Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Moritz Laupichler
Nikolai Maas

2. Übungsblatt zu Algorithmen II im WS 2023/2024

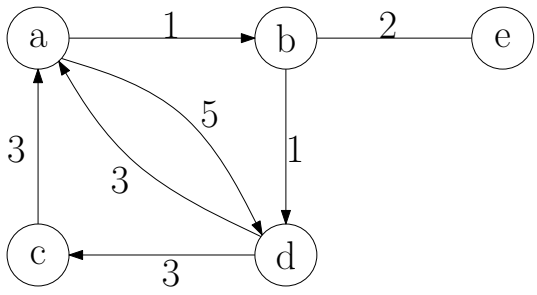
https://algo2.iti.kit.edu/AlgorithmenII_WS23.php
{sanders, moritz.laupichler, nikolai.maas}@kit.edu

Musterlösungen

Aufgabe 1 (*Einführung+Analyse: Bidirektionaler Dijkstra*)

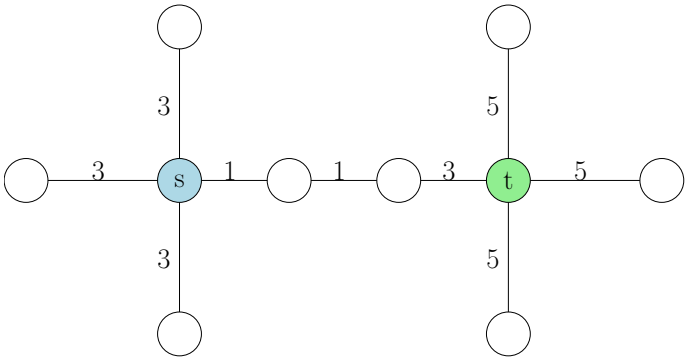
Gegeben sei – wie üblich – ein gerichteter Graph $G = (V,\ E)$ mit $|V| = n$ und $|E| = m$, sowie eine Kantengewichtsfunktion $c : E \to \mathbb{R}_0^+$. Gesucht ist der kürzeste Pfad $p = \langle s, \ldots, t\rangle$ zwischen zwei Punkten $s,\ t \in V$.
Eine bidirektionale Suche löst dieses Problem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

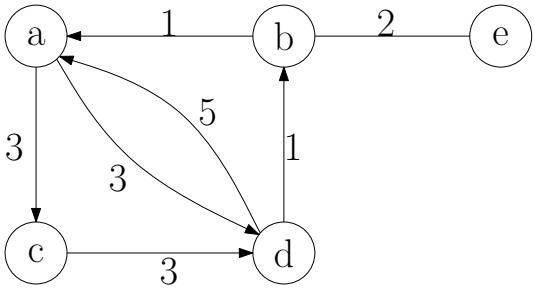


c) Zeigen Sie, dass die Abbruchbedingung korrekt ist.

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:

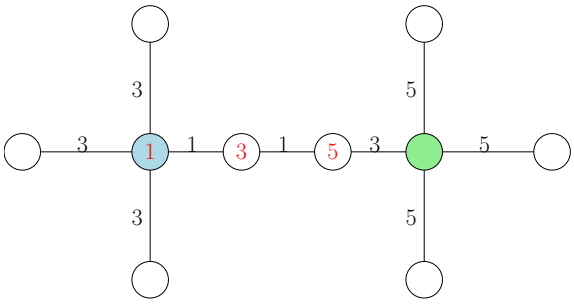
a) Rückwärtsgraph G^r:



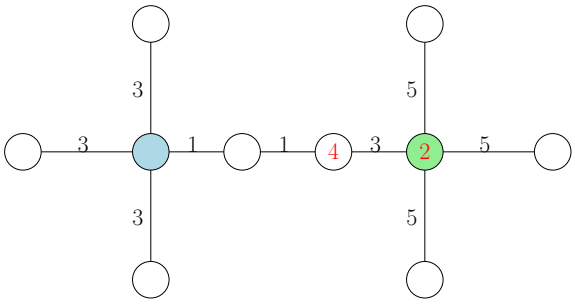
Es sind einfach alle Pfeile umgedreht worden.

