Institut für Theoretische Informatik
Lehrstuhl Prof. Dr. D. Wagner

Theorie-Übungsblatt 3 – Lösungsvorschläge

Algorithmen für Routenplanung, Sommer 2010

Problem 1: Arc-Flags, Aufwärmübung *

Gegeben sei ein Graph $G = (V, E, \text{len})$ mit einer Partition $\mathcal{P} := \{C_1, \ldots, C_k\}$ auf V. Das Arc-Flag einer Kante $e \in E$ bezüglich einer Zelle $C \in \mathcal{P}$ sei mit $\text{AF}_C(e)$ bezeichnet.

(a) Es sei $|\mathcal{P}| = 1$. Weiterhin werde die All-Pair-Shortest-Path-Variante für die Vorberechnung benutzt. Unterscheidet sich (wenn ja, worin) der Suchram, das heißt die Anzahl relaxierter Kanten bzw. abgearbeiter Knoten, des Arc-Flags-Algorithmus von Dijkstra's Algorithmus? Begründen Sie Ihre Antwort.

Lösung. Wegen $|\mathcal{P}| = 1$ gibt es genau eine Zelle C die *alle* Knoten enthält. Bei der All-Pair-ShortestPath-Variante der Vorberechnung werden nun für jeden Knoten $v \in C$ kürzeste Wege Bäume T_v auf dem Rückwärtsgraph $\overleftarrow{G}$ aufgebaut, und Flags auf Kanten $e \in T_v$ die auf dem kürzeste Wege Baum liegen auf `true` gesetzt.

Wir analysieren den Suchraum bezüglich abgearbeiteter Knoten und relaxierter Kanten.

- **Abgearbeitete Knoten.** Zu einer beliebigen s-t-Anfrage sei $v \in V$ ein Knoten im Suchraum von Dijkstra's Algorithmus (ohne Arc-Flags). Damit gilt $\text{dist}(s, v) \leq \text{dist}(s, t)$.

  Da nun bei der Arc-Flags-Vorberechnung die All-Pair-Shortest-Path-Methode angewandt wurde, wurde für alle Kanten $e \in T_v$ auf dem kürzeste Wege Baum von v das Arc-Flag $\text{AF}_C(e)$ auf `true` gesetzt. Damit existiert ein Weg P von s nach v in G mit $\text{len}\, P = \text{dist}(s, v) \leq \text{dist}(s, t)$ dessen Kanten alle offene Flaggen haben.

  Somit wird der Knoten v ebenfalls von der Arc-Flags-Query erreicht und wegen $\text{dist}(s, v) \leq \text{dist}(s, t)$ ebenfalls abgearbeitet.

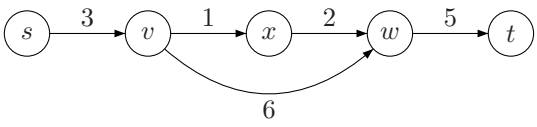
- **Relaxierte Kanten.** Sei $G_f = (V, E_f)$ mit $E_f \subseteq E$ der Subgraph von G der ausschließlich Kanten enthält deren Arc-Flag auf `true` gesetzt ist.

  Wegen der All-Pair-Shortest-Path-Vorberechnung wird E_f definiert durch

  $$E_f := \bigcup_{v \in V} T_v.$$

  Eine Kante $e = (v, w)$ ist in $E \setminus E_f$ genau dann wenn es keinen kürzeste Wege Baum T_v zu einem Knoten v gibt, der die Kante e enthält. Dies ist der Fall wenn es einen Pfad $P_{v,w} := [v, \ldots, w]$ in G gibt mit $\text{len}\, P_{v,w} < \text{len}(v, w)$.

  Wird also zu einer s-t-Anfrage der Knoten v abgearbeitet so ist die Kante (v, w) im Suchraum von Dijkstra's Algorithmus enthalten, nicht aber im Suchraum der Arc-Flags Query.



Die Kante (v, w) hat ihr Arc-Flag nicht gesetzt, da $\text{len}[v, x, w] < \text{len}(v, w)$.

□

(b) Es sei nun $|\mathcal{P}| = n$, das heißt jede Zelle enthält genau *einen* Knoten des Graphen. Zeigen Sie: Eine s-t-Anfrage besucht nur Knoten entlang des kürzesten s-t-Weges. Muss dazu der kürzeste Weg eindeutig sein?

