Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Moritz Laupichler
Hans-Peter Lehmann

2. Übungsblatt zu Algorithmen II im WS 2022/2023

http://algo2.iti.kit.edu/AlgorithmenII_WS22.php
{sanders, moritz.laupichler, hans-peter.lehmann}@kit.edu

Musterlösungen

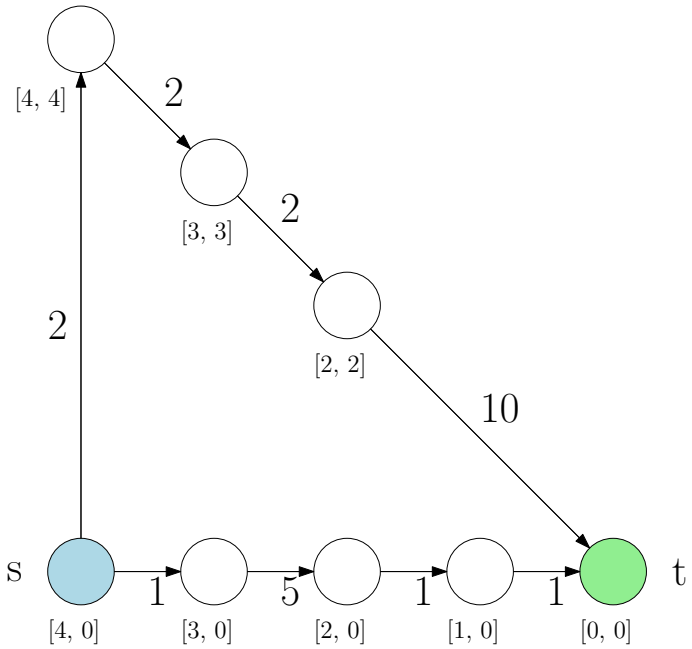
Aufgabe 1 (*Rechnen: A* Suche*)

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: $||\cdot||_1 : ||(x_1, y_1), (x_2, y_2)||_1 = y_2 - y_1 + x_2 - x_1$.

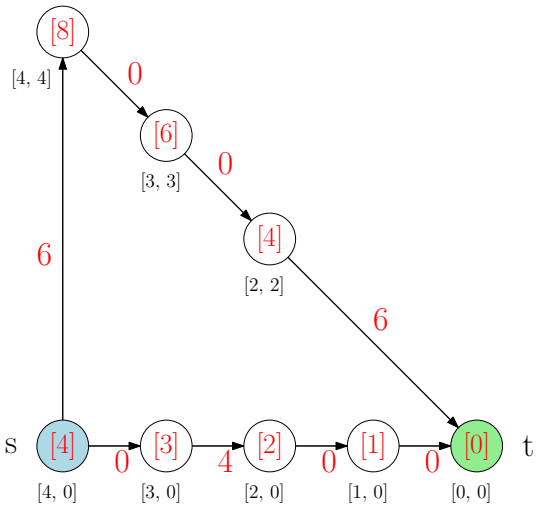
b) Tragen Sie die reduzierten Kantengewichte in den Graphen ein.

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



Musterlösung:

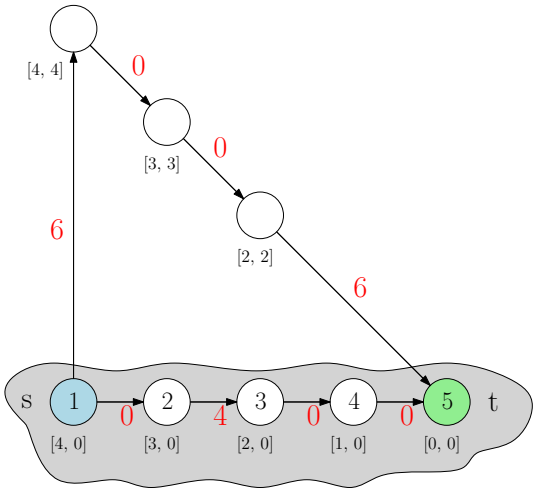
a) Knotenpotentiale pot(·) in Knoten eingetragen; Kantengewichte $c(\cdot)$ durch reduzierte Gewichte $\bar{c}(\cdot) : \bar{c}(u,v) = c(u,v) + \text{pot}(v) - \text{pot}(u)$ ersetzt:



