4 Kürzeste Wege

Ash Ketchum möchte die kürzeste Strecke von der Universität Trier (Campus 2) zur Pokémon-Arena Porta Nigra bestimmen. Er hat dafür eine Straßenkarte von Trier, auf der die Abstände zwischen allen Paaren benachbarter Kreuzungen eingezeichnet sind.
In diesem Szenario stellen die Kosten einer Kante die Distanz zweier Kreuzungen dar. In anderen Szenarien können die Kosten einer Kante aber auch andere Bedeutungen haben. Sie können z.B. tatsächliche Kosten darstellen, wenn es um Energieverbrauch oder das Herstellen von Produkten geht. Treten in diesen Szenarien negative Kosten auf, können diese Gewinnen entsprechen.

4.1 Allgemeines Problem

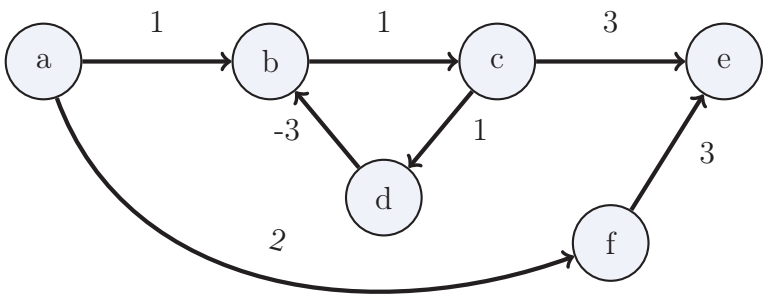
Gegeben:

Gerichteter Graph: $G = (V, E)$

Kostenfunktion: $cost : E \to \mathbb{R}$

$cost(e)$ = Kosten der Kante e
Anstatt $cost((v, w))$ *schreiben wir nur* $cost(v, w)$.

Beispiel 4.1



4.2 Definitionen

4.2.1 Pfad

Ein Pfad kann sowohl über die Knoten, als auch über die Kanten definiert werden, die er enthält.

Definition über Knoten

Ein Pfad von $s \in V$ nach $w \in V$ in einem gerichteten Graphen $G = (V, E)$ ist eine Folge $P = v_0, \ldots, v_k$ mit $v_0 = s$ und $v_k = w$.

Definition über Kanten

Ein Pfad von $s \in V$ nach $w \in V$ in einem gerichteten Graphen $G = (V, E)$ ist eine Folge $P = (e_1, e_2, \ldots, e_n)$ von Kanten $e_i \in E$.

Länge eines Pfades

Die Länge eines Pfades wird oft über die Anzahl der enthaltenen Kanten definiert. Hier allerdings **nicht**!
Trotzdem sprechen wir von kürzesten Pfaden, da das Problem im Englischen „shortest path“ genannt wird und es oft mit „kürzestem Weg“ übersetzt wird. Da oft von billigsten Pfaden gesprochen wird, wenn es um die Kosten eines Pfades geht, werden in diesem Skript die Bezeichnungen „kürzeste Wege“ und „billigste Pfade“ (oder Mischformen daraus) synonym verwendet.

Einfacher Pfad

Ein einfacher Pfad enthält keinen Knoten mehrfach.
Erfüllt ein Pfad diese Eigenschaft nicht nennt man ihn „nicht einfachen Pfad“.

4.2.2 Kosten eines Pfades

Die Kosten eines Pfades ist die Summe der Kosten der Kanten entlang des Pfades.

$$\boldsymbol{cost}(\boldsymbol{P}) := \sum_{\boldsymbol{e} \in \boldsymbol{P}} \boldsymbol{cost}(\boldsymbol{e})$$

Achtung: Eine Kante kann mehrfach vorkommen.

