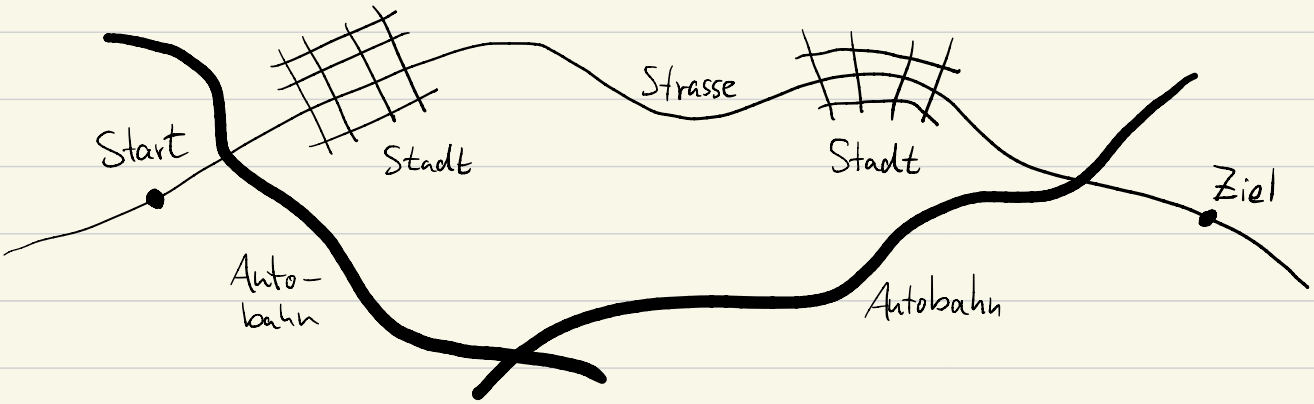
Strassennetzwerk



gesucht: Route mit möglichst wenig Kreuzungen (auch Auf-/Abfahrten)