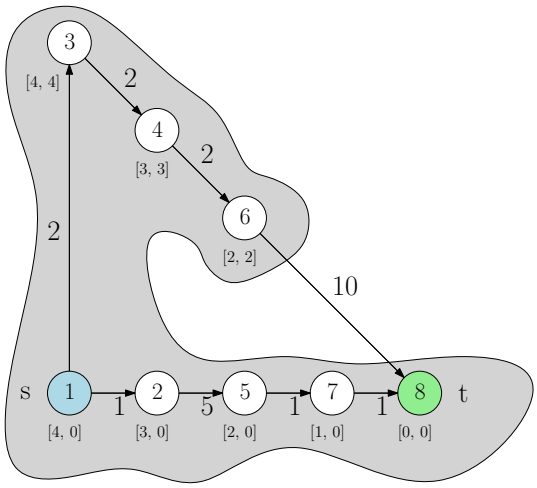
b) Siehe vorherige Teilaufgabe.

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*:



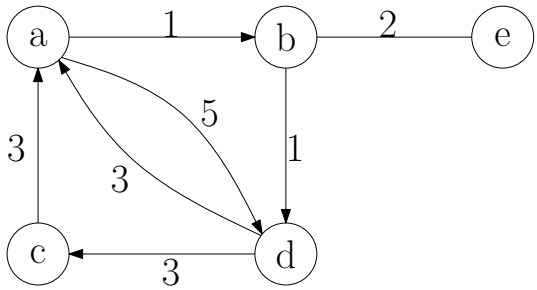
Dijkstra:



Aufgabe 2 (*Einführung+Analyse: Bidirektionaler Dijkstra*)

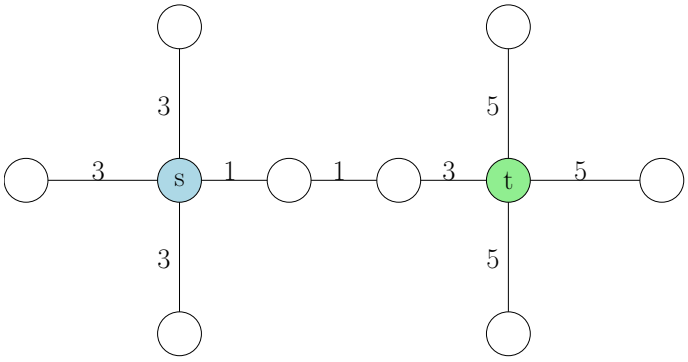
Gegeben sei – wie üblich – ein gerichteter Graph $G = (V,\ E)$ mit $|V| = n$ und $|E| = m$, sowie eine Kantengewichtungsfunktion $c : E \to \mathbb{R}_0^+$. Gesucht ist der kürzeste Pfad $p = \langle s, \ldots, t \rangle$ zwischen zwei Punkten $s,\ t \in V$.
Eine bidirektionale Suche löst dieses Problem 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.

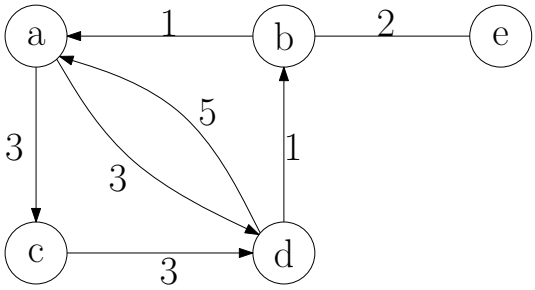


c) Zeigen Sie, dass die Abbruchbedingung korrekt ist.