Im Beispiel 4.1:

$$cost(a \to f \to e) = 5$$
$$cost(a \to b \to c \to e) = 5$$
$$cost(a \to b \to c \to d \to b \to c \to e) = 4$$

4.2.3 Distanz zweier Knoten

Die Distanz zweier Knoten entspricht dem kürzesten Weg zwischen diesen beiden Knoten.
Da es durch negative Zyklen unendlich viele Wege geben kann, wird hier nicht das Minimum, sondern das Infimum (s. 1.2.2) genutzt.

$$\boldsymbol{dist(v,w) := inf\{cost(P) \mid P \text{ ist Pfad von v nach w}\}}$$

dh. $dist(v,w) = -\infty$, falls negativer Zyklus auf P existiert.
$dist(v,w) = +\infty$, falls kein Pfad von v nach w existiert.

Im Beispiel 4.1:

$$dist(a,f) = 2$$
$$dist(e,a) = +\infty$$
$$dist(a,e) = -\infty \qquad \text{Pfad: } a \to b \to (c \to d \to b \to c)^i \to e$$
Kosten: $4 - i$ dh. beliebig klein.

4.3 Varianten des Problems

Es existieren verschiedene Varianten des Kürzesten-Wege Problems. Dabei wird unterschieden, wie viele source und target Knoten es gibt.

i) **Single-Source-Single-Target**
$dist(s,t) \quad s,t \in V$
Zwischen zwei gegebenen, festen Knoten wird die Distanz gesucht.

ii) **Single-Source-Shortest-Path**
$dist(s,v) \quad \forall\, v \in V$
Von einem gegebenen Knoten s werden die kürzesten Distanzen zu allen anderen Knoten gesucht.

iii) **All-Pairs-Problem**
$dist(v, w) \quad (v, w) \in VxV$
Gesucht ist die Distanz zwischen allen Knotenpaaren. ($\rightarrow$ Entfernungstabelle)

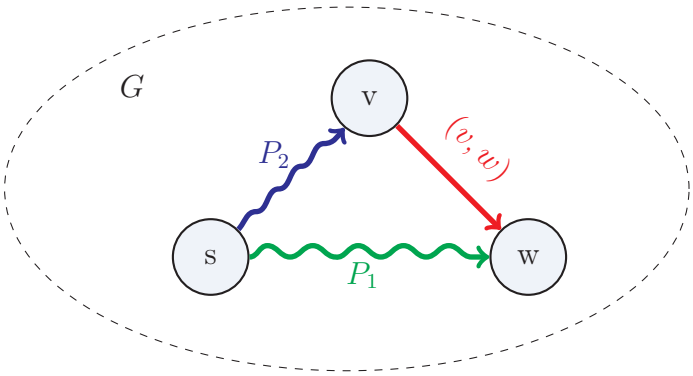
4.4 Single-Source-Shortest-Path

Ab hier wird das Single-Source-Shortest-Path Problem betrachtet.

Eingabe: $G = (V, E), \quad s \in V, \quad cost : E \rightarrow \mathbb{R}$
Ausgabe: $\forall\ v \in V \quad dist(s, v)$ (und entsprechende Pfade)

Beobachtung:
Die *dist*-Funktion erfüllt die Δ-Ungleichung.
Für jede Kante $(v, w) \in E$ gilt:

$$dist(s, w) \leq dist(s, v) + cost(v, w)$$



P_1 ist ein kürzester Pfad von s nach w.
P_2 ist ein kürzester Pfad von s nach v.

Herleitung der Δ-Ungleichung:
$dist(s, w) = cost(P_1)$
$dist(s, v) = cost(P_2)$

$dist(s, w) \leq cost(P_2 + (v, w))$ da P_1 kürzester Pfad von s nach w
$\Leftrightarrow dist(s, w) \leq cost(P_2) + cost(v, w)$
$\Leftrightarrow dist(s, w) \leq dist(s, v) + cost(v, w$

4.4.1 Allgemeiner Algorithmus (Label correcting)

Aus diesen Beobachtungen kann leicht ein Algorithmus abgeleitet werden.