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:



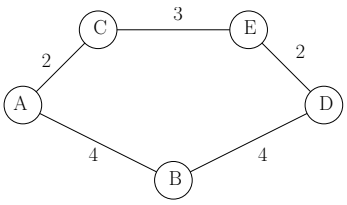
Rückwärtssuche:



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 2 (*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) Bei manchen Algorithmen kann es sinnvoll sein, unterschiedliche Potentialfunktionen zu kombinieren. Zeigen Sie in diesem Zusammenhang: Sind π_1 und π_2 gültige Potentialfunktionen, so ist auch $\pi = \frac{\pi_1+\pi_2}{2}$ eine gültige Potentialfunktion.

c) Kann es vorkommen, dass eine A*-Suche mehr Knoten absucht als eine Suche mit Dijkstra's Algorithmus für die gleiche Anfrage? Kann es auch dann vorkommen, wenn das Potential auf jedem kürzesten Pfad zu t monoton fallend ist (d.h. für jede Kante (u, v) mit $\mu(v, t) \leq \mu(u, t)$ gilt $\text{pot}(v) \leq \text{pot}(u)$)? Begründen Sie, warum nicht, oder geben Sie ein Beispiel an.

Musterlösung:

a) Es ist zu überprüfen, ob

$$\begin{aligned} c(u,v) + \text{pot}^c(v) - \text{pot}^c(u) &\geq 0 && (1)\\ \text{pot}^c(u) &\leq \mu(u,t) && (2)\\ \text{pot}^c(t) &= 0 && (3) \end{aligned}$$

gilt.

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

Bedingung (2) ist hingegen nur erfüllt, wenn $c \leq \mu(u,t) - \text{pot}(u)$ f.a. $u \in V$. Mit $\mu(t,t) = \text{pot}(t) = 0$, ist Bedingung (2) also nur erfüllt, wenn $c = 0$.

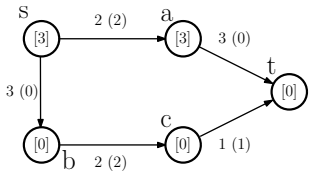
Dies folgt auch direkt aus Bedingung (3).

b) Es ist zu zeigen:

- π ist eine untere Schranke der Distanzfunktion: Es gilt: $\pi_1(u) \leq \mu(u,t)$, sowie $\pi_2(u) \leq \mu(u,t)$ für alle Knoten $u \in V$. Daraus folgt: $\pi_1(u) + \pi_2(u) \leq 2\mu(u,t)$ und damit $\pi \leq \mu(u,t)$.
- Die Kantengewichte sind nicht negativ: Für jede Kante $(u,v) \in E$ gilt: $c((u,v)) + \pi_1(v) \geq \pi_1(u)$, sowie $c((u,v)) + \pi_2(v) \geq \pi_2(u)$. Daraus folgt direkt: $2c((u,v)) + \pi_1(v) + \pi_2(v) \geq \pi_1(u) + \pi_2(u)$ und somit $c((u,v)) + \pi(v) \geq \pi(u)$.
- Es gilt $\pi(t) = 0$: Mit $\pi_1(t) = \pi_2(t) = 0$, folgt auch $\pi(t) = 0$.

c) Für allgemeine Potentialfunktionen lässt sich einfach ein Gegenbeispiel konstruieren, indem man einem Knoten negatives Potential gibt, der von Dijkstra nicht betrachtet würde. Bei nicht-negativen Potentialen hängt es von der Reihenfolge der betrachteten Knoten gleicher Distanz ab.