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:

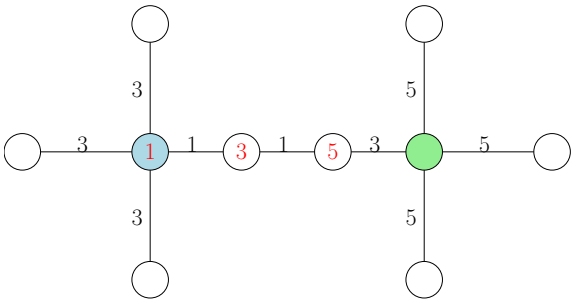
a) Rückwärtsgraph G^r:



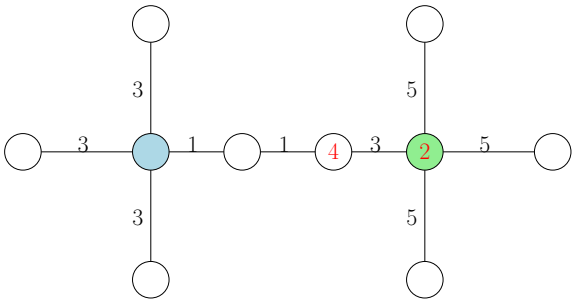
Es sind einfach alle Pfeile umgedreht worden.

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:



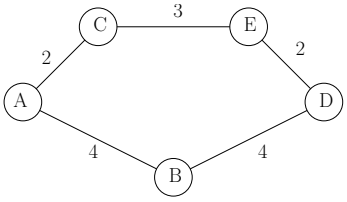
Rückwärtssuche:



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.

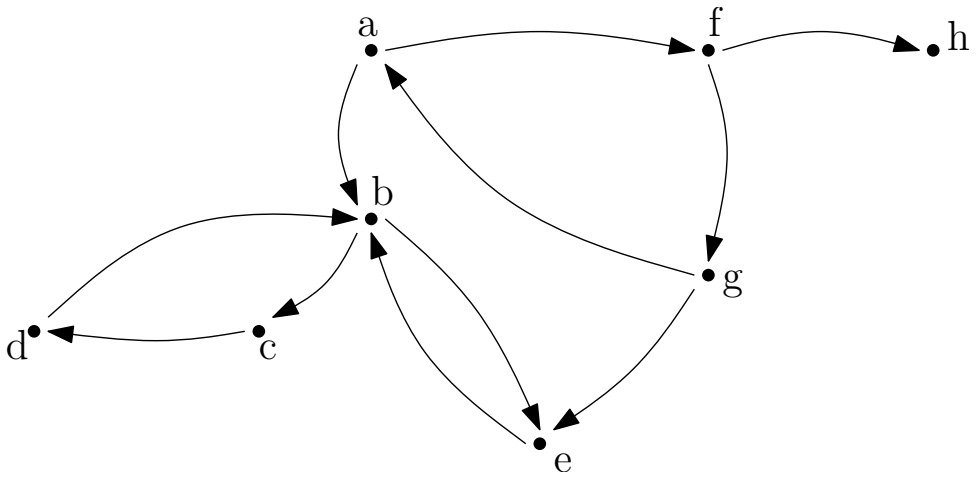
d) Der abgebildete Graph ist ein mögliches Beispiel.



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 3 (*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

