher bekannte Abstand `d[w]` größer als `d[v]+W(v,w)`, dann verbessere `d[w]` auf diesen Wert (Relaxierung).
- Wiederhole vorigen Schritt, bis sich nichts mehr ändert, bzw. breche ab, falls es einen negativen Zyklus gibt.

Bellman-Ford – Idee

- Wir wollen die Abstände aller Knoten `v` $\in V$ zum Startknoten `start` bestimmen.
- Dazu initialisieren wir `d[v]` mit ∞, sowie `d[start]=0`.
- Für alle Kanten $(v, w) \in E$ mit Gewicht $W(v, w)$:
  - Ist der bisher bekannte Abstand `d[w]` größer als `d[v]+W(v,w)`, dann verbessere `d[w]` auf diesen Wert (Relaxierung).
- Wiederhole vorigen Schritt, bis sich nichts mehr ändert, bzw. breche ab, falls es einen negativen Zyklus gibt.

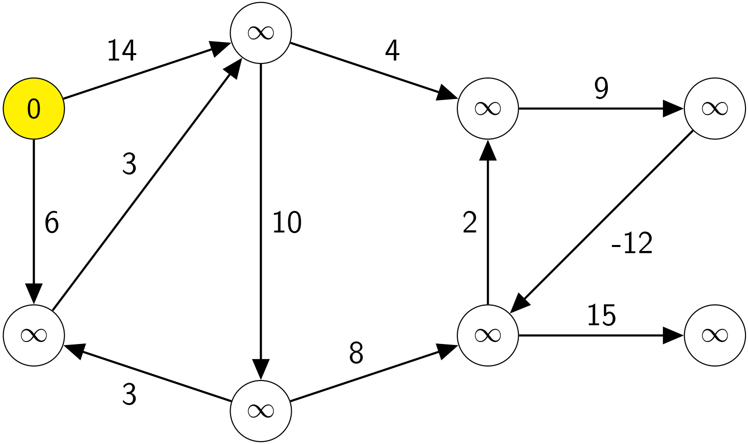
Theorem

Wenn nach $|V|-1$ Wiederholungen noch Verbesserungen möglich sind, dann gibt es einen negativen Zyklus. Andernfalls enthält `d[v]` *für alle* `v` *den kürzesten Abstand zum Startknoten.*

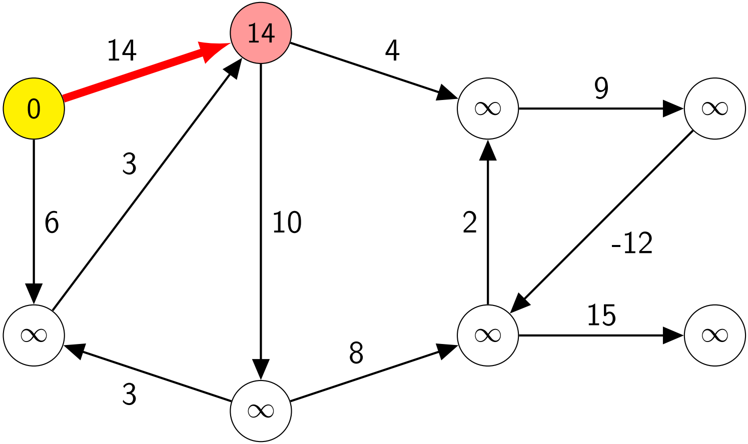
Beweisidee:

- Ein Pfad in (V, E) kann höchstens die Länge $|V|-1$ haben.

