Kürzeste Wege

Eines der ältesten dokumentierten **kürzesten-Wege-Probleme** stammt aus Schillers **"Wilhelm Tell"**: Tell befindet sich nach dem Apfelschuß am Ufer des Vierwaldstädter Sees nahe beim Ort **Altdorf**. Er muß vor dem Reichsvogt Hermann Geßler die Hohle Gasse in **Küßnacht** erreichen:

Tell: *Nennt mir den nächsten Weg nach Arth und Küßnacht.*

Fischer: *Die offene Straße zieht sich über Steinen. Doch einen kürzern Weg und heimlicheren kann euch mein Knabe über Lowrenz führen.*

Voraussetzungen:

Der **gerichtete Graph G=(V,R,α,ω)** ist **parallelenfrei** und wird üblich kurz mit **G=(V,R)** bezeichnet.

Die **Funktion c:R$\rightarrow\Re$** sei eine **Gewichtsfunktion**, welche den gerichteten Kanten (Pfeilen) von G **"Längen"** oder **"Kosten"** zuweist.

Parallelen spielen bei der Berechnung kürzester Wege keine Rolle. Man kann die kürzesten Parallelen im Graphen behalten und alle anderen vorab eliminieren.

Man unterscheidet folgende Typen von **kürzeste-Wege-Probleme**:

SPP (Single Pair Shortest Path Problem)

Instanz: Gerichteter Graph **G=(V,R)** mit Gewichten **c:R$\rightarrow\Re$**, sowie **zwei Knoten s,t $\in$ V**
Gesucht: Ein kürzester Weg (die Länge eines kürzesten Weges) von **s nach t** in **G**

SSP (Single Source Shortest Path Problem)

Instanz: Gerichteter Graph **G=(V,R)** mit Gewichten **c:R$\rightarrow\Re$**, sowie **ein Knoten s $\in$ V**
Gesucht: Kürzeste Wege (die Längen der kürzesten Wege) von **s nach v für alle v$\in$ G**

APSP (All Paires Shortest Path Problem)

Instanz: Gerichteter Graph **G=(V,R)** mit Gewichten **c:R$\rightarrow\Re$**

Gesucht: Für **jedes Paar u,v$\in$V** ein kürzester Weg (die Längen eines kürzesten Weges) von **u nach v**

Bei allen Problemen kann darüberhinaus nach **allen kürzesten Wegen** gefragt werden.

Definition Länge eines Weges, Distanz:

Sei **G=(V,R)** ein Graph und **c:R→ℜ** eine Gewichtsfunktion. Die **Länge c(P)** eines **Weges P=(v_0,r_1,r_2,...,r_k,v_k)** in **G** ist definiert durch

$$c(P) = \sum_{i=1}^{k} c(r_i)$$

Für einen Weg **P=(v_0)** ohne Kanten ergibt sich **c(P)=0**.

Die **Distanz $dist_c$(u,v,G)** zweier Knoten **u,v∈V** bzgl. der **Gewichtsfunktion c** definiert man durch

$dist_c$(u,v,G) := inf {c(P): P ist ein Weg in G von u nach v }

Falls **u ≠ v** und **kein Weg** von **u** nach **v** existiert, dann ist **$dist_c$(u,v,G)=+∞**

Falls **Graph G** aus dem Kontext klar ist, wird kürzer **$dist_c$(u,v)** geschrieben.

Sei **s∈V** ein Knoten aus **G**, dann gilt **$dist_c$(s,v) ≤ $dist_c$(s,u) + $dist_c$(u,v)** für alle **(u,v)∈R.**

Algorithmus zur Initialisierung der kürzeste Wegesuche

Werte **d[v] (v∈V)** sind die obere Schranke für **$dist_c$(s,v)**, die im Lauf des Algorithmus angepasst werden. Die Menge π[v] ist die Menge der direkten Vorgängerknoten des Knotens v.

```
INIT(G,s) {
   for all v∈V {
        d[v] := +∞
        π[v] := ∅
   }
   d[s] := 0
}
```

Bei Aufruf von **TEST(u,v)** für eine Kante **(u,v)** wird geprüft, ob es über **u** und die Kante **(u,v)** einen kürzeren Weg von **s** nach **v** als die aktuelle obere Schranke **d[v]** gibt:

```
TEST(u,v) {
    if (d[v] > d[u] + c(u,v))  {
        d[v] := d[u] + c(u,v);
        π[v] := u;
    }
}
```

Algorithmus von Dijkstra

Die kürzesten Wege von einem Knoten zu allen anderen Knoten in **V** werden gesucht. Es wird vorausgesetzt, daß $\mathbf{c(r) \geq 0}$ für alle $\mathbf{r \in R}$ ist. Es wird angenommen, daß alle Knoten von **s** aus erreichbar sind. Für alle nicht von **s** erreichbaren Knoten **v** gilt $\mathbf{dist_c(s,v) = +\infty}$.