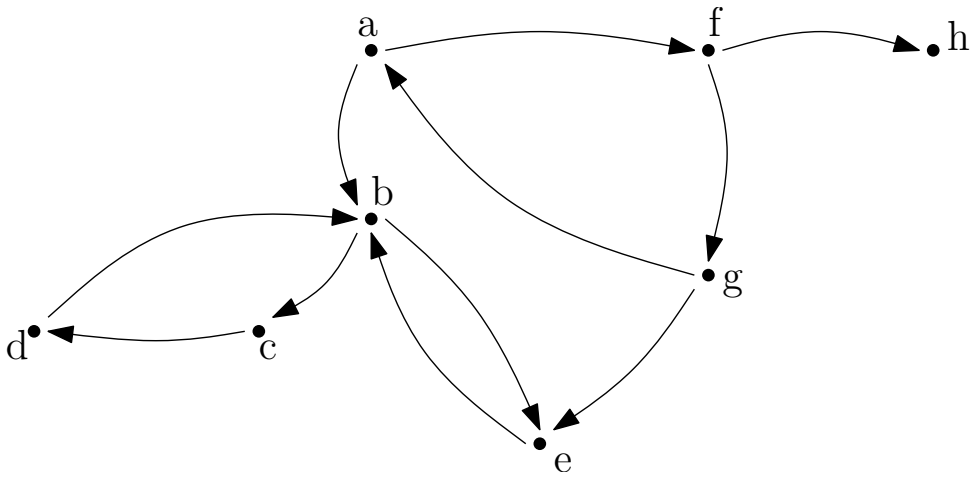
Nehmen wir eine FIFO Ordnung der Knoten gleichen Gewichtes an (z.B. in einer Bucket Queue), so ist folgender Graph ein Beispiel mit monoton fallendem Potential. Die Dijkstra Suche scannt die Knoten in der Reihenfolge: s, a, b, t, während A* die Knoten in der Reihenfolge s, b, a, c, t scannt.



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

Aufgabe 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

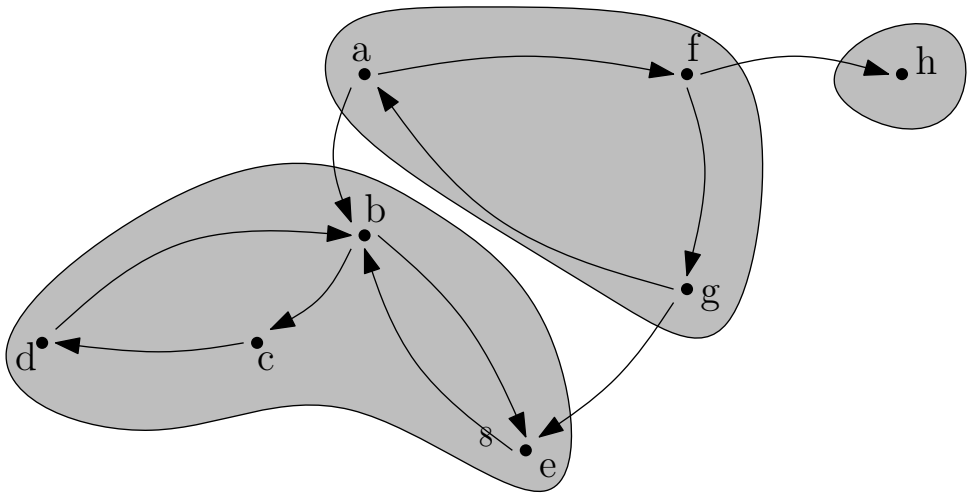
w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	h

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) Zeigen Sie, dass in jedem Graphen G *mit Gelenkpunkten* ein Knoten v existiert, 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 u ist immer dann ein Gelenkpunkt, wenn kein Kind (bzgl. des DFS-Baum) v von u einen Knoten erreichen kann, der eine niedrigere DFS-Nummer als u 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 edge weights*) in $O((n + m)\sqrt{m})$ liegt. Vergleichen Sie diese Laufzeit zum *Ford Fulkerson Algorithmus*. Für welche Graphen mit *unit edge weights* 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:

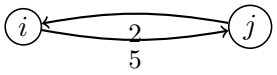
Entfernt man e' aus E und reduziert $c(e) := c(e) - c(e')$, so ändert sich der maximale Fluss nicht (d. h. man kann entgegengesetzte Kanten a-priori gegeneinander aufrechnen).