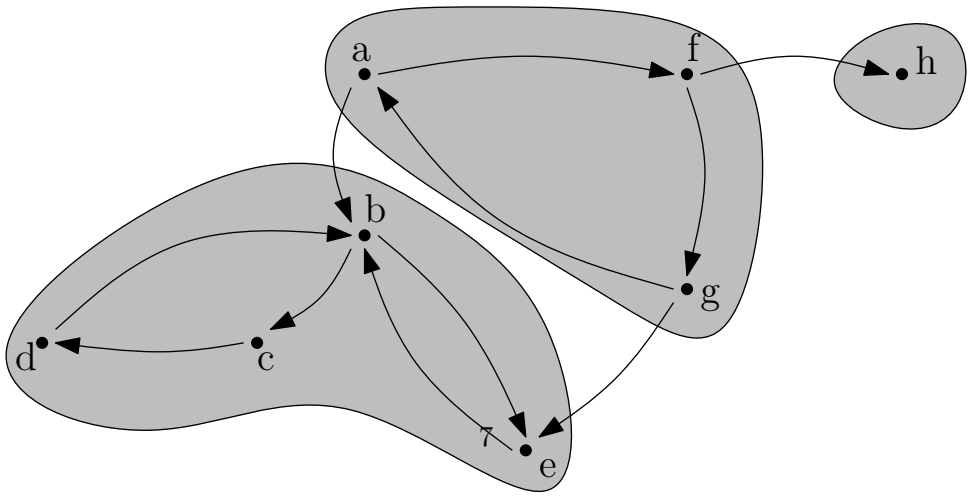
w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	h