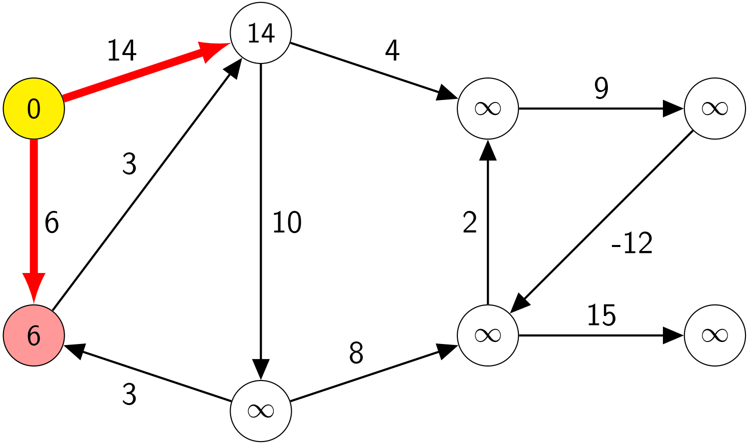
Bellman-Ford – Beispiel



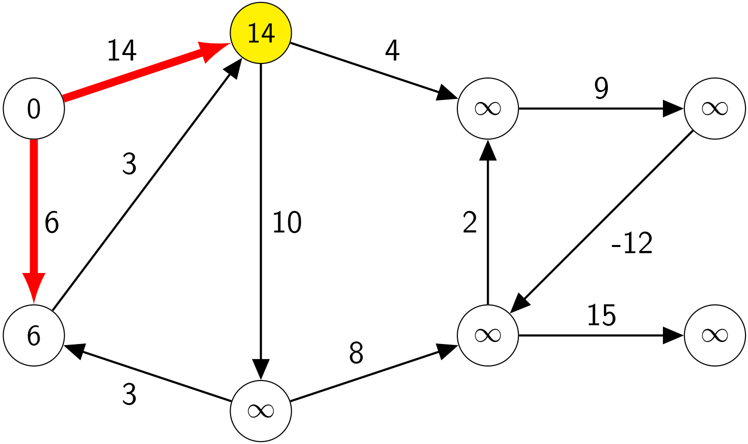
Bellman-Ford – Beispiel



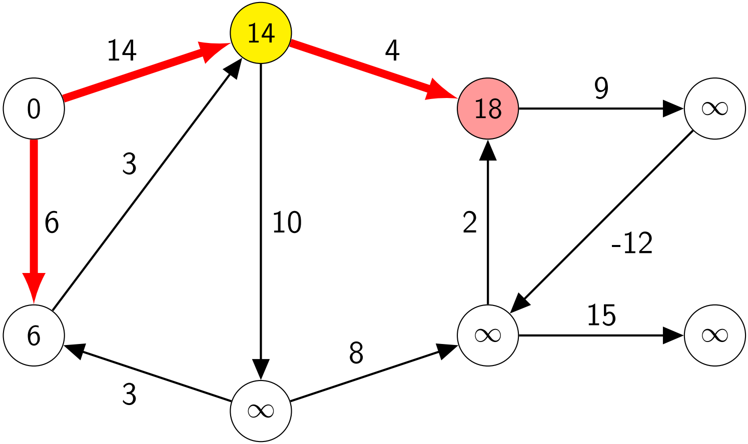
Bellman-Ford – Beispiel



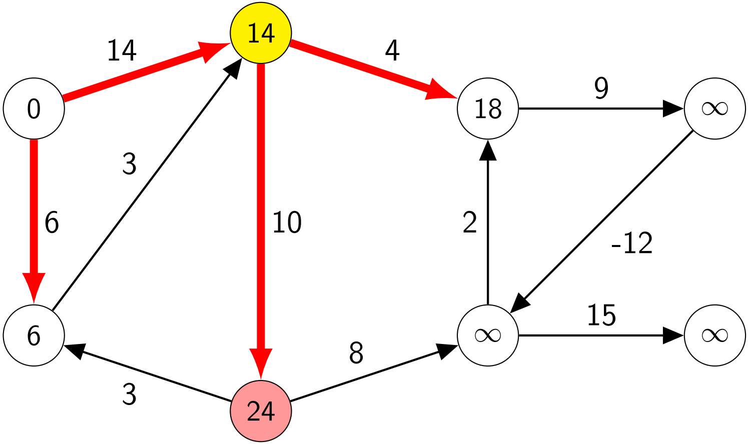
Bellman-Ford – Beispiel



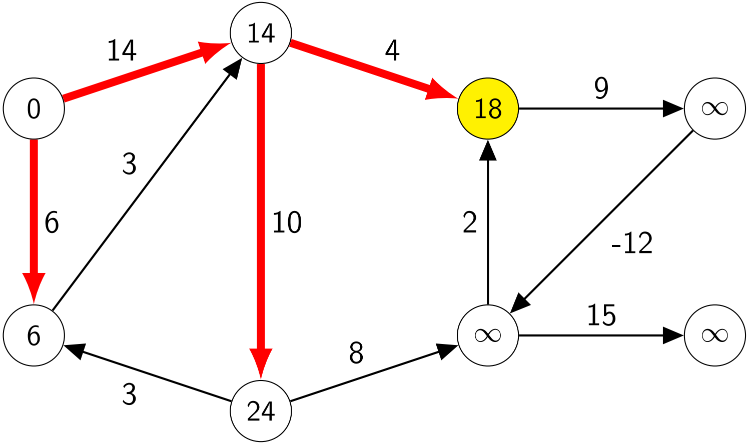
Bellman-Ford – Beispiel



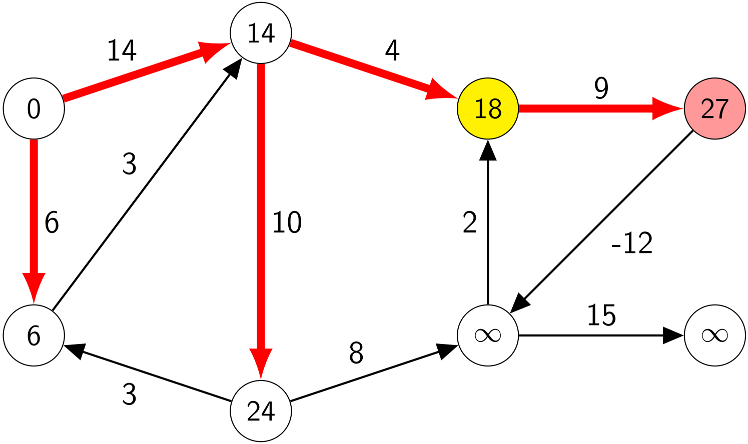
Bellman-Ford – Beispiel



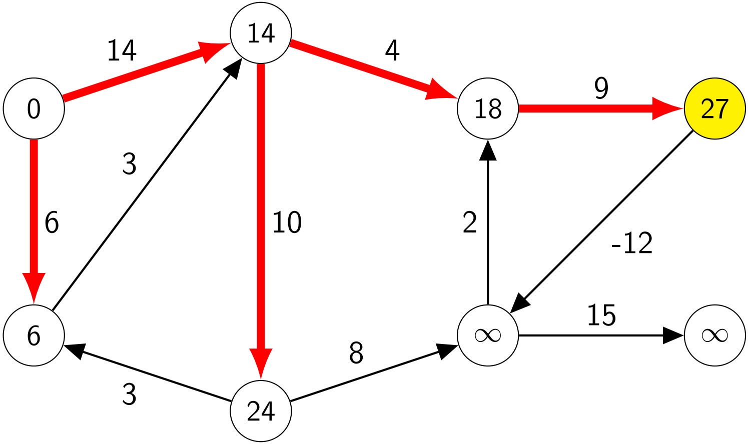
Bellman-Ford – Beispiel



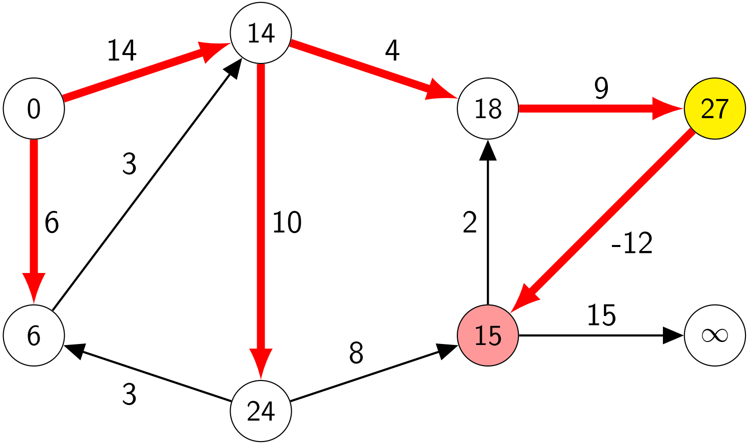
Bellman-Ford – Beispiel



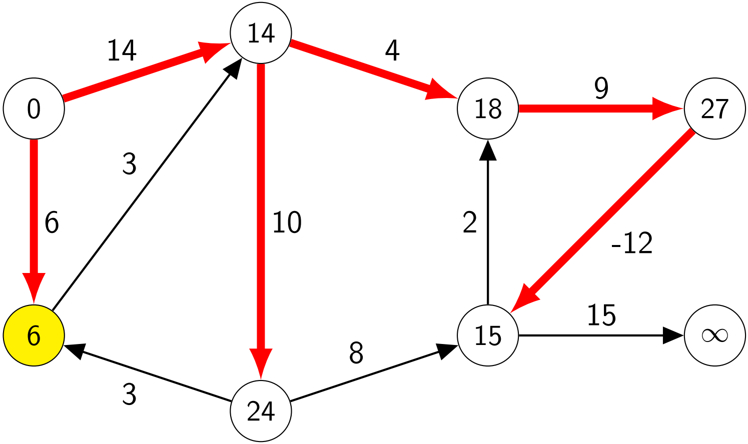
Bellman-Ford – Beispiel



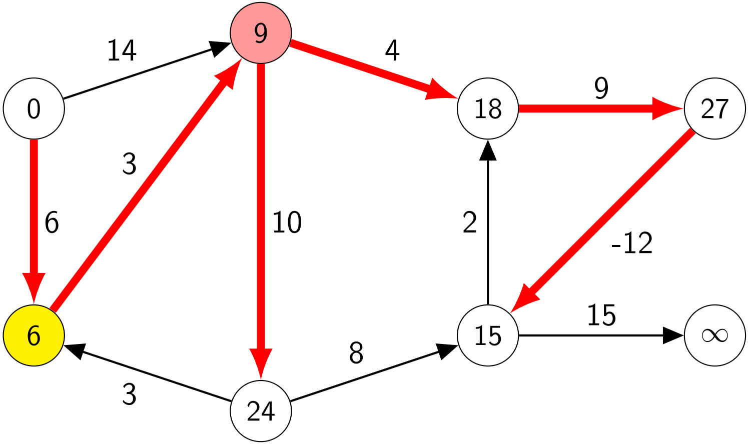
Bellman-Ford – Beispiel



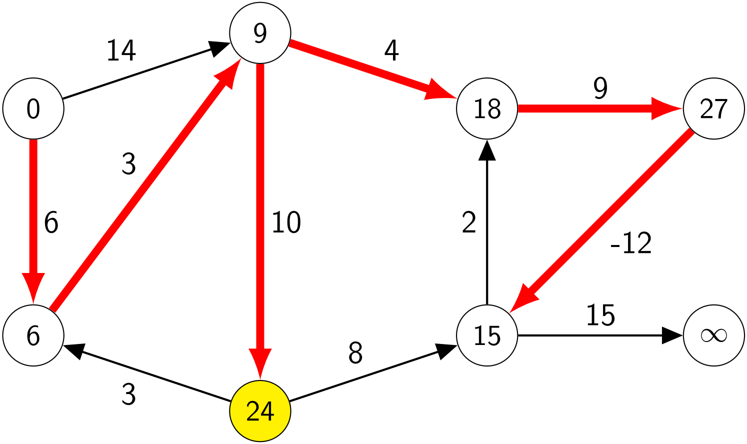
Bellman-Ford – Beispiel



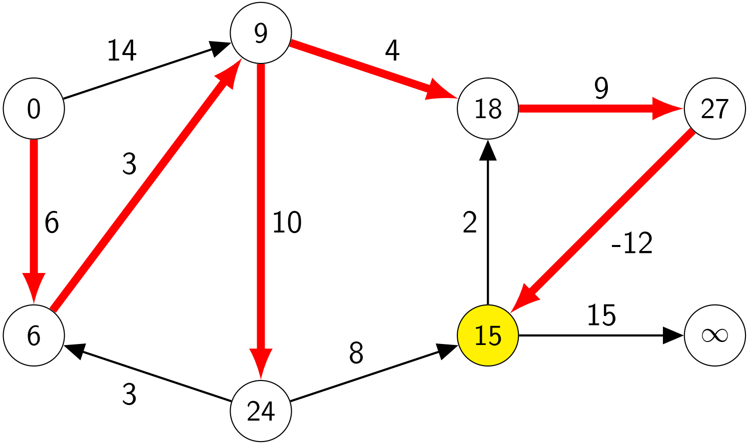
Bellman-Ford – Beispiel



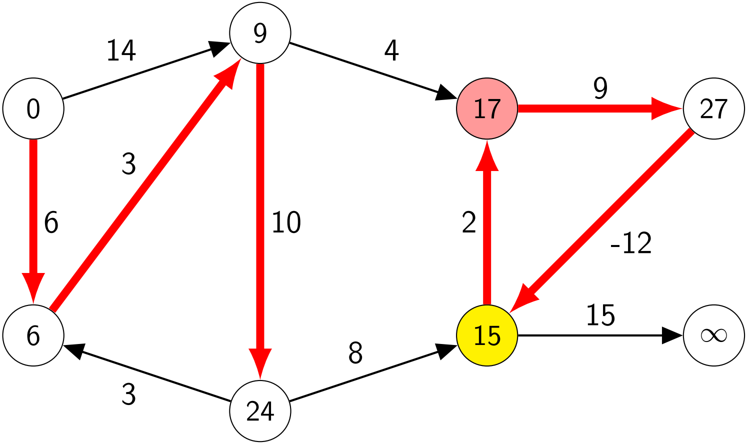
Bellman-Ford – Beispiel



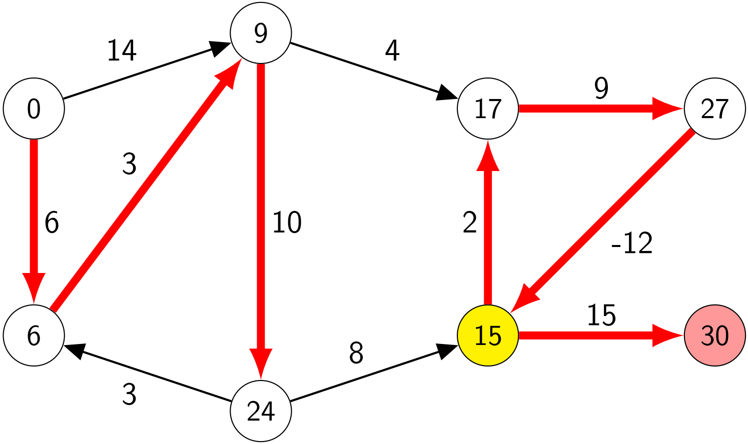
Bellman-Ford – Beispiel



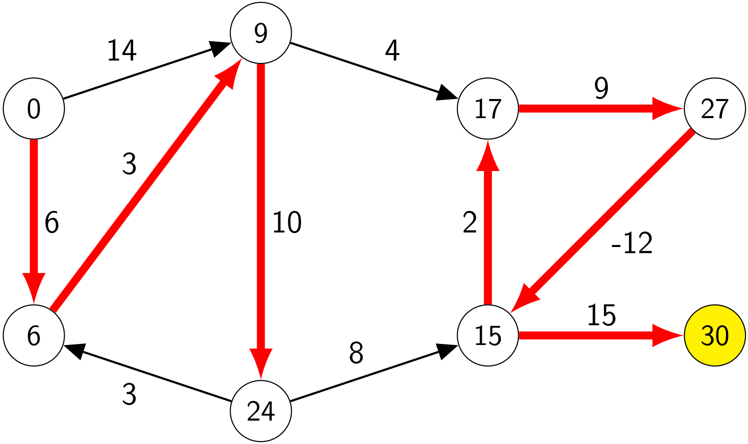
Bellman-Ford – Beispiel



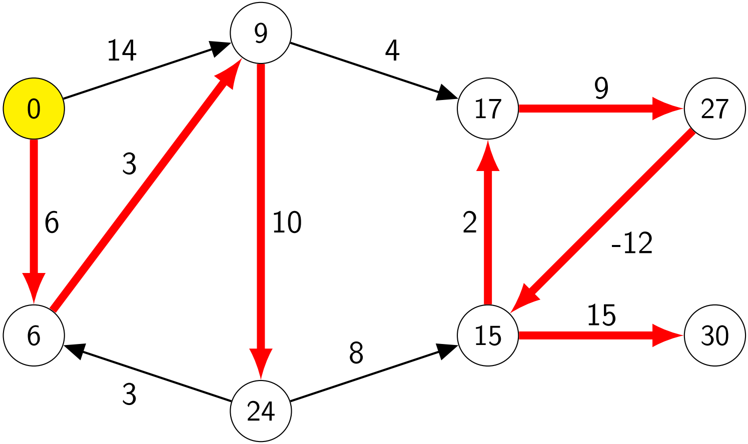
Bellman-Ford – Beispiel



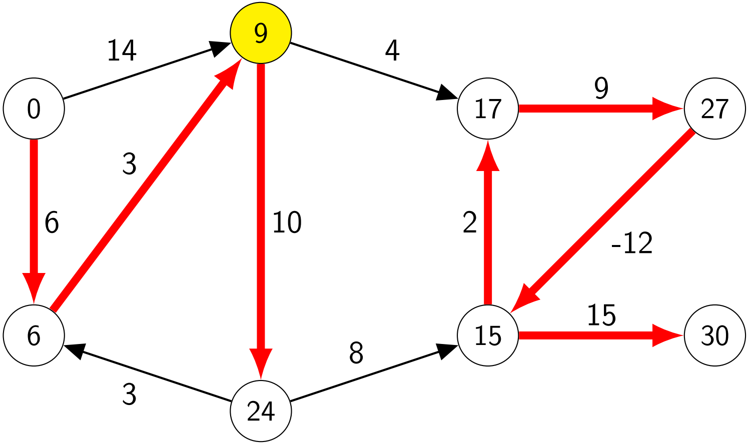