Musterlösung:

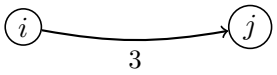
a) Ein Pfad in einem Graphen kann höchstens n unterschiedliche Knoten haben. Betrachte nun einen beliebigen augmentierenden Pfad in G_f. Für jede Kante auf diesem Weg wächst die Distanz für s höchstens um eins. Folglich kann ein augmentierender Pfad nur $d(s) \leq n - 1$ bedeuten.

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 weist 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:

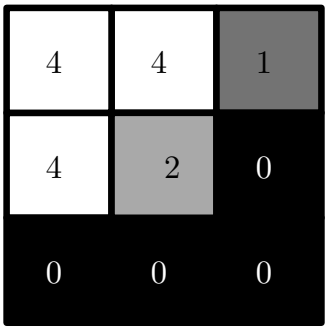
$$c(e=(u,v)) = \begin{cases} p_v(v) & u = s \\ p_h(u) & v = t \\ f(u,v) & \textit{sonst} \end{cases}$$

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.

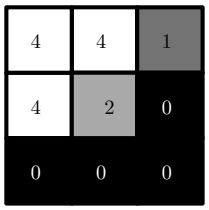
b) Führen Sie einen Augmenting-Path-Algorithmus auf dem entstandenen Graphen aus.

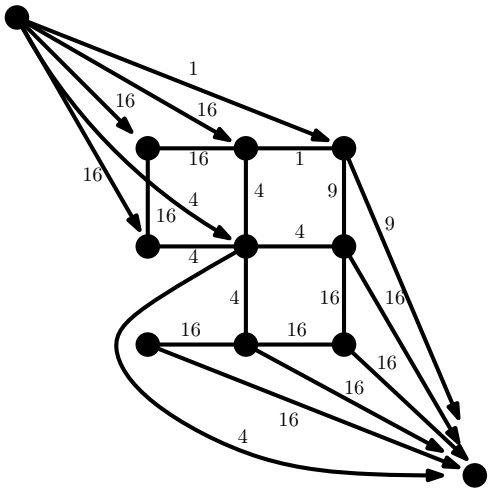
c) Wie würde die Segmentierung in Vorder- und Hintergrund im Bild als Ergebnis aussehen?



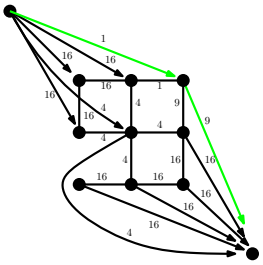
Musterlösung:

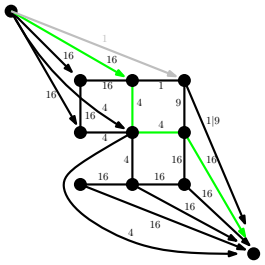
a) Als Transformation ergibt sich folgender Graph. Kanten ohne Kapazität wurden weggelassen.

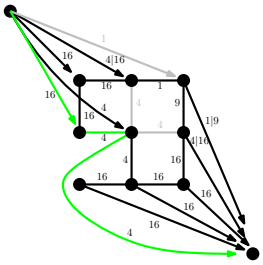


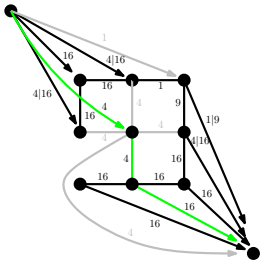


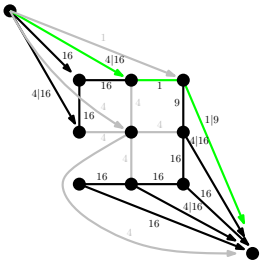
b) Der Algorithmus wird skizziert durch folgende Schritte:

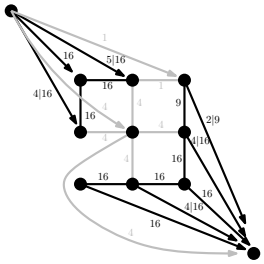


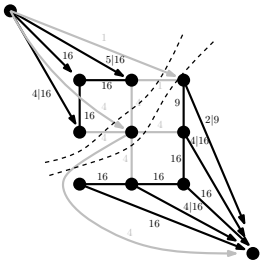




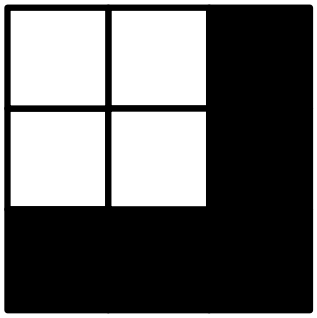


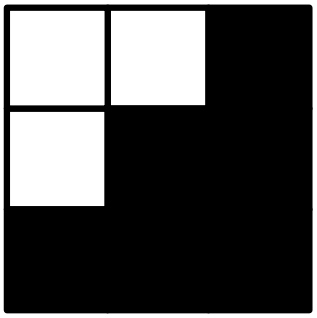






c) Die beiden gleichwertigen Lösungen nach unserer Modellierung sind:





Aufgabe 7 (*Analyse: Eigenschaften von Flüssen*)

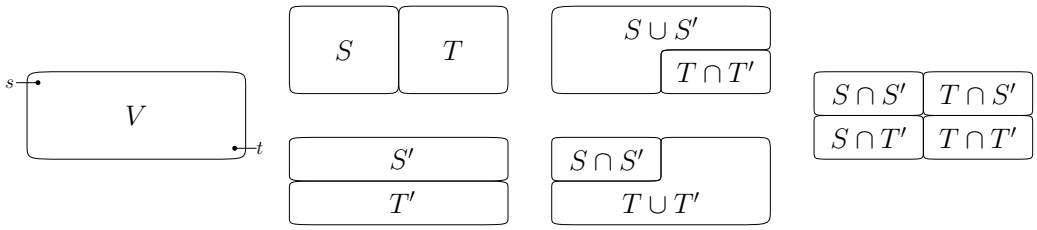
a) Seien (S, T) und (S', T') zwei minimale (s, t)-Schnitte in einem Flussgraphen G. Zeigen oder widerlegen Sie, dass $(S \cup S', T \cap T')$ und $(S \cap S', T \cup T')$ auch minimale (s, t)-Schnitte sind.

b) Sei (S, T) ein minimaler (s, t)-Schnitt in einem Flussgraphen G. Zeigen oder widerlegen Sie, dass (S, T) ein minimaler (x, y)-Schnitt ist für alle $(x, y) \in S \times T$.

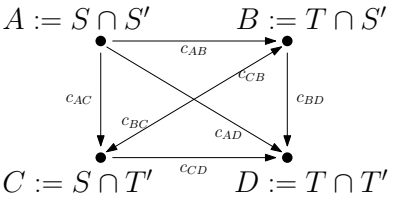
c) Wir betrachten den *Preflow-Push-Algorithmus* aus der Vorlesung mit beliebiger Wahl des nächsten Knotens. Zeigen Sie, dass es Eingaben gibt, für die immer $\Omega(n^2)$ *relabel*-Operationen benötigt werden.

Musterlösung:

a) Der Graph wird in vier Bereiche aufgeteilt, $S \cap S'$, $T \cap S'$, $T \cap T'$ und $S \cap T'$ (siehe Abbildung).



Zur Anschauung definiert man einen Graphen, dessen Knoten der obigen Aufteilung entsprechen und dessen Kanten die Schnitte zwischen den Bereichen darstellen.



Damit ergeben sich folgende Werte für die Schnitte:

- (S, T) Schnitt: $c_{ST} = c_{AB} + c_{AD} + c_{CB} + c_{CD}$
- (S', T') Schnitt: $c_{S'T'} = c_{AC} + c_{AD} + c_{BC} + c_{BD}$
- $(S \cup S', T \cap T')$ Schnitt: $c_{S \cup S', T \cap T'} = c_{AD} + c_{BD} + c_{CD}$
- $(S \cap S', T \cup T')$ Schnitt: $c_{S \cap S', T \cup T'} = c_{AB} + c_{AC} + c_{AD}$

Da (S, T) und (S', T') minimale Schnitte sind, gilt $c_{ST} = c_{S'T'}$. Außerdem sind $c_{S \cup S', T \cap T'}$ und $c_{S \cap S', T \cup T'}$ jeweils größer gleich c_{ST} bzw. $c_{S'T'}$. Löst man die sich ergebenden Ungleichungen, erhält man $c_{BC} = c_{CB} = 0$ und damit $c_{AB} = c_{BD}$, $c_{AC} = c_{CD}$ (Rechnung siehe nächste Seite).

