Korrektheit: ist klar.

Zeitkomplexität:

- n *ExtractMin*
- $\mathcal{O}(m)$ sonstige Operationen inclusive *DecreaseKey*

Implementierung der Priority Queue mittels Fibonacci-Heaps:

Initialisierung	$\mathcal{O}(n)$
ExtractMins	$\mathcal{O}(n \log n)$ $(\leq n$ Stück$)$
DecreaseKeys	$\mathcal{O}(m)$ $(\leq m$ Stück$)$
Sonstiger Overhead	$\mathcal{O}(m)$

Satz 192

Sei $G = (V, E)$ ein ungerichteter Graph (zusammenhängend, einfach) mit Kantengewichten w. Prim's Algorithmus berechnet, wenn mit Fibonacci-Heaps implementiert, einen minimalen Spannbaum von (G, w) in Zeit $\mathcal{O}(m + n \log n)$ (wobei $n = |V|,\ m = |E|$). Dies ist für $m = \Omega(n \log n)$ asymptotisch optimal.

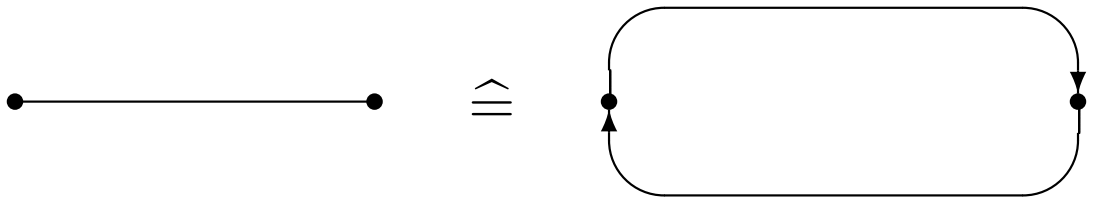
Beweis:

s.o.

8. Kürzeste Pfade

8.1 Grundlegende Begriffe

Betrachte Digraph $G = (V, A)$ oder Graph $G = (V, E)$.



Kante im ungerichteten Graph

entgegengesetzte (antiparallele) Kanten

Distanzfunktion: $\mathrm{d} : A \longrightarrow \mathbb{R}_0^+ \cup \{+\infty\}$ (bzw. $\longrightarrow \mathbb{R} \cup \{+\infty\}$)
O.B.d.A.: $A = V \times V$, $\mathrm{d}(x, y) = +\infty$ für Kanten, die eigentlich nicht vorhanden sind.
$\mathrm{dis}(v, w) :=$ Länge eines kürzesten Pfades von v nach w
$\in \mathbb{R}_0^+ \cup \{+\infty\}$.

Arten von Kürzeste-Pfade-Problemen:

1. single-pair-shortest-path (spsp). Beispiel: Kürzeste Entfernung von München nach Frankfurt.
2. single-source-shortest-path: gegeben G, d und $s \in V$, bestimme für alle $v \in V$ die Länge eines kürzesten Pfades von s nach v (bzw. einen kürzesten Pfad von s nach v) (sssp). Beispiel: Kürzeste Entfernung von München nach allen anderen Großstädten.
3. all-pairs-shortest-path (apsp). Beispiel: Kürzeste Entfernung zwischen allen Großstädten.

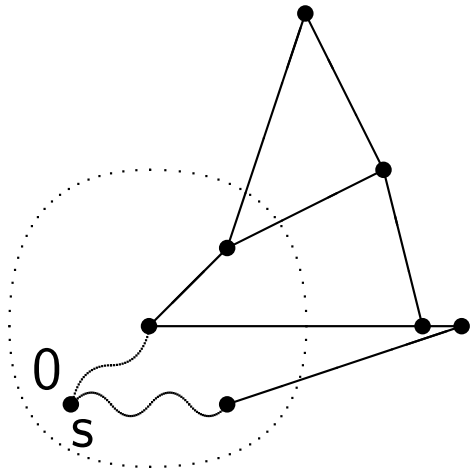
Bemerkung: Wir kennen keinen Algorithmus, der das single-pair-shortest-path berechnet, ohne nicht gleichzeitig (im worst-case) das single-source-shortest-path-Problem zu lösen.

