Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Sebastian Lamm, Demian Hespe

2. Übungsblatt zu Algorithmen II im WS 2018/2019

http://algo2.iti.kit.edu/AlgorithmenII_WS18.php
{sanders, lamm, hespe}@kit.edu

Musterlösungen

Aufgabe 1 (*Rechnen: Monotone ganzzahlige* Priority Queues)

Bei einer Ausführung von *Dijkstra's Algorithmus* wird folgender Ausschnitt an *Priority Queue* Operationen protokolliert:

...

- `insert(a, 06 [00110] )` (Parameter: Knotenbezeichnung, Distanz [Distanz binär])
- `insert(b, 10 [01010] )`
- `insert(c, 07 [00111] )`
- `deleteMin()`
- `deleteMin()`
- `insert(d, 12 [01100] )`
- `deleteMin()`
- `insert(e, 16 [10000] )`

...

Zusätzlich wissen Sie, dass das maximale Kantengewicht im Graphen $C = 6$ beträgt und dass vor der ersten protokollierten Operation das letzte enthaltene Element aus der *Priority Queue* entfernt wurde. Dieses hatte den Wert $min = 5$.

a) Führen Sie die Operationen auf einer *Bucket Queue* aus. Geben Sie den Zustand der Datenstruktur nach jeder Operation an.

b) Wieviele *Buckets* werden für eine Ausführung auf einem *Radix Heap* benötigt? Führen Sie die Operationen auf einem *Radix Heap* aus. Geben Sie den Zustand der Datenstruktur und den Wertebereich der *Buckets* nach jeder Operation an.

Musterlösung:

a) *Bucket Queue*:

`insert(a, 06 [00110] ):`

0	1	2	3	4	5	6
						(a, 6)

min = 5

`insert(b, 10 [01010] ):`

0	1	2	3	4	5	6
			(b, 10)			(a, 6)

min = 5

`insert(c, 07 [01000] ):`

(für monotone *Priority Queues* nur wichtig, dass Elemente aus $[min, min + C]$ stammen!)

0	1	2	3	4	5	6
(c, 7)			(b, 10)			(a, 6)

min = 5

`deleteMin():`

0	1	2	3	4	5	6
(c, 7)			(b, 10)			

min = 6

`deleteMin():`

0	1	2	3	4	5	6
			(b, 10)			

min = 7

`insert(d, 12 [01100] ):`

0	1	2	3	4	5	6
			(b, 10)		(d, 12)	

min = 7

`deleteMin():`

0	1	2	3	4	5	6
					(d, 12)	

min = 10

`insert(e, 16 [10000] ):`

0	1	2	3	4	5	6
		(e, 16)			(d, 12)	

min = 10

Musterlösung:

b) *Radix Heap*:
(Es werden $K+2$ *Buckets* benötigt: $B[-1]$, $B[0]$, …, $B[K]$; mit $K = 1 + \lfloor \log_2 C \rfloor = 3$ ergeben sich 5 *Buckets*.)

`insert(a, 06 [00110] )`:

-1	0	1	2	3
		(a, 06 [00110])		
5	–	6–7	–	8–11

min = 5 [00101]

`insert(b, 10 [01010] )`:

-1	0	1	2	3
		(a, 06 [00110])		(b, 10 [01010])
5	–	6–7	–	8–11

min = 5 [00101]

`insert(c, 07 [00111] )`:

-1	0	1	2	3
		(a, 06 [00110]) (c, 07 [00111])		(b, 10, [01010])
5	–	6–7	–	8–11

min = 5 [00101]

`deleteMin()`:
($B[1]$ ist der erste gefüllte *Bucket*; min wird auf das kleinste enthaltene Element (6) gesetzt; die Elemente in $B[1]$ werden neu verteilt und anschließend das Element in $B[-1]$ entfernt)

-1	0	1	2	3
	(c, 07 [00111])			(b, 10 [01010])
6	7	–	–	8–12

min = 6 [00110]

`deleteMin()`:

-1	0	1	2	3
				(b, 10 [01010])
7	–	–	–	8–13

min = 7 [00111]

`insert(d, 12 [01100] )`:

-1	0	1	2	3
				(b, 10 [01010]) (d, 12 [01100])
7	–	–	–	8–13

min = 7 [00111]

`deleteMin()`:

-1	0	1	2	3
			(d, 12 [01100])	
10	11	–	12–15	16

min = 10 [01010]

`insert(e, 16 [10000] )`:

