

Algorithmen und Datenstrukturen

Sommersemester 2021

Stefan Näher

Universität Trier

naeher@uni-trier.de

Vorlesung 20

24. Juni 2021

Spezialfälle (ohne negative Zyklen)

1. Azyklische Netzwerke

Es existieren überhaupt keine Kreise.

—→ **Topologisches Sortieren**

Zeile 5: wähle $u \in U$ mit $\text{topnum}[u]$ minimal

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

2. Nicht-negative Netzwerke

$\text{cost}(e) \geq 0$ für alle $e \in E$.

—→ **Algorithmus von Dijkstra**

Zeile 5: wähle $u \in U$ mit $\text{DIST}[u]$ minimal

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

Azyklische Netzwerke

In azyklischen Graphen treten keine Zyklen auf. Daher ist es auch, trotz 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)

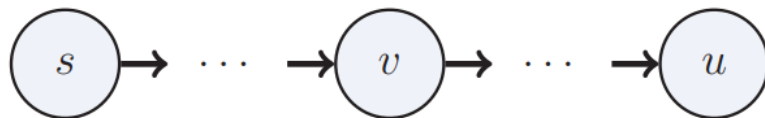
Wähle $u \in U$ mit $topnum[u]$ minimal.

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) $v \in U$



$\Rightarrow topnum[v] < topnum[u] \quad \nexists$

Widerspruch zur kleinsten Wahl von u (kleinste $topnum$ in U)

$\Rightarrow DIST[u] = dist(s, u)$

□

Mögliche Implementierungen:

- 1) Topsort \rightarrow Liste aller Knoten in U mit aufsteigenden *topnum*'s
Durchlaufe diese Knoten
- 2) Kombiniertes Algorithmus aus Topsort und kürzeste Wege

Ein Algorithmus für azyklische Netzwerke (Erweitertes TOPSORT)

```
1. forall  $v \in V$  do INDEG[ $v$ ]  $\leftarrow 0$ ; od
2. forall  $(v, w) \in E$  do INDEG[ $w$ ] ++; od
3. forall  $v \in V$  do
4.   if INDEG[ $v$ ] = 0 then ZERO.push( $v$ ); fi
5.   DIST[ $v$ ]  $\leftarrow \infty$ ;
6. od
7. DIST[ $s$ ]  $\leftarrow 0$ ;
8. while  $\neg$ ZERO.empty() do
9.    $u \leftarrow$  ZERO.pop();
10.  forall  $v \in V$  mit  $(u, v) \in E$  do
11.    if -- INDEG[ $v$ ] == 0 then ZERO.push( $v$ ); fi ;
12.     $d \leftarrow$  DIST[ $u$ ] + cost( $u, v$ );
13.    if  $d <$  DIST[ $v$ ] then
14.      DIST[ $v$ ]  $\leftarrow d$ ;
15.    fi
16.  od
17. od
```

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!)

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 $PQ.delmin()$.

→ 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:

$$\begin{array}{ll} dist[s, u] \geq dist(s, v) & \text{da alle Kosten nicht-negativ sind} \\ = DIST[v] & \text{nach Lemma 2 b)} \\ \geq DIST[u] & \text{da } u \text{ minimal gew\u00e4hlt wurde} \end{array}$$

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

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

□

Priority Queues

speichern allgemein Paare von **Werten** und **Prioritäten**.

Bei der Anwendung auf kürzeste Wege sind die Werte die **Knoten** und die Prioritäten ihre **DIST-Labels**.

Operationen auf einer PQ

`PQ.insert(v, d)`: fügt Knoten v mit Priorität d hinzu

`PQ.delmin()`: entfernt und liefert einen Knoten mit minimaler Priorität

`PQ.decrease(v, d)`: vermindert die Priorität von v auf d

`PQ.empty(v, d)`: Test auf leer

Dijkstra's Algorithmus für nicht-negative Netzwerke

```
1. forall  $v \in V$  do  $\text{DIST}[v] \leftarrow \infty$ ; od
2.  $\text{DIST}[s] \leftarrow 0$ ;
3.  $\text{PQ.insert}(s, 0)$ ;
4. while  $\neg \text{PQ.empty}()$  do
5.    $u \leftarrow \text{PQ.delmin}()$ ;
6.   forall  $v \in V$  mit  $(u, v) \in E$  do
7.      $d \leftarrow \text{DIST}[u] + \text{cost}(u, v)$ ;
8.     if  $d < \text{DIST}[v]$  then
9.       if  $\text{DIST}[v] = \infty$ 
10.        then  $\text{PQ.insert}(v, d)$ ;
11.        else  $\text{PQ.decrease}(v, d)$ ;
12.      fi
13.       $\text{DIST}[v] \leftarrow d$ ;
14.   fi
15. od
16. od
```

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.

Laufzeitanalyse:

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

ii) Priority Queue:

1 · Konstruktor

$n \cdot \text{insert}()$

$n \cdot \text{delmin}()$

$n \cdot \text{empty}()$

$m \cdot \text{decrease}()$

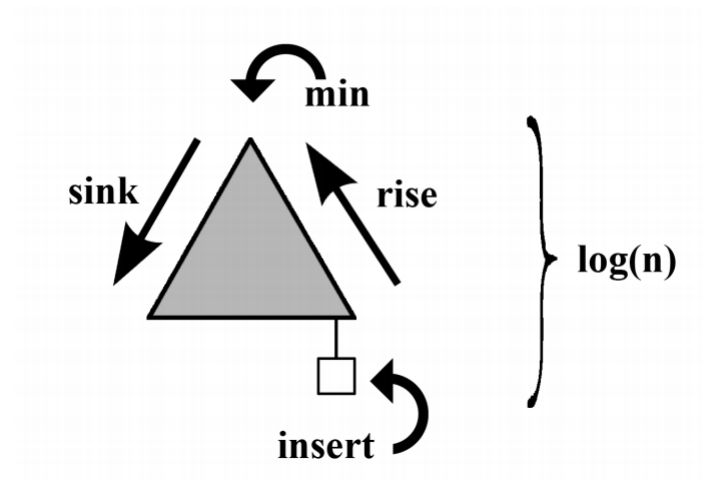
⇒ Gesamtlaufzeit: $\mathcal{O}(n \cdot (T_{\text{insert}}(n) + T_{\text{delmin}}(n) + T_{\text{empty}}(n)) + m \cdot T_{\text{decrease}}(n))$

$T_{\text{op}}(n)$ = Laufzeit der Operation op auf PQ der Größe n

Varianten von Datenstrukturen:

- binärer Min-Heap
- balancierter Baum

⇒ 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:

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