8.2 Das single-source-shortest-path-Problem

Zunächst nehmen wir an, dass $\mathrm{d} \geq 0$ ist. Alle kürzesten Pfade von a nach b sind o.B.d.A. einfache Pfade.

8.2.1 Dijkstra's Algorithmus

Gegeben: $G = (V, A)$, $(A = V \times V)$, Distanzfunktion $\mathrm{d} : A \to \mathbb{R}_0^+ \cup \{+\infty\}$, Startknoten s, G durch Adjazenzlisten dargestellt.



algorithm sssp:=

$S := \{s\}$; $\mathrm{dis}[s] := 0$; initialisiere eine Priority Queue PQ, die alle Knoten $v \in V \setminus \{s\}$ enthält mit Schlüssel $\mathrm{dis}[v] := \mathrm{d}(s, v)$

for alle $v \in V - \{s\}$ **do** from$[v] := s$ **od**

while $S \neq V$ **do**

$v :=$ *ExtractMin*(PQ)

$S := S \cup \{v\}$

for alle $w \in V \setminus S$, $\mathrm{d}(v, w) < \infty$ **do**

if $\mathrm{dis}[v] + \mathrm{d}(v, w) < \mathrm{dis}[w]$ **then**

DecreaseKey$(w, \mathrm{dis}[v] + \mathrm{d}(v, w))$

co *DecreaseKey* aktualisiert $\mathrm{dis}[w]$ **oc**

from$[w] := v$

fi

od

od

Seien $n = |V|$ und $m =$ die Anzahl der wirklichen Kanten in G.

Laufzeit (mit Fibonacci-Heaps):

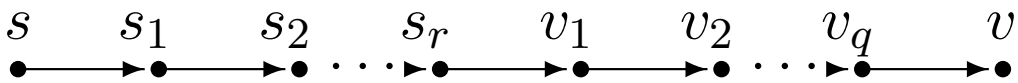
Initialisierung:	$\mathcal{O}(n)$
ExtractMin:	$n \cdot \mathcal{O}(\log n)$
Sonstiger Aufwand:	$m \cdot \mathcal{O}(1)$ (z.B. *DecreaseKey*)

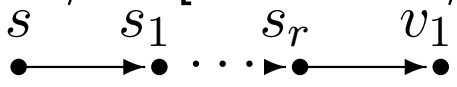
$\Rightarrow$ Zeitbedarf also: $\boxed{\mathcal{O}(m + n \log n)}$

Korrektheit: Wir behaupten, dass in dem Moment, in dem ein $v \in V \setminus \{s\}$ Ergebnis der *ExtractMin* Operation ist, der Wert $\mathrm{dis}[v]$ des Schlüssels von v gleich der Länge eines kürzesten Pfades von s nach v ist.

Beweis:

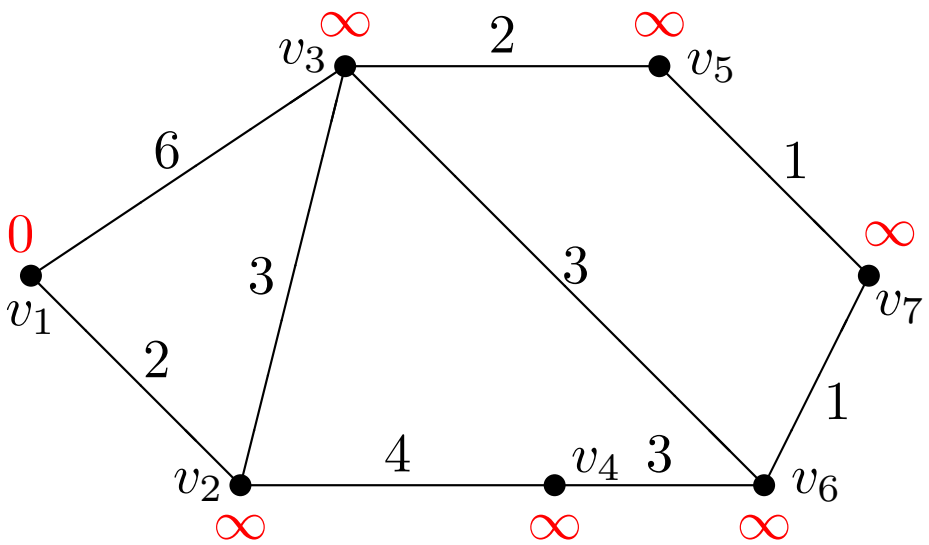
[durch Widerspruch] Sei $v \in V \setminus \{s\}$ der erste Knoten, für den diese Behauptung nicht stimmt, und sei



ein kürzester Pfad von s nach v, mit einer Länge $< \mathrm{dis}[v]$. Dabei sind $s_1, \ldots, s_r \in S$, $v_1 \notin S$ [$r = 0$ und/oder $q = 0$ ist möglich]. Betrachte den Pfad 

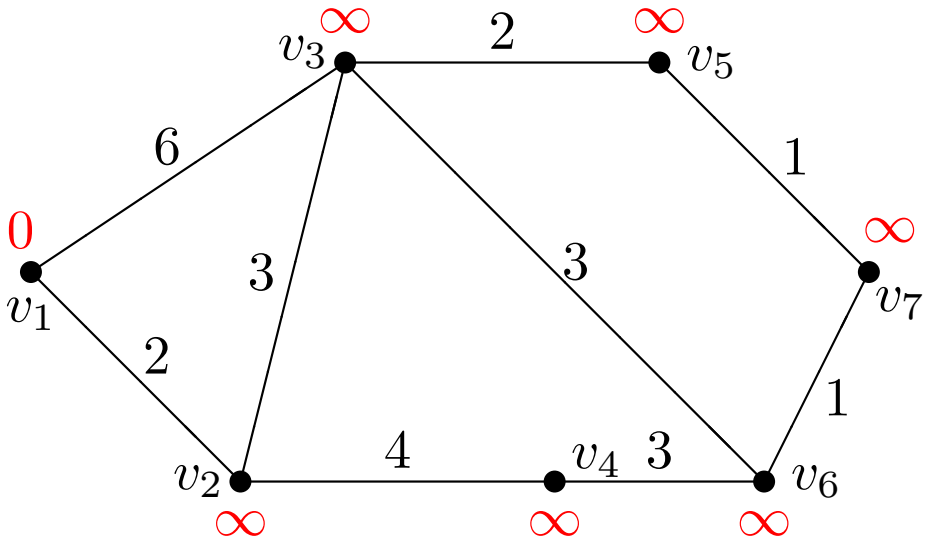
; seine Länge ist $< \mathrm{dis}[v]$, für $q \geq 1$ (ebenso für $q = 0$) ist also $\mathrm{dis}[v_1] < \mathrm{dis}[v]$, im Widerspruch zur Wahl von v.