-1	0	1	2	3
			(d, 12 [01100])	(e, 16 [10000])
10	11	–	12–15	16

min = 10 [01010]

Aufgabe 2 (*Analyse: Laufzeit von Dijkstra's Algorithmus*)

Gegeben sei ein gerichteter Graph $G = (V,\ E)$ mit $|V| = n$ und $|E| = m$, sowie eine Kantengewichtungsfunktion $c : E \to \mathbb{R}_0^+$.

a) Beweisen Sie die Behauptung aus der Vorlesung, dass für $m = \Omega(n \log n \log\log n)$ Dijkstra's Algorithmus mit einem *binary heap* eine durchschnittliche Laufzeit von $O(m)$ besitzt.

b) Eine spezielle *Priority Queue* habe folgende Laufzeiteigenschaften:

- `insert`: $O(\log n)$
- `decreaseKey`: $O(1)$
- `deleteMin`: $O(\sqrt{m})$

(ob eine Datenstruktur mit diesen Eigenschaften existiert und Dijkstra's Algorithmus mit ihr korrekt arbeitet, ist eine andere Frage, aber wir nehmen für diese Aufgabe an es ginge :-))

Geben Sie eine kleinste obere Schranke für die Laufzeit von Dijkstra's Algorithmus unter Verwendung dieser *Priority Queue* an. Unter welcher Bedingung an das Verhältnis der Anzahl Knoten n zu Kanten m wird die Laufzeit linear in der Eingabegröße? Die Eingabe erfolgt in Form einer Adjazenzliste.

Musterlösung:

a) Für die durchschnittliche Laufzeit von Dijkstra's Algorithmus mit einem *binary heap* gilt:

$$O(m + n \log \frac{m}{n} \log n)$$

Zu zeigen ist, ob diese Laufzeit in $O(m)$ liegt für die gegebene Wahl von $m = \Omega(n \log n \log\log n)$. Wähle den kleinstmöglichen Wert für m. Falls die Aussage diesen Wert gilt, gilt Sie sicher auch für alle größeren m. Eingesetzt und umgeformt ergibt sich:

$$\begin{aligned}
& O(n \log n \log\log n + n \log \frac{n \log n \log\log n}{n} \log n) \\
\overset{\text{kürzen}}{=}\ & O(n \log n \log\log n + n \log(\log n \log\log n) \log n) \\
\overset{\log ab = \log a + \log b}{=}\ & O(n \log n \log\log n + n \log\log n + n \log\log\log n \log n) \\
\overset{\log\log\log n = O(\log\log n)}{=}\ & O(n \log n \log\log n) \\
=\ & O(m)
\end{aligned}$$

Damit liegt die Laufzeit in $O(m)$.

Für eine geringere Abhängigkeit, z.B. $m = O(n)$ würde der zweite Term den ersten im O-Kalkül dominieren und die Umformung würde nicht zu $O(m)$ führen.

b) Allgemein gilt für die Laufzeit von Dijkstra's Algorithmus:

$$O(m + m \cdot T_{decreaseKey}(n) + n \cdot (T_{deleteMin}(n) + T_{insert}(n)))$$

Mit den angegebenen Laufzeiten eingesetzt ergibt sich:

$$O(m + n\sqrt{m} + n \log n)$$

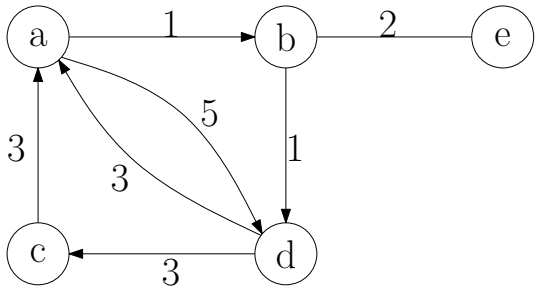
Unter den Forderungen $n \cdot \sqrt{m} = O(m)$ und $n \log n = O(m)$ ist die Laufzeit linear in m. Dies lässt sich umformen zu $\Omega(n) = \sqrt{m}$ und $\Omega(n \log n) = m$. Damit ergibt sich $m = \Omega(n^2)$.

Aufgabe 3 (*Einführung+Analyse: Bidirektionaler Dijkstra*)

In Vorlesung und Saalübung wurde eine bidirektionale Variante von Dijkstra's Algorithmus angesprochen, die in dieser Aufgabe näher untersucht werden soll.

