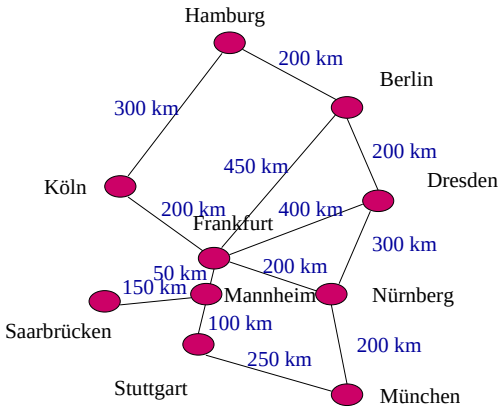
Routenplaner



Berechne den optimalen (kürzesten, schnellsten) Weg von X nach Y.

Das Gewicht eines Pfades

Gegeben ein Graph G=(V,E) und eine Gewichtsfunktion w

$$w : E \rightarrow \Re$$

die jeder Kante ein Gewicht (eine reelle Zahl) zuordnet.

Das Gewicht eines Pfades

$$p = (v_0, v_1, \ldots , v_k)$$

ist die Summe der Kantengewichte der aufeinander folgenden Knotenpaare.

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

Single-Source Shortest Paths: SSSP

Ein kürzester Pfad p* von s nach v ist ein Pfad mit minimalem Gewicht:

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

$$w(p^*) = d(s,v)$$

Gegeben ein Graph G=(V,E) und eine Gewichtsfunktion w: E $\rightarrow \Re$. Berechne die kürzesten Pfade von einem vorgegebenen Knoten s (source) zu allen anderen Knoten von G.

Single-Destination Shortest Paths: SDSP

SDSP —Umkehrung der Kantenrichtung→ SSSP

Single-Pair Shortest Path: SPSP

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

All-Pairs Shortest Paths: APSP

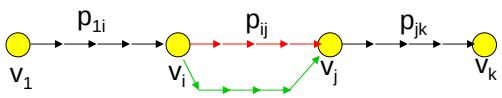
Dieses Problem kann mit SSSP für alle Knoten(paare) gelöst werden. Es gibt jedoch effizientere Verfahren, die nicht auf SSSP basieren.

Shortest Paths

Sei $p = v_1\ v_2\ ..v_i\ ...\ v_j\ ...\ v_k$ ein kürzester Pfad von v_1 nach v_k und sei

$\forall i, j : 1 \le i \le j \le k$ $p_{ij} = v_i\ v_{i+1}\ ...\ v_j$ der Teilpfad in p von v_i nach v_j.

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



Wir nehmen an, dass p^*_{ij} ein kürzerer Pfad von v_i nach v_j ist.

$$w(p^*_{ij}) < w(p_{ij}).$$

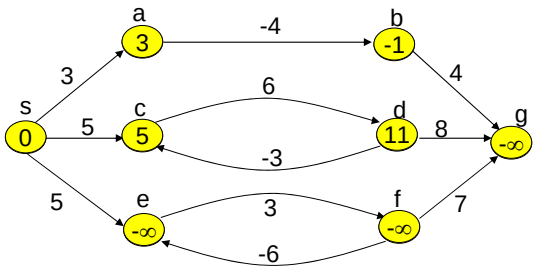
$$w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$$
$$> w(p_{1i}) + w(p^*_{ij}) + w(p_{jk})$$ Widerspruch

Teilpfade von kürzesten Pfaden sind kürzeste Pfade (bezüglich der entsprechenden Knotenpaare).

Shortest Paths

Kanten mit negativen Gewichten

Auch negative reelle Zahlen tauchen manchmal als Kantengewichte auf.



Falls es einen negativen Zyklus gibt, kann man die Pfadlängen beliebig klein machen kann => Der kürzeste Pfad ist dann nicht wohldefiniert!!!