Lösung. Gegeben dem Fall dass jede Zelle $C_v \in \mathcal{P}$ ausschließlich den Knoten v enthält, besteht der Subgraph G_f zu einer s-t-Anfrage gerade aus dem kürzeste Wege Baum T_t. Das heißt es werden nur Kanten (und Knoten) entlang des kürzesten s-t-Weges relaxiert. Da in einem Baum zwischen zwei Knoten (insbesondere also auch von s nach t) genau ein Weg existiert, wird im Falle der Mehrdeutigkeit eines kürzesten s-t-Weges nur einer der beiden Wege abgearbeitet. □

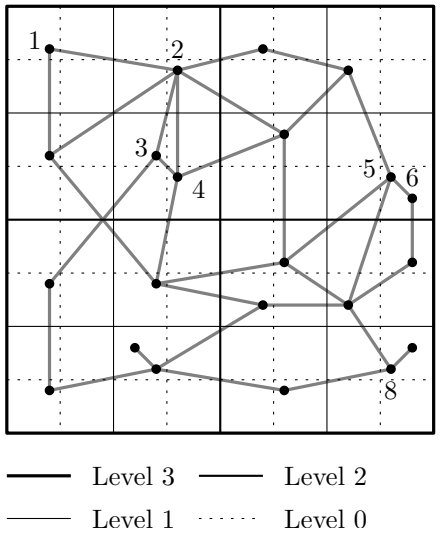
Problem 2: Multi-Level Partitionierung **

Gegeben sei ein Graph $G = (V, E, \text{len})$. Eine *multi-level Partition* $\mathcal{P}$ mit k Levels sei rekursiv wie folgt definiert:

- $\mathcal{P}_k := \{V\}$
- $\mathcal{P}_i = \bigcup_{C \in \mathcal{P}_{i+1}} P_C$ wobei P_C eine Partition der Zelle C ist. Jede Zelle C der Partition $\mathcal{P}_{i+1}$ wird also "verfeinert".

Bemerkung: Arc-Flags $\text{AF}_C^{(i)}(e)$ auf Level i beziehen sich auf Zellen $C \in \mathcal{P}_{i-1}$.

Weiterhin sei der *gemeinsame Level* zweier Knoten u und v das kleinste Level i für das eine Zelle $C \in \mathcal{P}_i$ existiert die sowohl u als auch v enthält.



(a) Betrachten Sie den Graphen aus obiger Abbildung. Geben Sie das gemeinsame Level der Knotenpaare $(1, 2)$, $(1, 4)$, $(5, 6)$ und $(3, 8)$ an.

Lösung. Es bezeichne $\text{comlev} : V \times V \to \{0, \ldots, k\}$ die Abbildung die einem Knotenpaar seinen gemeinsamen Level zuordnet. Aus dem Bild lässt sich ablesen, dass die gemeinsamen Level der jeweiligen Knoten $\text{comlev}(1, 2) = 2$, $\text{comlev}(1, 4) = 2$, $\text{comlev}(5, 6) = 0$ und $\text{comlev}(3, 8) = 3$ sind. □

(b) Geben Sie ein effizientes (algorithmisches) Verfahren zur Bestimmung des gemeinsamen Levels zweier Knoten u und v an.

Hinweis: Benutzen Sie eine geschickte Nummerierung der Zellen auf dem untersten Level, die Sie an die Knoten speichern.

Lösung. Die Idee für ein effizientes Verfahren zur Berechnung von $\text{comlev}(u, v)$ für zwei beliebige Knoten $u, v \in V$ besteht darin, die Zellen $C_j \in \mathcal{P}_0$ in Level 0 geschickt zu nummerieren. Jeder Knoten $v \in V$ bekommt dann die Zellennummer auf Level 0 zugewiesen in der er liegt, also

$$\text{cell}(v) = j \Leftrightarrow v \in C_j \quad \text{wobei} \quad C_j \in \mathcal{P}_0.$$

Der Trick besteht nun darin die Nummerierung derart zu wählen, dass $\text{cell}(v)$ implizit die Informationen enthält in welcher Zelle auf welchem Level sich der Knoten v befindet.

Unser Verfahren soll für allgemeine multi-level Partitionen platzsparend funktionieren. Insbesondere ist es im Allgemeinen nicht erforderlich dass alle Unterteilungen der Zellen die gleiche Größe haben. Es bezeichne also $\phi(i, j)$ die Größe der Unterpartition der j-ten Zelle auf Level i (es gibt also $\phi(i, j)$ paarweise verschiedene Zellen $C \in \mathcal{P}_{i-1}$ für die gilt $C \subset C_j$ mit $C_j \in \mathcal{P}_i$). Dann wählen wir für jedes Level i die minimale Anzahl Bits

$$\psi(i) := \max_j \left\{ \lceil \log_2(\phi(i, j)) \rceil \right\}$$

die erforderlich ist um die Zellen auf Level i, die zu einer Unterteilung einer Level $i + 1$ Zelle gehören, zu nummerieren.