Ideen:

- Überschätze die *dist*-Werte ($DIST = \infty$)
- Solange eine Kante (u, v) die Δ-Ungleichung verletzt: Korrigiere $DIST[v]$ dh. $DIST[v] \leftarrow DIST[u] + cost(u, v)$

Algorithmus 1 : Allgemeiner Algorithmus

```
1 foreach v ∈ V do
2 |   DIST[v] ← ∞
3 end
4 DIST[s] ← 0
5 while ∃ Kante (u, v) ∈ E mit DIST[v] > DIST[u] + cost((u, v)) do
6 |   DIST[v] ← DIST[u] + cost((u, v))
7 end
```

Lemma 1

a) Es gilt immer $DIST[v] \geq dist(s, v)$

b) Wenn $DIST[v] < \infty$, dann existiert ein Pfad von s nach v mit Kosten $DIST[v]$

c) Die kürzesten Pfade bilden einen Baum T mit Wurzel s.

d) Für jede Kante (v, w) auf einem kürzesten Pfad gilt:
$dist(s, w) = dist(s, v) + cost((v, w))$ (Δ-Ungleichung mit Gleichheit)

Erklärungen zu Lemma 1:

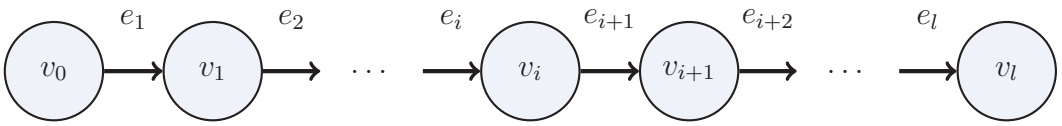
a) $dist(s, v)$ sind die Kosten des kürzesten Pfades von s nach v. $DIST[v]$ ist zu Beginn $+\infty$ und wird im Laufe des Algorithmus, falls mindestens ein Pfad von s nach v existiert, auf die Kosten dieses Pfades verringert. Somit kann

der Wert von $DIST[v]$ auch nie kleiner als die Kosten des kürzesten Pfades ($dist(s, v)$) werden. Ab dem Zeitpunkt, an dem der Algorithmus den kürzesten Pfad findet, gilt: $DIST[v] = dist(s, v)$

b) Sobald der Algorithmus einen Pfad von s nach v findet, berechnet er die Kosten dieses Pfades und setzt dieses Ergebnis als neuen $DIST$-Wert von v. Da der Pfad und die Kosten der einzelnen Kanten endlich sind, ist das Ergebnis und somit $DIST[v]$ echt kleiner als $+\infty$.

c) In jedem Knoten mit $DIST[v] < \infty$ mündet genau ein kürzester Pfad. Bei Korrektur des $DIST$-Wertes wird die eingehende Kante von T definiert.

d) Die Δ-Ungleichung mit Gleichheit ergibt sich aus der Definition von $dist(s, w)$.



$$dist(s, v_i) = \sum_{k=1}^{i} e_k$$

$$dist(s, v_{i+1}) = \sum_{k=1}^{i} e_k + cost(v_i, v_{i+1})$$

$$dist(s, v_{i+1}) = \sum_{k=1}^{i} e_k + e_{i+1}$$

$$dist(s, v_{i+1}) = \sum_{k=1}^{i+1} e_k$$

(v_i, v_{i+1}) verbindet den Knoten v_{i+1} mit dem auf dem kürzesten Weg davor liegenden Knoten.

4.4.2 Verfeinerter Algorithmus

Der allgemeine Algorithmus lässt viel Freiheit bei der Wahl des Knotens. Dadurch kann der Knoten sehr ungeschickt gewählt werden. Liegt der Knoten sehr weit hinten im Graph, müssen die $DIST$-Werte der hinteren Knoten immer wieder angepasst werden, wenn bei einem auf dem Pfad davor liegenden Knoten der $DIST$-Wert verringert wurde. Daher beginnt der folgende Algorithmus beim Startknoten, sodass

die $DIST$-Werte wie eine Welle durch den Graphen vom Startknoten aus verändert werden.

Damit der Algorithmus nicht mehr jedes mal alle Kanten überprüfen muss, wird eine Kandidatenmenge gebildet. In dieser Kandidatenmenge U werden alle Knoten, aus denen Kanten ausgehen, die die Δ-Ungleichung verletzen können, gespeichert. Dadurch muss der Algorithmus immer nur die Kandidatenmenge abarbeiten und arbeitet somit effizienter.
Am Anfang besteht U nur aus dem Startknoten s. $U = \{s\}$
Immer, wenn $DIST[v]$ vermindert wird, wird v in die Menge U aufgenommen, da die Δ-Ungleichung verletzt worden sein könnte.