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

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

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

- Datenstruktur für die Gewichte der Kanten (Teil der Eingabe):

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

- Datenstruktur für die Gewichte (Längen) der aktuellen kürzesten Pfade:

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

- Datenstruktur für die Vorgängerknoten auf den aktuellen kürzesten Pfaden zu den Knoten:

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

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

Shortest Paths

Die Shortest-Path-Algorithmen berechnen den Vorgänger-Graphen

$$G_p = (V_p, E_p)$$ wobei

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

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

Vorgänger-Graphen sind Bäume mit den folgenden Eigenschaften:

(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.

Shortest Paths

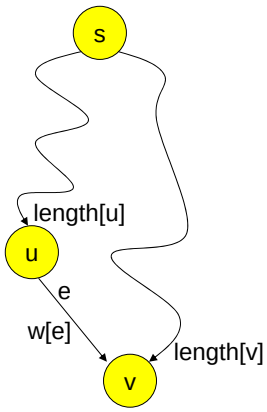
Relaxation

- wichtigste Prozedur bei der Suche nach kürzesten Pfaden!!!
- length[v] = Länge des kürzesten **bisher gefundenen** Pfads von s nach v.
- parent[v] = Vorgängerknoten von v
  auf dem kürzesten **bisher gefundenen** Pfad von s nach v.
- Die Knoten werden hierbei wie folgt initialisiert

```
// Initialize-Single-Source(G,s) :
node_array<float> length(G, INFINITY);
node_array<node> parent(G, nil);
length[s] = 0;
```

Was bedeutet es, eine Kante (u,v) zu „entspannen“ ?

Shortest Paths

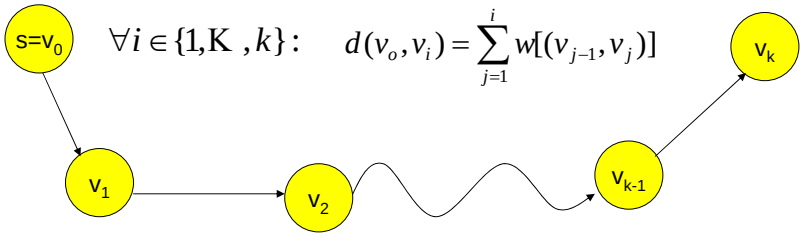



```
// e sei gleich (u,v):  source(e) = u : target(e) = v

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

Shortest Paths

Sei p = v_0 v_1 ... v_k ein kürzester Pfad von s = v_0 nach v_k



Entspannt man zunächst die erste Kante (v_0 , v_1), so gilt:

$$length[v_1] = length[v_0] + w[(v_0, v_1)]$$
$$= 0 + w[(v_0, v_1)] = d(v_0, v_1)$$

Entspannt man anschließend die zweite Kante (v_1 , v_2), so gilt:

$$length[v_2] = length[v_1] + w[(v_1, v_2)]$$
$$= d(v_0, v_1) + w[(v_1, v_2)] = d(v_0, v_2)$$

Durch Induktion folgt:

Shortest Paths

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

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

Induktions annahme →

$$\leq d(s, v_{i-1}) + w[(v_{i-1}, v_i)]$$

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

$$\leq \sum_{k=1}^{i} w[(v_{k-1}, v_k)] = d(s, v_i)$$

■

Shortest Paths

Lemma [1]: (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

$$length[v_k] = d(s,v_k).$$

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

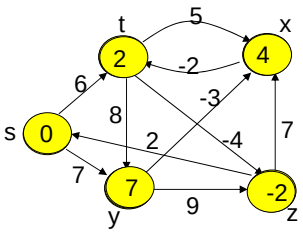
Shortest Paths

Der Bellman-Ford-Algorithmus

Der Bellman-Ford-Algorithmus löst den allgemeinen Fall des SSSP-Problems, d.h., negative Kantengewichte sind erlaubt.

Rückgabewert: TRUE falls es keinen negativen Zyklus
FALSE sonst

```
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] + w[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.