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 4 (*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)
```

Aufgabe 5 (*Kleinaufgaben: Eigenschaften von Flüssen*)

a) Nach Vorlesung ist eine gültige Distanzfunktion $d(\cdot)$ für *Dinitz Algorithmus* gegeben durch:

- $d(t) = 0$
- $d(u) \leq d(v) + 1 \quad \forall\ (u, v) \in G_f$

Zeigen Sie, falls $d(s) \geq n$, existiert kein *augmentierender Pfad*.

b) In der Vorlesung wurde gezeigt, dass die Laufzeit von *Dinitz Algorithmus* für Graphen mit Kantengewichten gleich 1 (*unit edgeweights*) in $O((n+m)\sqrt{m})$ liegt. Vergleichen Sie diese Laufzeit zum *Ford Fulkerson Algorithmus*. Für welche Graphen mit *unit edgeweights* ist welcher der beiden Algorithmen schneller?

c) Sei $G = (V, E)$ ein gerichteter Graph, in dem maximale Flüsse berechnet werden sollen. Sei $e = (i, j) \in E$ ebenso wie $e' = (j, i) \in E$, d. h. G besitzt ein Paar entgegengesetzter Kanten. Außerdem sei $c(e) \geq c(e')$. Widerlegen Sie durch ein Gegenbeispiel:

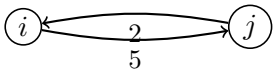
Entfernt man e' aus E und reduziert $c(e) := c(e) - c(e')$, ändert sich der maximale Fluss nicht, d. h. man kann entgegengesetzte Kanten a-priori (für beliebige s und t) gegeneinander aufrechnen.

Musterlösung:

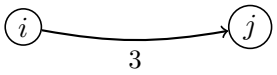
a) Ein Pfad in einem Graphen kann höchstens n unterschiedliche Knoten haben. Betrachte nun einen beliebigen augmentierenden Pfad in G_f. Für jede Kante auf diesem Weg wächst die Distanz für s höchstens um eins. Folglich kann ein augmentierender Pfad nur $d(s) \leq n - 1$ bedeuten.

b) Auf Graphen mit Kantengewichten gleich 1 ist die Laufzeit des *Ford Fulkerson Algorithmus* in $O(nm)$ (da $U = 1$!). *Dinics Algorithmus* ist damit schneller als der *Ford Fulkerson Algorithmus* falls $O((n+m)\sqrt{m}) < O(nm)$. Ausgerechnet ergibt sich $n > O(\frac{m}{\sqrt{m}-1}) = O(\sqrt{m})$.

c) Rechnet man die Kanten in folgendem Graph gegeneinander auf,



so ergibt sich



Für $s := j$ und $t := i$ ist kein Fluss mehr möglich, während im Originalgraphen ein maximaler Fluss von 2 möglich war. Dies ist somit ein Gegenbeispiel zur Behauptung.

Aufgabe 6 (*Rechnen: Segmentierung mit Flüßen*)