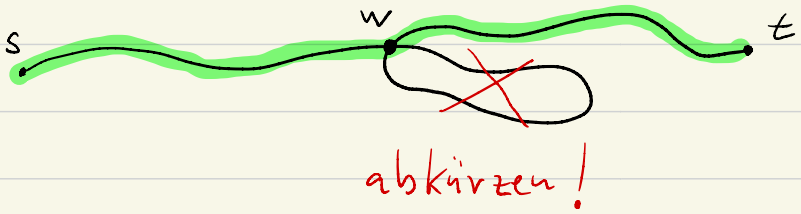
als Graph: Knoten ≃ Kreuzungen

Kanten ≃ Stassenabschnitte ohne Kreuzung

gesucht: Weg im Graph mit wenig Kanten

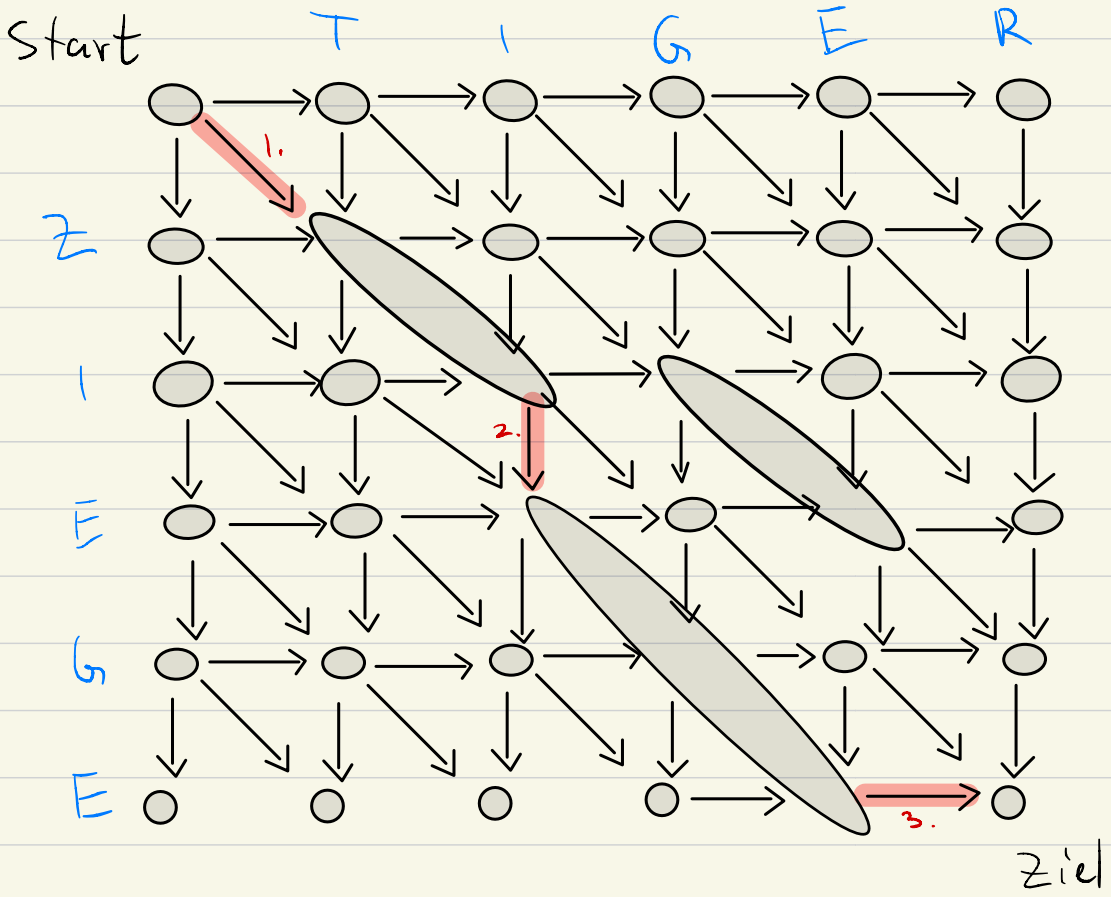
gerichtete Kanten möglich (z.B. Einbahnstrasse)

Beobachtung: Kürzester Weg ist immer Pfad



weiteres Beispiel:

minimale Editierdistanz



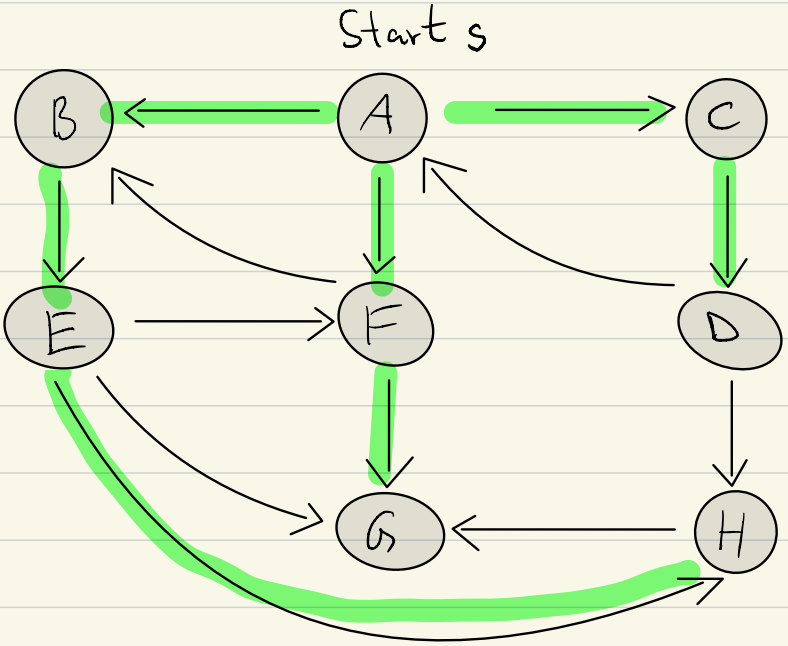
Kanten: mögliche Editieroperationen

kürzester Weg: minimale Sequenz von Editierop.

