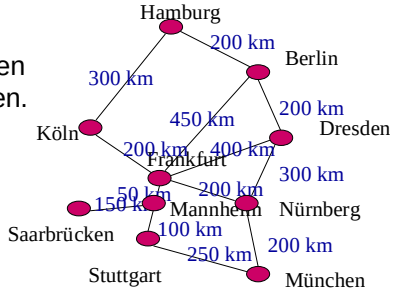
Single-Source Shortest Paths: SSSP

Gegeben ein Graph G=(V,E) und eine Gewichtsfunktion w: E $\rightarrow \Re$, die jeder Kante ein Gewicht in Form einer reellen Zahl zuordnet. Für einen vorgegebenen Knoten s (source) berechne man die kürzesten Pfade zu allen anderen Knoten.

Das Gewicht eines Pfades p = (v_0, v_1, ... , v_k) ist die Summe der Kantengewichte, die zu den aufeinander folgenden Knotenpaaren gehören.

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$



Wir definieren das Gewicht des kürzesten Pfades von s nach v als

$$\delta(s,v) = \begin{cases} \min\{w(p) \mid s \xrightarrow{p} v\} & \text{falls ein Pfad existiert} \\ \infty & \text{sonst} \end{cases}$$

Ein kürzester Pfad p von s nach v ist natürlich ein Pfad mit minimalem Gewicht: w(p) = δ(s,v)

Single-Destination Shortest Paths: SDSP

Gegeben ein Graph G=(V,E) und eine Gewichtsfunktion w: E $\rightarrow \Re$, die jeder Kante ein Gewicht in Form einer reellen Zahl zuordnet. Für einen vorgegebenen Knoten v (destination) berechne man die kürzesten Pfade von allen anderen Knoten zu v.

SDSP —Umkehrung der Kantenrichtung→ SSSP

Single-Pair Shortest Path: SPSP

Gegeben ein Graph G=(V,E) und eine Gewichtsfunktion w: E $\rightarrow \Re$, die jeder Kante ein Gewicht in Form einer reellen Zahl zuordnet. Für ein vorgegebenes Paar (s,v) von Knoten berechne man den kürzesten Pfad von s nach v.

Wende SSSP für Quelle s an und bestimme alle Pfade, unter anderem auch den kürzesten Pfad nach v.

All-Pair Shortest Paths: APSP

Gegeben ein Graph G=(V,E) und eine Gewichtsfunktion w: E $\rightarrow \Re$, die jeder Kante ein Gewicht in Form einer reellen Zahl zuordnet. Berechne für jedes Paar s,v von Knoten den kürzesten Weg von s nach v.

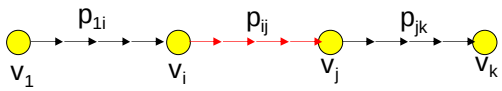
Dieses Problem kann mit SPSP für alle Knoten(paare) gelöst werden. Es gibt jedoch effizientere Verfahren, die nicht auf SPSP basieren.

Lemma [1]: (Teilpfade von kürzesten Pfaden sind kürzeste Pfade)

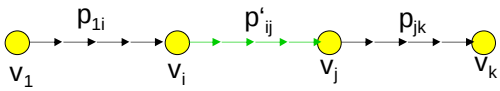
Gegeben ein gewichteter Graph G=(V,E) mit Gewichtsfunktion w: E $\rightarrow \Re$. Sei p =(v_1, v_2, ... , v_k) ein kürzester Pfad von Knoten v_1 nach v_k und sei für jedes Paar i, j mit $1 \leq i \leq j \leq k$ $p_{ij} = (v_i , v_{i+1} , ... , v_j)$ der Teilpfad von Knoten v_i nach Knoten v_j. Dann ist p_{ij} ein kürzester Pfad von v_i nach v_j.

Beweis:

$$w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$$



Nehmen wir an, dass p_{ij} nicht ein kürzester Pfad von v_i nach v_j ist, so gibt einen kürzeren Pfad p'_{ij} mit $w(p'_{ij}) < w(p_{ij})$.