Algorithmus 2 : Verfeinerter Algorithmus

```
1  foreach v ∈ V do
2  |  DIST[v] ← ∞
3  end
4  DIST[s] ← 0
5  U = {s}
6  while U ≠ ∅ do
7  |  wähle und entferne ein u ∈ U
8  |  foreach v ∈ V mit (u, v) ∈ E do
9  |  |  c = DIST[u] + cost((u, v))
10 |  |  if c < DIST[v] then
11 |  |  |  DIST[v] = c
12 |  |  |  U = U ∪ {v}
13 |  |  end
14 |  end
15 end
```

Lemma 2

Falls G keine negativen Zyklen enthält, dann gilt folgendes:

a) Falls $v \notin U$, dann gilt für alle ausgehenden Kanten (v, w):
$DIST[w] \leq DIST[v] + c((v, w))$ (dh. Δ-Ungleichung erfüllt)

b) Sei $v_o, \ldots, v_k$ ein kürzester Pfad von s nach v (dh. $v_0 = S,\ v_k = v$).
Falls nun $DIST[v] > dist(s, v)$, dann existiert ein i $(0 \leq i \leq k-1)$ mit $DIST[v_i] = dist(s, v_i)$ und $v_i \in U$ z.B. $s = 0$

c) Es existiert immer ein $u \in U$ mit $DIST[u] = dist(s, u)$

d) Wenn in Zeile 7 des Algorithmus 2 stets ein $u \in U$ mit $DIST[u] = dist(s, u)$ gewählt wird ("perfekte Wahl"), dann wird die while-Schleife für jeden Knoten höchstes einmal ausgeführt.

Beweis

a) Induktion über die Schleifendurchläufe (while):

$\underline{i = 0:}$ Vor dem ersten Lauf $DIST[s] = 0$, $DIST[v] = \infty$ für $v \neq s$ und $U = \{s\}$

$\underline{i \to i + 1:}$ Betrachte ein beliebiges $v \notin U$ nach der $(i + 1)$ten Ausführung.

Fall 1: $v \notin U$ vor $(i + 1)$ten Ausführung. Nach I.A. galt:
$DIST[w] \leq DIST[v] + c((v, w))$ für alle ausgehende Kanten (v, w).
$DIST[v]$ wurde im $(i+1)$ten Lauf nicht verändert(da $v \notin U$ danach) und die $DIST$-Werte aller Nachbarn w von v wurden eventuell vermindert
$\Rightarrow DIST[w] \leq DIST[v] + c((v, w))$ für alle ausgehenden Kanten (nach der $(i + 1)$ten Ausführung)

Fall 2: $v \in U$ vor $(i + 1)$Ausführung
$\Rightarrow v$ wurde in Zeile 7 ausgewählt (dh. $u = v$)
$\Rightarrow$ Die innere Schleife stellt die Δ-Ungleichung für alle ausgehenden Kanten her.

b) Sei $s = v_0, \ldots, v_k = v$ ein kürzester Pfad von s nach v mit
$DIST[v_k] > dist(s, v_k)$.
Sei i maximal mit $DIST[v_i] = dist(s, v_i) \Rightarrow 0 \leq i < k$
Z.z. $v_i \in U$
Widerspruchsbeweis:
Annahme: $v_i \notin U$

Teil a) $\Rightarrow DIST[v_{i+1}] \leq DIST[v_i] + c((v_i, v_{i+1}))$
Außerdem gilt: $dist(s, v_{i+1}) = dist(s, v_i) + c((v_i, v_{i+1}))$ (auf kürzesten Pfaden ist die Δ-Ungleichung mit Gleichheit erfüllt)
Dann folgt:

$$\begin{aligned} dist(s, v_{i+1}) &= dist(s, v_i) + c((v_i, v_{i+1})) \\ &= DIST[v_i] + c((v_i, v_{i+1})) \\ &\geq DIST[v_{i+1}] (\text{da } v_i \notin U) \end{aligned}$$

Zusammen: $dist(s, v_{i+1}) \geq DIST[v_{i+1}]$
$\Rightarrow dist(s, v_{i+1}) = DIST[v_{i+1}]$ $\lightning$ Widerspruch zur maximalen Wahl von i da $DIST$-Werte nie zu klein werden.

c) Sei $v \in U$ beliebig.

