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: $\mathcal{O}(n+m)$

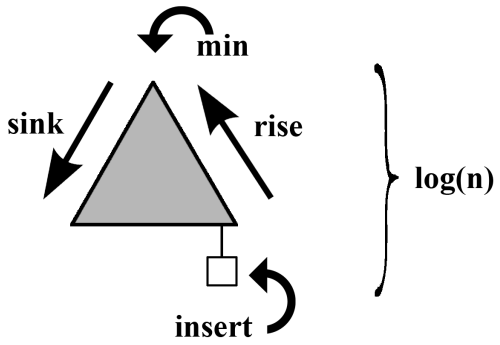
ii) Priority Queue:
$1 \cdot$ Konstruktor
$n \cdot insert()$
$n \cdot delmin()$
$n \cdot empty()$
$m \cdot decrease()$

$\Rightarrow$ 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:

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