Der Algorithmus von Dijkstra arbeitet mit einer "Wellenfront"-Strategie: im Verfahren halten wir eine Menge $\mathbf{PERM \subseteq V}$ von "permanent markierten" Knoten, d.h. von Knoten **v**, für die bereits $\mathbf{d[v] = dist_c(s,v)}$ gilt. Anfangs ist $\mathbf{PERM = \emptyset}$. In jeder Iteration entfernen wir einen Knoten **u** aus **Q := V \ PERM** mit **minimalem Schlüsselwert d[u]** und fügen **u** $\in$ **PERM** hinzu. Anschließend testen wir alle Kanten $\mathbf{(u,v) \in R}$.

Bei Abbruch des Algorithmus von Dijkstra gilt $\mathbf{d[v] = dist_c(s,v)}$ für alle $\mathbf{v \in V}$. Der Graph $\mathbf{G_\pi}$ ist ein Baum kürzester Wege bezüglich **s**.

```
DIJKSTRA(G, c, s) {
  Input:  Gerichteter Graph G=(V,R) in Adjazenzlistendarstellung, eine nichtnegative
          Gewichtsfunktion c: R → ℜ+  und ein Knoten s∈V, von dem alle anderen Knoten
          erreichbar sind
  Output: Für alle v∈V die Distanz d[v] = dist_c(s,v) sowie ein Baum G_π kürzester Wege
          von s aus.
  INIT(G,s)
  PERM := ∅   // PERM ist Menge der "permanent markierten" Knoten
  while ( PERM ≠ V) {
        Wähle u ∈ Q := V \ PERM mit minimalem Schlüsselwert d[u]
        PERM := PERM ∪ {u}
        for all v ∈ Adj[u] \ PERM
            TEST(u,v)
  }
  return d[] und G_π              // G_π wird durch die Vorgängerknoten π aufgespannt
}
```

Priority Queue

Warteschlange Q, in der die Elemente x auf- bzw. absteigend eingeordnet werden:

MAKE()	erstellt eine leere Prioritätenwarteschlange
INSERT(Q,x)	fügt ein Element x in Schlange Q ein
MINIMUM(Q)	liefert (einen Zeiger auf) das Element mit minimalem Schlüsselwert
MAXIMUM(Q)	liefert (einen Zeiger auf) das Element mit maximalem Schlüsselwert
EXTRACT-MIN(Q)	liefert (Zeiger auf) das Element mit minimalem Schlüsselwert und entfernt das Element aus der Warteschlange
EXTRACT-MAX(Q)	liefert (Zeiger auf) das Element mit maximalem Schlüsselwert und entfernt das Element aus der Warteschlange
DECREASE-KEY(Q,x,k)	Weist dem Element **x** in der Schlange den neuen Schlüsselwert **k** zu. Dabei wird vorausgesetzt, dass **k** nicht größer als der aktuelle Schlüsselwert von **x** ist
INCREASE-KEY(Q,x,k)	Weist dem Element **x** in der Schlange den neuen Schlüsselwert **k** zu. Dabei wird vorausgesetzt, dass **k** nicht kleiner als der aktuelle Schlüsselwert von **x** ist

