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: $dist(s, v_{i+1}) \geq DIST[v_{i+1}]$
$\Rightarrow dist(s, v_{i+1}) = DIST[v_{i+1}]$ $\quad \lightning$ 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

□

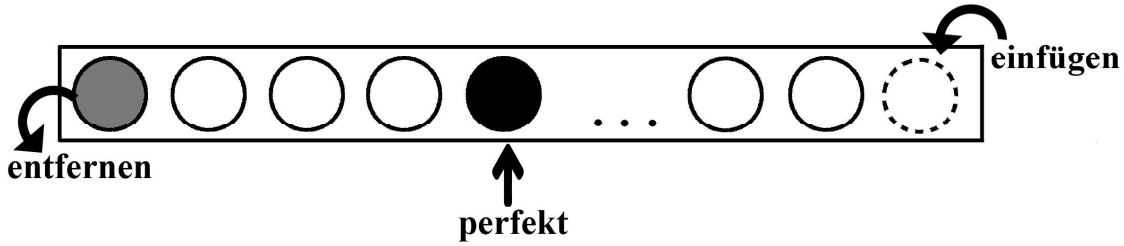
4.5 Perfekte Wahl

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:

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

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.

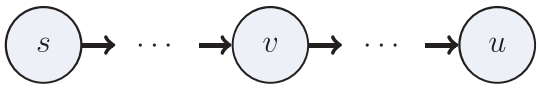
4.5.2 Azyklische Graphen

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!)