Bellman-Ford – Beispiel



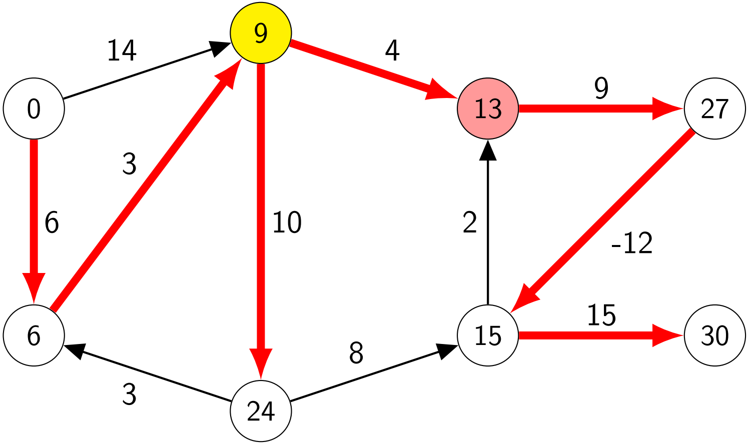
Bellman-Ford – Beispiel



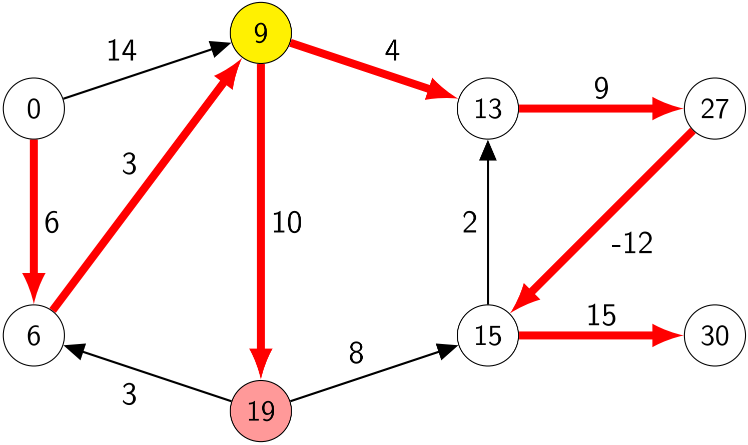
Bellman-Ford – Beispiel



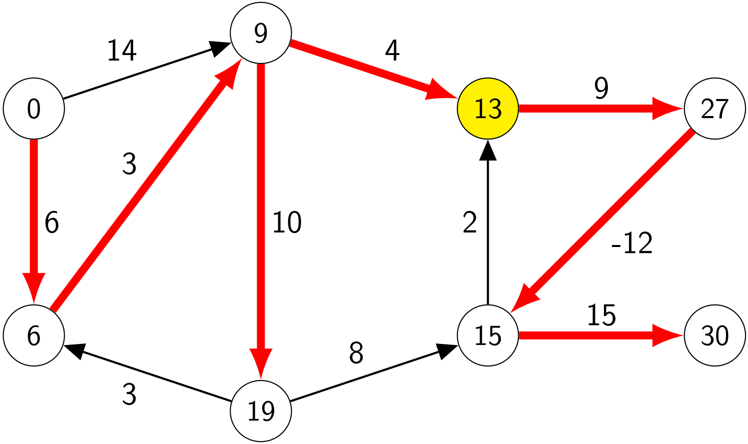
Bellman-Ford – Beispiel



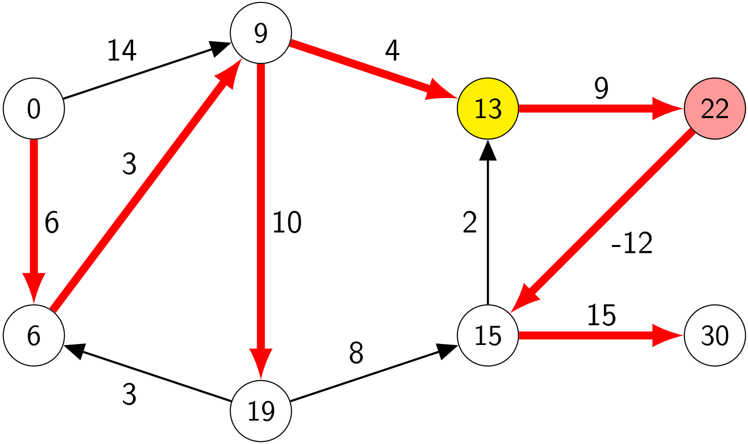
Bellman-Ford – Beispiel



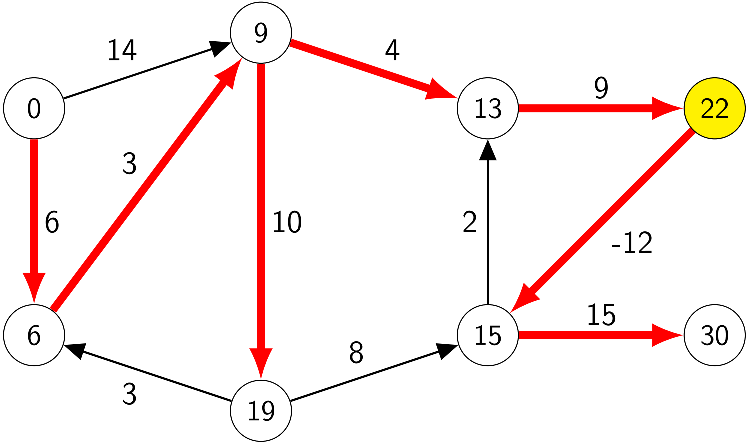
Bellman-Ford – Beispiel



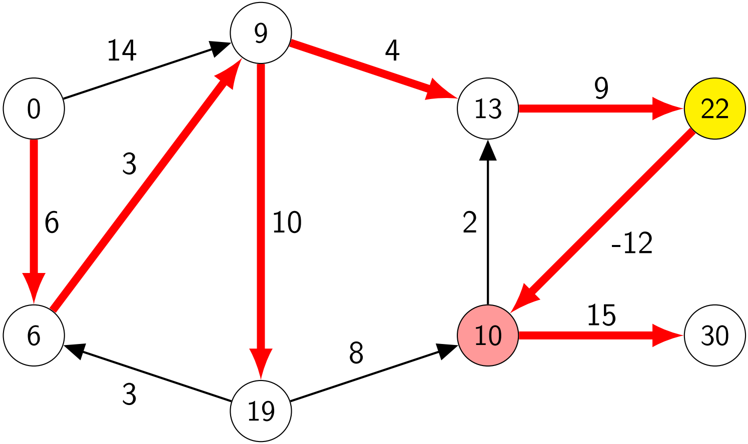
Bellman-Ford – Beispiel



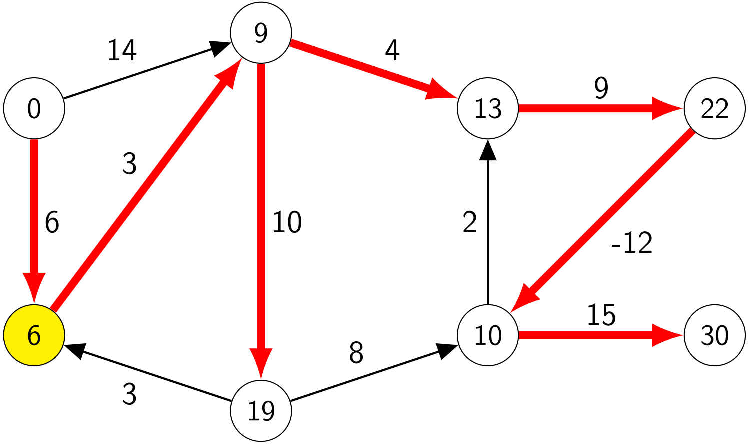
Bellman-Ford – Beispiel



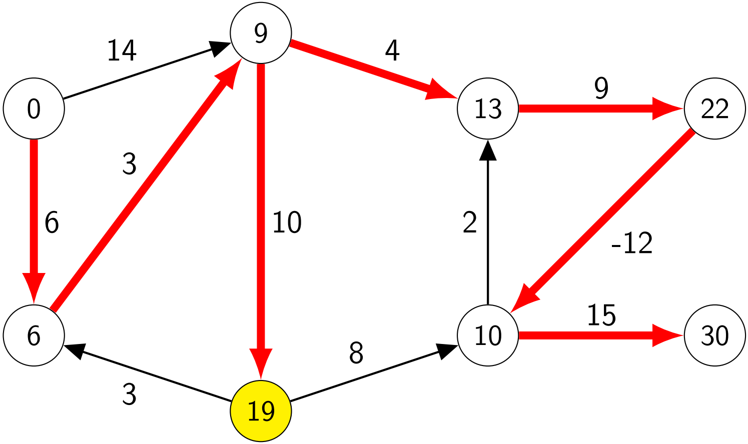
Bellman-Ford – Beispiel



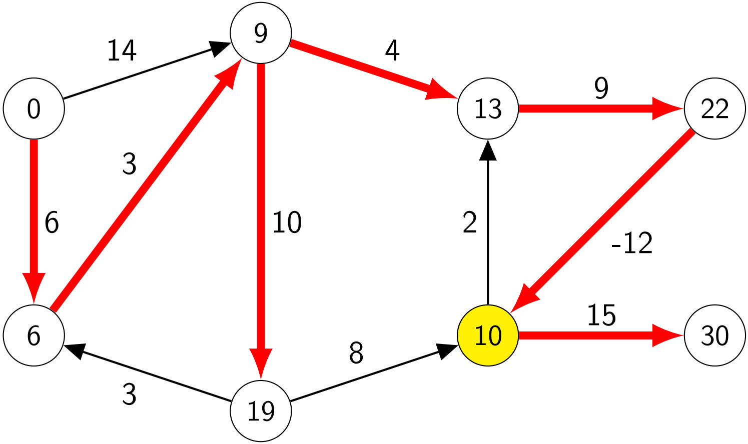
Bellman-Ford – Beispiel



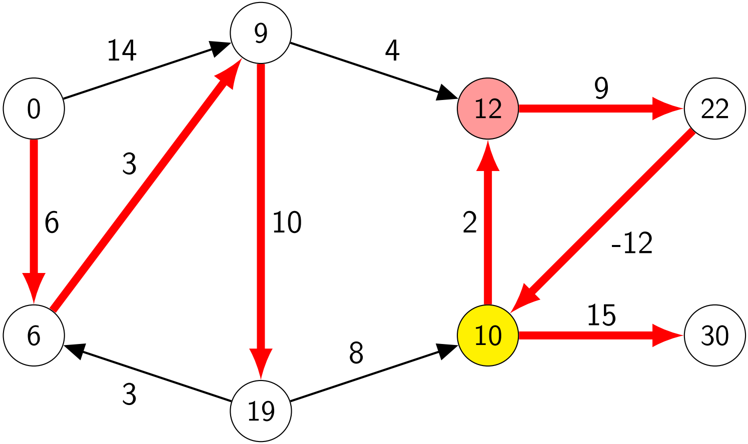
Bellman-Ford – Beispiel



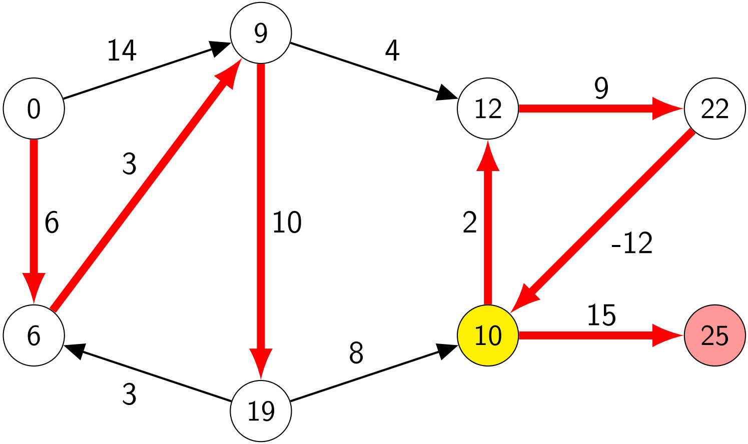
Bellman-Ford – Beispiel



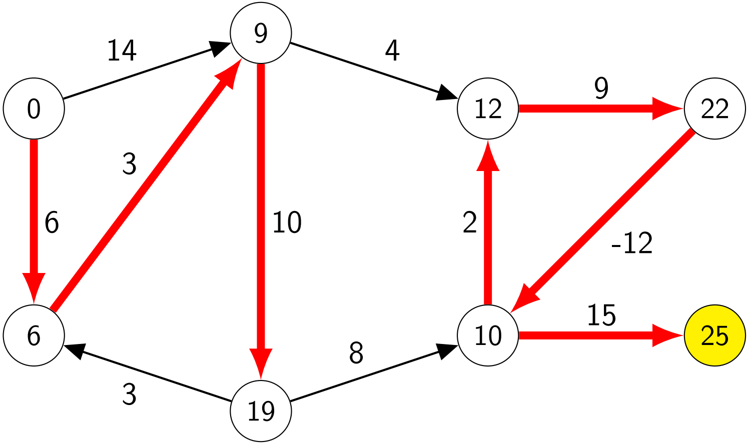
Bellman-Ford – Beispiel



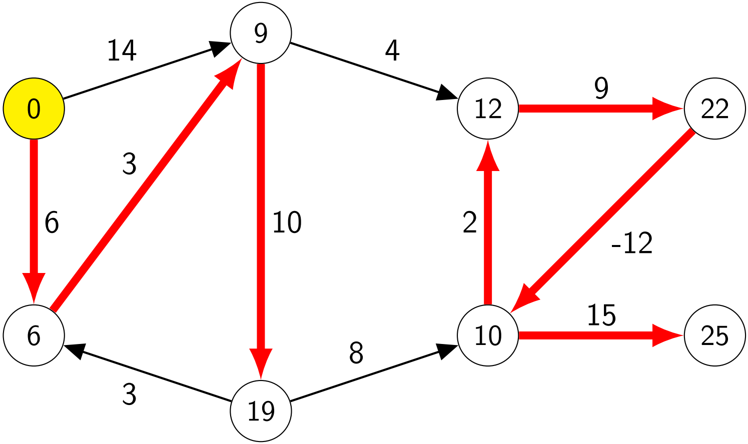
Bellman-Ford – Beispiel



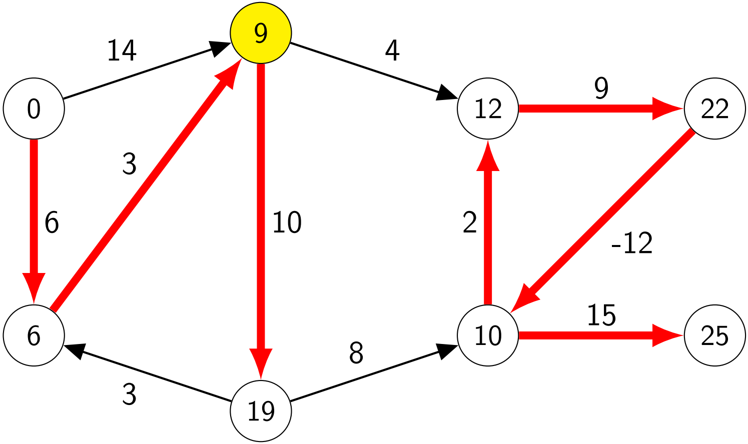
Bellman-Ford – Beispiel



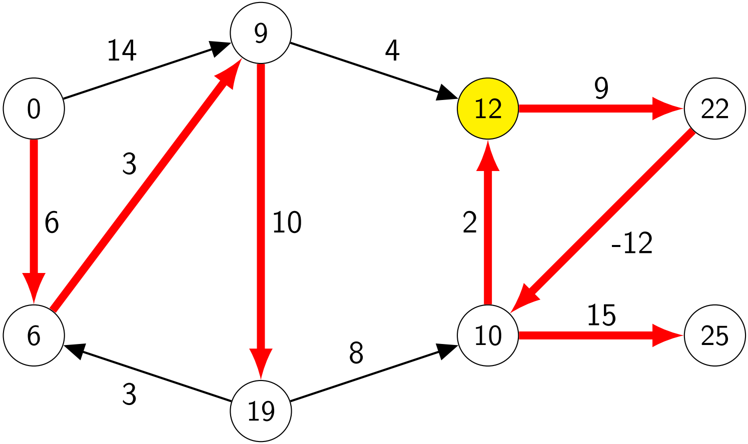
Bellman-Ford – Beispiel



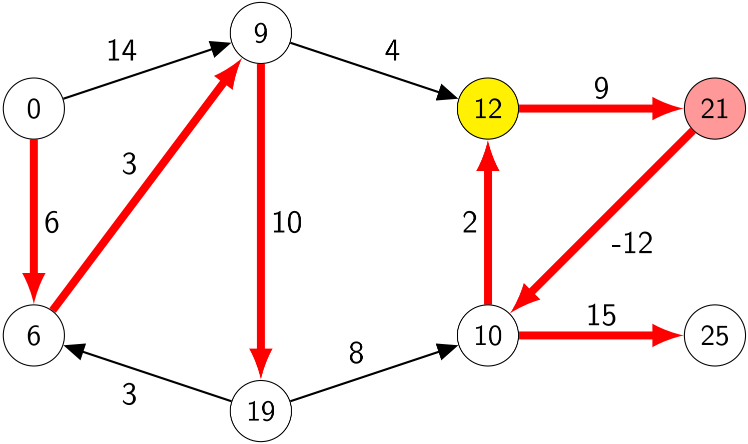
Bellman-Ford – Beispiel



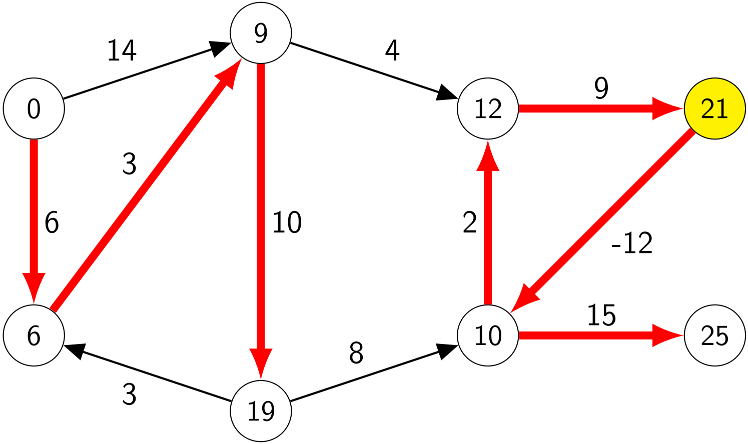