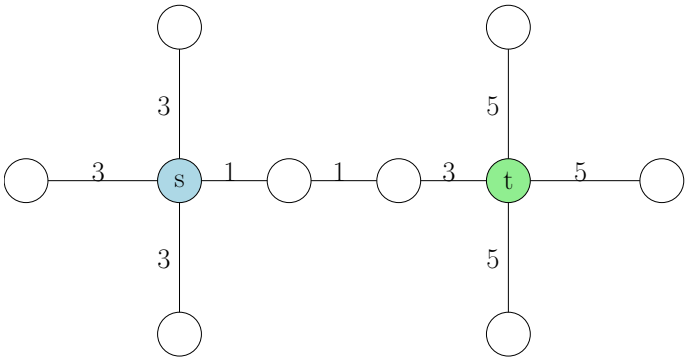
Zur Wiederholung:
Gegeben sei –wie üblich– ein gerichteter Graph $G = (V,\ E)$ mit $|V| = n$ und $|E| = m$, sowie eine Kantengewichtungsfunktion $c : E \to \mathbb{R}_0^+$. Gesucht ist der kürzeste Pfad $p = \langle s, \ldots, t \rangle$ zwischen zwei Punkten $s,\ t \in V$.
Eine bidirektionale Suche löst dieses Problem wie folgt: Es werden zwei unidirektionale Suchen mit Dijkstra's Algorithmus gestartet. Die *Vorwärtssuche* beginnt bei Knoten s und operiert auf dem normalen Graphen G, auch *Vorwärtsgraph* genannt. Die *Rückwärtssuche* beginnt bei Knoten t und operiert auf dem *Rückwärtsgraph* $G^r = (V, E^r)$ mit Kantengewichtungsfunktion c^r. Dieser Graph entsteht aus G durch Umkehrung aller Kanten. Der Algorithmus scannt abwechselnd einen Knoten in der Vorwärtssuche und in der Rückwärtssuche, beginnend mit der Vorwärtssuche.
Wird während des Scans von Knoten u Kante $(u,\ v)$ relaxiert, so wird überprüft, ob die Distanz $d_{\text{forward}}[v] + d_{\text{backward}}[v]$ kleiner ist als die momentan minimale gefundene Distanz von s nach t und diese gegebenenfalls angepasst ($d_{\text{forward}}[v]$ gibt die bisher kürzeste gefundene Distanz von s nach v in der Vorwärtssuche und $d_{\text{backward}}[v]$ die bisher kürzeste gefundene Distanz von v nach t in der Rückwärtssuche an).
Sobald ein Knoten in einer Richtung gescannt werden soll, der bereits in der anderen Richtung gescannt worden ist, kann die Suche beendet werden (*Abbruchbedingung*). Die aktuelle minimale gefundene Distanz ist dann die tatsächliche minimale Distanz zwischen s und t.

a) Zeichnen Sie den Rückwärtsgraph G^r zum angegebenen Graphen. Geben Sie die Kantengewichte $c(a, d)$, $c^r(a, d)$ sowie $c(b, e)$, $c^r(b, e)$ an.



(Kante (b, e) ist eine bidirektionale [bzw. ungerichtete] Kante)

b) Geben Sie an, in welcher Reihenfolge der unten angegebene Graph durchlaufen wird.

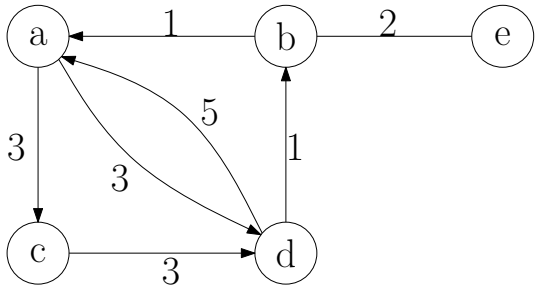


c) Zeigen Sie, dass die Abbruchbedingung korrekt ist.

d) Wann kann es passieren, dass die Suche nach dem Scan von Knoten u beendet wird, dieser aber nicht Teil des kürzesten Weges ist. Geben Sie ein Beispiel an.