Sei nun $C \in \mathcal{P}_{i+1}$ eine Zelle auf Level $i + 1$ mit der Unterteilung $\mathcal{U}_C := \{C_1, \ldots, C_{\phi(i,j)}\} \subset \mathcal{P}_i$ in Level i. Dann sei für einen Knoten v die Funktion $\Gamma(v, i)$ definiert als

$$\Gamma(v, i) := \begin{cases} 2^{\sum_{\nu=0}^{i} \psi(\nu)} \cdot \gamma & \text{wenn } v \in C_\gamma \text{ und } C_\gamma \in \mathcal{U}_C \\ 0 & \text{sonst} \end{cases}.$$

Es wird also der Index der Zelle der Unterpartition auf Level i berechnet und an die korrekte Stelle in der Bitdarstellung der Nummerierung verschoben. Die Zelleninformation ergibt sich damit insgesamt durch

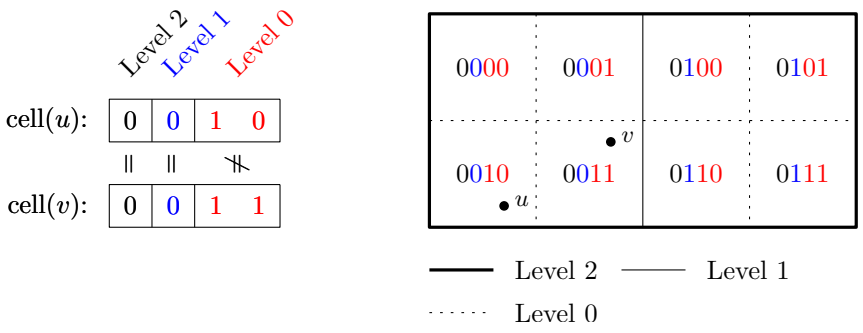
$$\text{cell}(v) := \sum_{i=0}^{k} \Gamma(v, i).$$

Damit folgt dass für zwei Knoten $u, v \in V$ in der Bitdarstellung von $\text{cell}(u)$ bzw. $\text{cell}(v)$ diejenigen Blöcke übereinstimmen für die u und v in einer gemeinsamen Zelle liegen. Um schließlich den gemeinsamen Level zweier Knoten $u, v \in V$ zu berechnen, können wir also nach dem niederwertigsten Block suchen in dem $\text{cell}(u)$ und $\text{cell}(v)$ gerade noch übereinstimmen. Formal also

$$\text{comlev}(u, v) := \min\{i \mid \text{cell}(u)/2^{\sum_{\nu=0}^{i} \psi(\nu)} = \text{cell}(v)/2^{\sum_{\nu=0}^{i} \psi(\nu)}\}.$$

Dabei entspricht die Division einer Ganzzahldivision; Im Falle von $\cdot/2^n$ also gerade einem Rechtsshift um n Bits.

Das folgende Bild verdeutlicht das Verfahren.



Das Bild illustriert eine multi-level Partition mit unterschiedlich großen Unterteilungen. Die Nummerierung der Zellen nach obigem Schema ist farbig dargestellt. Außerdem ist beispielhaft für zwei Knoten u und v aufgeführt wie sich der gemeinsame Level berechnen lässt indem nach dem niederwertigsten Block in der Bitdarstellung von cell(u) bzw. cell(v) gesucht wird der gerade noch übereinstimmt.

Effizientes Berechnen von comlev. Um das Berechnen von comlev(u, v) zu beschleunigen kann folgendermaßen vorgegangen werden. In einer Vorberechnung speichern wir in einem Array level eine Abbildung die jedem Bit-Index zuordnet welcher Level der Partition durch das jeweilige Bit beschrieben wird.

Zum Bestimmen von comlev(u, v) berechnen wir nun zunächst

$$b := \text{MSB}(\text{cell}(u)\,\text{XOR}\,\text{cell}(v)),$$

wobei MSB den Index des *most significant Bits* bezeichnet, also das am weitesten links liegende Bit das auf 1 gesetzt ist. Ist keines der Bits auf 1 gesetzt, so sei das Ergebnis von MSB als -1 definiert.