Bellman-Ford – Beispiel



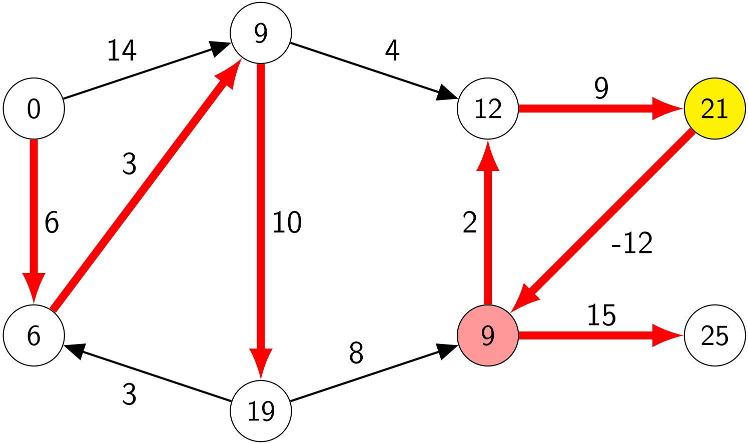
Bellman-Ford – Beispiel



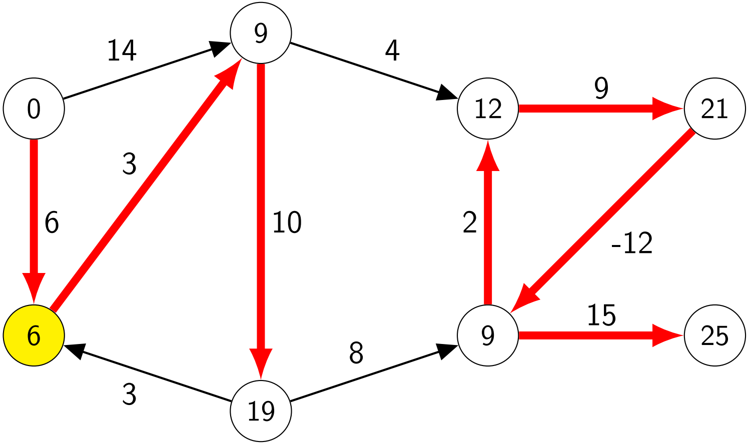
Bellman-Ford – Beispiel



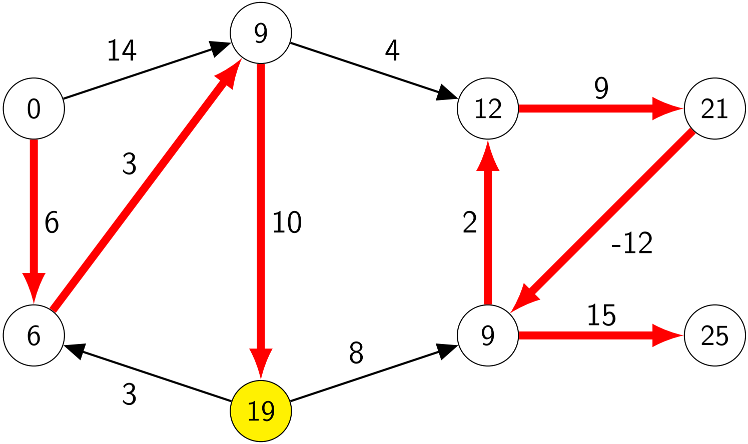
Bellman-Ford – Beispiel



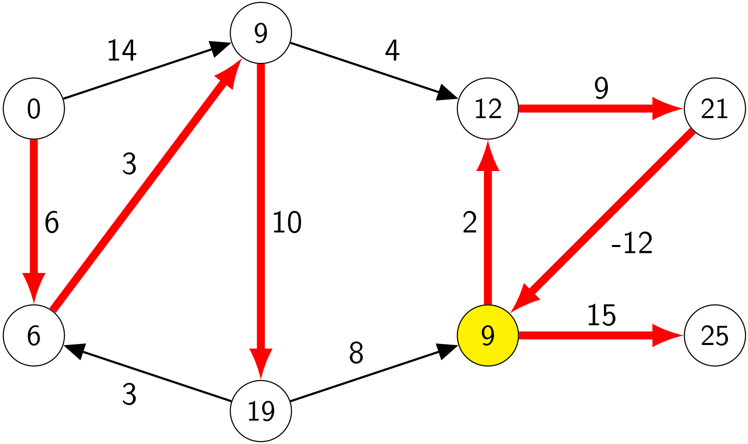
Bellman-Ford – Beispiel



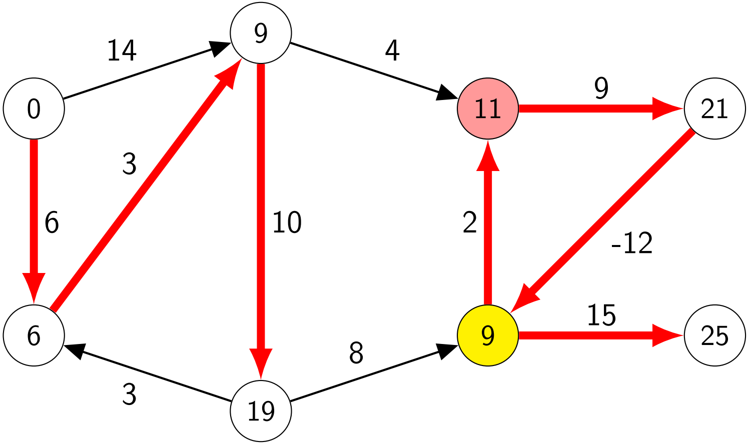
Bellman-Ford – Beispiel



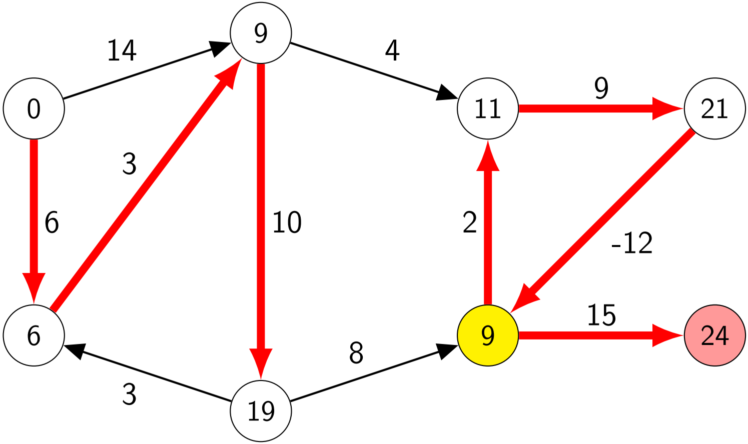
Bellman-Ford – Beispiel



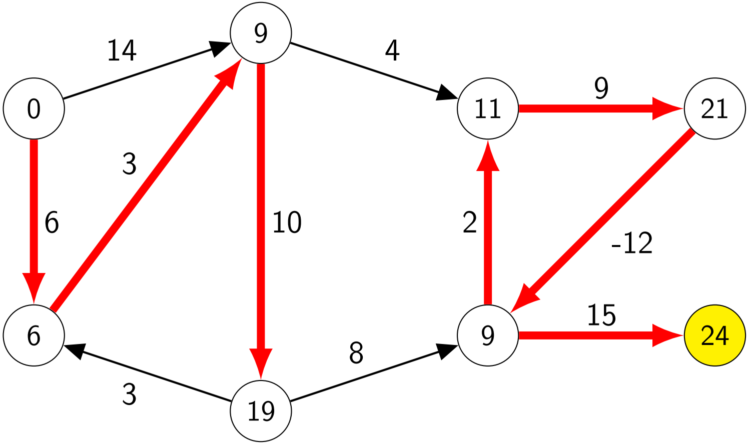
Bellman-Ford – Beispiel



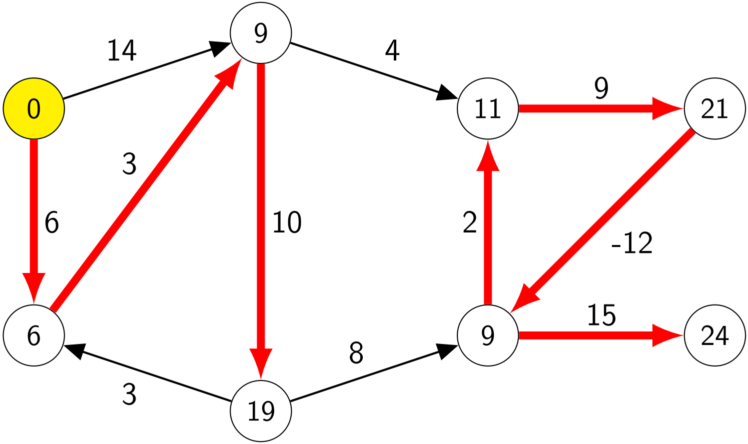
Bellman-Ford – Beispiel



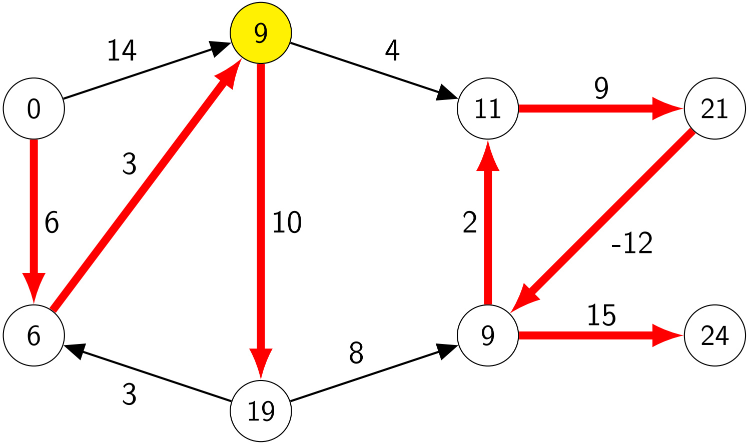
Bellman-Ford – Beispiel



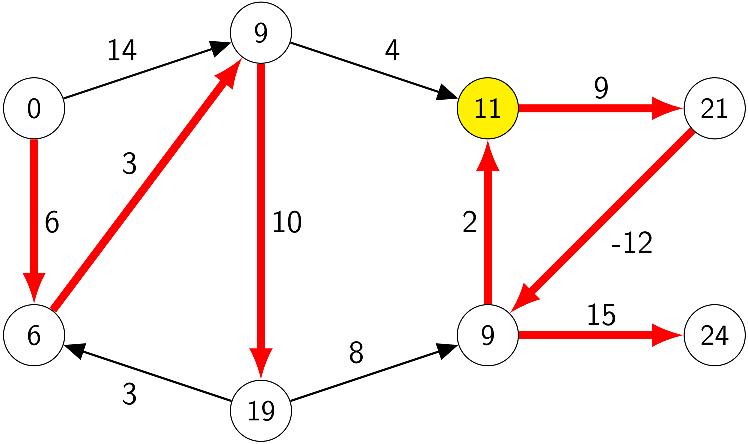
Bellman-Ford – Beispiel



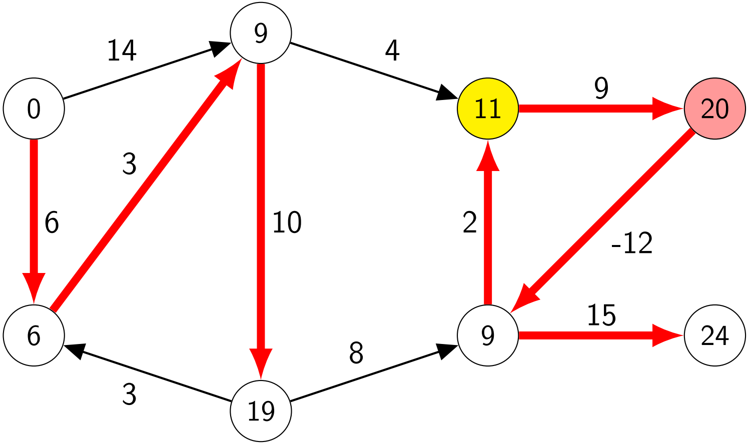
Bellman-Ford – Beispiel



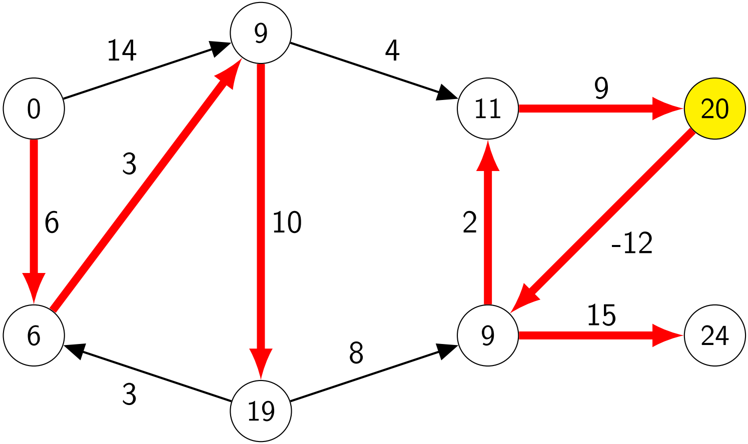
Bellman-Ford – Beispiel



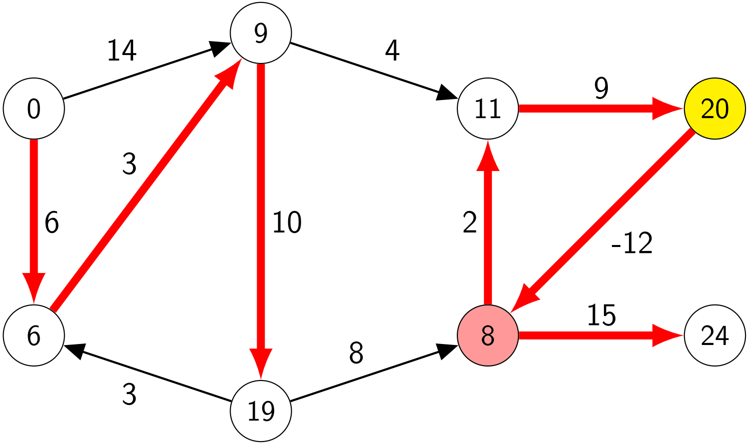
Bellman-Ford – Beispiel



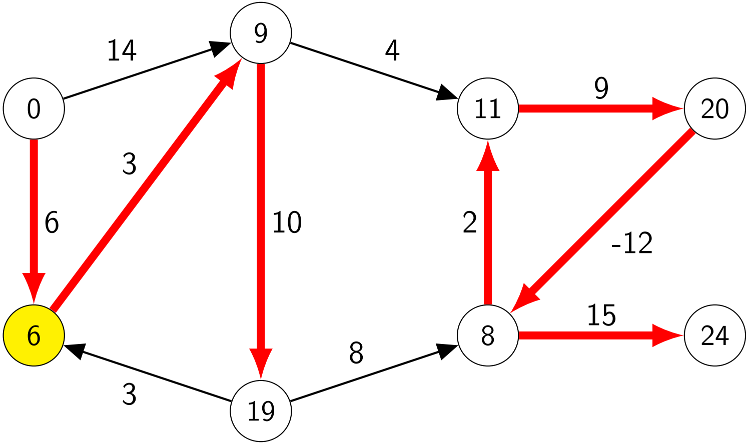
Bellman-Ford – Beispiel



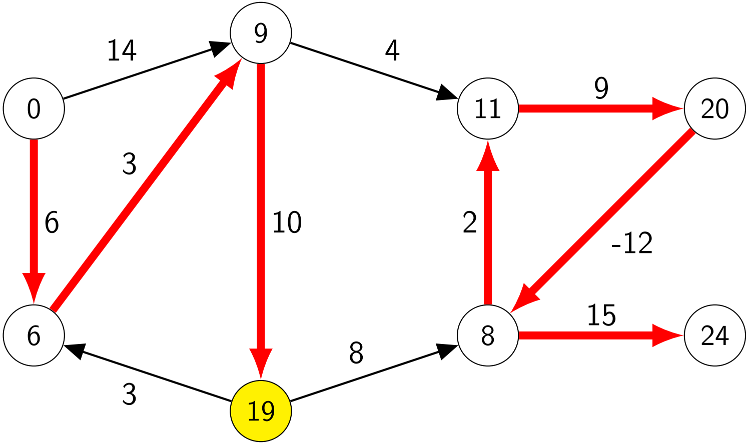
Bellman-Ford – Beispiel



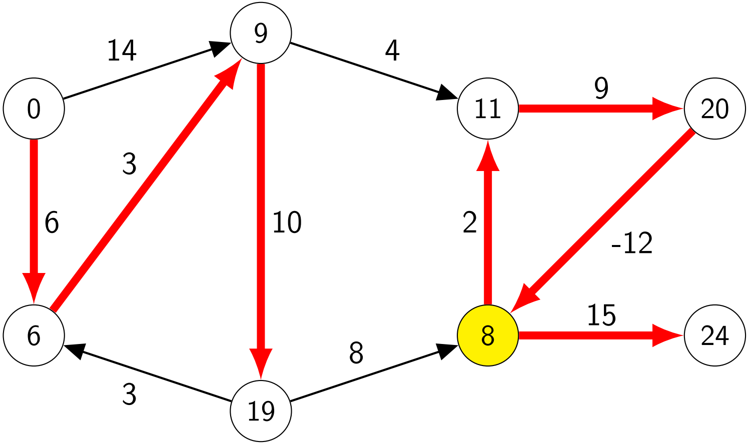
Bellman-Ford – Beispiel