Fall 1: $DIST[v] = dist(s, v) \Rightarrow$ fertig.

Fall 2: $DIST[v] > dist(s, v)$
Betrachte einen kürzesten Pfad von s nach v. Dann existiert auf diesem Pfad (nach Teil b)) ein $v_i \in U$ mit $DIST[v_i] = dist(s, v_i)$.

d) Falls in Zeile 7 des Algorithmus 2 immer eine perfekte Wahl getroffen wird, dann kann u kein zweites Mal in U eingefügt werden.

Perfekte Wahl
$\rightarrow$ Gesamtaufwand: $\mathcal{O}\Big(\underbrace{\sum_{v \in V} (1 + outdeg(v))}_{n+m} + \underbrace{\text{Verwaltung der Menge } \mathcal{U}}_{?} \Big)$

$n = \#$Knoten
$m = \#$Kanten

□

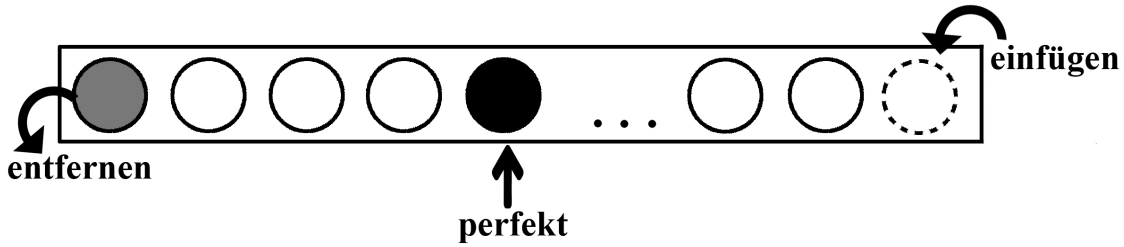
4.5 Perfekte Wahl

Frage: Wie findet man effizient ein perfektes u, um es in Zeile 7 des Algorithmus 2 zu wählen?

4.5.1 Allgemeiner Fall

Bei einem beliebigen Graph sind Zyklen möglich. Da auch die Kostenfunktion beliebig ist und somit negative Kosten auftreten können, können negative Zyklen entstehen.

Idee: Realisiere U als Schlage (FIFO)



Beinhaltet der Graph keine negativen Zyklen, so existiert nach Lemma 2 mindestens ein perfekter Knoten ($DIST[v] = dist(s, v)$) in der Schlange.
$\Rightarrow$ Zwischen zwei Entnahmen des selben Knoten u wurde mindestens ein perfekter Knoten entfernt. Dieser kann nicht wieder in die Schlage eingefügt werden.
$\Rightarrow$ Jeder Knoten wird maximal n-mal entfernt. Spätestens danach gilt $U = \emptyset$.

Beobachtung:
Wird ein Knoten öfter ($> n$) entfernt, dann muss nach Lemma 2 ein negativer Zyklus existieren.
$\rightarrow$ Algorithmus von Bellman/Ford

Algorithmus von Bellman/Ford

```
1  bool BELLMANFORD(const graph& G, node s, const
      edge_array<int>& cost, node_array<int>& DIST,
      node_array<edge>& PRED){
2     queue<node> Q;
3     node_array<bool> inQ(G,false);
4     node_array<int> count(G,0);
5     node v;
6     forall_nodes(v,G){
7           DIST[v] = MAXINT;
8           PRED[v] = NULL;
9     }
10    DIST[s] = 0;
11    Q.append(s);
12    inQ[s] = true;
13    while(!Q.empty()){  // Solange Q nicht leer ist
14          node u = Q.pop()
15          inQ[u] = false;
16          if(++count[u] > G.numbers_of_nodes())
17                return false;
18          edge e;
19          forall_out_edges(e,u){
20                node v = G.target(e);
21                int d = DIST[u] + cost(e);
22                if(d < DIST[v]){
23                      DIST[v] = d;
24                      PRED[v] = e;
25                      if(!inQ[v]){
26                            Q.append(v);
27                            inQ[v] = true;
28                      }
29                }
30          }
31    }
32    return true;
33 }
```