In einem zweiten Schritt können wir mit level(b) den größten Level bestimmen indem sich u und v unterscheiden. Das heißt der gemeinsame Level der Knoten ergibt sich durch

$$\text{comlev}(u, v) := \text{level}(b) + 1.$$

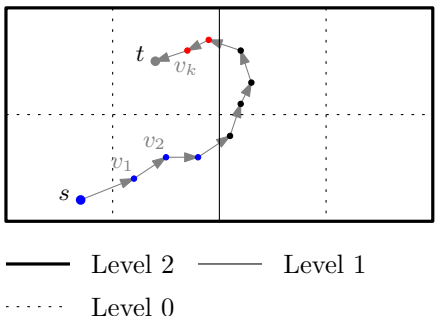
Moderne Prozessoren können sowohl die XOR-Operation, das Berechnen des MSB und das Inkrementieren um 1 in jeweils einem Takt berechnen, wodurch mit diesem Verfahren ein möglichst geringer Berechnungsoverhead entsteht. □

Problem 3: Multi-Level Arc-Flags ***

Gegeben sei ein Graph $G = (V, E, \text{len})$ und eine multi-level Partition $\mathcal{P} = \{\mathcal{P}_0, \ldots, \mathcal{P}_k\}$.

(a) Sei $P := [s, v_1, v_2, \ldots, v_k, t]$ ein kürzester s-t-Weg in G. Zeigen Sie: Der gemeinsame Level der Knoten v_i und t nimmt mit zunehmendem i nicht notwendigerweise monoton ab.

Lösung. Wir zeigen die Aussage durch ein Beispiel. Gegeben sei ein Graph $G = (V, E, \text{len})$ mit der multi-level Partition wie in der folgenden Abbildung angegeben.



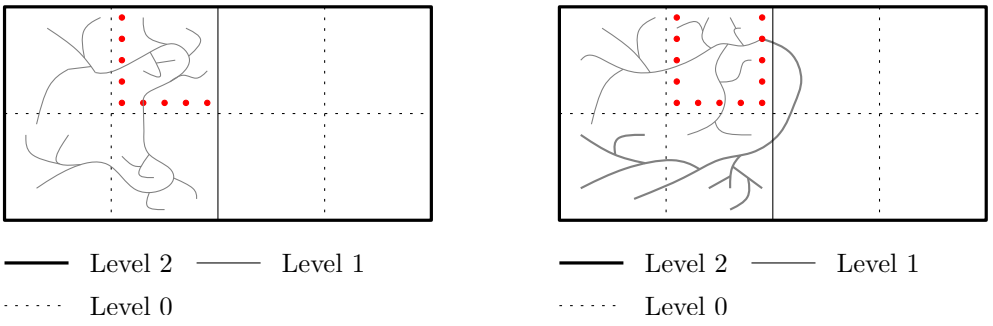
Der kürzeste Weg $P = [s, v_1, \ldots, v_k, t]$ für zwei Knoten $s, t \in V$ ist in der Abbildung eingezeichnet. Zu Beginn ist der gemeinsame Level von s und t gerade 1 (blaue Knoten). Da

der kürzeste Weg jedoch die gemeinsame Zelle in Level 1 verlässt, steigt der gemeinsame Level auf 2 an (schwarze Knoten), bevor der Weg die Zelle des Zielknotens t erreicht (rote Knoten), wodurch der gemeinsame Level auf 0 sinkt. Der Verlauf von $\text{comlev}(v_i, t)$ ist also nicht-monoton. □

(b) Welche Konsequenz ergibt sich daraus für die Vorberechnung von Arc-Flags basierend auf Randknoten?

Lösung. Für die Vorberechnung ergeben sich folgende Konsequenzen. Sei $C \in \mathcal{P}_i$ eine Zelle auf Level i und C^* die zu C gehörende Superzelle, also $C^* \in \mathcal{P}_{i+1}$ mit $C \subset C^*$.

Für die Berechnung der Level-$(i+1)$ Arc-Flags reicht es für C *nicht* aus nur die Randknoten bezüglich Level i zu betrachten, das heißt Knoten $v \in C$ zu denen es eine Kante $(u, v) \in E$ gibt mit $u \in C'$ wobei $C' \neq C$ und $C' \in \mathcal{P}_i$. Würde man sich auf solche Randknoten beschränken, so würde man Flaggen entlang kürzester Wege nicht öffnen die zwar in der gleichen Superzelle C^* beginnen, aber die Zelle C^* temporär verlassen.