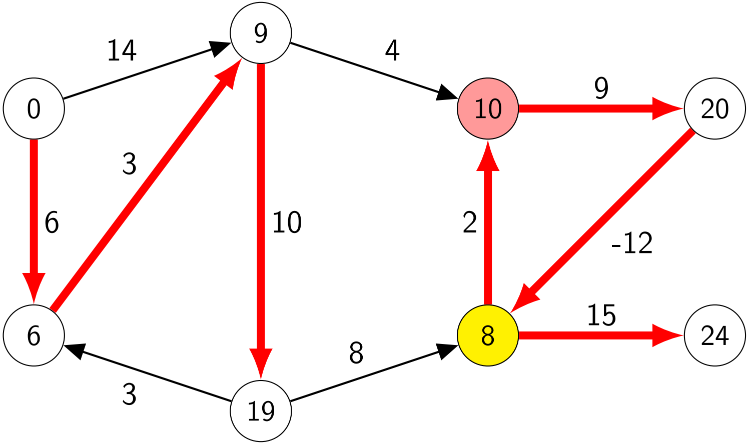
Bellman-Ford – Beispiel



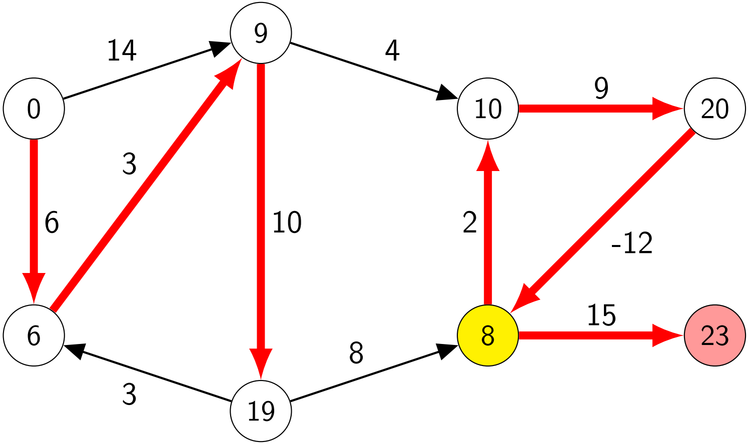
Bellman-Ford – Beispiel



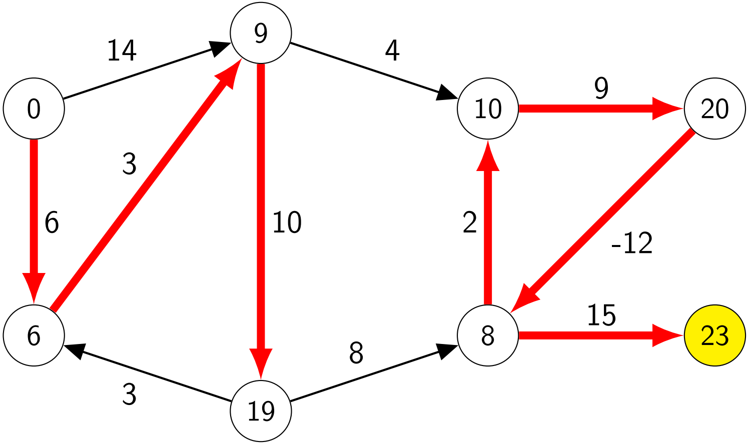
Bellman-Ford – Beispiel



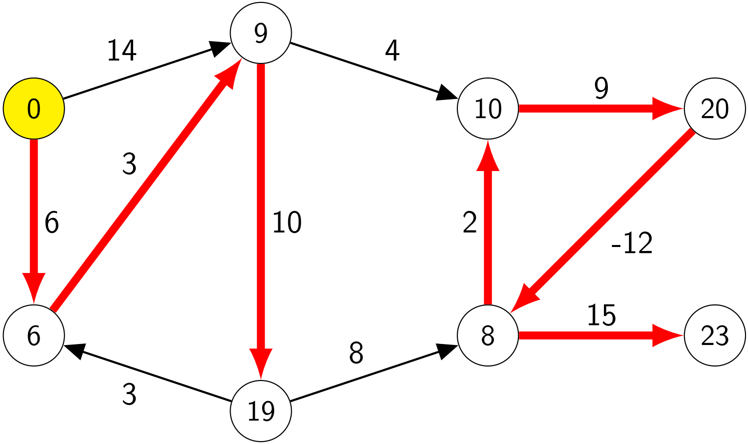
Bellman-Ford – Beispiel



Bellman-Ford – Beispiel



Bellman-Ford – Beispiel



Bellman-Ford – Implementierung

```
// Keine Zyklen mit negativem Gewicht?
bool bellFord(List adjLst[n], int n, int start) {
  int d[n] = +inf;
  d[start] = 0;
  for (int i = 1; i < n; i++) // n-1 Durchläufe
    for (int v = 0; v < n; v++) // alle Kanten
      foreach (edge in adjLst[v])
        if (d[edge.w] > d[v] + edge.weight) {
          d[edge.w] = d[v] + edge.weight;
        }
  for (int v = 0; v < n; v++) // alle Kanten
    foreach (edge in adjLst[v])
      if (d[edge.w] > d[v] + edge.weight)
        return false; // "noch kürzerer Weg"
  return true;
}
```

Bellman-Ford – Implementierung

```
// Keine Zyklen mit negativem Gewicht?
bool bellFord(List adjLst[n], int n, int start) {
  int d[n] = +inf;
  d[start] = 0;
  for (int i = 1; i < n; i++) // n-1 Durchläufe
    for (int v = 0; v < n; v++) // alle Kanten
