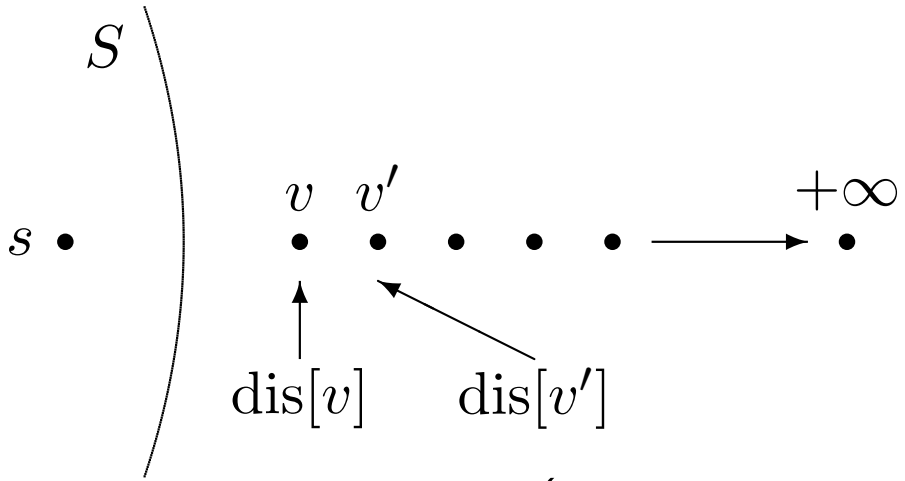
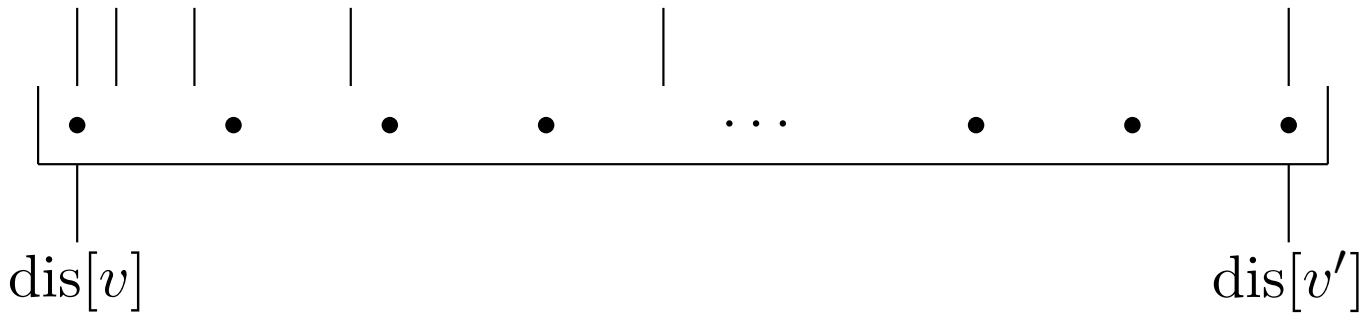
2.2 Dijkstra's Algorithmus mit Radix-Heaps



$\mathrm{d}(v) \in [0, \ldots, C]$ (alle Werte sind zwischen $\mathrm{dis}[v]$ und nC)



Wir verwenden Radix-Heaps (siehe dort), in der einstufigen Variante wie vorgestellt.

Satz 109

Dijkstra's Algorithmus mit einstufigen Radix-Heaps hat Zeitkomplexität $\mathcal{O}(m + n \log C)$.

Beweis:

siehe Satz 71. □

Verbesserungen:

1. zweistufige Heaps: $\boxed{\mathcal{O}(m + n \cdot \frac{\log C}{\log \log C})}$
2. mehrstufige Heaps (+Fibonacci-Heaps): $\boxed{\mathcal{O}(m + n\sqrt{\log C})}$

R.K. Ahuja, Kurt Mehlhorn, J.B. Orlin, R.E. Tarjan:
Faster algorithms for the shortest path problem
J. ACM **37**(2), pp. 213–223 (1990)

2.3 Bellman-Ford-Algorithmus

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 110

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

Beweis:

s.o. □

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 $\mathrm{d} : A \to \mathbb{R}_0^+$ 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** ; $1 \leq i,j \leq n$

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 $= c_{i,k+1}^{(k)} + c_{k+1,j}^{(k)}$. □