Beispiel 193 (Dijkstra's Algorithmus)

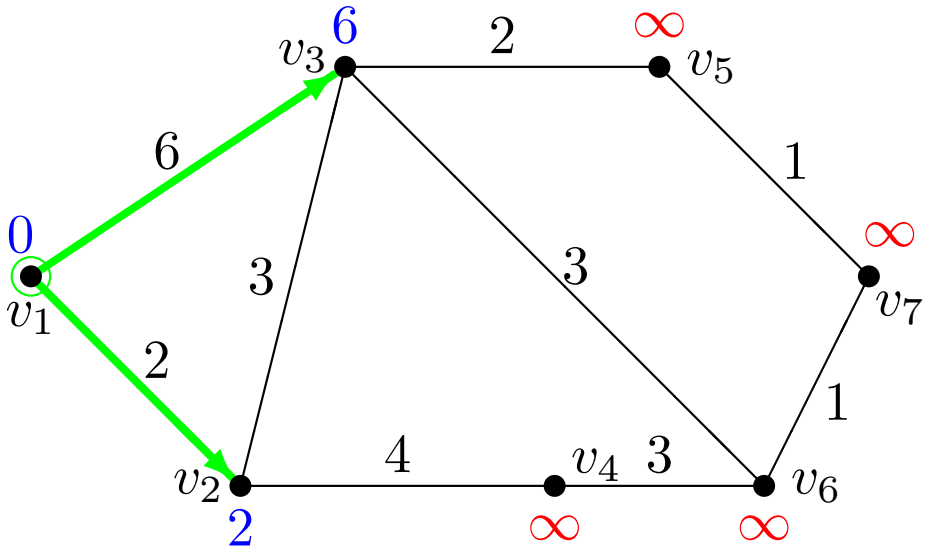


gegeben Graph G;
v_1 ist der Startknoten;
setze v_1 als Bezugsknoten;
setze $\mathrm{dis}[v_1] = 0$;
setze $\mathrm{dis}[\text{Rest}] = +\infty$;

Beispiel 193 (Dijkstra's Algorithmus)

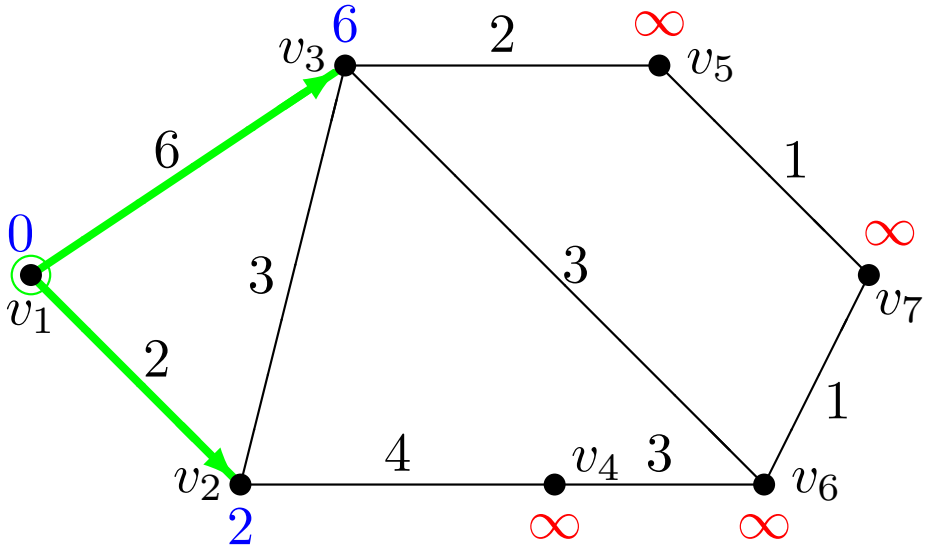


gegeben Graph G;
v_1 ist der Startknoten;
setze v_1 als Bezugsknoten;
setze $\mathrm{dis}[v_1] = 0$;
setze $\mathrm{dis}[\mathrm{Rest}] = +\infty$;