      foreach (edge in adjLst[v])
        if (d[edge.w] > d[v] + edge.weight) {
          d[edge.w] = d[v] + edge.weight;
        }
  for (int v = 0; v < n; v++) // alle Kanten
    foreach (edge in adjLst[v])
      if (d[edge.w] > d[v] + edge.weight)
        return false; // "noch kürzerer Weg"
  return true;
}
```

- Erweiterbar durch Speichern des Vorgängers auf die Rückgabe der kürzesten Wege (Übung).

Bellman-Ford – Implementierung

```
// Keine Zyklen mit negativem Gewicht?
bool bellFord(List adjLst[n], int n, int start) {
  int d[n] = +inf;
  d[start] = 0;
  for (int i = 1; i < n; i++) // n-1 Durchläufe
    for (int v = 0; v < n; v++) // alle Kanten
      foreach (edge in adjLst[v])
        if (d[edge.w] > d[v] + edge.weight) {
          d[edge.w] = d[v] + edge.weight;
        }
  for (int v = 0; v < n; v++) // alle Kanten
    foreach (edge in adjLst[v])
      if (d[edge.w] > d[v] + edge.weight)
        return false; // "noch kürzerer Weg"
  return true;
}
```

- Erweiterbar durch Speichern des Vorgängers auf die Rückgabe der kürzesten Wege (Übung).
- Komplexität: $O(|V| \cdot |E|) = O(n \cdot m) \in O(n^3)$.