Musterlösung:

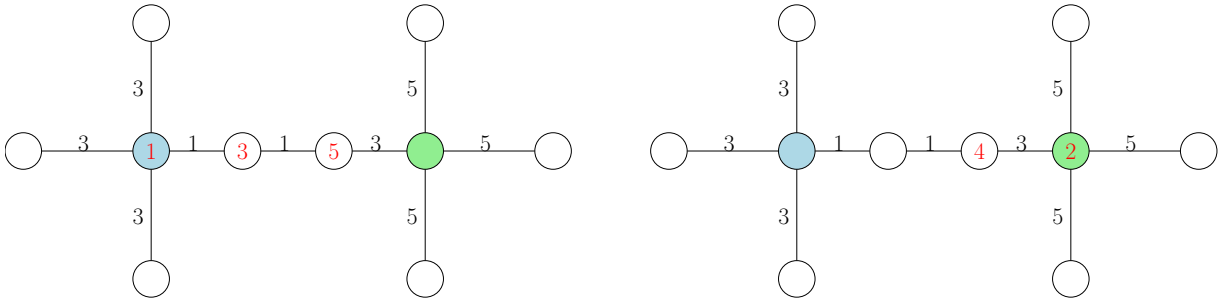
a) Rückwärtsgraph G^r:



Es sind einfach alle Pfeile umgedreht worden.

Kantengewichte:
$c(a,d) = 5$, $c^r(a,d) = 3$
$c(b,e) = 2$, $c^r(b,e) = 2$
Allgemein gilt $c(u,v) = c^r(v,u)$.

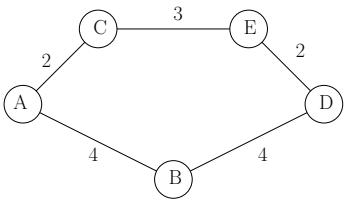
b) Vorwärtssuche: Rückwärtssuche:



Die Zahlen in den Knoten geben die Reihenfolge an, in der Sie gescannt worden sind. Sobald die Vorwärtssuche den Knoten mit Nummer 5 scannt, ist die Suche beendet, da er schon in Rückwärtsrichtung gescannt wurde.

c) Nehmen wir an, es existiere ein Knoten u, der in beiden *Queues* gelöscht wurde, aber $d(s,t) < d(s,u)+d(u,t)$ sei noch nicht bekannt. Da die Knoten in streng monotoner Reihenfolge gescannt werden, sind in der Vorwärtssuche bereits alle Knoten v mit $d(s,v) < d(s,u)$ gescannt worden. Gleiches gilt für Knoten v mit $d(v,t) < d(u,t)$ in der Rückwärtssuche. Betrachten wir den kürzesten Pfad $p = \{s = n_1, \ldots, n_k = t\}$. Weiterhin betrachten wir den Knoten mit maximalem i, so dass $d(s,n_i) < d(s,u)$ sowie den Knoten mit minimalem j, so dass $d(n_j,t) < d(u,t)$. Da $d(s,t)$ noch nicht bekannt ist, muss gelten: $i < j-2$ (sonst wäre die Distanz bekannt). Folglich existiert aber ein Knoten n_x in p mit $d(s,n_x) \geq d(s,u)$ sowie $d(n_x,t) \geq d(u,t)$. Damit wäre aber auch $d(s,t) \geq d(s,u) + d(u,t) > d(s,t)$, was ein Widerspruch ist.

d) Der abgebildete Graph ist ein mögliches Beispiel.



Die Vorwärtssuche bearbeitet die Knoten in der Reihenfolge A,C,B,E,D. Die Rückwärtssuche bearbeitet die Knoten in der Reihenfolge D,E,B,C,A. Nach drei abwechselnden Schritten wurde B folglich in beiden Suchräumen gescannt. Der kürzeste Weg folgt aber der Route A,C,E,D.

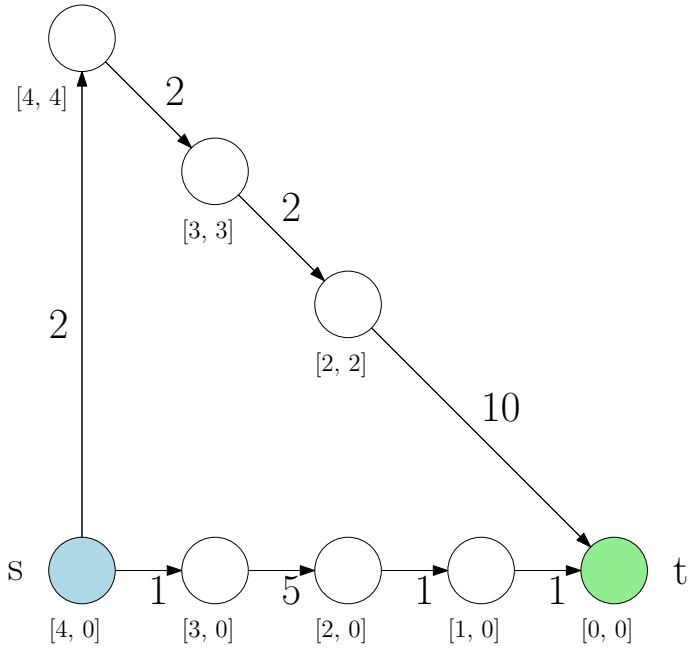
Aufgabe 4 (*Rechnen: A* Suche*)