Listing 4.1: Bellman/Ford

PRED-Verweise:
Die $PRED$-Verweise werden dazu genutzt, später den kürzesten Weg nachverfolgen zu können. $pred[v]$ gibt dabei an, über welche Kante v auf dem kürzesten Weg erreicht wird. $PRED$-Verweise ändern sich daher, wenn der kürzeste Weg sich ändert.

Erklärung:
Zuerst wird s in die Menge Q aufgenommen, da sich dessen $dist$-Wert nicht mehr ändert, außer s liegt auf einem negativen Zyklus. Nun beginnt der Algorithmus richtig und läuft so lange, bis entweder Q leer ist und damit zu jedem Knoten ein kürzester Weg gefunden wurde oder der Algorithmus bricht ab, da er einen negativen Zyklus erkannt hat.
Innerhalb der $while$-Schleife wird ein beliebiger Knoten u aus Q gewählt und entfernt. Für alle ausgehenden Kanten von u wird die Δ-Ungleichung überprüft und im Fall, dass sie verletzt wird, wird der $DIST$-Wert des folgenden Knoten korrigiert, dessen $PRED$-Verweis auf u gesetzt und der Knoten in die Menge Q aufgenommen, da dessen ausgehenden Kanten die Δ-Ungleichung verletzen könnten.

Laufzeitanalyse:

$$n \cdot \Big(\underbrace{\text{Iterationen über alle Knoten} + \text{ausgehende Kanten}}_{\mathcal{O}\Big(\underbrace{\sum_{v \in V}(1 + outdeg(v))}_{n+m}\Big)} \Big)$$

Gesamtlaufzeit: $\mathcal{O}(n \cdot (n + m)) = \mathcal{O}(n^2 + n \cdot m)$
Wenn der Graph zusammenhängend ist, ist $m \geq n - 1$
$\Rightarrow$ Laufzeit: $\mathcal{O}(nm)$
dh. $\mathcal{O}(n^2)$ für dünne Graphen (Graphen mit wenig Kanten)
$\mathcal{O}(n^3)$ für dichte Graphen (Graphen mit vielen Kanten)

Ist der Graph nicht zusammenhängend, so kann die Zusammenhangskomponente, in der sich s befindet, in linearer Zeit gefunden werden.

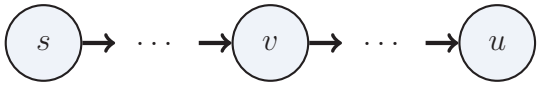
4.5.2 Azyklische Graphen

In azyklischen Graphen treten keine Zyklen auf. Daher ist es auch, trotzt beliebiger Kostenfunktion und dadurch möglicher negativer Kosten, nicht möglich, dass negative Zyklen auftreten. Dadurch existiert eine topologische Sortierung der Knoten.

Lemma 3
Der Knoten $u \in U$ mit kleinster topologischer Nummer, ist eine perfekte Wahl dh. $DIST[u] = dist(s, u)$

Beweis (indirekt)
Wähle $u \in U$ mit $topnum[u]$ minimal.
Annahme: $DIST[u] > dist(s, u)$
$\xRightarrow{\text{Lemma 2 b)}}$ $\exists$ Knoten v auf dem kürzesten Pfad von s nach u mit:

1) $DIST[v] = dist(s, v)$
2) $v \in U$



$\Rightarrow topnum[v] < topnum[u]$ ↯
Widerspruch zur kleinsten Wahl von u (kleinste $topnum$ in U)
$\Rightarrow DIST[u] = dist(s, u)$ □

Mögliche Implementierungen:

1) Topsort → Liste aller Knoten in U mit aufsteigenden $topnum$'s
   Durchlaufe diese Knoten
2) Kombinierter Algorithmus aus Topsort und kürzeste Wege