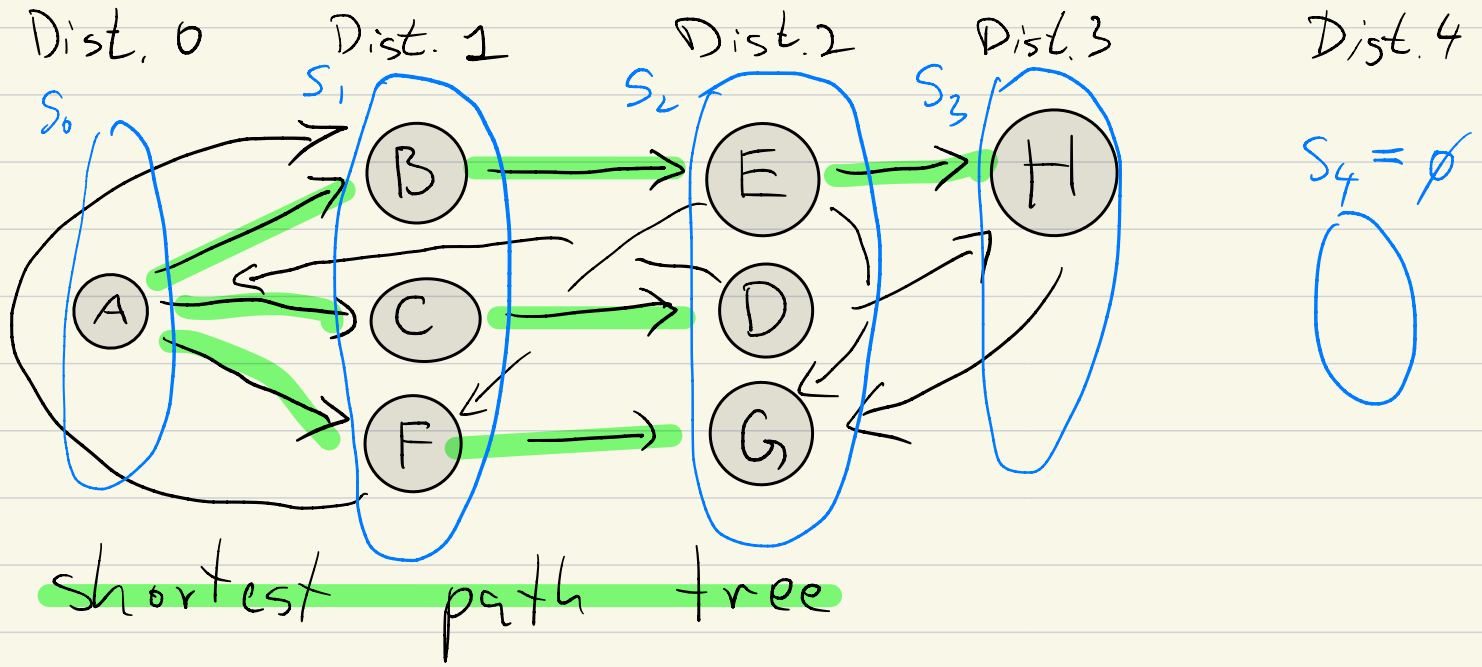
T I G E R
Z I G E R — 1. Tausche T und Z
Z I E G E R — 2. Füge E hinzu
Z I E G E — 3. Entferne R

Definition: ger. Graph $G = (V, E)$, $n = |V|$

Distanz: $dist(u,v)$ = kürzeste Länge eines Wegs von u nach v in G



gesucht: $dist(s,v)$ für alle $v \in V$



level k: $S_k := \{ v \in V \mid dist(s,v) = k \}$

Kante von S_k nach $S_{k'}$ $\Rightarrow$ $k' \leq k+1$

ungerichteter Graph: $k-1 \leq k' \leq k+1$

angenommen: $S_0, \dots, S_{k-1}$ berechnet

wie S_k berechnen?

$v \in S_k \iff v \notin S_0 \cup \dots \cup S_{k-1}$

und v Nachfolger eines Knoten in S_{k-1}

Algorithmus:

$S_0 \leftarrow \{s\}, S_1 \leftarrow \emptyset, \dots, S_{n-1} \leftarrow \emptyset$

n-1 ist grösstmögliche (endliche) Distanz von s

For $k = 1 .. n-1$:

berechne S_k:

For $u \in S_{k-1}$: (2)

For $(u,v) \in E$, $v \notin S_0 \cup \dots \cup S_{k-1}$: (1)

$S_k \leftarrow S_k \cup \{v\}$ (3)