Implementierung des Algorithmus von Dijkstra

DIJKSTRA(G, c, s) {
  Input: Gerichteter Graph **G=(V,R)** in Adjazenzlistendarstellung, eine nichtnegative Gewichtsfunktion **c:** $\mathbf{R} \to \Re_+$ und ein Knoten $s \in V$, von dem alle anderen Knoten erreichbar sind
  Output: Für alle $\mathbf{v} \in \mathbf{V}$ die Distanz **d[v] = dist$_c$(s,v)** sowie ein Baum $\mathbf{G}_\pi$ kürzester Wege von **s** aus.
  INIT(G,s)
  PERM := $\varnothing$ // PERM ist Menge der "permanent markierten" Knoten
  Q:=MAKE() // Erzeuge leere Prioritätenwarteschlange
  INSERT(Q,s) // Füge s mit Schlüsselwert d[s]=0 in die Schlange ein
  while (|PERM| < n) {
    u := EXTRACT-MIN(Q)
    PERM := PERM $\cup$ **{u}**
    for all **v** $\in$ **Adj[u] \ PERM {**
      if (d[v] == +∞ **)** // v ist noch nicht in Q enthalten
        INSERT(Q,v)
      Prüfe Kante **(u,v)** mit **TEST(u,v)**, wenn dabei **d[v]** auf **d[u]+c(u,v)** herabgesetzt wird, dann führe **DECREASE-KEY(Q,v,d[u]+c(u,v)** aus
    }
  }
  return **d[]** und $\mathbf{G}_\pi$ // $\mathbf{G}_\pi$ wird durch die Vorgängerknoten π aufgespannt
}

Informeller Algorithmus Dijkstra

Die Grundidee des Algorithmus ist, ab einem Startknoten die kürzest möglichen Wege weiter zu verfolgen und längere Wege beim Updaten auszuschließen. Er besteht aus diesen Schritten:

1. Weise allen Knoten die beiden Eigenschaften "Distanz" und "Vorgänger" zu. Initialisiere die Distanz im Startknoten mit 0 und in allen anderen Knoten mit ∞.
2. Solange es noch unbesuchte Knoten gibt, wähle darunter denjenigen mit minimaler Distanz aus und
    1. Speichere, dass dieser Knoten schon besucht wurde
    2. Berechne für alle noch unbesuchten Nachbarknoten die Summe des jeweiligen Kantengewichtes und der Distanz im aktuellen Knoten
    3. Ist dieser Wert für einen Knoten kleiner als die dort gespeicherte Distanz, aktualisiere sie und setze den aktuellen Knoten als Vorgänger.
    Dieser Schritt wird auch als Update bezeichnet und ist die zentrale Idee von Dijkstra.

In dieser Form berechnet der Algorithmus ausgehend von einem Startknoten die kürzesten Wege zu allen anderen Knoten. Ist man dagegen nur an dem Weg zu einem ganz bestimmten Knoten interessiert, so kann man in Schritt (2) schon abbrechen, wenn der gesuchte Knoten der aktive ist.

Algorithmus von Dijkstra in Pseudocode

```
function Dijkstra(Graph, Quelle) {
    for each Knoten v in Graph {  // Variablen initialisieren
        abstand[v] := inf         // Abstand der Quelle zu v
        vorgänger[v] := null      // Vorheriger Knoten auf dem Weg zu v
    }
    abstand[Quelle] := 0  // Der Abstand der Quelle zu sich selbst ist 0
    Q := Die Menge aller Knoten in Graph
    while Q nicht leer {          // Der eigentliche Algorithmus
        u := Knoten in Q mit kleinstem Wert in abstand[]
        if abstand[u] = ∞:
            break         // alle verbleibenden Knoten sind unerreichbar
        entferne u aus Q
        for each nachbar v von u { // nur die v,die noch in Q enthalten
            alt := abstand[u] + gewicht(u, v)
            // gewicht bestimmt das Kantengewicht zwischen u und v
            if alt < abstand[v] {              // Verwende (u,v,a)
                abstand[v] := alt
                vorgänger[v] := u
            }
        }
    }
    return vorgänger[]
}
```

Falls man nur am kürzesten Weg zwischen zwei Knoten interessiert ist, kann man den Algorithmus abbrechen lassen, falls *u = Zielknoten* ist. Den kürzesten Weg kann man nun durch Iteration über die *vorgänger* ermitteln:

```
1  S := []
2  u := Zielknoten
3  while vorgänger[u] definiert:
4      füge u am Anfang von S ein
5      u := vorgänger[u]
```

Algorithmus Bellman und Ford

Der Algorithmus von Bellman-Ford arbeitet auch mit negativen Kantengewichten. Der Algorithmus arbeitet in n-1 Phasen, in jeder Phase wird jede Kante genau einmal getestet:

BELLMAN-FORD (G, c, s) {
Input: Gerichteter Graph **G=(V,R)** in Adjazenzlistendarstellung, eine (auch negative) Gewichtsfunktion **c: R** $\rightarrow \Re$ und ein Knoten s$\in$V
Output: Für alle **v**$\in$**V** die Distanz **d[v] = dist$_c$(s,v)** sowie ein Baum $\mathbf{G}_\pi$ kürzester Wege von **s** aus.
INIT(G,s)
for (k:=1, ... , n-1) { // **n** ist die Kantenzahl
for all (u, v) $\in$ **R**
TEST(u,v)
}
return **d[]** und $\mathbf{G}_\pi$ // $\mathbf{G}_\pi$ wird durch die Vorgängerknoten π aufgespannt
}

Am Ende von Phase **k = 1, 2, 3, ..., n-1** gilt für alle **v∈V**:
d[v] ≤ min { c(P) : P ist ein Weg von s nach v mit höchstens k Kanten }

Wenn **G** keinen Kreis negativer Länge enthält, der von **s** aus erreichbar ist, dann gilt bei Abbruch des Algorithmus: **d[v] = dist (s, v)** für alle **v∈V**.

Ferner ist $\mathbf{G}_\pi$ ein **Baum kürzester Wege** bzgl. **s**.

Bei **negativen Gewichten** tritt ein grundsätzliches Problem auf: Es kann ein **Kreis negativer Länge** von **s** aus erreichbar sein und damit $\mathbf{dist_c(s,v)}=-\infty$ für alle Knoten **v** gelten, die von **V(C)** aus erreichbar sind.

Am Ende des Verfahrens von Bellman-Ford können noch einmal alle Kanten **(u,v)** durchlaufen werden und die Bedingung **d[v] ≤ d[u] + c(u,v)** überprüft werden. Falls **d[v] > d[u] + c(u,v)** für eine Kante gilt, dann ist ein **Kreis negativer Länge** erreicht worden.

Algorithmus, der nach Abschluß des Algorithmus von Bellman und Ford überprüft, ob G einen Kreis negativer Länge enthält:

TEST-NEGATIVE-CYCLE(G, c, d) {
Input: Gerichteter Graph **G=(V,R)** in Adjazenzlistendarstellung, eine (auch negative) Gewichtsfunktion **c: R→ℜ**, Distanzen **d** aus dem Bellman-Ford-Algorithmus
Output: Information, ob G einen Kreis negativer Länge enthält

```
  for all (u, v) ∈ R {
     if ( d[v] > d[u] + c(u,v) )
          return "ja"
  }
  return "nein"
}
```

Alternative Darstellung des Algorithmus von Bellman-Ford in Pseudocode:

```
algorithm Bellmann-Ford(G, s) { berechne alle kürzesten Wege von s zu
                                allen anderen Knoten in G }

ROT := {s}; GRÜN := {}; dist(s) := 0;
while ROT <> {} do
      ALT_ROT := ROT
      for each v in ALT_ROT do
          färbe v grün;
          for each Nachfolger w von v do
              if dist(w) > dist(v) + c(v, w)
              { w kann kürzer über v erreicht werden }
              then dist(w) := dist(v) + c(v, w);
                   färbe w rot;
              fi
          od
      od
od.
```

Algorithmus von Floyd zur Bestimmung aller kürzesten Wege eines Graphen

Gegeben:

Graph **G(X,U)** mit **Knotenmenge X** mit **kn** Knoten und **Kantenmenge U** mit **ka** Kanten
Gewicht $\mathbf{g_{ij}}$ für jede **Kante** $\mathbf{(x_i,x_j)}$ aus **U**

Gesucht:

kürzeste Wege $\underline{w}(x_q,x_s)$ für alle Knotenpaare (x_q,x_s).

Bezeichnungen:

$\underline{G}$: bewertete Adjazenzmatrix des Graphen, mit Elementen g_{ij} (Gewichte)
$\underline{L}$: Matrix, mit den Längen der kürzesten Wege; das Element l_{ij} enthält die Länge des kürzesten Weges zwischen den Knoten x_i und x_j
$\underline{P}$: Matrix, mit dem Baum des kürzesten Wege, das Element p_{qs} enthält den Vorgänger des Knotens x_s auf dem Weg $\underline{w}(x_q,x_s)$.

Verbale Beschreibung:

Der Algorithmus überprüft, ob sich ein Weg von einem Knoten x_i zu einem Knoten x_j über den Knoten x_k verkürzen läßt. Ist das der Fall, wird der neue Weg über Knoten x_k in $\underline{P}$ eingetragen.