Links der naive Ansatz der fehlschlägt. Kürzeste Wege in die Zelle C auf Level i können temporär die gemeinsame Superzelle C^* verlassen. Es ist also für die Vorberechnung der Level-$(i+1)$ Arc-Flags notwendig kürzeste Wege Bäume von *allen* Randknoten r zu berechnen für die es eine Kante $(u, r) \in E$ gibt mit $u \notin C$ und $r \in C$. Die Flaggen dürfen jedoch weiterhin ausschließlich auf Kanten $(u, v) \in E$ gesetzt werden für die gilt $u \in C^*$. □

(c) Geben Sie den Algorithmus zur Vorberechnung basierend auf Randknoten in Pseudo-Code an. Sie können dabei Dijkstra's Algorithmus als Baustein verwenden.

Lösung. Algorithmus 1 leistet das Gewünschte.

Für jeden Level werden die Arc-Flags seperat in einer Schleife berechnet. Da sich Level-i Arc-Flags auf Zellen in Level $i-1$ beziehen, betrachten wir jede Superzelle $C^* \in \mathcal{P}_i$ und berechnen die Arc-Flags für alle Unterzellen C in Level $i-1$ mit $C \subset C^*$.

Zunächst werden *alle* Randknoten ermittelt und dabei gleichzeitig die Intrazellenkanten in C auf `true` gesetzt. In einem zweiten Schritt wird von allen Randknoten $r \in R$ ein kürzeste Wege Baum auf dem Rückwärtsgraph mit `backDijkstra` aufgebaut. Dieser Dijkstra-Algorithmus kann stoppen sobald alle Knoten aus C^* abgearbeitet sind.

Schließlich wird für alle Kanten (u, v) auf dem kürzeste Wege Baum das Level-i Arc-Flag zur Zelle C auf `true` gesetzt wenn der Knoten u sich in der gleichen Superzelle C^* befindet wie r. □

(d) Erweitern Sie Dijkstra's Algorithmus zu einer multi-level Arc-Flags-Query. Beschreiben Sie

Algorithmus 1: ML-Arc-Flags Vorberechnung

Eingabe : Graph $G = (V, E, \text{len})$ und eine multi-level Partition $\mathcal{P} = \{\mathcal{P}_0, \ldots, \mathcal{P}_k\}$
Vorbedingung : Alle Flaggen sind mit `false` initialisiert.
Seiteneffekte : Multi-level Arc-Flags AF auf G zu $\mathcal{P}$.

```
1  für alle Level i ← 1...k tue
2      für alle Zellen C* ∈ P_i tue
3          für alle Zellen C ∈ P_{i-1} mit C ⊂ C* tue
               // Setzen der Flaggen auf Intrazellenkanten und
               // Bestimmen der Randknoten der Zelle C
4              R ← ∅
5              für alle Kanten (u, v) ∈ E tue
6                  wenn u ∉ C und v ∈ C dann
7                      R ← R ∪ {v}
8                  sonst wenn u ∈ C und v ∈ C dann
9                      AF_C^(i)(u, v) ← true
               // Kürzeste Wege Bäume berechnen und Flaggen setzen
10             für alle r ∈ R tue
11                 T ← backDijkstra(r)
12                 für alle Kanten (u, v) ∈ T tue
13                     wenn u ∈ C* dann
14                         AF_C^(i)(u, v) ← true
```

die notwendigen Änderungen in Pseudo-Code.

Lösung. Dijkstra's Algorithmus kann fast ohne Änderungen übernommen werden. Die einzige Änderung betrifft das Relaxieren von Kanten: Eine Kante $e = (u, v) \in E$ wird nur dann relaxiert, wenn für das Arc-Flag auf e

$$\text{AF}_{\text{cell}(t)}^{(\text{comlev}(u,t))}(e) = \texttt{true}$$

erfüllt ist.

Algorithmus 2: Arc-Flag Dijkstra

Eingabe : Graph $G = (V, E, \text{len})$, multi-level Arc-Flags AF und zwei Knoten $s, t \in V$
Ausgabe : Ein kürzester s-t-Weg in G

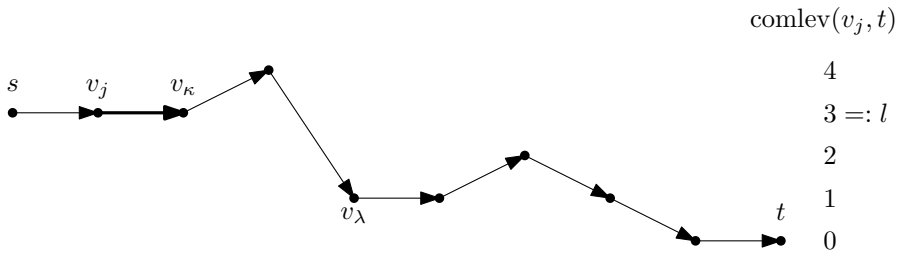
```
   // ...
1  u ← Q.deleteMin()
   // ...
2  für alle Kanten e = (u, v) tue
3      wenn AF_cell(t)^(comlev(u,t))(e) dann
4          relaxEdge(e)
   // ...
```

□

(e) Zeigen Sie die Korrektheit Ihres Algorithmus aus Aufgabe (d) wenn Sie die Vorberechnung aus Aufgabe (c) benutzen.