Schnelle Laufzeit

(1) markiere Knoten sobald Distanz bekannt

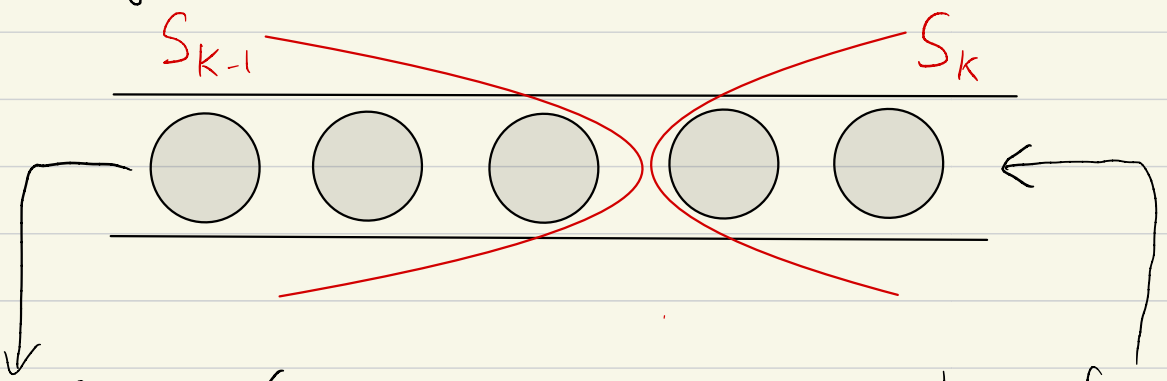
- gemeinsame Datenstruktur um

(2) S_{k-1} abzuarbeiten

(3) S_k aufzubauen

Schlange Q (queue)

im Speicher z.B. als Array mit Zähler



entfernen / dequeue

hinzufügen / enqueue

FIFO: first-in first-out

LIFO: last-in first-out für Stapel

breath-first search / Breitensuche

BFS(s): [Zuse '45, Moore 50er]

$Q \leftarrow \{s\}$ • $dist[s] \leftarrow 0$

$enter[s] \leftarrow 0$; $T \leftarrow 1$

WHILE $Q \neq \emptyset$

$u \leftarrow dequeue(Q)$

$leave[u] \leftarrow T$; $T \leftarrow T+1$

FOR $(u,v) \in E$, $enter[v]$ nicht zugewiesen

$enqueue(Q,v)$ • $dist[v] \leftarrow dist[u]+1$

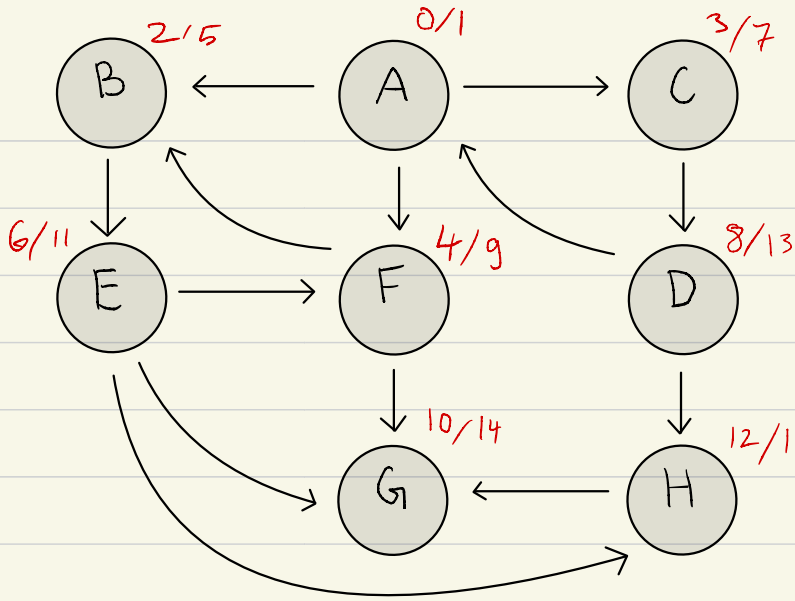
$enter[v] \leftarrow T$; $T \leftarrow T+1$