Übersicht

Dijkstras Kürzeste-Wege-Algorithmus

Annahme

Alle Kantengewichte sind nicht-negativ, d. h. $W(v, w) \geqslant 0$.

Dijkstras Kürzeste-Wege-Algorithmus

Annahme

Alle Kantengewichte sind nicht-negativ, d. h. $W(v, w) \geqslant 0$.

- Kürzeste Wege können nun nur noch Pfade (jeder Knoten wird höchstens einmal besucht) sein.

Dijkstras Kürzeste-Wege-Algorithmus

Annahme

Alle Kantengewichte sind nicht-negativ, d. h. $W(v, w) \geqslant 0$.

- Kürzeste Wege können nun nur noch Pfade (jeder Knoten wird höchstens einmal besucht) sein.

Dijkstras Kürzeste-Wege-Algorithmus basiert auf folgender Eigenschaft:

Dijkstras Kürzeste-Wege-Algorithmus

Annahme

Alle Kantengewichte sind nicht-negativ, d. h. $W(v, w) \geqslant 0$.

- Kürzeste Wege können nun nur noch Pfade (jeder Knoten wird höchstens einmal besucht) sein.

Dijkstras Kürzeste-Wege-Algorithmus basiert auf folgender Eigenschaft:

- Angenommen der kürzeste Pfad von x nach z geht über Knoten y.

Dijkstras Kürzeste-Wege-Algorithmus

Annahme

Alle Kantengewichte sind nicht-negativ, d. h. $W(v, w) \geqslant 0$.

- Kürzeste Wege können nun nur noch Pfade (jeder Knoten wird höchstens einmal besucht) sein.

Dijkstras Kürzeste-Wege-Algorithmus basiert auf folgender Eigenschaft:

- Angenommen der kürzeste Pfad von x nach z geht über Knoten y.

$\Rightarrow$ Dann ist der Teilpfad von x nach y auch *ein* kürzester Pfad von x nach y.

Dijkstras Kürzeste-Wege-Algorithmus

Annahme

Alle Kantengewichte sind nicht-negativ, d. h. $W(v, w) \geqslant 0$.

- Kürzeste Wege können nun nur noch Pfade (jeder Knoten wird höchstens einmal besucht) sein.

Dijkstras Kürzeste-Wege-Algorithmus basiert auf folgender Eigenschaft:

- Angenommen der kürzeste Pfad von x nach z geht über Knoten y.

$\Rightarrow$ Dann ist der Teilpfad von x nach y auch *ein* kürzester Pfad von x nach y.

$\Rightarrow$ Auch der Teilpfad von y nach z ist ein kürzester Pfad von y nach z.

Dijkstras Kürzeste-Wege-Algorithmus – Übersicht

Wie beim Algorithmus von Prim ordnen wir die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

Dijkstras Kürzeste-Wege-Algorithmus – Übersicht

Wie beim Algorithmus von Prim ordnen wir die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

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

Dijkstras Kürzeste-Wege-Algorithmus – Übersicht

Wie beim Algorithmus von Prim ordnen wir die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

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

Rand-knoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Dijkstras Kürzeste-Wege-Algorithmus – Übersicht

Wie beim Algorithmus von Prim ordnen wir die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

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

Dijkstras Kürzeste-Wege-Algorithmus – Übersicht

Wie beim Algorithmus von Prim ordnen wir die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

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

Rand-knoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Ungesehene Knoten: Alle anderen Knoten.

Grundkonzept:

Dijkstras Kürzeste-Wege-Algorithmus – Übersicht

Wie beim Algorithmus von Prim ordnen wir die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

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

Rand-knoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Ungesehene Knoten: Alle anderen Knoten.

Grundkonzept:

- Fange mit einen Baum aus nur einem Knoten an, in dem der Quellen-Knoten s die Wurzel ist.

Dijkstras Kürzeste-Wege-Algorithmus – Übersicht

Wie beim Algorithmus von Prim ordnen wir die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

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

Rand-knoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Ungesehene Knoten: Alle anderen Knoten.

Grundkonzept:

- Fange mit einen Baum aus nur einem Knoten an, in dem der Quellen-Knoten s die Wurzel ist.
- Finde die Kante mit kürzestem Abstand von s, die den bisherigen Baum verlässt.

Dijkstras Kürzeste-Wege-Algorithmus – Übersicht

Wie beim Algorithmus von Prim ordnen wir die Knoten in drei Kategorien (BLACK, GRAY, WHITE) ein:

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

Rand-knoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Ungesehene Knoten: Alle anderen Knoten.

Grundkonzept:

- Fange mit einen Baum aus nur einem Knoten an, in dem der Quellen-Knoten s die Wurzel ist.
- Finde die Kante mit kürzestem Abstand von s, die den bisherigen Baum verlässt.
- Füge den über diese Kante erreichten (Rand-)Knoten dem Baum hinzu, zusammen mit der Kante.

Dijkstras Kürzeste-Wege-Algorithmus – Übersicht

Wie beim Algorithmus von Prim ordnen wir die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

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

Rand-knoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Ungesehene Knoten: Alle anderen Knoten.

Grundkonzept:

- Fange mit einen Baum aus nur einem Knoten an, in dem der Quellen-Knoten s die Wurzel ist.
- Finde die Kante mit kürzestem Abstand von s, die den bisherigen Baum verlässt.
- Füge den über diese Kante erreichten (Rand-)Knoten dem Baum hinzu, zusammen mit der Kante.
- Fahre fort, bis keine weiteren Randknoten mehr vorhanden sind.

Dijkstras Kürzeste-Wege-Algorithmus – Übersicht

Wie beim Algorithmus von Prim ordnen wir die Knoten in drei Kategorien (`BLACK`, `GRAY`, `WHITE`) ein:

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

Rand-knoten: Nicht im Baum, jedoch adjazent zu Knoten im Baum.

Ungesehene Knoten: Alle anderen Knoten.

Grundkonzept:

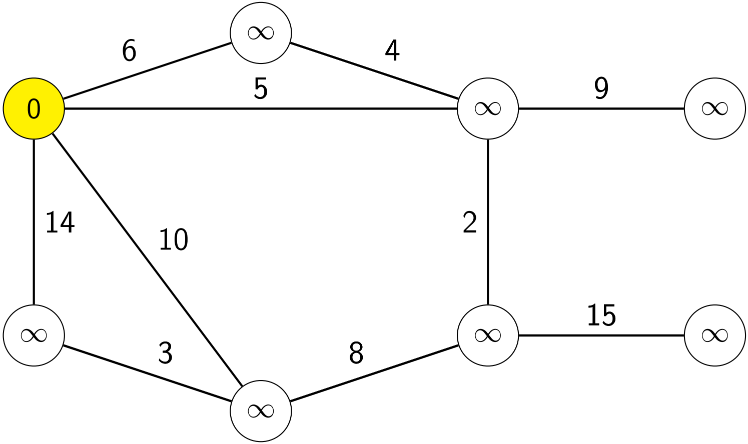
- Fange mit einen Baum aus nur einem Knoten an, in dem der Quellen-Knoten s die Wurzel ist.
- Finde die Kante mit kürzestem Abstand von s, die den bisherigen Baum verlässt.
- Füge den über diese Kante erreichten (Rand-)Knoten dem Baum hinzu, zusammen mit der Kante.
- Fahre fort, bis keine weiteren Randknoten mehr vorhanden sind.

Genauso wie der MST-Algorithmus von Prim handelt es sich um einen Greedy-Algorithmus.

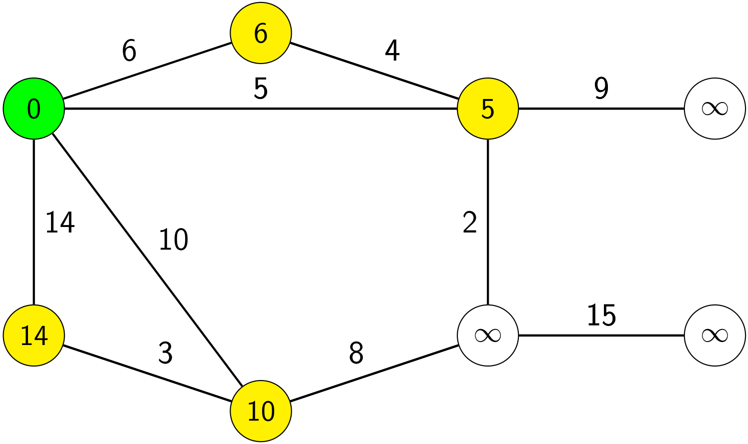
Dijkstras Kürzeste-Wege-Algorithmus – Grundgerüst

```
// ungerichteter Graph G mit n Knoten
void dijkstraSP(Graph G, int n) {
  initialisiere alle Knoten als ungesehen (WHITE);
  markiere s als Baum (BLACK) und setze d(s, s) = 0;
  reklassifiziere alle zu s adjazenten Knoten als Rand (GRAY);
  while (es gibt Randknoten) {
    wähle von allen Kanten zwischen einem Baumknoten t und
      einem Randknoten v die mit minimalem d(s, t) + W(t, v);
    reklassifiziere v als Baum (BLACK);
    füge Kante tv zum Baum hinzu;
    setze d(s, v) = d(s, t) + W(t, v);
    reklassifiziere alle zu v adjazenten ungesehenen Knoten
      mit Rand (GRAY);
  }
}
```

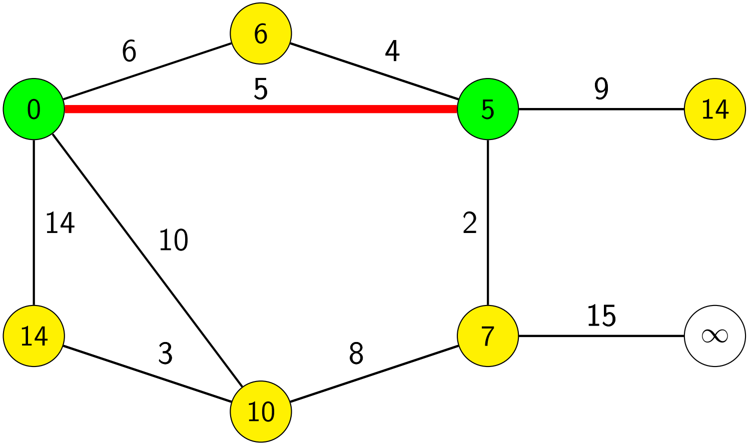
Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



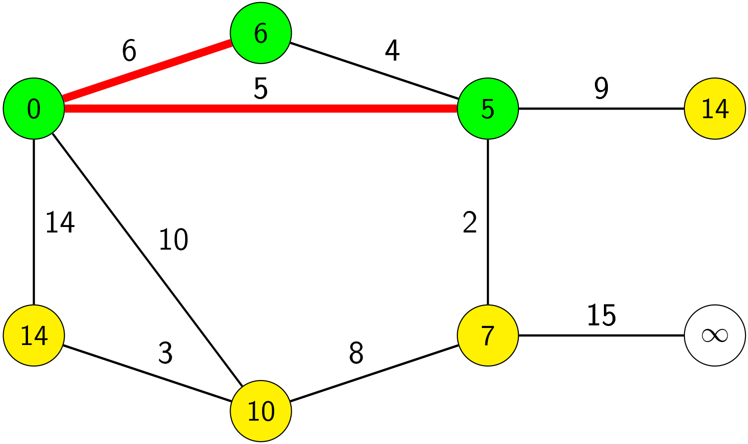
Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