Wir betrachten einen einfachen Fall für Bildbearbeitung. Die Vorder-/Hintergrundsegmentierung. Das Ziel dieses Prozesses ist es, ein Bild in Vorder und Hintergrund zu zerlegen. Die Transformation dafür weißt jedem Pixel $p_{i,j}$ des Bildes einen Knoten $v_{i,j}$ im Graphen zu. Für jedes Paar von benachbarten Pixeln $p_{i,j}$ und $p_{k,l}$ ($|i-k|+|j-l|=1$) fügen wir eine ungerichtete Kante $(v_{i,j}, v_{k,l})$ ein. Zusätzlich fügen wir je einen Knoten s für Vordergrund (Quelle) und einen Knoten t für Hintergrund (Senke) ein. Von Knoten s existiert eine gerichtete Kante zu jedem Knoten $p_{i,j}$ und von jedem Knoten $p_{i,j}$ existiert eine gerichtete Kante zu Knoten t. Wir definieren darüber hinaus folgende Kantengewichte:

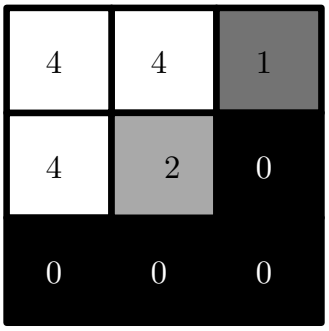
$$c(e=(u,v)) = \begin{cases} p_v(v) & u = s \\ p_h(u) & v = t \\ f(u,v) & sonst \end{cases}$$

Wobei mit $p_v(x)$ die Wahrscheinlichkeit gegeben ist, dass x Vordergrundknoten ist, mit $p_h(x)$ die Wahrscheinlichkeit für einen Hintergrundknoten und mit $f(x,y)$ eine Penaltyfunktion für das Trennen der beiden Knoten x und y. Für ein Graustufenbild B definieren wir
$p_v(x,y) = B[x,y]^2$, $p_h(x,y) = (4 - B[x,y])^2$ sowie $f((x_1,y_1),(x_2,y_2)) = (4 - |B[x_1,y_1] - B[x_2,y_2]|)^2$.
Hinweis: Diese Modellierung ist nur ein Beispiel und keine allgemeingültige Modellierung. Sie soll nur verdeutlichen, wie Flow-Algorithmen für andere Probleme eingesetzt werden können.

a) Geben Sie den Flussgraphen für das unten angegebene Graustufenbild an.

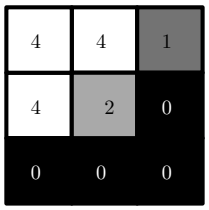
b) Führen Sie einen augmenting Path Algorithmus auf dem entstandenen Graphen aus.

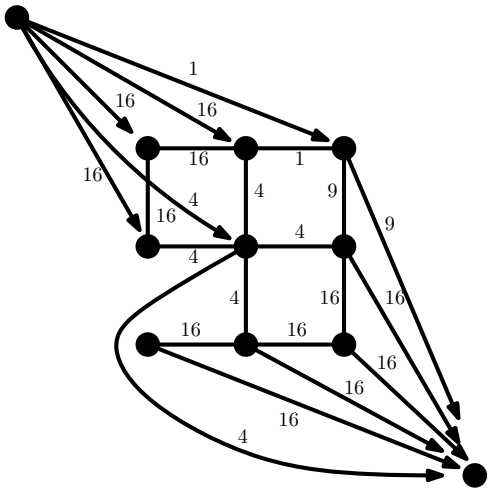
c) Wie würde die Segmentierung in Vorder- und Hintergrund im Bild als Ergebnis aussehen?



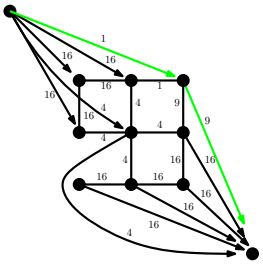
Musterlösung:

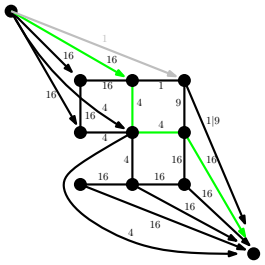
a) Als Transformation ergibt sich folgender Graph. Kanten ohne Kapazität wurden weggelassen.

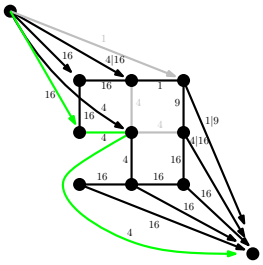


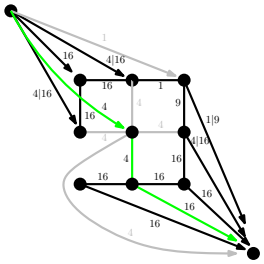


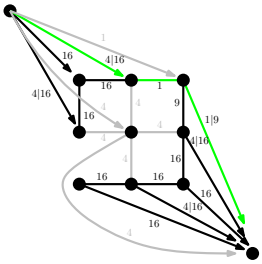
b) Der Algorithmus wird skizziert durch folgende Schritte:

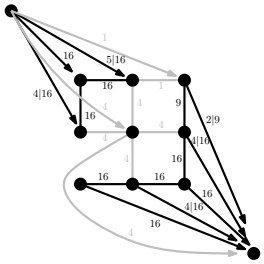


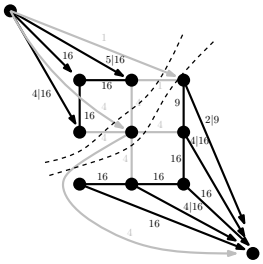




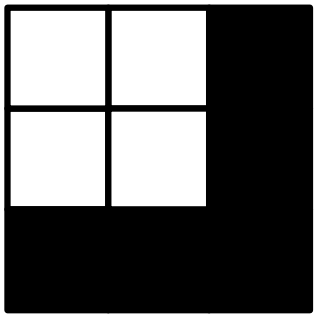


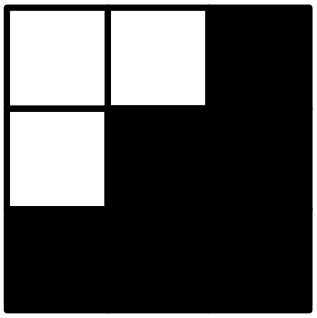






c) Die beiden gleichwertigen Lösungen nach unserer Modellierung sind:





Aufgabe 7 (*Analyse+Entwurf+Rechnen: Grenzüberwachung*)

Eine (eindimensionale) Grenzlinie soll durch ein Sensornetz überwacht werden. Zu diesem Zweck wurde eine große Anzahl an Sensorknoten unregelmäßig an der Grenze ausgebracht. Jeder Knoten kann einen Bereich der Grenze für eine gewisse Zeit proportional zu seiner Batteriekapazität überwachen. Die Grenze gilt als vollständig gesichert, wenn jeder Abschnitt der Grenzlinie von mindestens einem Sensorknoten abgedeckt ist. Aufgrund der großen Menge an Knoten sind ihre Überwachungsbereiche stark überlappend. Daher müssen nicht immer alle Knoten aktiv sein, um eine vollständige Sicherung der Grenze zu gewährleisten. So kann Energie gespart werden und die maximale Dauer der Grenzsicherung erhöht werden.