Es ergibt sich $c_{S \cup S', T \cap T'} = c_{ST} = c_{S \cup S', T \cap T'} = c_{S'T'}$.

Musterlösung:

a) (fortgesetzt)
Auflösen der Ungleichungen:

$$
\begin{aligned}
c_{S\cup S',T\cap T'} &\geq c_{ST} && (1)\\
c_{S\cup S',T\cap T'} &\geq c_{S'T'} && (2)\\
c_{S\cap S',T\cup T'} &\geq c_{ST} && (3)\\
c_{S\cap S',T\cup T'} &\geq c_{S'T'} && (4)
\end{aligned}
$$

$$
\begin{aligned}
c_{AD}+c_{BD}+c_{CD} &\geq c_{AB}+c_{AD}+c_{CB}+c_{CD} && (1)\\
c_{AD}+c_{BD}+c_{CD} &\geq c_{AC}+c_{AD}+c_{BC}+c_{BD} && (2)\\
c_{AB}+c_{AC}+c_{AD} &\geq c_{AB}+c_{AD}+c_{CB}+c_{CD} && (3)\\
c_{AB}+c_{AC}+c_{AD} &\geq c_{AC}+c_{AD}+c_{BC}+c_{BD} && (4)
\end{aligned}
$$

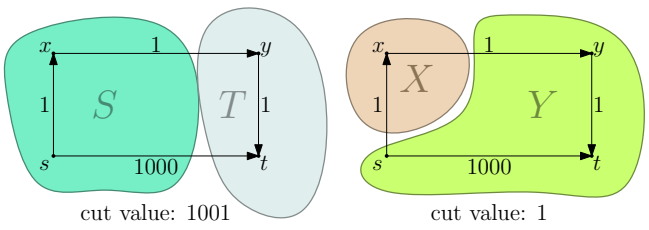
$$
\begin{aligned}
c_{BD} &\geq c_{AB}+c_{CB} && (1)\\
c_{CD} &\geq c_{AC}+c_{BC} && (2)\\
c_{AC} &\geq c_{CB}+c_{CD} && (3)\\
c_{AB} &\geq c_{BC}+c_{BD} && (4)
\end{aligned}
$$

Setze (2) in (3) ein und erhalte $c_{AC} \geq c_{CB}+c_{AC}+c_{BC} \Leftrightarrow 0 \geq c_{CB}+c_{BC}$. Da Kapazitäten nicht negativ sein können, gilt $c_{CB} = c_{BC} = 0$. Dies eingesetzt in die anderen Formeln liefert

$$
\begin{aligned}
c_{BD} &\geq c_{AB} && (1)\\
c_{CD} &\geq c_{AC} && (2)\\
c_{AC} &\geq c_{CD} && (3)\\
c_{AB} &\geq c_{BD} && (4)
\end{aligned}
$$

und damit $c_{AB} = c_{BD}$, $c_{AC} = c_{CD}$.

b) Unten abgebildetes Flussnetzwerk mit dem eingezeichneten Schnitt ist ein Gegenbeispiel. Links ist ein minimaler (s,t) Schnitt mit Wert 1001 abgebildet. Der minimale (x,y) Schnitt mit dem Wert 1 ist rechts zu sehen.



c) Wir betrachten folgenden Graphen, wobei die Anzahl Knoten ungerade sei:

Um einen Fluss auf diesem Graphen zu berechnen, muss der Fluss einmal durch den ganzen Graphen und wieder zurückfließen. Damit der zusätzliche Fluss vom vorletzten Knoten wieder in die Quelle zurückfließen kann, muss der Knoten mindestens Level $n+1$ haben. Somit ergibt sich über alle Knoten $\#relabel \geq \sum_{i=1}^{n-2} n+1 = n^2-n-2 \in \Omega(n^2)$.