Lösung. Seien in G ohne Beschränkung der Allgemeinheit die kürzesten Wege eindeutig. Seien weiterhin $s, t \in V$ Knoten und $P = [s, v_1, \ldots, v_r, t]$ der kürzeste s-t-Weg. Insbesondere sei $v_0 = s$. Wir zeigen, dass jede Kante $e = (v_j, v_{j+1})$ entlang des Weges P von der Arc-Flags-Query relaxiert wird.

Sei $l := \text{comlev}(v_j, t)$ der gemeinsame Level der Knoten v_j und t. Sei für $l > 0$ zur Abkürzung $f := \text{AF}^{(l)}_{C_t^{l-1}}(v_j, v_{j+1})$ wobei C_t^{l-1} die Zelle auf Level $l-1$ ist die t enthält.



Wir unterscheiden zwei Fälle:

1. $l = 0$: Der Knoten v_j befindet sich in der selben Zelle $C \in \mathcal{P}_0$ wie t. Dann wird wegen der gesetzten Intrazellenflags die Kante relaxiert.

2. $l > 0$: Wegen $\text{comlev}(t, t) = 0$ existiert ein Index $\kappa \geq j$ derart, dass v_κ der letzte Knoten auf dem Pfad P mit $\text{comlev}(v_\kappa, t) = l$ ist. Des Weiteren muss es einen *kleinsten* Index $\lambda > \kappa$ geben, so dass $\text{comlev}(v_\lambda, t) < l$ ist. Sei $e' := (v_{\lambda-1}, v_\lambda)$. Da λ der kleinste Index mit der geforderten Eigenschaft ist, gilt $\text{comlev}(v_{\lambda-1}, t) > \text{comlev}(v_\lambda, t)$, das heißt v_λ ist ein Randknoten bezüglich Level $\text{comlev}(v_{\lambda-1}, t)$ und damit insbesondere auch bezüglich Level $l-1$.

   Während der Vorberechnung der Level l Arc-Flags wurde also von v_λ ein kürzeste Wege Baum T_{v_λ} aufgebaut. Des Weiteren wurde für alle Kanten $(u, v) \in T_{v_\lambda}$ (auf dem kürzeste Wege Baum) mit $\text{comlev}(u, v_\lambda) = l$ das Arc-Flag $f' := \text{AF}^{l}_{C^{l-1}_{v_\lambda}}(u, v)$ gesetzt wobei $C^{l-1}_{v_\lambda} \in \mathcal{P}_{l-1}$ die Zelle auf Level $l-1$ ist, die v_λ enthält. Wegen

$$\text{comlev}(u, v_\lambda) = \text{comlev}(v_j, v_\lambda) = \text{comlev}(v_j, t)$$

   ist $f' = f = \texttt{true}$. Das heißt, die Kante (v_j, v_{j+1}) wird relaxiert.

Da diese Argumentation für alle Kanten (v_j, v_{j+1}) auf dem kürzesten Weg geführt werden kann, sind die relevanten Flags entlang aller Kanten des kürzesten Weges gesetzt, und der kürzeste Weg wird von der Arc-Flags-Query gefunden. Der Algorithmus ist somit korrekt. □

Problem 4: Self-Bounding Bidirectional Reach-Algorithmus ***

Gegeben sei ein Graph $G = (V, E, \text{len})$. Der *reach* eines Knotens u bezüglich eines kürzesten Weges $P := [s, \ldots, u, \ldots, t]$ ist definiert als $r_P(u) := \min(\text{dist}(s, u), \text{dist}(u, t))$. Der reach in G von u ist

das Maximum aller reach-Werte von u bezüglich aller kürzesten Wege durch u, also

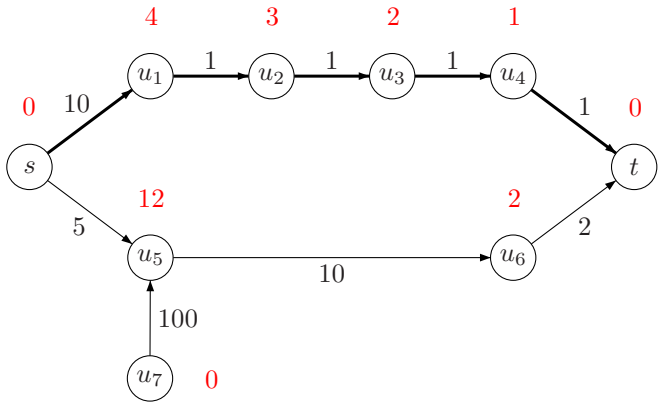
$$r(u) := \max\{r_P(u) \mid P \text{ ist ein kürzester Weg der } u \text{ enthält}\}.$$

Studieren Sie den Self-Bounding Bidirectional Reach-Algorithmus aus der Vorlesung.