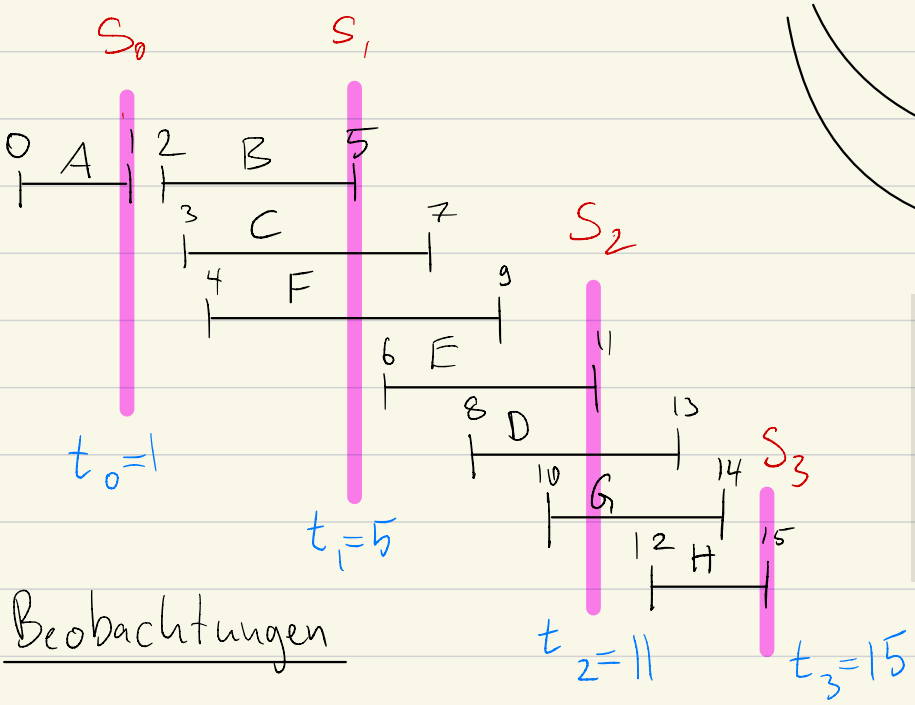
- merke wann Knoten Schlange betreten und verlassen
- Knoten darf Schlange nicht mehrmals betreten

Bemerkung: DFS kann ähnlich iterativ implementiert werden mit Stapel statt Schlange

Beispiel

BFS(A) enter / leave





Intervalle: (vgl. Tiefensuche)

$$I_v := \{ enter[v], \ldots, leave[v] \}$$

Beobachtungen

$enter[v] < leave[v]$

$enter\text{-}order = leave\text{-}order$ (wegen LIFO Eigenschaft von S)

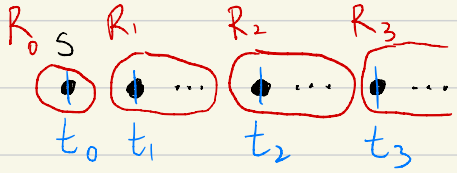
$t_k = \min \{ leave[v] \mid dist(s,v) \geq k \}$ erstes Mal dass ein Knoten mit dist $\geq k$ dequeued wird

$t_0 = 1 \leq t_1 \leq \cdots \leq t_{n-1} \leq t_n = \infty$ ⇝ definieren Epochen für Verlauf von BFS(s)

$R_k := \{ v \mid t_k \leq leave[v] < t_{k+1} \}$ Knoten, die in Epoche k Schlange verlassen

Bildlich:

leave-order



Behauptung $\forall k \in \mathbb{N}_0.$ $\underbrace{S_k = R_k = \{v \mid t_k \in I_v\}}_{\text{Aussage } A(k)}$ (S_k: Inhalt von Q direkt vor Zeit t_k)

Beweis: per Induktion

$A(0)$: (Ind.anfang)

$S_0 = \{s\}$, $leave[s] = 1$, $t_0 = 1$, $\forall v \neq s.\ leave[v] > 1$

$\rightsquigarrow t_1 > 1$, $R_0 = \{s\}$, $\{v \mid t_0 \in I_v\} = \{s\}$ ✓

$A(0), \dots, A(k) \Rightarrow A(k+1)$: (Ind. Schritt)

Q verlassen vor Zeit t_k: $S_0 \cup \dots \cup S_{k-1}$ $(leave[v] < t_k)$

in Q zur Zeit t_k: S_k $(enter[v] \leq t_k \leq leave[v])$

} Ind. hyp.