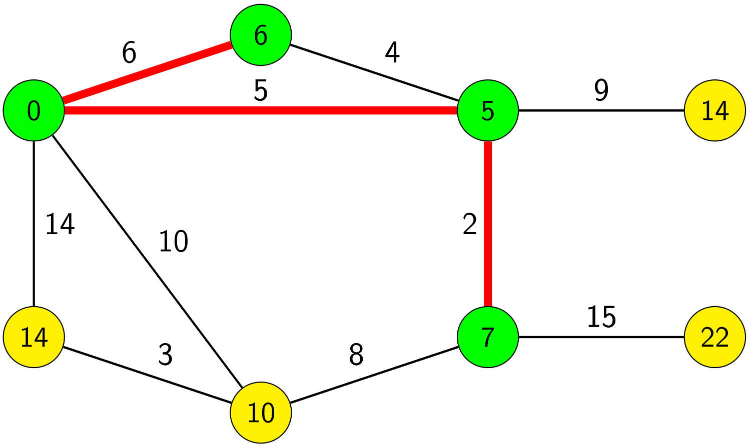
Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



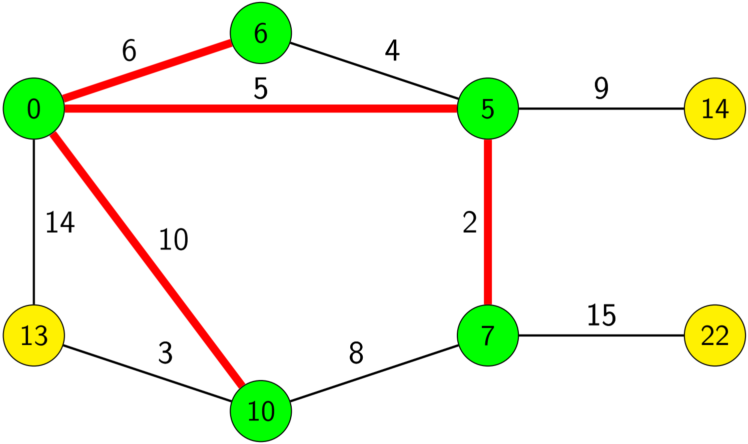
Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



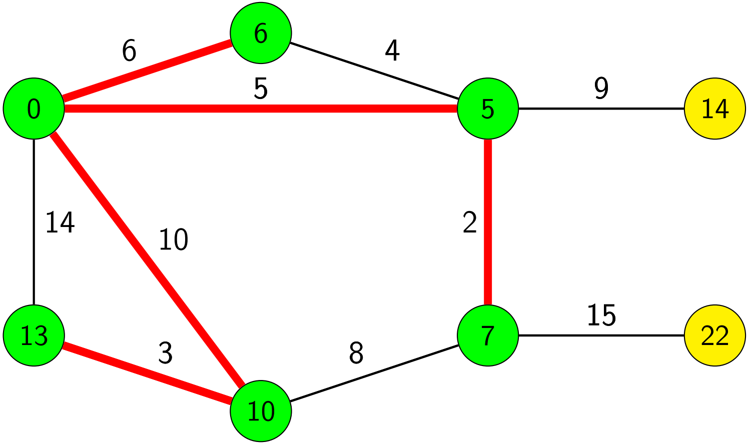
Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



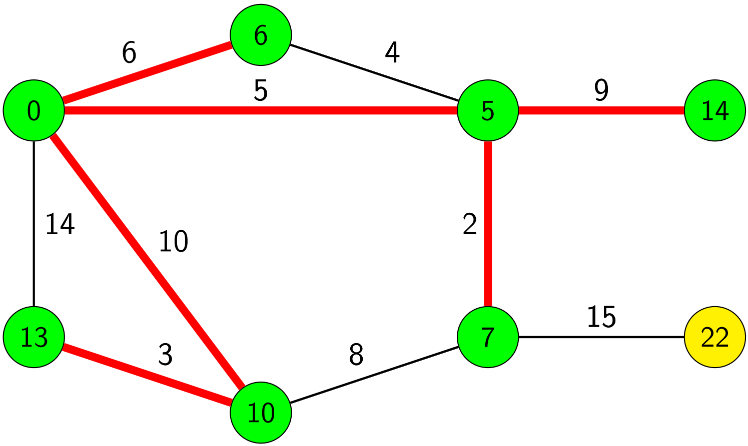
Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



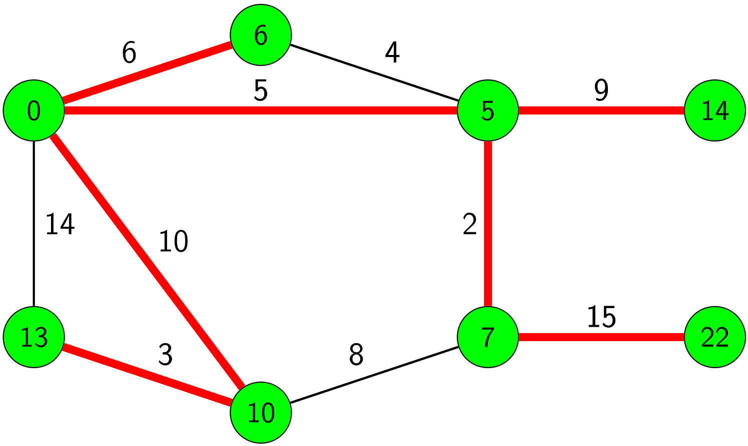
Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



Dijkstras Kürzeste-Wege-Algorithmus – Beispiel



Theorem

Sei $s \in V' \subseteq V$ mit kürzestem Abstand $d(s, y)$ von s nach $y \in V'$.

Theorem

Wenn (y, z) die Kante mit minimalem $d(s, y) + W(y, z)$ über alle Kanten mit $y \in V'$ und $z \in V \setminus V'$ ist, dann ist der zusammengesetzte Weg bestehend aus dem kürzesten Weg von s nach y gefolgt von der Kante (y, z) auch der kürzeste Weg von s nach z.

Beweis

Beweis.

Sei P der kürzeste Weg von s nach y, erweitert um die Kante (y, z), die „am nahesten an s liegt“.

Beweis

Beweis.

Sei P der kürzeste Weg von s nach y, erweitert um die Kante (y, z), die „am nahesten an s liegt".

Sei nun $P' = s, z_1, \ldots, z_k, \ldots, z$ der kürzeste Weg von s nach z, wobei z_k der erste Knoten $\notin V'$ ist.

Beweis

Beweis.

Sei P der kürzeste Weg von s nach y, erweitert um die Kante (y, z), die „am nahesten an s liegt“.

Sei nun $P' = s, z_1, \ldots, z_k, \ldots, z$ der kürzeste Weg von s nach z, wobei z_k der erste Knoten $\notin V'$ ist.

- Wegen der Wahl von (y, z) gilt:

$$W(P) = d(s, y) + W(y, z) \leqslant d(s, z_{k-1}) + W(z_{k-1}, z_k).$$

Beweis

Beweis.

Sei P der kürzeste Weg von s nach y, erweitert um die Kante (y, z), die „am nahesten an s liegt".

Sei nun $P' = s, z_1, \ldots, z_k, \ldots, z$ der kürzeste Weg von s nach z, wobei z_k der erste Knoten $\notin V'$ ist.

- Wegen der Wahl von (y, z) gilt:

$$W(P) = d(s, y) + W(y, z) \leqslant d(s, z_{k-1}) + W(z_{k-1}, z_k).$$

- Da $s, z_1, \ldots, z_k$ ein Prefix von P' ist und alle verbleibenden Kanten *nicht-negatives* Gewicht haben, gilt:

$$d(s, z_{k-1}) + W(z_{k-1}, z_k) \leqslant W(P')$$

Beweis

Beweis.

Sei P der kürzeste Weg von s nach y, erweitert um die Kante (y, z), die „am nahesten an s liegt“.

Sei nun $P' = s, z_1, \ldots, z_k, \ldots, z$ der kürzeste Weg von s nach z, wobei z_k der erste Knoten $\notin V'$ ist.

- Wegen der Wahl von (y, z) gilt:

$$W(P) = d(s, y) + W(y, z) \leqslant d(s, z_{k-1}) + W(z_{k-1}, z_k).$$

- Da $s, z_1, \ldots, z_k$ ein Prefix von P' ist und alle verbleibenden Kanten *nicht-negatives* Gewicht haben, gilt:

$$d(s, z_{k-1}) + W(z_{k-1}, z_k) \leqslant W(P')$$

$\Rightarrow$ Daher ist $W(P) \leqslant W(P')$, d. h. P ist der kürzeste Weg! $\square$

Korrektheit

Theorem (Korrektheit)

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

Korrektheit

Theorem (Korrektheit)

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

Beweis.

Induktion nach der Sequenz von hinzugefügte Knoten im SSSP-Baum. (Übung.) $\square$

Dijkstras Kürzeste-Wege-Algorithmus – Implementierung (I)

```
// Wie Prim. Ergebnis als Vorgängerliste(-baum) in .parent
// curWeight[v] enthält gerade d(s,v)
VertexState[n] dijkSP(List adjLst[n], int n, int start) {
  VertexState state[n] = // (eigentlich im Konstruktor von pq)
      { color: WHITE, parent: -1, curWeight: +inf };
  PriorityQueue pq = VS_PriorityQueue<&VS.curWeight>(&state);

  pq.insert(start, {parent: -1, curWeight: 0});
  while (!pq.isEmpty()) { // solange es Randknoten gibt
    int v = pq.getMin(); // günstigste Kante, bzw. Randknoten
    pq.delMin(); // setzt auch Farbe auf BLACK
    updateFringe(pq, adjList, v); // update den Rand
  }
  return state;
}
```

Dijkstras Kürzeste-Wege-Algorithmus – Implementierung (II)

```
void updateFringe(PriorityQueue &pq, List adjLst[], int v) {
  // kürzester Weg von s nach v
  float ownWeight = pq.getWeight(v);
  foreach (edge in adjLst[v]) {
    // Distanz von s nach w über v
    float newWeight = edge.weight + ownWeight;

    if (pq.getColor(edge.w) == WHITE) { // -> GRAY
      pq.insert(edge.w, {parent: v, curWeight: newWeight});
    } else if (pq.getColor(edge.w) == GRAY) {
      if (newWeight < pq.getWeight(edge.w)) {
        // Randknoten-update: Weg über v ist besser
        pq.decrKey(edge.w, {parent: v, curWeight: newWeight});
      }
    }
  }
}
```

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: $O(|V|)$.