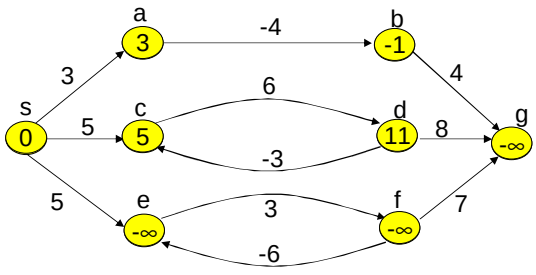
Dann folgt: $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk}) > w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$

Der Pfad p wäre also nicht der kürzeste Pfad von v_1 nach v_k . Widerspruch.

■

Kanten mit negativen Gewichten

Auch negative reelle Zahlen tauchen manchmal als Kantengewichte auf. Eine besondere Rolle können unter diesen Umständen Kreise bzw. Rundgänge mit negativen Gesamtgewicht spielen.



Falls es einen negativen Zyklus auf einem Pfad gibt, dann ist der kürzeste Pfad nicht wohldefiniert, da man durch Hinzufügen weiterer Zyklusrunden das Pfadgewicht beliebig klein machen kann:

(s, e, f, e, f, e, f, e, f, ...)

Dann existiert kein kürzester Pfad und man setzt $\delta(s,v) = -\infty$.

Gibt es kürzeste Pfade, die Zyklen enthalten?

(1) Zyklen mit negativen Gewicht? Nein, denn es existiert kein wohldefinierter kürzester Pfad!

(2) Zyklen mit Gewicht größer 0 ? Nein, denn man kann diese Zyklen aus den Pfaden streichen und dadurch das Gesamtgewicht verkleinern.

(3) Zyklen mit Gewicht gleich 0 ? Ja, aber es gibt auch kürzeste Pfade ohne die Zyklen mit gleichem Gewicht.

Da wir nur kürzeste Pfade suchen werden, die keine Zyklen enthalten, können die kürzesten Pfade höchstens jeden Knoten einmal enthalten.

=> Die gesuchten kürzesten Pfade enthalten höchstens n Knoten und n-1 Kanten.

Datenstrukturen für die Berechnung der kürzesten Pfade

- Die Gewichte der Kanten werden in dem Knotenfeld

```
edge_array<float> weight(G);
```

gespeichert.

- Die Gewichte (Längen) der aktuellen kürzesten Pfade zu den Knoten werden in dem Knotenfeld

```
node_array<float>  length(G, INFINITY);
```

gespeichert.

- Die Vorgängerknoten auf den aktuellen kürzesten Pfaden zu den Knoten werden wie üblich in dem Knotenfeld

```
node_array<node> parent(G, nil);
```

gespeichert. Mit Hilfe des parent-Feldes kann man den kürzesten Pfad zu einem Knoten v bestimmen.

Wie bei Breadth–First Search (BFS) berechnen wir wieder den Vorgänger-Graphen oder Predecessor-Graph $G_p=(V_p,E_p)$ mit den folgenden Knoten und Kantenmengen:

$$V_p = \{v \in V \mid parent[v] \neq nil\} \cup \{s\}$$

$$E_p = \{(parent[v], v) \in E \mid v \in V_p\}$$

Wir werden zeigen, dass die später in diesem Abschnitt diskutierten Algorithmen Bäume mit den folgenden Eigenschaften als Vorgänger-Graphen besitzen:

(1) V_p ist die Menge aller Knoten, die von s aus erreicht werden können.

(2) Der eindeutige (Baum)Pfad in G_p von s zu einem Knoten v ist ein kürzester Pfad von s nach v.

Relaxation

- Die in diesem Abschnitt vorgestellten Techniken verwenden eine Prozedur mit dem Namen Relaxation
- Solange man den kürzesten Pfad von s zu einem Knoten v noch nicht gefunden hat, speichert man in length[v] eine obere Schranke für die Länge des kürzesten Pfades von s nach v.
- Die Knoten werden hierbei am Start wie folgt initialisiert (Initialize-Single-Source(G,s) :

```
node_array<float> length(G, INFINITY);
node_array<node> parent(G, nil);
length[s] = 0;
```