Shortest Paths

```
bool Bellmann_Ford(     const graph& G,
                        const edge_array<float>&  w,
                        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, w, length, parent);
        }
        forall_edges(e,G)
        {
                if (length[G.target(e)] > (length[G.source(e)] + w[e]))
                        return false;
        }
        return true;
}
```

Shortest Paths

Korrektheit des Bellman-Ford-Algorithmus

Sei $p = v_0 v_1 .... v_k$ ein kürzester Pfad von s zu einem anderen Knoten.

Da der Pfad höchstens n-1 Kanten hat, gilt $k < n$.

Die Relaxierung aller Kanten im Algorithmus wurde n-1 mal durchgeführt.

n-1 {

$e_1\ e_2\ ....\ e_i = (v_0,v_1)\ .....\ e_m$

$e_1\ e_2\ .....e_j=(v_1,v_2)\ .....\ e_m$

$e_1\ e_2\ .....e_l = (\ v_2,v_3)\ .....\ e_m$

...

...

...

$e_1\ e_2\ ..e_z = (\ v_{k-1}\ ,v_k)\ .....\ e_m$

Korrektheit des Bellman-Ford-Algorithmus folgt aus Lemma [1].

Shortest Paths

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.

Shortest Paths

Der Algorithmus von Dijkstra

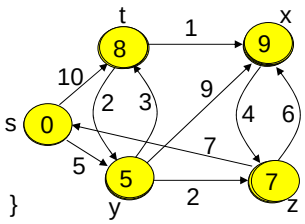
Für alle Kanten e =(u,v) $\in$ E gilt: w[(u,v)] = w[e] $\geq$ 0.

Der Algorithmus verwaltet eine Knotenmenge S

S = { v $\in$V | ein kürzester Pfad von s zu v wurde bereits identifiziert }

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

Shortest Paths



Q={ , , , , }

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

Shortest Paths

21

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] = d(s,v)

Beweis:

Wir beweisen die folgende Invariante für die Iteration:

Für jeden Knoten v gilt: Wenn v zu S hinzugefügt wird, dann gilt

length[v] = d(s,v)

Initialisierung: S = {s}. Es gilt: length[s] = 0 = d(s,s)

Indirekter Beweis: 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] ≠ d(s,u) ⟹

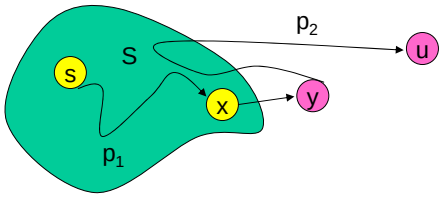
Shortest Paths

22

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

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

Sei p der kürzeste Pfad von s nach u.



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.

$p = s \xrightarrow{p_1} x \rightarrow y \xrightarrow{p_2} u$

Es gilt: length[x] = d(s,x)

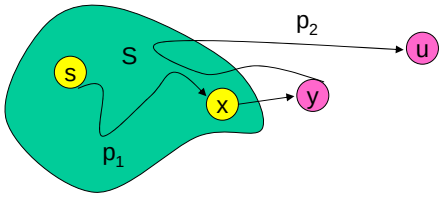
Als x zu S addiert wurde, wurde die Kante (x,y) entspannt. ⟹

Shortest Paths

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

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

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



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

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

$$length[y] = d(s,y) = d(s,u) = length[u]$$

Da d(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:

$$length[v] = d(s,v)$$

■

Shortest Paths

Satz [3]:
Verwendet man einen binären Min-Heap, so ist die Laufzeit von Dijkstras Algorithmus für einen gerichteten Graphen G = (V,E) mit n Knoten und m Kanten O((n+m) log(n)).

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 log n)

Shortest Paths

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 [4]:

Die amortisierte Laufzeit von Dijkstras Algorithmus (mit Fibonacci-Heap) 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)