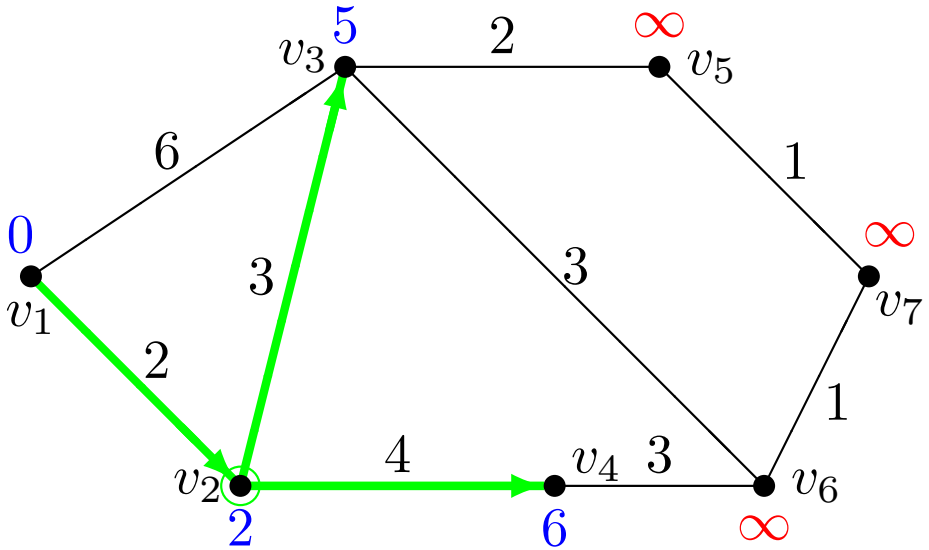


setze $\mathrm{dis}[v_2] = 2$;
markiere (v_1, v_2);
setze $\mathrm{dis}[v_3] = 6$;
markiere (v_1, v_3);
setze v_2 als Bezugsknoten,
da $\mathrm{dis}[v_2]$ minimal;

Beispiel 193 (Dijkstra's Algorithmus)

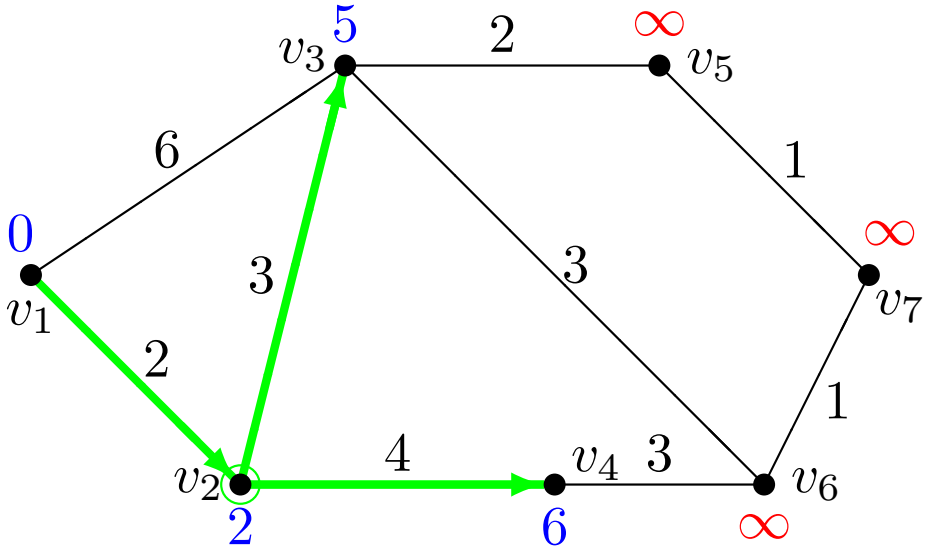


setze $\mathrm{dis}[v_2] = 2$;
markiere (v_1, v_2);
setze $\mathrm{dis}[v_3] = 6$;
markiere (v_1, v_3);
setze v_2 als Bezugsknoten,
da $\mathrm{dis}[v_2]$ minimal;