Gegeben sei der unten abgebildete Graph. An den Kanten sind Kosten für die Nutzung der Verbindung eingetragen und die Knoten tragen Ortskoordinaten.

a) Ergänzen Sie den gegebenen Graphen um Knotenpotentiale für eine A* Suche von s nach t. Verwenden Sie Knoten t als Landmarke und die Manhatten-Distanz ($\hat{=}$ Einsnorm $||\cdot||_1$) als Abschätzung für die Entfernung zum Ziel.
Hinweis: $||\cdot||_1 : ||(x_1, y_1), (x_2, y_2)||_1 = y_2 - y_1 + x_2 - x_1$.

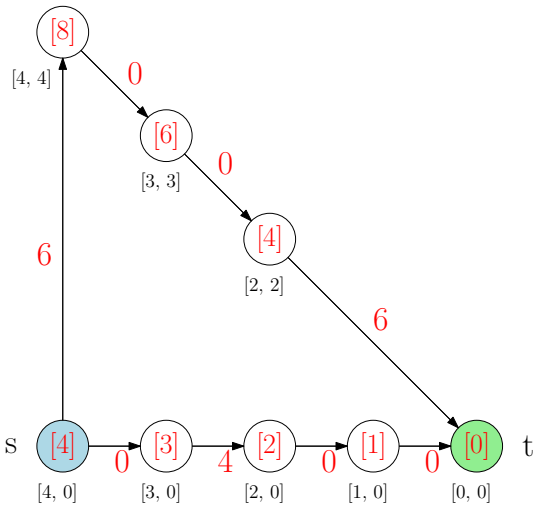
b) Tragen Sie die reduzierten Kantengewichte in den Graphen ein.

c) Wieviele `deleteMin` Operationen führt die A* Suche auf dem Graphen aus? Wieviele eine normale Suche mit Dijkstra's Algorithmus?



Musterlösung:

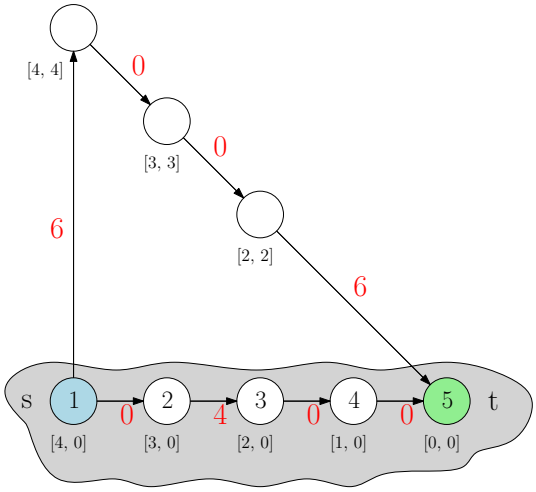
a) Knotenpotentiale pot(·) in Knoten eingetragen; Kantengewichte $c(\cdot)$ durch reduzierte Gewichte $\bar{c}(\cdot) : \bar{c}(u, v) = c(u, v) + \text{pot}(v) - \text{pot}(u)$ ersetzt:



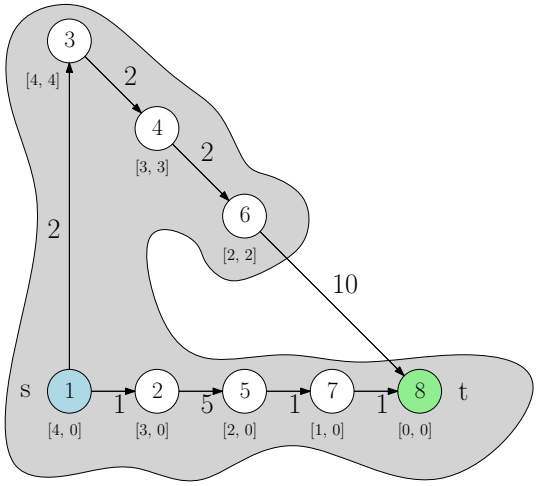
b) Siehe vorherige Teilaufgabe.

c) Die A* Suche benötigt 5 `deleteMin` Operationen, die normale Suche hingegen 8. Die entsprechenden Suchräume sind in den folgenden Abbildungen eingezeichnet. Die Knotennummerierung gibt die Reihenfolge der `deleteMin` Operationen an.