Algorithmus Topsort und kürzeste Wege

```
1  void Topsort(const graph& G, node s, const edge_array<
     int> cost, node_array<int>& Dist){
2      node_array<int> INDEG(G,0);
3      forall_nodes(v,G){
4           INDEG[v] = indeg(v);
5           if(INDEG[v] == 0)
6                ZERO.append(v);
7           DIST[v] = MAXINT;
8      }
9      DIST[s] = 0;
10     while(!ZERO.empty()){
11          u = ZERO.pop();
12          edge e;
13          forall_out_edges(e,u){
14               v = G.target(e);
15               if(--INDEG[v] == 0)
16                    ZERO.append(v);
17               int d = DIST[u] + cost(u,v);
18               if(d < DIST[v])
19                    DIST[v] = d;
20          }
21     }
22 }
```

Listing 4.2: Topsort und kürzeste Wege

Erklärung:
Die Grundstruktur entspricht dem normalen Topsort-Algorithmus (ZERO-Liste). In die innere Schleife wurde die Überprüfung der Δ-Ungleichung eingebaut. Da dieser zusätzliche Aufwand in konstanter Zeit ausführbar ist, ändert sich die Laufzeit von Topsort nicht.
Der Knoten s muss hierbei nicht gesondert behandelt werden, da der Algorithmus von alleine erst richtig startet, wenn s betrachtet wird, da die *dist*-Werte aller anderen Knoten auf $MAXINT$ gesetzt wurden und somit die Δ-Ungleichung nie verletzt wird.

Laufzeit:
$\mathcal{O}(n+m)$ (perfekte Wahl!)

4.5.3 Nicht-negative Netzwerke

In nicht-negativen Netzwerken können Zyklen vorkommen, aber aufgrund der fehlenden negativen Kosten, können keine negativen Zyklen entstehen.

Idee:
Wähle in Zeile 7 des Algorithmus 2 einen Knoten $u \in U$ mit $DIST[u]$ minimal. Dafür eignet sich eine Priority Queue und deren Funktion $PQ.delmin()$.
$\rightarrow$ Dijkstra

Lemma 4
Der Knoten $u \in U$ mit $DIST[u]$ minimal, dh. $DIST[u] = dist(s,u)$, ist eine perfekte Wahl.

Beweis (indirekt)
Annahme: $DIST[u] > dist(s,u) \xRightarrow{\text{Lemma 2 b)}} \exists\ v \in U$ mit $DIST[v] = dist(s,v)$ auf einem kürzesten Pfad von s nach u $(v \neq u)$.

Dann gilt:

$$\begin{aligned} dist[s,u] &\geq dist(s,v) && \text{da alle Kosten nicht-negativ sind} \\ &= DIST[v] && \text{nach Lemma 2 b)} \\ &\geq DIST[u] && \text{da u minimal gewählt wurde} \end{aligned}$$

$\Rightarrow dist(s,u) = DIST[u]$ da $DIST$-Werte nie zu klein werden. $\lightning$ Widerspruch

Daher gilt: $DIST[u] = dist(s,u)$ $\square$

Implementierung
Datenstruktur: Priority Queue

LEDA-Datentyp:

i) allgemein: $p\_queue < I, P >$
I = Information, P = Priorität

hier: $I = node$
$P = int$ $DIST$-Werte

ii) Für Graphen: $node\_pq < P > PQ$ (Knoten-Priority-Queue)
PQ = Menge von Knoten mit dazugehörigen Prioritäten

Operationen:

- Konstruktor: $node\_pq < P > PQ(graph\ G)$
  $\rightarrow$ leere PQ für Knoten von G
- $\underline{void}$ $PQ.insert(node\ v,\ P\ p)$ fügt einen Knoten v mit der Priorität p zu PQ hinzu.
- $\underline{node}$ $PQ.delmin()$ entfernt einen Knoten v mit kleinster Priorität und gibt v zurück.
- $\underline{node}$ $PQ.find\_min()$ gibt einen Knoten v mit kleinster Priorität zurück.
- $\underline{void}$ $PQ.decrease\_p(node\ v,\ P\ q)$ vermindert die Priorität von v auf q.
- $\underline{bool}$ $PQ.empty()$ testet, ob PQ leer ist.