(a) Zeigen Sie, dass der kürzeste Weg unter Benutzung des Abbruchkriteriums des normalen bidirektionalen DIJKSTRA-Algorithmus nicht notwendigerweise im Suchraum enthalten ist.

Lösung. Wir zeigen die Aussage durch ein Gegenbeispiel. Als Abbruchkriterium benutzen wir $\text{minKey}(\overrightarrow{Q}) + \text{minKey}(\overleftarrow{Q}) \geq \mu$ wobei μ die Länge des vorläufigen kürzesten Weges ist.

Gegeben sei folgender Graph $G = (V, E, \text{len})$. Die Reachwerte $r(v)$ sind bereits ausgerechnet und stehen an den Knoten.



Es vermöge die bidirektionale Suche abwechselnd einen Vorwärts- und einen Rückwärtssuchschritt durchführen. Der Ablauf des Algorithmus ist dann wie folgt.

Vorwärtssuche: `settle`(s)

- `insert`(u_1, 10)
- `insert`(u_5, 5)

Rückwärtssuche: `settle`(t)

- `insert`(u_4,1)
- `insert`(u_6,2)

Vorwärtssuche: `settle`(u_5)

- `insert`(u_6, 15)
  Da u_6 bereits von der Rückwärtssuche entdeckt ist, setze $\mu =: 17$ und $M := u_6$.

Rückwärtssuche: `settle`(u_4)

- `insert`(u_3,2)

Vorwärtssuche: `settle`(u_1)

Wegen $r(u_1) = 4 < \text{dist}(s, u_1)$ prune diesen Pfad.

Rückwärtssuche:

Es ist

$$\underbrace{\text{minKey}(\overrightarrow{Q})}_{=15} + \underbrace{\text{minKey}(\overleftarrow{Q})}_{=2} \geq \mu = 17$$

stop.

Der gefundene Weg $[s, \ldots, u_6, t]$ hat jedoch Länge 17, während der kürzeste Weg Länge 14 hat. □

(b) Zeigen Sie die Korrektheit des Abbruchkriteriums aus der Vorlesung.

Lösung. Das Abbruchkriterium aus der Vorlesung ist wie folgt definiert. Die beiden Suchen werden nach einer beliebigen Strategie abgewechselt. Eine Suche stoppt sobald gilt $\text{minKey}(Q) \geq \mu/2$. Sind beide Suchen gestoppt, wird der kürzeste Weg der Länge μ ausgegeben.

Bevor wir die Aufgabe beweisen, sollten wir uns noch zwei Sachen klar machen:

(i) Zu einer s-t-Anfrage ist stets $\text{dist}(s, t) \leq \mu$. Der vorläufig gefundene kürzeste Weg kann nie kürzer als der echte kürzeste Weg sein.

(ii) Sei $\text{dist}(s, t)$ die Länge eines kürzesten s-t-Weges und v ein Knoten auf dem kürzesten s-t-Weg. Dann folgt aus $r(v) < \text{dist}(s, v)$ stets $\text{dist}(v, t) < \text{dist}(s, v)$.

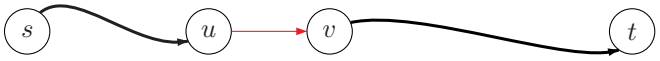
Beweis. Da diese Aussage vielleicht nicht sofort einsichtig ist, sei hier ein kurzer Beweis. Stumpfes Rechnen liefert

$$\begin{aligned}
\text{dist}(s, v) &> r(v) \\
&= \max\{r_P(v) \mid P \text{ ist ein kürzester Weg der } v \text{ enthält}\} \\
&\geq r_{s,t}(v) \\
&= \min\big(\text{dist}(s, v), \text{dist}(v, t)\big) \\
&= \text{dist}(v, t).
\end{aligned}$$

□

Seien ohne Beschränkung der Allgemeinheit die kürzesten Wege in G eindeutig. Sei $P := [s, v_1, \ldots, v_{k-1}, t]$ der kürzeste s-t-Weg. Wir zeigen die Aussage durch Widerspruch, nehmen wir also an der Weg P würde durch den Self-Bounding Bidirectional Reach-Algorithmus nicht gefunden werden. Das bedeutet, es existiert mindestens ein Knoten $v \in P$ der von mindestens einer Suche nicht in die Schlange eingefügt wurde. Wir unterscheiden die folgenden Fälle.