A*:



Dijkstra:



Aufgabe 5 (*Kleinaufgaben: A* Suche*)

a) Sei pot$(\cdot)$ eine gültige Potentialfunktion für die A* Suche nach Knoten t in Graph $G(V,\ E)$. Überprüfen Sie, ob

$$\text{pot}^c = \text{pot} + c,\ c = const.$$

ebenfalls eine gültige Potentialfunktion darstellt.

b) Kann es vorkommen, dass eine A* Suche mehr Knoten absucht als eine Suche mit Dijkstra's Algorithmus für die gleiche Anfrage? Begründen Sie warum nicht oder geben Sie ein Beispiel an.

Musterlösung:

a) Es ist zu überprüfen, ob

$$\begin{aligned} c(u,v) + \text{pot}^c(v) - \text{pot}^c(u) &\geq 0 \quad (1) \\ \text{pot}^c(u) &\leq \mu(u,t) \quad (2) \end{aligned}$$

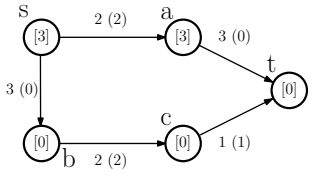
gilt.

Bedingung (1) ist immer erfüllt. Nach Einsetzen ergibt sich $c(u,v) + \text{pot}(v) - \text{pot}(u) \geq 0$. Da nach Voraussetzung pot$(\cdot)$ eine gültige Potentialfunktion ist, ist dies erfüllt.

Bedingung (2) ist hingegen nur erfüllt, wenn $c \leq \mu(u,t) - \text{pot}(u)$ f.a. $u \in V$.

Damit ist pot$^c(\cdot)$ nur für geeignete Wahl von c eine gültige Potentialfunktion.
(Bemerkung: Falls $\text{pot}(t) = 0$ f.a. Potentiale gefordert ist (anstatt nur $\text{pot}(t) \leq 0$), gilt $c = 0$!)

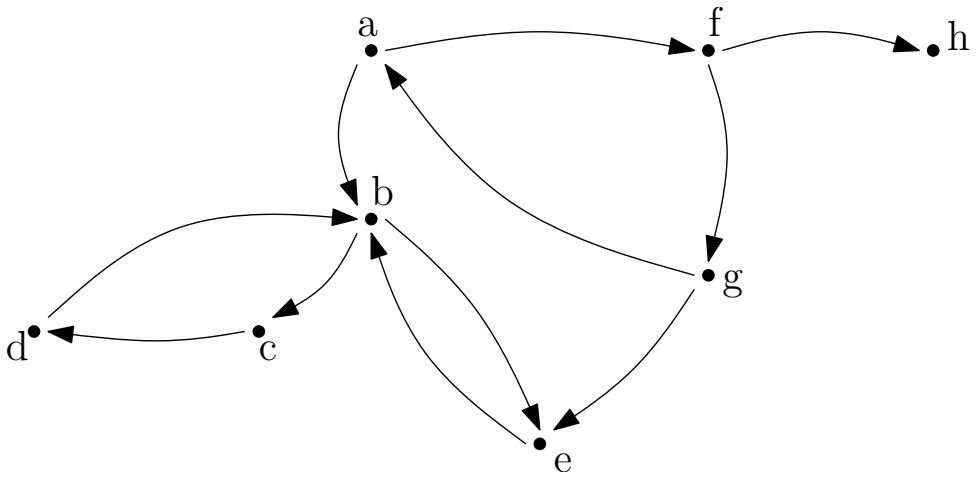
b) Im bidirektionalen Fall kann dies durchaus einfach vorkommen. Im unidirektionalen Fall hängt es von der Reihenfolge der betrachteten Knoten gleicher Distanz ab. Nehmen wir eine FIFO Ordnung der Knoten gleichen Gewichtes an (z.B. in einer Bucket Queue), so ist folgender Graph ein Beispiel. Die Dijkstra Suche scannt die Knoten in der Reihenfolge: s, a, b, t, während A* die Knoten in der Reihenfolge s, b, a, c, t scannt.



Legende: Werte in eckigen Klammern geben Knotenpotentiale an, Werte in runden Klammern reduzierte Kantengewichte.