Zeit t_k bis t_{k+1}:

Q verlassen: S_k $(t_k \leq leave[v] < t_{k+1})$

Q betreten: alle Nachfolger von S_k nicht in $S_0 \cup \dots \cup S_k$

$\rightsquigarrow$ genau S_{k+1} (siehe Charakterisierung auf S. 8)

in Q zur Zeit t_{k+1}: $S_k \setminus S_k \cup S_{k+1} = S_{k+1}$ ✓

Q verlassen von Zeit t_{k+1} bis t_{k+2}: S_{k+1} ✓

} $A(k+1)$

Laufzeitanalyse

$T_u :=$ Laufzeit der WHILE Iteration für Knoten u

$$T_u \leq O(1 + \deg_{out}(u))$$

$$\sum_{u \in V} T_u \leq O\left(\underbrace{\sum_{u \in V} (1 + \deg_{out}(u))}_{} \right)$$

$$= |V| + \underbrace{\sum_{u \in V} \deg_{out}(u)}_{}$$

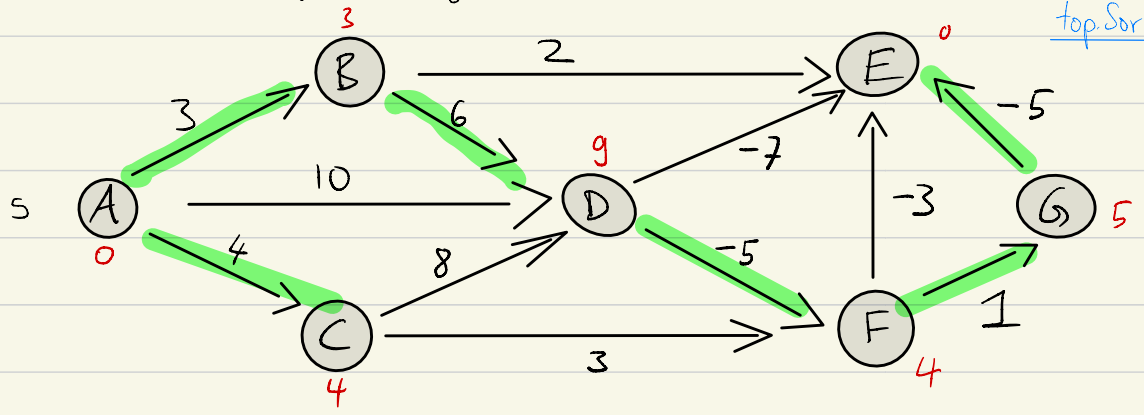
$$= |E| \quad \text{(Handschlaglemma)}$$

$$\rightsquigarrow \sum_{u \in V} T_u \leq O(|V| + |E|)$$

$$\rightsquigarrow \text{BFS}(s) \text{ Laufzeit } O(1) + \sum_{u \in V} T_u \leq O(|V| + |E|)$$

günstigste Wege in gewichteten Graphen

top. Sort.: A B C D F G E



gerichteter Graph $G=(V,E)$, Startknoten $s \in V$, $n=|V|$, $m=|E|$

Kantenkosten: $c(e) \in \mathbb{R}$, $e \in E$

negative Kosten erlaubt (algorithmisch aber schwerer)

Beispiel: E-Autos gewinnen Energie zurück wenn Strecke bergab geht

gesucht: günstigste Wege von s (shortest path)

Wegkosten: Summe der Kantenkosten

Weg $W = (v_0, v_1, \dots, v_\ell)$

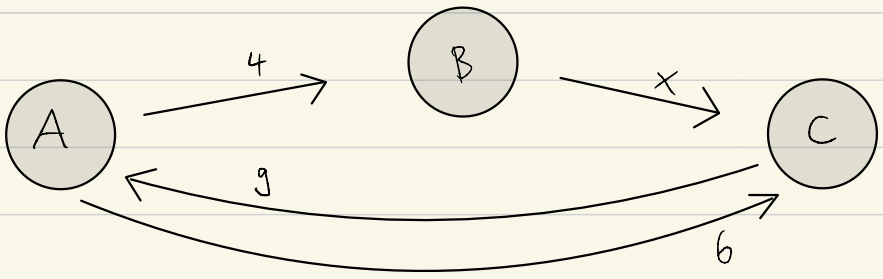
$$c(W) := c(v_0,v_1) + \dots + c(v_{\ell-1}, v_\ell)$$