Eine Kante (u,v) zu „entspannen“, bedeutet, dass man testet, ob man einen neuen kürzesten Pfad von s nach v bekommt, wenn man den aktuellen kürzesten Pfad von s nach u betrachtet und die Kante (u,v) hinzufügt.

```
void relax(    edge e,
               const edge_array<float>& weight,
               node_array<float>& length,
               node_array<node>& parent)
{
        float help = length[source(e)] + weight[e];
        if ( length[target(e)] > help)
        {
               length[target(e)] = help;
               parent[target(e)] = source(e);
        }
}
```


Lemma [2]: (Path-Relaxation Property)

Falls $p = (v_0, v_1, \ldots, v_k)$ ein kürzester Pfad von $s = v_0$ nach v_k ist und falls die Kanten von p in der Reihenfolge $(v_0,v_1), (v_1,v_2), \ldots, (v_{k-1}, v_k)$ relaxiert werden, dann ist

$$length[v_k] = \delta(s,v_k).$$

Diese Eigenschaft gilt unabhängig von anderen Relaxationsschritten, die eventuell vorher, nachher oder auch zwischendurch ausgeführt werden.

Beweis:
Man zeigt mittels vollständiger Induktion, dass nach der Relaxation von (v_{i-1},v_i) folgendes gilt (hierbei setzen wir voraus, dass die anderen Kanten bereits in der im Lemma angegebenen Reihenfolge relaxiert wurden):

$$length[v_i] = \delta(s,v_i)$$

Induktionsstart: i = 0: length[s] = 0 = δ(s,s).
Induktionsannahme: Die Aussage gilt für alle k < i.

Induktionsschluss:

$$\delta(s,v_i) = \sum_{k=1}^{i} w(v_{k-1},v_k)$$

$$= \sum_{k=1}^{i-1} w(v_{k-1},v_k) + w(v_{i-1},v_i)$$

$$= \delta(s,v_{i-1}) + w(v_{i-1},v_i)$$

$$= length[v_{i-1}] + w(v_{i-1},v_i) = length[v_i]$$

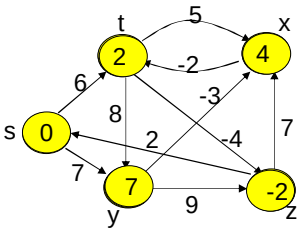
Induktionsannahme

nach der Relaxation der Kante (v_{i-1},v_i)

Der Bellman-Ford-Algorithmus

Der Bellman-Ford-Algorithmus löst den allgemeinen Fall des SSSP-Problems, d.h., negative Kantengewichte sind erlaubt. Der Algorithmus gibt den Wert einer boolschen Variable zurück. Die Variable ist TRUE, falls es keinen negativen Zyklus gibt und der Algorithmus dann erfolgreich alle kürzesten Pfade zu den von s aus erreichbaren Knoten bestimmt hat. Gibt es einen negativen Zyklus, der von s aus erreichbar ist, so gibt der Algorithmus FALSE zurück.

```
Bellman-Ford(G,w,s):
(1) Initialize-Single-Source(G,s)
(2) Für i = 1, ... , n-1 :
(3)       Für alle Kanten e = (u,v) ∈ E: relax(e, ...)
(4) Für alle Kanten e = (u,v):
(5)       Falls length[v] > length[u] + weight[e] :
(6)             Gib FALSE zurück.
(7) Gib TRUE zurück.
```



Satz [1]: Der Bellman-Ford-Algorithmus hat Laufzeit O(nm), wobei n die Zahl der Knoten in V und m die Zahl der Kanten in E ist.

```
bool Bellmann_Ford(     const graph& G,
                        const edge_array<float>&  weight,
                        node s,
                        node_array<float>& length,
                        node_array<node>& parent)
{
// Wir nehmen an, dass die Initialisierung schon erfolgt ist!!!
        node v;
        edge e;
        for (int i = 1; i < G.number_of_nodes()-1; i++)
        {
                forall_edges(e,G)  relax(e, weight, length, parent);
        }
        forall_edges(e,G)
        {
                if (length[G.target(e)] > (length[G.source(e)] + weight[e]))
                        return false;
        }
        return true;
}
```


Korrektheit des Bellman-Ford-Algorithmus

Es muss gezeigt werden, dass der auf der vorhergehenden Seite beschriebene Algorithmus die kürzesten Pfade (Gewichte der kürzesten Pfade) korrekt berechnet und dass er auch negative Zyklen durch den beschriebenen Test identifizieren kann (Ausgabe: FALSE).