Durch die unregelmäßige Anbringung der Knoten und durch große Fertigungstoleranzen in der Batteriekapazität und dem Überwachungsbereich (*man hat unbedingt beim billigsten Hersteller einkaufen müssen...*) ist zunächst nicht klar, wie lange die Grenze maximal vollständig gesichert werden kann. Glücklicherweise wurden die Positionen der Knoten und ihre jeweiligen Kapazitäten und Detektionsbereiche protokolliert und können verwendet werden, um diese Frage zu beantworten.

a) In der Vorlesung haben Sie Flussprobleme mit beschränkten Kantenkapazitäten $c(e)$ kennengelernt. Ebenso können Flussprobleme mit beschränkten Knotenkapazitäten $c(v)$ sinnvoll sein. In diesem Fall darf für einen gültigen Fluss die Summe der in den Knoten ankommenden bzw. ausgehenden Flüsse die Kapazität des Knotens nicht überschreiten. Außerdem muss wie bisher für jeden Knoten (außer der Quelle und Senke) die Summe der ankommenden Flüsse gleich der Summe der ausgehenden Flüsse sein.

   Erklären Sie, wie maximale Flüsse mit Knotenkapazitäten berechnet werden können. Begründen Sie kurz, warum Ihr Ansatz einen zulässigen und optimalen Fluss berechnet.

b) Konstruieren Sie ein Flussnetzwerk, das das oben beschriebene Problem der Bestimmung einer maximalen Dauer für die vollständige Grenzüberwachung lösen kann.

   Hinweis: Jeder Knoten entspricht einem Sensorknoten. Batteriekapazität kann als äquivalent zur Flussmenge betrachtet werden.

c) Erstellen Sie ein Flussnetz, das dem folgenden Sensornetz entspricht. Wie lange kann dieses Netz die Grenze im Bereich $[0, 13]$ überwachen? Welche Sensorknoten müssen wann aktiv sein?

   Format der Angaben: $x_{nodeID} = \{[begin\_range, end\_range], capacity\}$

   $x_1 = \{[0, 5], 4\}$
   $x_2 = \{[0, 7], 3\}$
   $x_3 = \{[4, 9], 2\}$
   $x_4 = \{[3, 8], 5\}$
   $x_5 = \{[8, 13], 5\}$
   $x_6 = \{[7, 11], 3\}$
   $x_7 = \{[11, 15], 2\}$

   Hinweis: Bevor Sie langwierig einen maximalen Fluss berechnen, versuchen Sie ihn durch *scharfes Hinschauen* zu bestimmen.

Musterlösung:

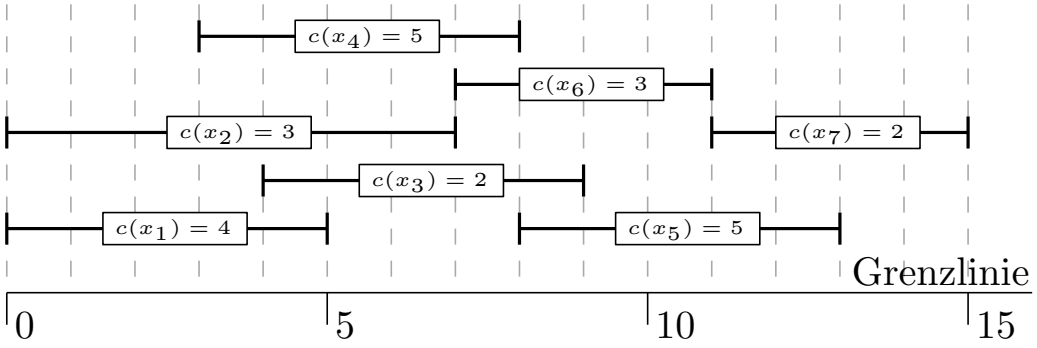