Aufgabe 6 (*Rechnen: SCC mit Tiefensuche*)

Gegeben sei folgender Graph $G = (V, E)$:



Führen Sie den Algorithmus zur Bestimmung aller starken Zusammenhangskomponenten aus der Vorlesung auf dem Graph G aus. Geben Sie nach jedem Schritt den Zustand von `oReps`, `oNodes` und `component` an.

Musterlösung:

Schritt 1: `root(a)`

oReps	a
oNodes	a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 2: `traverseTreeEdge(a,b)`

oReps	b a
oNodes	b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 3: `traverseTreeEdge(b,c)`

oReps	c b a
oNodes	c b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 4: `traverseTreeEdge(c,d)`

oReps	d c b a
oNodes	d c b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 5: `traverseNonTreeEdge(d,b)`

oReps	b a
oNodes	d c b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 6, 7: `backtrack(c,d)`, `backtrack(b,c)`

oReps	b a
oNodes	d c b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 8: `traverseTreeEdge(b,e)`

oReps	e b a
oNodes	e d c b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 9: `traverseNonTreeEdge(e,b)`

oReps	b a
oNodes	e d c b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

(Fortsetzung auf nächster Seite)

Musterlösung:

Schritt 10: `backTrack(b,e)`

oReps	b a
oNodes	e d c b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 11: `backTrack(a,b)`

oReps	a
oNodes	a

w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	-

Schritt 12: `traverseTreeEdge(a,f)`

oReps	f a
oNodes	f a

w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	-

Schritt 13: `traverseTreeEdge(f,h)`

oReps	h f a
oNodes	h f a

w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	-

Schritt 14: `backtrack(f,h)`

oReps	f a
oNodes	f a

w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	h

Schritt 15: `traverseTreeEdge(f,g)`

oReps	g f a
oNodes	g f a

w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	h

Schritt 16: `traverseNonTreeEdge(g,e)`

oReps	g f a
oNodes	g f a

w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	h

Schritt 17: `traverseNonTreeEdge(g,a)`

oReps	a
oNodes	g f a

w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	h

Schritt 18: `backtrack(f,g)`, `backtrack(a,f)`

oReps	a
oNodes	g f a

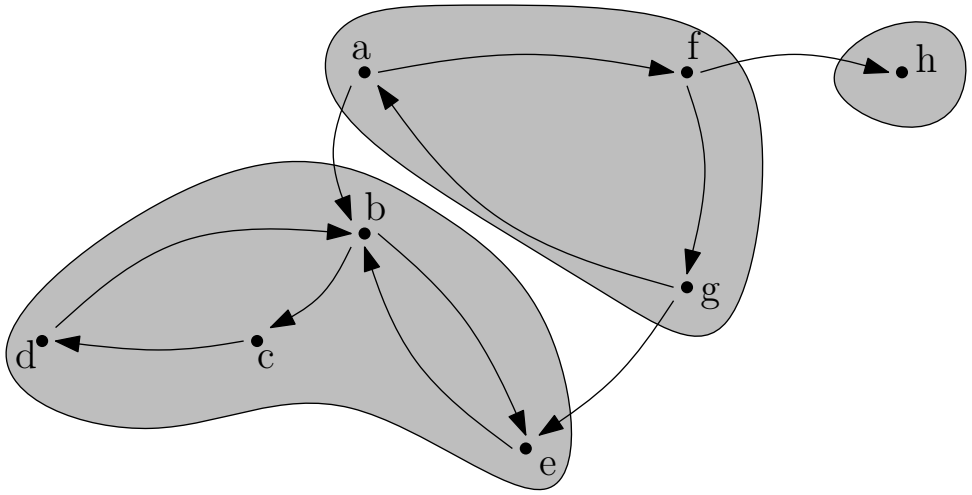
w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	h

Schritt 19: `backtrack(a,a)`

oReps	
oNodes	

w	a	b	c	d	e	f	g	h
component[w]	a	b	b	b	b	a	a	h

Die starken Zusammenhangskomponenten sind also wie folgt:



Aufgabe 7 (*Analyse+Entwurf: Artikulationspunkte*)

Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph. Ein Knoten v des Graphen G wird als *Gelenkpunkt* bezeichnet, wenn dessen Entfernen die Zahl der Zusammenhangskomponenten erhöht.

a) Zeigen Sie, dass es in jedem Graphen G *ohne Gelenkpunkte* und mit $|V| \geq 3$ immer mindestens ein Knotenpaar (i, j), $i, j \in V$, $i \neq j$ gibt, so dass zwei Pfade $P_1 = \langle i, \ldots, j \rangle$ und $P_2 = \langle i, \ldots, j \rangle$ existieren, die bis auf die Endpunkte knotendisjunkt sind, d.h.: $P_1 \cap P_2 = \{i, j\}$.