Notationen: $u \xrightarrow{e} v$ wenn $e=(u,v) \in E$

$u \overset{W}{\rightsquigarrow} v$ wenn W Weg von u nach v

Distanz: $d(u,v) := \min \{ c(W) \mid u \overset{W}{\rightsquigarrow} v \}$

Beispiel:



$c(A \to C) = 6$

$c(A \to B \to C) = 4 + x$

$c(A \to B \to C \to A \to B \to C) = 4 + x + (13 + x)$

$$\text{dist}(A, C) = \begin{cases} 6 & \text{falls } x \geq 2 \\ 4 + x & \text{falls } -13 \leq x < 2 \\ -\infty & \text{falls } x < -13 \end{cases}$$

Zyklus $A \to B \to C \to A$ negativ

$\rightsquigarrow$ wiederhole Zyklus beliebig oft

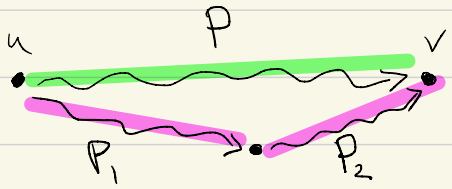
$\rightsquigarrow$ beliebig kleine negative Kosten

Annahme: $\nexists$ negativer Zyklus

$\rightsquigarrow$ kann jeden Weg zu Pfad abkürzen

$\rightsquigarrow$ $d(s, s) = 0$

Dreiecksungleichung: $d(u,v) \leq d(u,w) + d(w,v)$

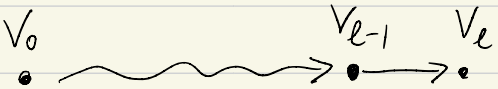


$\rightsquigarrow (P_1, P_2)$ ist Weg von u nach v

$\rightsquigarrow c(P_1, P_2) = d(u,w) + d(w,v)$

wähle günstigste Pfade P, P_1, P_2

Struktur günstigster Wege:



$(\ell \geq 1)$

Falls $v_0, \ldots, v_{\ell-1}, v_\ell$ günstigst, dann auch $v_0, \ldots, v_{\ell-1}$

Rekurrenz:

$$\forall v \neq s. \quad d(s,v) = \min_{u \to v} d(s,u) + c(u,v)$$

Dynamische Programmierung ? Berechnungsreihenfolge ?

Azyklische Graphen

Idee: topologische Sortierung

FOR $v \in V$ in topologischer Reihenfolge:

$$d[v] \leftarrow \begin{cases} 0 & \text{falls } v = s \\ \infty & \text{falls } v \neq s,\ \deg_{in}(v) = 0 \\ \min\limits_{u \to v} d[u] + c(u,v) & \text{sonst} \end{cases}$$

$d[u]$: u kommt vor v in top Sort. $\rightsquigarrow$ $d[u]$ schon berechnet

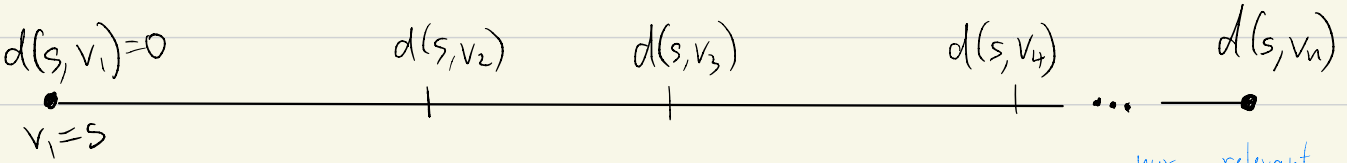
Beispiel: siehe S. 1

Laufzeit: $O(|V| + |E|)$ falls Adj.-liste gegeben

(Analyse wie DFS, BFS)

Nicht-negative Kantenkosten

Idee: sortiere Knoten nach Distanz von s



Annahme: alle Distanzen von s verschieden

nur relevant für Diskussion Algorithmus/Analyse funktioniert auch ohne Annahme

Rekurrenz: $d(s,v_k) = \min_{\substack{v_i \to v_k \\ i<k}} d(s,v_i) + c(v_i,v_k) \quad (k \geq 2)$