Algorithmusbeschreibung:

1. Setze $\underline{L}=\underline{G}$ und initialisiere $\underline{P}$ wie folgt:

$$p_{ij} = \begin{cases} 0 \text{ für } g_{ij} = \\ i \text{ sonst} \end{cases}$$

für alle i = 1 .. kn und j = 1 .. kn

```
for (k=1 ; k<= kn ; k=k+1) {
    for (i=1 ; i<= kn ; i=i+1) {
            for (j=1 ; j<= kn ; j=j+1) {
            // Überprüfung des neuen Weges über x_k
                    if ( l_ij > l_ik + l_kj ) {
                                        l_ij = l_ik + l_k
                                        p_ij = p_kj
                    }
             }
     }
}
```

Nach Beendigung des Algorithmus befindet sich in der Matrix $\underline{P}$ der Baum aller kürzesten Wege. Dieser ist in einer Rückwärtsverkettung abgespeichert.

Algorithmus von Floyd

Die Idee für den **Floyd-Algorithmus** ist eine Rekursion für die Distanzen. Dabei darf **G** keinen Kreis negativer Länge enthalten.

Die Knoten $\mathbf{V = \{v_1, v_2, ..., v_n\}}$ werden in einer beliebigen Reihenfolge notiert.

Für $\mathbf{v_i, v_j \in V}$ und **k=0, 1, ..., n** wird der Wert $\mathbf{d_k(v_i, v_j)}$ als **Länge eines kürzesten Weges** von **u** nach **v** definiert, der nur die Knoten $\mathbf{\{v_i, v_j, v_1, v_2, ..., v_k\}}$ berührt. Falls kein solcher Weg existiert, dann sei $\mathbf{d_k(v_i, v_j) := +\infty}$. Die Distanz $\mathbf{dist_c(v_i, v_j)}$ entspricht dann $\mathbf{d_n(v_i, v_j)}$.

Für **k=0** darf der Weg nur $\mathbf{v_i}$ und $\mathbf{v_j}$ berühren, d.h. $\mathbf{d_0(v_i, v_j) = c(v_i,v_j)}$, falls $\mathbf{(v_i,v_j) \in R}$, sonst ∞

Sei nun **P** ein **kürzester Weg** von $\mathbf{v_i}$ nach $\mathbf{v_j}$, der nur $\{\mathbf{v_i, v_j, v_1, v_2, ..., v_{k+1}}\}$ berührt. Da **G** nach Annahme keine Kreise negativer Länge enthält, können wir ohne Beschränkung davon ausgehen, daß **P elementar** (keinen Knoten mehr als einmal berührt) ist.

Falls $\mathbf{v_{k+1}}$ nicht von **P** berührt wird, so ist $\mathbf{c(P) = d_k(v_i, v_j)}$.

Falls $\mathbf{v_{k+1} \in s(P)}$, so können wir **P** in Wege $\mathbf{Pv_i,v_{k+1}}$ von $\mathbf{v_i}$ nach $\mathbf{v_{k+1}}$ und $\mathbf{Pv_{k+1},v_j}$ von $\mathbf{v_{k+1}}$ nach $\mathbf{v_j}$ aufspalten.

Dann gilt $\mathbf{c(Pv_i,v_{k+1}) = d_k(v_i,v_{k+1})}$ und $\mathbf{c(Pv_{k+1},v_j) = d_k(v_{k+1},v_j)}$.

Also gilt für $\mathbf{k \geq 1}$: $\mathbf{d_{k+1}(v_i, v_j) = \min \{ d_k(v_i, v_j) , d_k(v_i,v_{k+1}) + d_k(v_{k+1},v_j) \}}$

```
FLOYD (G, c) {
  Input:  Gerichteter Graph G=(V,R) in Adjazenzlistendarstellung, eine
          Gewichtsfunktion c: R → ℜ
  Output: Für alle u, v∈V die Distanz D_n[u,v] = dist_c(u,v)
  for all v_i, v_j ∈ V {  D_0[v_i, v_j ] := +∞ }
  for all (v_i,v_j ) ∈ R {  D_0[v_i, v_j ] := c(v_i,v_j) }
  for k=0, ..., n-1 {
      for i=1, ...,n {
          for j=1,...,n {
              D_{k+1}(v_i, v_j) = min { D_k(v_i, v_j) , D_k(v_i,v_{k+1}) + D_k(v_{k+1},v_j) }
          }
      }
   }
   return D_n[ ]
}
```