Lemma [3]:

Sei G = (V,E) ein gewichteter und gerichteter Graph mit Quelle s und Gewichtsfunktion w: $E \rightarrow \Re$. Enthält der Graph G keine negativen Zyklen, die von s aus erreicht werden können, dann berechnet der Algorithmus die Gewichte (Längen) der kürzesten Pfade korrekt, d.h., für alle erreichbaren Knoten gilt: length[v] = $\delta(s,v)$.

Beweis: Anwendung der „Path-Relaxation-Property“.

Sei v irgend ein Knoten, der von s aus erreichbar ist, und sei p = $(v_0, \ldots, v_k)$ mit $s = v_0$ und $v = v_k$ ein azyklischer, kürzester Pfad von s nach v.
Der Pfad p hat höchstens n-1 Kanten. Daher ist k<n.
In Zeile (2) und (3) wird jede Kante in jeder der n-1 Iterationen entspannt.
Unter den Kanten, die in der i-ten Iteration entspannt werden, ist auch (v_{i-1},v_i).
Aus der Path-Relaxation-Property folgt daher:

$$length[v] = length[v_k] = \delta(s,v_k) = \delta(s,v)$$ ■

Korrektheit des Bellman-Ford-Algorithmus

Aus Zeitgründen verzichten wir hier auf den vollständigen Korrektheitsbeweis (FALSE, falls negativer Zyklus vorliegt, usw.), den Sie auf den Seiten 590-591 in dem Buch „Introduction to Algorithms“ von Cormen et al. nachlesen können.

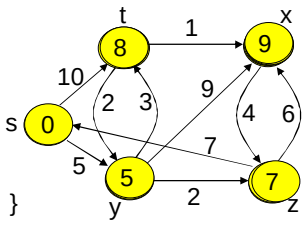
Der Algorithmus von Dijkstra

Dijkstra‘s Algorithmus löst das SSSP-Problem für gerichtete Graphen mit Gewichtsfunktionen, die den Kanten nur positive Kantengewichte zuordnen.

Für alle Kanten $e = (u,v) \in E$ gilt also: $w(u,v) \geq 0$.

Der Algorithmus verwaltet eine Knotenmenge S, für die bereits ein kürzester Pfad von der Quelle s aus identifiziert wurde.
Aus der restlichen Knotenmenge $V \setminus S$ wählt man in jeder Iteration den Knoten u mit dem kleinsten gespeicherten Wert für den „aktuellen“ kürzesten Pfad, addiert u zu S und entspannt alle Kanten, die u verlassen.

```
Dijkstra(G,w,s):
(1) Initialize-Single-Source(G,s)
(2) S = ∅
(3) Q = V
(4) Solange Q != ∅:
(5)       u = Extract-Min(Q),  d.h., u ∈ Q hat den kürzesten Pfad  length[u]
(6)       S = S ∪ {u}
(7)       Für jede Kante e = (u,v) ∈ E, die u verlässt:  relax(e, ....)
```



Q={ , , , , }

Dijkstra(G,w,s):
(1) Initialize-Single-Source(G,s)
(2) S = ∅
(3) Q = V
(4) Solange Q != ∅:
(5) u = Extract-Min(Q), d.h., u ∈ Q hat den kürzesten Pfad length[u]
(6) S = S ∪ {u}
(7) Für jede Kante e = (u,v) ∈ E, die u verlässt: relax(e)

Satz [2]:

Der Algorithmus von Dijkstra berechnet für einen gewichteten und gerichteten Graphen G = (V,E), eine Gewichtsfunktion w: E→ℜ mit nicht negativen Gewichten und eine Quelle s die kürzesten Pfade von s zu allen erreichbaren Knoten v von V korrekt, d.h., length[v] = δ(s,v)

Beweis:

Wir beweisen die folgende Invariante für die Iteration:

Am Start jeder Iteration in der „Solange"-Schleife (while-loop) gilt für alle Knoten v ∈ S: length[v] = δ(s,v)

Es genügt zu zeigen, dass für jeden Knoten v, der zu S hinzugefügt wird, gilt:

length[v] = δ(s,v)

Initialisierung: S = ∅. Daher ist die Invariante erfüllt.