a) Man definiert einen neuen Flussgraph $G' = (V', E')$. Für jeden Knoten $v \in V$ fügt man zwei Knoten v_{in} und v_{out} sowie eine Kante (v_{in}, v_{out}) mit Kapazität $c(v_{in}, v_{out}) = c(v)$ in G' ein. Für jede Kante $(u, v) \in E$ fügt man eine neue Kante (u_{out}, v_{in}) in E' ein. Die Kapazität der Kante wird übernommen (bzw. auf ∞ gesetzt falls sie keine Kapazität hatte).

Nun berechnet man auf G' einen Fluss von s_{in} nach t_{out} und transferiert die Flusswerte zurück auf die Kanten in G. Der Fluss respektiert die Knotenkapazitäten, da sie in G' durch die Kanten (v_{in}, v_{out}) passend beschränkt wurden. Außerdem ist der Fluss optimal. Angenommen es gäbe noch einen augmentierenden Pfad in G, dann gäbe es auch einen in G' und der berechnete Fluss wäre kein maximaler Fluss in G': Die Restkapazitäten der ursprünglichen Kanten sind per Konstruktion gleich zu ihren Entsprechungen in G. Eine zu einem nicht voll ausgelasteten Knoten gehörende Kante hätte ebenfalls noch Restkapazität.

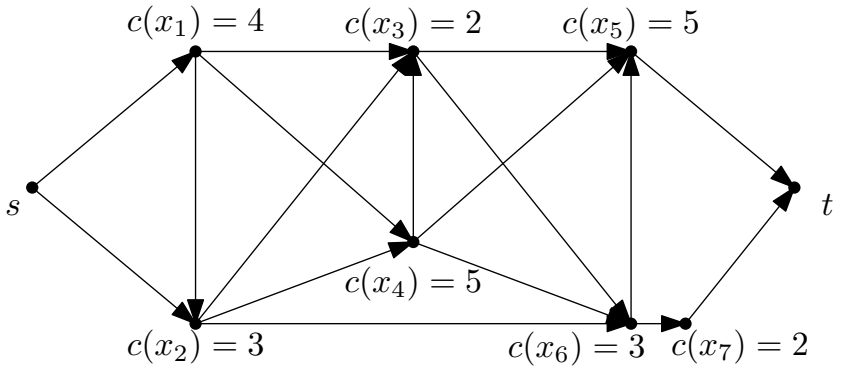
b) Das Flussnetz kann mit Hilfe von Knotenkapazitäten–wie in der letzten Teilaufgabe besprochen–konstruiert werden. Für jeden Sensorknoten x_i fügt man eine Knoten i mit Kapazität $c(i)$ gleich der Batteriekapazität des Sensorknotens ein. Außerdem fügt man eine Quelle s und eine Senke t ein. Anschließend fügt man Kanten (i, j) ein, wenn $x_i.end\_range \in [x_j.start\_range, x_j.end\_range]$ (für s und t entsprechen die *range* Werte dem Anfang und dem Ende des Grenzverlaufs). Alle Kanten sind ohne Kapazität.

Jeder Flusspfad durch das Netz entspricht einer Konfiguration von aktiven Sensorknoten, die den gesamten Grenzverlauf überwachen können und die Flussmenge der Überwachungsdauer für diese Konfiguration. Der gesamte Fluss entspricht der maximalen Überwachungsdauer.

c) Eingezeichnete Überwachungsbereiche und Batteriekapazitäten der Sensorknoten:



Sich ergebendes Flussnetzwerk:



Musterlösung:

c) *(Fortsetzung)*
Durch geschicktes Hinschauen muss man den Flussalgorithmus nicht ausführen und kann direkt eine maximale Überwachungsdauer von 7 ablesen. Diese wird durch folgende Knotenmengen erreicht, die jeweils gleichzeitig für die angegebene Dauer aktiv sind:

aktive Knoten	Dauer
x_1, x_4, x_5	4
x_2, x_4, x_5	1
x_2, x_6, x_7	2

Jede aktive Knotenmenge deckt offensichtlich den kompletten Bereich $[0, 13]$ ab. Fast alle Knoten verbrauchen ihre komplette Energie –aber auch nicht mehr– außer Knoten x_3 (gar nicht verwendet) und x_6 (noch 1 Restkapazität). Mit den restlichen Knoten kann keine weitere vollständige Überdeckung erreicht werden.