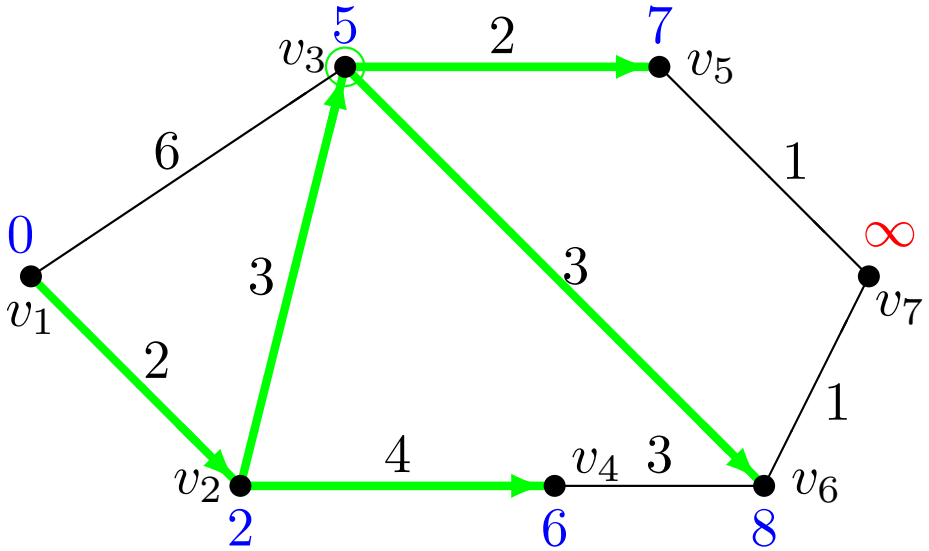


setze $\mathrm{dis}[v_3] = 2 + 3 = 5$;
markiere (v_2, v_3);
unmarkiere (v_1, v_3);
setze $\mathrm{dis}[v_4] = 2 + 4 = 6$;
markiere (v_2, v_4);
setze v_3 als Bezugsknoten,
da $\mathrm{dis}[v_3]$ minimal;

Beispiel 193 (Dijkstra's Algorithmus)

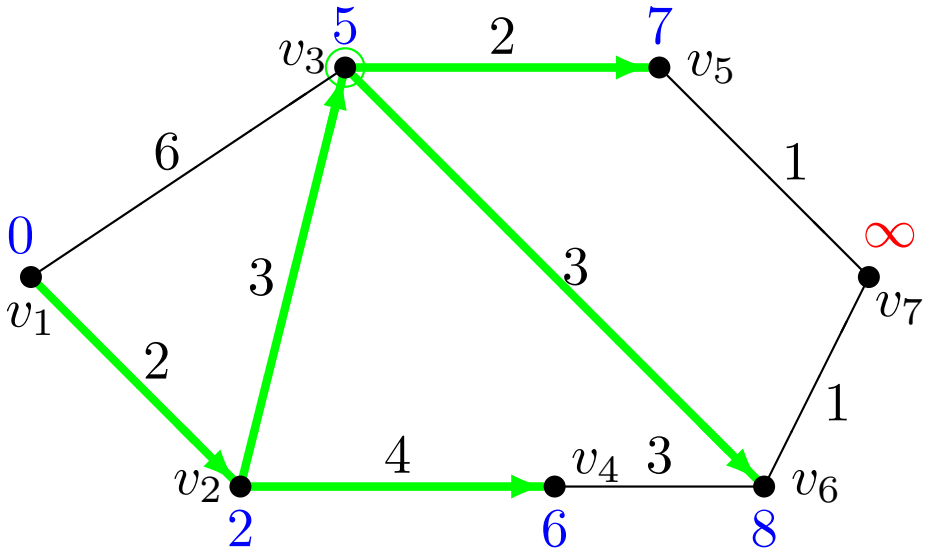


setze $\mathrm{dis}[v_3] = 2 + 3 = 5$;
markiere (v_2, v_3);
unmarkiere (v_1, v_3);
setze $\mathrm{dis}[v_4] = 2 + 4 = 6$;
markiere (v_2, v_4);
setze v_3 als Bezugsknoten,
da $\mathrm{dis}[v_3]$ minimal;