Aufrechterhaltung: Wie wollen zeigen, dass in jeder Iteration für den hinzugefügten Knoten v gilt: length[v] = δ(s,v)

Wir nehmen an, dass es Knoten in S gibt, die diese Eigenschaft nicht besitzen. Sei u der erste in S eingefügte Knoten, der diese Eigenschaft nicht besitzt, d.h.,

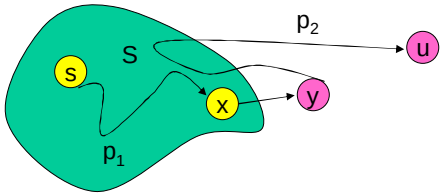
length[u] ≠ δ(s,u)

Shortest Paths

Wir betrachten den Zeitpunkt, zu dem Knoten u in S eingefügt wurde:

Es gilt: $u \neq s$ und $S \neq \emptyset$.

Es muss einen Pfad von s nach u geben, denn sonst wäre aufgrund der Initialisierung length[u] = δ(s,u) = ∞ .Daher gibt es auch einen kürzesten Pfad p.



Bevor u zu S addiert wird, gilt:

$s \in S$ und $u \in V \setminus S$

Sei y der erste Knoten auf dem kürzesten Pfad p von s nach u, der zu diesem Zeitpunkt nicht zu S gehört und x sein Vorgänger in S.

Wir zerlegen p in die folgenden Komponenten: $s \xrightarrow{p_1} x \rightarrow y \xrightarrow{p_2} u$

Wir zeigen nun, dass zu dem Zeitpunkt, als u zu S hinzugefügt wurde, gilt:

length[y] = δ(s,y)

Da $x \in S$ ist und vor u in S eingefügt wurde, gilt für x: length[x] = δ(s,x)

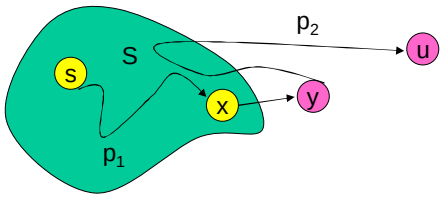
Als x zu S addiert wurde, wurde die Kante (x,y) entspannt.

Shortest Paths

length[y] = length[x] + w(x,y) = δ(s,x) + w(x,y) = δ(s,y)

Da y vor u auf einem kürzesten Pfad von s nach u auftaucht und alle Kantengewichte größer gleich 0 sind , folgt :

$\delta(s,y) \leq \delta(s,u)$



$length[y] = \delta(s,y) \leq \delta(s,u) \leq length[u]$

Da aber beide Knoten u und y in Q=V \ S waren, als u als Knoten mit minimalem Gewicht bestimmt wurde, ist

$length[y] = \delta(s,y) = \delta(s,u) = length[u]$

Da δ(s,u) = length[u] ist, erhalten wir hier einen Widerspruch zur Wahl von u.

Ende des Algorithmus: Am Ende ist Q leer und S folglich gleich der Knotenmenge V, so dass also für alle Knoten von V gilt:

$length[v] = \delta(s,v)$

■

Laufzeit von Dijkstra

Implementiert man die Extract-Min-Funktion mittels eines Fibonacci-Heap, so kann man in Zeit O(log(n)) das Minimum „extrahieren“. Entspannt man eine Kante, so ändert sich eventuell der aktuelle kürzeste Pfad und man muss im Fibonacci-Heap eine Decrease-Key-Operation ausführen, die amortisiert Zeit O(1) kostet.

Satz [3]:

Die amortisierte Laufzeit von Dijkstras Algorithmus für einen gerichteten Graphen G = (V,E) mit n Knoten und m Kanten ist O(n log(n) + m).

Dijkstra(G,w,s):	
(1) Initialize-Single-Source(G,s)	O(n)
(2) S = ∅	O(1)
(3) Q = V	O(n log(n))
(4) Solange Q != ∅:	O(n)
(5) u = Extract-Min(Q)	O(n log(n))
(6) S = S ∪ {u}	O(n * 1) = O(n)
(7) Für jede Kante e = (u,v) ∈ E, die u verlässt: relax(e)	O(m * 1) = O(m)