b) Beweisen Sie in jedem Graphen G *mit Gelenkpunkten* die Existenz eines Knotens v, für den gilt: Man kann einen Knoten w entfernen, so dass es von v aus keine Pfade mehr zu mindestens der Hälfte der verbleibenden Knoten gibt.

c) Zeigen Sie, dass in einem zusammenhängenden Graphen $G = (V, E)$ stets ein Knoten v existiert, so dass G nach Entfernen von v weiterhin zusammenhängend ist.

d) Vervollständigen Sie den angegebenen allgemeinen DFS-Algorithmus, so dass er in $O(|V| + |E|)$ alle Gelenkpunkte eines ungerichteten Graphen berechnet. Geben Sie an, was die Funktionen `init`, `root(s)`, `traverseTreeEdge(v,w)`, `traverseNonTreeEdge(v,w)` und `backTrack(u,v)` machen.
Überlegen Sie sich zunächst, wie Sie mit Hilfe der DFS-Nummerierung Gelenkpunkte erkennen können.

```
Depth-first search of graph G = (V, E)
unmark all nodes
init
for all s ∈ V do
    if s is not marked then
        mark s
        root(s)
        DFS(s,s)
    end if
end for

procedure DFS(u,v : NodeID)
    for all (v, w) ∈ E do
        if w is marked then
            traverseNonTreeEdge(v,w)
        else
            traverseTreeEdge(v,w)
            mark w
            DFS(v,w)
        end if
    end for
    backtrack(u,v)
end procedure
```

Musterlösung:

a) Sei v kein Gelenkpunkt. Da G ein zusammenhängendener, ungerichteter Graph mit $|V| \geq 3$ ist, existieren zwei zu v benachbarte Knoten i und j mit $i \neq j$. Ein Weg zwischen i und j geht offensichtlich über den Pfad $P_1 = \langle i, v, j \rangle$. Wird v nun entfernt, fallen die Kanten (i, v) und (v, j) weg. G bleibt zusammenhängend, sonst wäre v ein Gelenkpunkt. Dies bedeutet, dass es einen weiteren Pfad P_2 zwischen i und j geben muß und dass dieser disjunkt zu P_1 ist, da Knoten v nicht mehr vorhanden ist.

b) Sei w ein Gelenkpunkt. Dann zerfällt G nach Wegnahme von v in zwei oder mehr Komponenten. Eine Komponente K hat die minimale Anzahl von Knoten unter allen Komponenten. Da es mindestens zwei Komponenten gibt und K die kleinere ist, kann K nicht mehr als $|K| := \frac{|V \setminus \{v\}|}{2}$ Knoten besitzen. Wählt man aus K einen Knoten v, so hat dieser offensichtlich zu weniger als der Hälfte der verbleibenden Knoten einen Pfad.

c) Betrachte einen spannenden Baum des Graphen G. Jeder Blattknoten dieses Baumes kann entfernt werden ohne dass der Graph zerfällt.

d) Das Problem kann per DFS gelöst werden. Folgende Beobachtung liefert den Schlüssel zur Lösung: Ein Knoten v ist immer dann ein Gelenkpunkt, wenn er keinen anderen Knoten erreichen kann, der eine niedrigere DFS-Nummer besitzt. Um dies festzustellen, müssen im DFS-Algorithmus die minimal erreichbaren DFS-Nummern aller Unterbäume nach oben propagiert werden. Der Startknoten der DFS ist allerdings ein Sonderfall und nur Gelenkpunkt, wenn er mindestens zwei Kanten im DFS-Baum besitzt.

```
init:                          dfsPos= 1; finishingTime= 1; rootTreeEdgeCount= 0

root(s):                       dfsNum[s]=dfsPos++; minimum[s] = dfsNum[s]; tree_root = s

traverseTreeEdge(v,w):         dfsNum[w]:=dfsPos++; minimum[w] = dfsnum[w]
                               if( v == tree_root )
                                  rootTreeEdgeCount++

traverseNonTreeEdge(v,w):      minimum[v] = min( dfsNum[w], minimum[v] )
backtrack(u,v):                minimum[u] = min( minimum[u], minimum[v] )
                               if( minimum[v] ≥ dfsNum[u] )
                                  if ( tree_root ≠ u )
                                    output(u)
                                  if ( tree_root == u && rootTreeEdgeCount == 2)
                                    output(u)
```