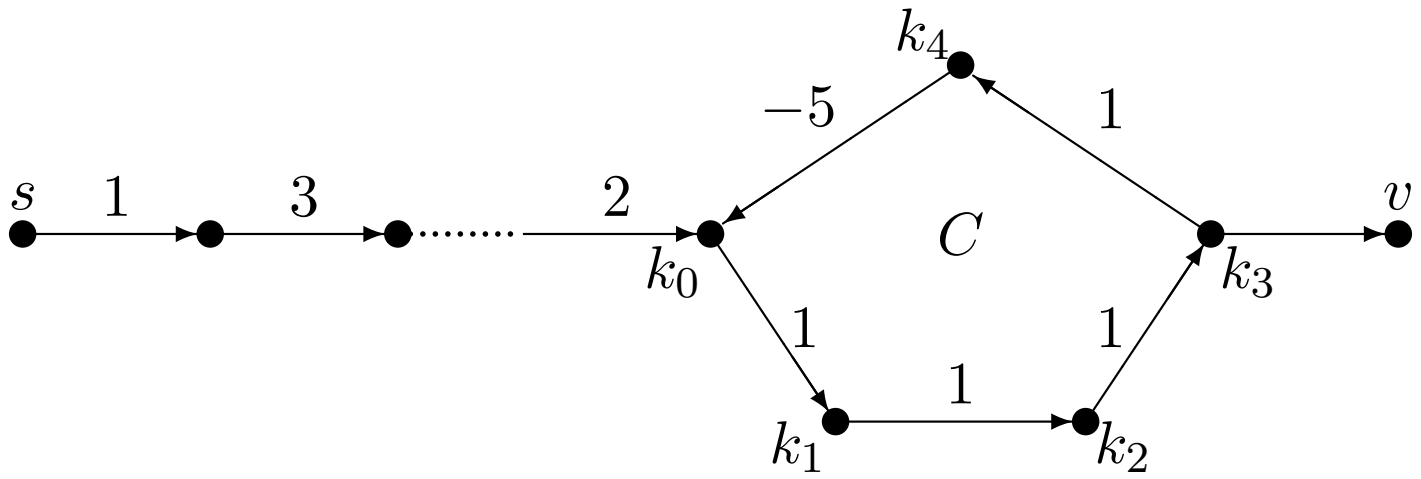
Satz 111

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

4. Digraphen mit negativen Kantengewichten

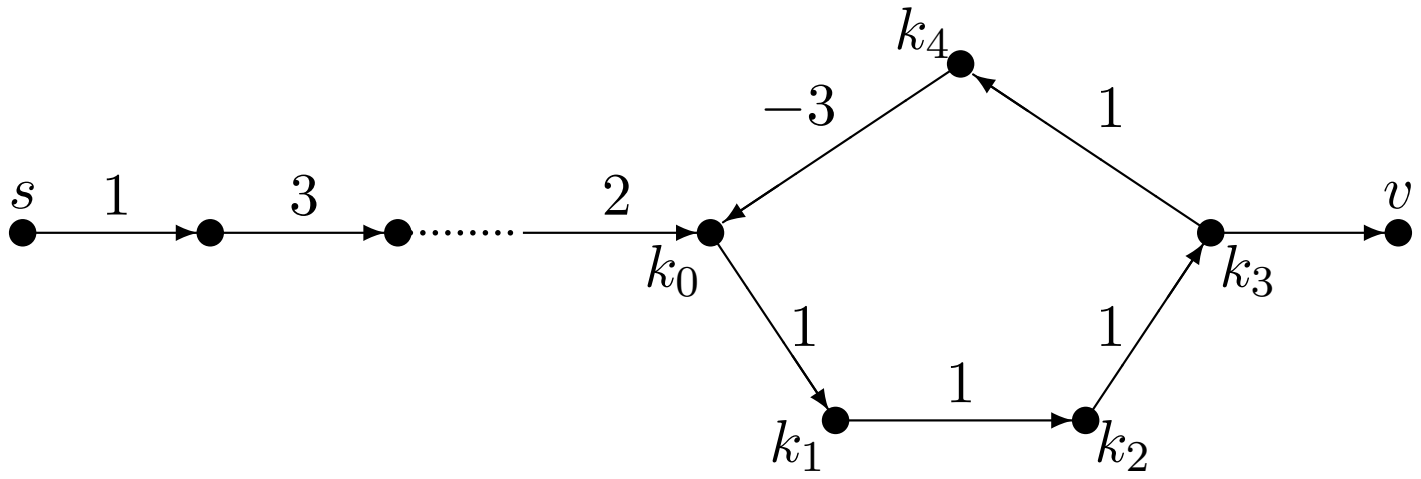
4.1 Grundsätzliches

Betrachte Startknoten s und einen Kreis C mit Gesamtlänge < 0.