Beweis: $\forall i > k.\ \underbrace{d(s,v_i)}_{> d(s,v_k)} + \underbrace{c(v_i,v_k)}_{\geq 0} > d(s,v_k)$

$\rightsquigarrow$ Rekurrenz auf S.3: Vorgänger v_i mit $i > k$ irrelevant

wie Reihenfolge $v_1, \dots, v_n$ berechnen?

angenommen: $S = \{v_1, \dots, v_{k-1}\}$ bekannt

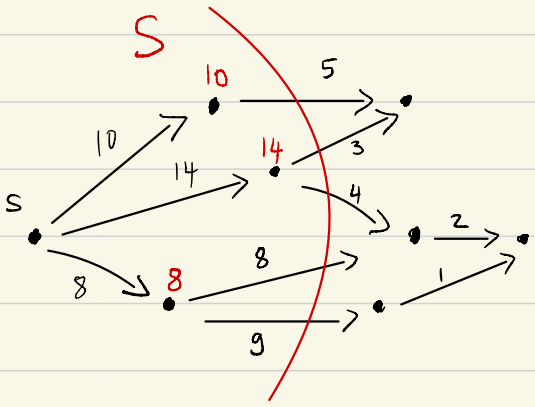
berechne v_k:

wähle Kante $u^* \to v^*$,

$u^* \in S$, $v^* \notin S$ mit

$d(s,u^*) + c(u^*,v^*)$ minimal

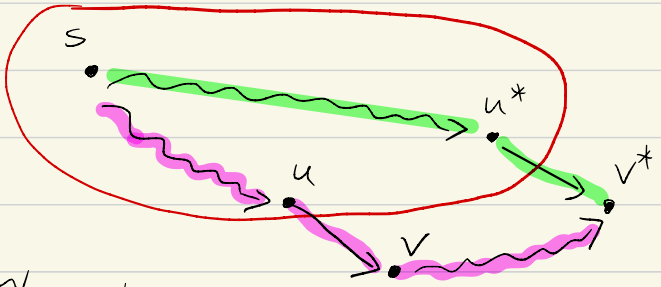
$\rightsquigarrow$ $v_k = v^*$



neu berechnet zuvor bekannt

Behauptung: $d(s,v^*) = d(s,u^*) + c(u^*,v^*)$

Beweis



W muss S an einer Stelle verlassen da $s \in S$ und $v^* \notin S$

$\forall W, s \overset{W}{\rightsquigarrow} v^*$.

$$c(W) \geq d(s,u) + c(u,v) + \underbrace{d(v,v^*)}_{\geq 0} \quad (u \in S, v \notin S)$$

$$\geq d(s,u) + c(u,v)$$

$$\geq d(s,u^*) + c(u^*,v^*) \quad \text{(Wahl von } u^* \to v^*\text{)}$$

$\rightsquigarrow$ kein Weg günstiger ▨

Schnelle Laufzeit: wie v^* schnell berechnen?

alle Kanten $u \to v$ mit $u \in S$, $v \in V \setminus S$ aufzählen?

verwalte Array $d[\,]$ von (oberen) Schranken

$$d[v] = \min_{\substack{u \to v \\ u \in S}} d(s,u) + c(u,v) \qquad (v \neq s)$$

- Wahl von v^*: Knoten in $V \setminus S$ mit minimaler Schranke
- v^* zu S hinzugefügt: alle Schranken bleiben gleich bis auf Nachfolger von v^*

Dijkstra (s):

$d[s] \leftarrow 0$, $d[v] \leftarrow \infty$ für $v \in V \setminus \{s\}$

$S \leftarrow \emptyset$ [$H \leftarrow$ make-heap(V), decrease-key$(H, s, 0)$]

WHILE $S \neq V$:

wähle $v^* \in V \setminus S$ mit $d[v^*]$ minimal

[$v^* \leftarrow$ extract-min(H)]

$S \leftarrow S \cup \{v^*\}$

FOR $(v^*, v) \in E$, $v \notin S$:

$d[v] \leftarrow \min\{d[v], d[v^*] + c(v, v^*)\}$

[decrease-key$(H, v, d[v])$]

Beispiel:

shortest-path tree