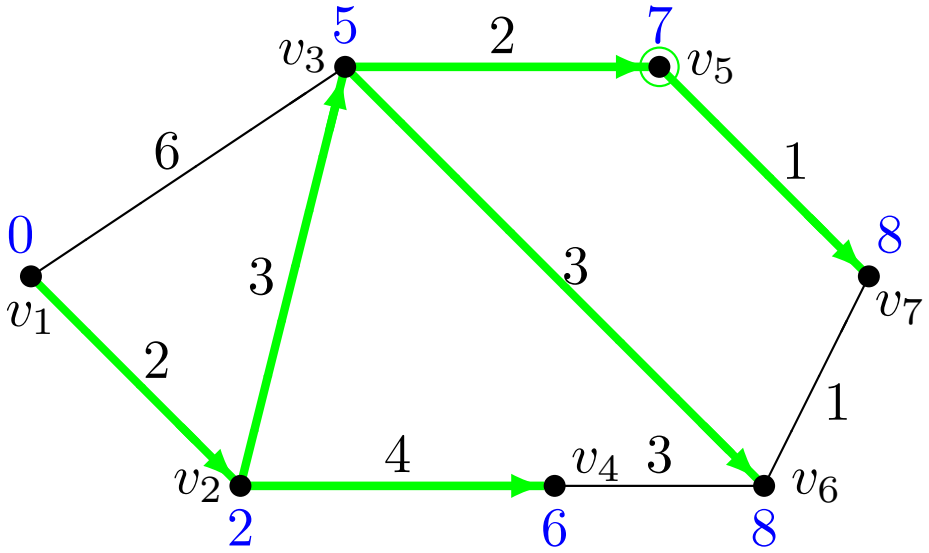


setze $\mathrm{dis}[v_5] = 5 + 2 = 7$;
markiere (v_3, v_5);
setze $\mathrm{dis}[v_6] = 5 + 3 = 8$;
markiere (v_3, v_6);
setze v_4, dann v_5
als Bezugsknoten;

Beispiel 193 (Dijkstra's Algorithmus)



setze $\mathrm{dis}[v_5] = 5 + 2 = 7$;
markiere (v_3, v_5);
setze $\mathrm{dis}[v_6] = 5 + 3 = 8$;
markiere (v_3, v_6);
setze v_4, dann v_5
als Bezugsknoten;



setze $\mathrm{dis}[v_7] := 7 + 1 = 8$;
markiere (v_5, v_7);
alle Knoten wurden erreicht:
$\Rightarrow$ Algorithmus zu Ende

Beobachtung:

- *ExtractMin* liefert eine (schwach) monoton steigende Folge von Schlüsseln $\mathrm{dis}[\cdot]$;
- Die Schlüssel $\neq \infty$ in PQ sind stets $\leq \mathrm{dis}[v] + C$, wobei v das Ergebnis der vorangehenden *ExtractMin*-Operation (bzw. s zu Beginn) und $C := \max_{(u,w)\in A}\{\mathrm{dis}(u,w)\}$ ist.

Satz 194

Dijkstra's Algorithmus (mit Fibonacci-Heaps) löst das single-source-shortest-path-Problem in Zeit $\mathcal{O}(m + n\log n)$.

8.2.2 Bellman-Ford-Algorithmus

Wir setzen (zunächst) wiederum voraus:

$$\mathrm{d} \geq 0 \; .$$

Dieser Algorithmus ist ein Beispiel für dynamische Programmierung.

Sei $B_k[i] :=$ Länge eines kürzesten Pfades von s zum Knoten i, wobei der Pfad höchstens k Kanten enthält.
Gesucht ist $B_{n-1}[i]$ für $i = 1, \ldots, n$ (o.B.d.A. $V = \{1, \ldots, n\}$).