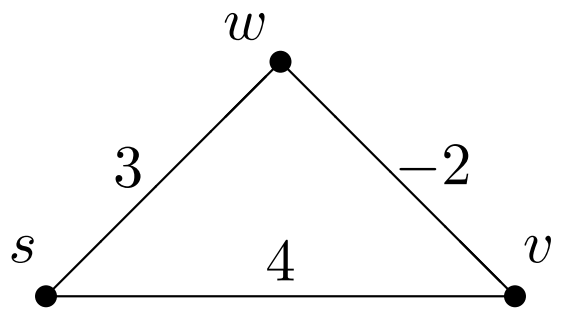
Sollte ein Pfad von s nach C und von C nach v existieren, so ist ein kürzester Pfad von s nach v nicht definiert.

Falls aber die Gesamtlänge des Kreises $C \geq 0$ ist,



dann ist der kürzeste Pfad wohldefiniert. Probleme gibt es also nur dann, wenn G einen Zyklus negativer Länge enthält.

Dijkstra's Algorithmus funktioniert bei negativen Kantenlängen nicht:



Bei diesem Beispielgraphen (der nicht einmal einen negativen Kreis enthält) berechnet der Dijkstra-Algorithmus die minimale Entfernung von s nach w fälschlicherweise als 3 (statt 2).

4.2 Modifikation des Bellman-Ford-Algorithmus

$B_k[i]$ gibt die Länge eines kürzesten gerichteten s-i-Pfades an, der aus höchstens k Kanten besteht. Jeder Pfad, der keinen Kreis enthält, besteht aus maximal $n-1$ Kanten. In einem Graphen ohne negative Kreise gilt daher:

$$\forall i \in V : B_n[i] \geq B_{n-1}[i]$$

Gibt es hingegen einen (von s aus erreichbaren) Kreis negativer Länge, so gibt es einen Knoten $i \in V$, bei dem ein Pfad aus n Kanten mit der Länge $B_n[i]$ diesen Kreis häufiger durchläuft als jeder Pfad aus maximal $n-1$ Kanten der Länge $B_{n-1}[i]$. Demnach gilt in diesem Fall:

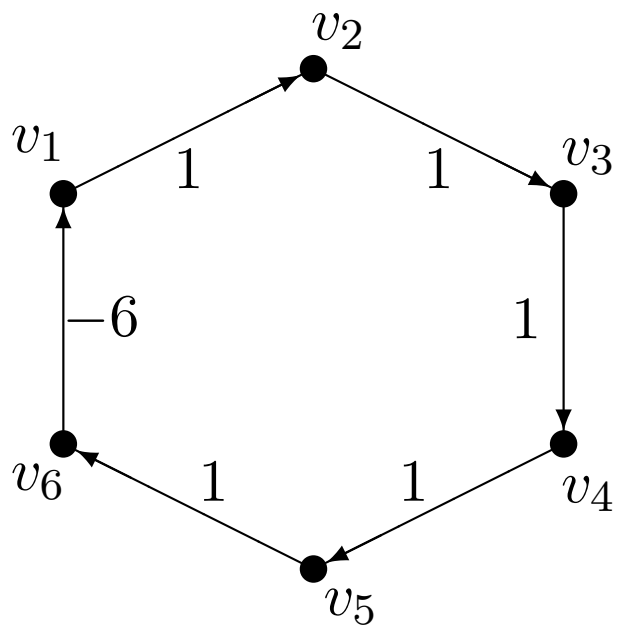
$$B_n[i] < B_{n-1}[i]$$

Man kann also in den Algorithmus von Bellman-Ford einen Test auf negative Kreise einbauen, indem man auch für alle $i \in V$ $B_n[i]$ berechnet und am Ende den folgenden Befehl einfügt:

for $i := 1$ **to** n **do**
  if $B_n[i] < B_{n-1}[i]$ **then** stop „Negativer Kreis“ **fi**

4.3 Modifikation des Floyd-Algorithmus

Falls kein negativer Kreis existiert, funktioniert der Algorithmus weiterhin korrekt.



$$c_{16}^6 = 5 = c_{16}^5$$

$$c_{61}^6 = -6 = c_{61}^5$$

$$c_{11}^6 = \min\{c_{11}^5, c_{16}^5 + c_{61}^5\} = -1$$

$\Rightarrow$ der Graph enthält einen negativen Kreis, gdw ein $c_{ii}^n < 0$ existiert.