Algorithmus von Dijkstra

```
1  void Dijkstra(const graph& G, node s, const edge_array<
       int>& cost, node_array<int>& DIST, node_array<edge>&
        PRED){
2        node_pq<int> PQ(G);
3        node v;
4        forall_nodes(v,G){
5              DIST[v] = MAXINT;
6              PRED[v] = NULL;
7        }
8        DIST[s] = 0;
9        PQ.insert(s,0);
10       while(!PQ.empty()){
11             node u = PQ.del_min(); // perfekte Wahl
12             egde e;
13             forall_out_edges(e,u){
14                   node v = G.target(e);
15                   int d = DIST[u] + cost[e];
16                   if(d < DIST[v]){
17                         if(DIST[v] == MAXINT)
18                               PQ.insert(v,d);
19                         else // v ist in U
20                               PQ.decrease_p(v,d)
21                         DIST[v] = d;
22                         PRED[v] = e;
23                   }
24             }
25       }
26 }
```

Listing 4.3: Dijkstra

Erklärung:
Zuerst wird s in die Priority Queue PQ aufgenommen. Die *while*-Schleife läuft nun so lange, bis PQ leer ist. Ist dies der Fall, ist zu jedem Knoten ein kürzester Weg bekannt. In der *while*-Schleife wird zuerst aus der PQ ein Knoten u gewählt. Dabei wird darauf geachtet, dass es eine perfekte Wahl ist. Dann werden alle von u ausgehenden Kanten betrachtet, die jeweilige Δ-Ungleichung überprüft und falls sie verletzt wurde, der erreichte Knoten entweder in PQ aufgenommen, falls er noch nicht in PQ drin ist oder sonst seine Priorität auf seinen neuen DIST-Wert gesetzt.

Danach wird noch sein $DIST$-Wert angepasst und sein $PRED$-Verweis auf die Kante gesetzt, über die er erreicht wurde.

Laufzeitanalyse:

i) Operationen auf dem Graphen: $\mathcal{O}(n+m)$

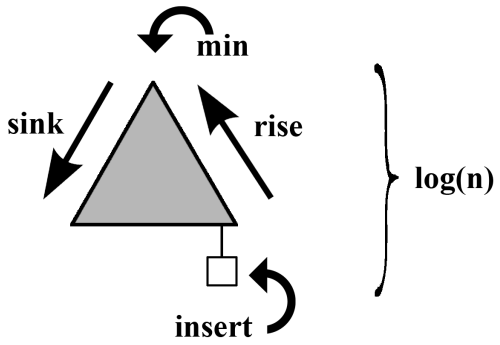
ii) Priority Queue:
$1 \cdot$ Konstruktor
$n \cdot insert()$
$n \cdot delmin()$
$n \cdot empty()$
$m \cdot decrease()$

$\Rightarrow$ Gesamtlaufzeit: $\mathcal{O}\big(n \cdot (T_{insert}(n) + T_{delmin}(n) + T_{empty}(n)) + m \cdot T_{decrease}(n)\big)$
$T_{op}(n) =$ Laufzeit der Operation op auf PQ der Größe n

Varianten von Datenstrukturen:

- binärer Min-Heap
- balancierter Baum

  $\Rightarrow$ Dijkstra $\mathcal{O}((n+m) \cdot log(n))$



Spezielle Heap-Datenstruktur:
Ein Fibonacci-Heap ist eine Menge von binomischen Bäumen. Daher ist der maximale Rang eines Fibonacci-Heaps kleiner gleich $log(n)$, wobei n die Anzahl der Knoten ist. Zudem besitzt der Fibonacci-Heap einen Pointer auf eine Wurzel mit minimaler Priorität. Hier entspricht die Priorität den DIST-Werten. Dadurch ergibt sich der folgende Aufwand:

Amortisierte Analyse:

$$\left.\begin{array}{l}\text{Insert } \mathcal{O}(1)\\ \text{Delmin } \mathcal{O}(log(n))\\ \text{Decrease } \mathcal{O}(1)\end{array}\right\} \rightarrow \mathcal{O}(n \cdot log(n) + m)$$