Fall I: Kein Pruning. Es wurde kein Knoten $v \in P$ wegen des Reachwertes $r(v)$ geprunt (weder von der Vorwärts- noch von der Rückwärtssuche). Damit sich die Suchen also bei v nicht treffen konnten sei also ohne Beschränkung der Allgemeinheit (u, v) die Kante die von einer der Suchen nicht relaxiert wurde.



Dann muss gelten

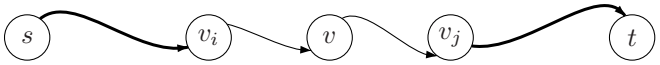
$$\text{dist}(s,u) > \frac{\mu}{2} \qquad \text{und} \qquad \text{dist}(u,t) > \frac{\mu}{2}.$$

Dies hat zur Konsequenz

$$\begin{aligned}\text{dist}(s,t) &= \text{dist}(s,u) + \underbrace{\text{len}(u,v)}_{\geq 0} + \text{dist}(v,t)\\ &> \frac{\mu}{2} + \frac{\mu}{2}\\ &= \mu\\ &\overset{(i)}{\geq} \text{dist}(s,t),\end{aligned}$$

was nicht sein kann.

Fall II: Pruning in einer Suche. Sei ohne Beschränkung der Allgemeinheit v_j ein Knoten in der Rückwärtssuche, der geprunt wurde. Da der Weg P nicht gefunden wurde, muss es einen Knoten v_i mit $i < j$ geben, der von der Vorwärtssuche nicht abgearbeitet wurde.



Da v_i nicht abgearbeitet wurde, muss gelten $\text{dist}(s,v_i) \geq \mu/2$ und weil v_j geprunt wurde, muss gelten $\text{dist}(v_j,t) > r(v_j)$. Mit (ii) folgt $\text{dist}(s,v_j) < \text{dist}(v_j,t)$, also insgesamt

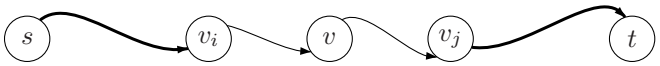
$$\text{dist}(s,v_i) \geq \frac{\mu}{2} \qquad \text{und} \qquad \text{dist}(v_j,t) > \frac{\mu}{2}.$$

Mit der gleichen Argumentation wie in Fall I ist

$$\begin{aligned}\text{dist}(s,t) &= \text{dist}(s,v_i) + \underbrace{\text{dist}(v_i,v_j)}_{\geq 0} + \text{dist}(v_j,t)\\ &> \frac{\mu}{2} + \frac{\mu}{2}\\ &= \mu\\ &\overset{(i)}{\geq} \text{dist}(s,t),\end{aligned}$$

was wieder zu einem Widerspruch führt.

Fall III: Pruning in beiden Suchen. Bleibt also noch der Fall, dass beide Suchen geprunt haben. Es gibt also Knoten v_i und v_j mit $i < j + 1$, so dass die Vorwärtssuche v_i und die Rückwärtssuche v_j geprunt hat.



Das heißt, dass gilt $r(v_i) < \text{dist}(s, v_i)$ und $r(v_j) < \text{dist}(v_j, t)$. Mit (ii) können wir ableiten

$$\text{dist}(v_i, t) < \text{dist}(s, v_i) \qquad \text{und} \qquad \text{dist}(s, v_j) < \text{dist}(v_j, t).$$

Insgesamt ist damit

$$\begin{aligned}
\text{dist}(s,t) &= \text{dist}(s, v_i) + \text{dist}(v_i, v_j) + \text{dist}(v_j, t) \\
&> \text{dist}(v_i, t) + \text{dist}(s, v_j) + \text{dist}(v_i, v_j) \\
&= \text{dist}(s,t) + \underbrace{2 \cdot \text{dist}(v_i, v_j)}_{\geq 0} \\
&\geq \text{dist}(s,t),
\end{aligned}$$

was ebenfalls ein Widerspruch ist.

Wir erhalten also insgesamt dass der kürzeste s-t-Weg P von dem Self-Bounding Bi-Directional Reach-Algorithmus mit dem Abbruchkriterium aus der Vorlesung gefunden wird. □