Man kann also in den Algorithmus von Floyd einen Test auf negative Kreise einbauen, indem man am Ende den folgenden Befehl einfügt:

for $i := 1$ **to** n **do**
    if $c_{ii}^n < 0$ **then** stop „Negativer Kreis“ **fi**

4.4 Der Algorithmus von Johnson

Definition 112
Sei $\mathrm{d} : A \to \mathbb{R}$ eine Distanzfunktion. Eine Abbildung

$$r : V \to \mathbb{R}$$

heißt Rekalibrierung, falls gilt:

$$(\forall (u,v) \in A)[r(u) + \mathrm{d}(u,v) \geq r(v)]$$

Beobachtung: Sei r eine Rekalibrierung (für d). Setze $\mathrm{d}'(u,v) := \mathrm{d}(u,v) + r(u) - r(v)$. Dann gilt:

$$\mathrm{d}'(u,v) \geq 0$$

Sei $u = v_0 \to \cdots \to v_k = v$ ein Pfad. Dann ist:

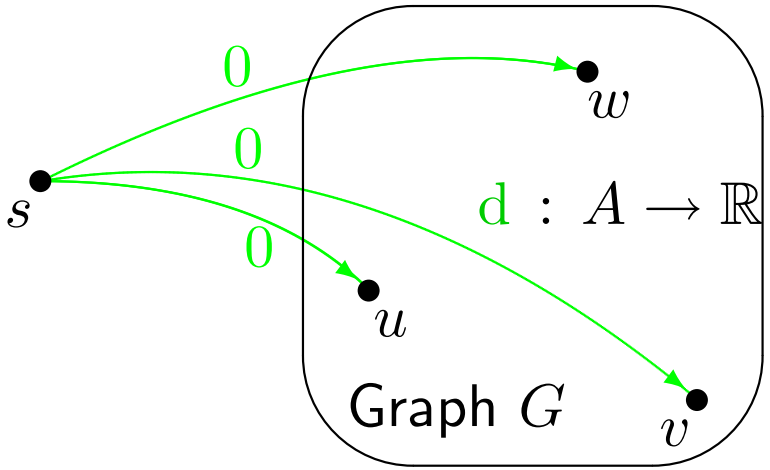
$$\text{d-Länge} := \sum_{i=0}^{k-1} \mathrm{d}(v_i, v_{i+1})$$

Demnach ist:

$$\begin{aligned}
\text{d}'\text{-Länge} &= \sum_{i=0}^{k-1} \mathrm{d}'(v_i, v_{i+1}) \\
&= \sum_{i=0}^{k-1} (\mathrm{d}(v_i, v_{i+1}) + r(v_i) - r(v_{i+1})) \\
&= \sum_{i=0}^{k-1} \mathrm{d}(v_i, v_{i+1}) + r(v_0) - r(v_k)
\end{aligned}$$

Also ist ein d-kürzester Pfad von u $(= v_0)$ nach v $(= v_k)$ auch ein d'-kürzester Pfad und umgekehrt. Nach einer Rekalibrierung kann man also auch die Algorithmen anwenden, die eine nichtnegative Distanzfunktion d voraussetzen (z.B. Dijkstra).

Berechnung einer Rekalibrierung:



Füge einen neuen Knoten s hinzu und verbinde s mit jedem anderen Knoten $v \in V$ durch eine Kante der Länge 0.

Berechne sssp von s nach allen anderen Knoten $v \in V$ (z.B. mit Bellman-Ford). Sei $r(v)$ die dadurch berechnete Entfernung von s zu $v \in V$. Dann ist r eine Rekalibrierung, denn es gilt:

$$r(u) + \mathrm{d}(u, v) \geq r(v) \,.$$

5. Zusammenfassung

	$d \geq 0$	d allgemein
sssp	D (Fibonacci): $\mathcal{O}(m + n \cdot \log n)$ D (Radix): $\mathcal{O}(m + n\sqrt{\log C})$	B-F: $\mathcal{O}(n \cdot m)$
apsp	D: $\mathcal{O}(n \cdot m + n^2 \min\{\log n, \sqrt{\log C}\})$ F: $\mathcal{O}(n^3)^{(*)}$	J: $\mathcal{O}(n \cdot m + n^2 \log n)$ F: $\mathcal{O}(n^3)$

Bemerkung$^{(*)}$: In der Praxis ist der Floyd-Algorithmus für kleine n besser als Dijkstra's Algorithmus.