Initialisierung:

$$B_1[i] := \begin{cases} \mathrm{d}(s,i) & , \text{ falls } \mathrm{d}(s,i) < \infty, i \neq s \\ 0 & , \text{ falls } i = s \\ +\infty & , \text{ sonst} \end{cases}$$

Iteration:

for $k := 2$ **to** $n-1$ **do**
  for $i := 1$ **to** n **do**

$$B_k[i] := \begin{cases} 0 & , \text{falls } i = s \\ \min_{j \in N^{-1}(i)} \{B_{k-1}[i], B_{k-1}[j] + \mathrm{d}(j,i)\} & , \text{sonst} \end{cases}$$

  od
od

Bemerkung: $N^{-1}(i)$ ist die Menge der Knoten, von denen aus eine Kante zu Knoten i führt.

Korrekheit:
klar (Beweis durch vollständige Induktion)

Zeitbedarf:
Man beachte, dass in jedem Durchlauf der äußeren Schleife jede Halbkante einmal berührt wird.

Satz 195

Der Zeitbedarf des Bellman-Ford-Algorithmus ist $\mathcal{O}(n \cdot m)$.

Beweis:

s.o.

8.3 Floyd's Algorithmus für das all-pairs-shortest-path-Problem

Dieser Algorithmus wird auch als „Kleene's Algorithmus" bezeichnet. Er ist ein weiteres Beispiel für dynamische Programmierung.

Sei $G = (V, E)$ mit Distanzfunktion $\mathrm{d} : A \to \mathbb{R}_0^+ \cup \{+\infty\}$ gegeben. Sei o.B.d.A. $V = \{v_1, \ldots, v_n\}$.

Wir setzen nun

$c_{ij}^k :=$ *Länge eines kürzesten Pfades von* v_i *nach* v_j*, der als innere Knoten (alle bis auf ersten und letzten Knoten) nur Knoten aus* $\{v_1, \ldots, v_k\}$ *enthält.*

algorithm floyd :=

for alle (i,j) **do** $c_{ij}^{(0)} := \mathrm{d}(i,j)$ **od** **co** $1 \leq i,j \leq n$ **oc**

for $k := 1$ **to** n **do**

for alle (i,j), $1 \leq i,j \leq n$ **do**

$$c_{ij}^{(k)} := \min\left\{c_{ij}^{(k-1)},\ c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\right\}$$

od

od

Laufzeit: $\mathcal{O}(n^3)$

Korrektheit:

Zu zeigen: $c_{ij}^{(k)}$ des Algorithmus $= c_{ij}^{k}$ (damit sind die Längen der kürzesten Pfade durch $c_{ij}^{(n)}$ gegeben).

Beweis:

Richtig für $k = 0$. Induktionsschluss: Ein kürzester Pfad von v_i nach v_j mit inneren Knoten $\in \{v_1, \ldots, v_{k+1}\}$ enthält entweder v_{k+1} gar nicht als inneren Knoten, oder er enthält v_{k+1} genau einmal als inneren Knoten. Im ersten Fall wurde dieser Pfad also bereits für $c_{ij}^{(k)}$ betrachtet, hat also Länge $= c_{ij}^{(k)}$. Im zweiten Fall setzt er sich aus einem kürzesten Pfad P_1 von v_i nach v_{k+1} und einem kürzesten Pfad P_2 von v_{k+1} nach v_j zusammen, wobei alle inneren Knoten von P_1 und $P_2 \in \{v_1, \ldots, v_k\}$ sind. Also ist die Länge des Pfades $= c_{i,k+1}^{(k)} + c_{k+1,j}^{(k)}$.

Satz 196

Floyd's Algorithmus für das all-pairs-shortest-path-Problem hat Zeitkomplexität $\mathcal{O}(n^3)$.