Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Dr. Thomas Worsch, Dr. Simon Gog
Demian Hespe, Yaroslav Akhremstev

6. Übungsblatt zu Algorithmen II im WS 2017/2018

http://algo2.iti.kit.edu/AlgorithmenII_WS17.php
{hespe,sanders,simon.gog,worsch,yaroslav.akhremtsev}@kit.edu

Musterlösungen

Aufgabe 1 (*Analyse: Kleinaufgaben*)

a) Geben Sie die wesentlichen Unterschiede –laut Vorlesung– zwischen einer (normalen) *Priority Queue* und einer adressierbaren *Priority Queue* an.

b) Vergleichen Sie die Laufzeit einer `merge`-Operation für *Pairing Heaps* und *Binary Heaps*.

Musterlösung:

a) Im Gegensatz zur normalen *Priority Queue* erlaubt die adressierbare *Priority Queue* den direkten Zugriff auf beliebige Elemente über ein *Handle* h. Dieses wird von der `insert`-Operation als Rückgabewert geliefert. Außerdem ermöglicht sie einige zusätzliche Operationen: `remove(h)`, `decreaseKey(h,k)`. Die `merge`-Operation kann prinzipiell auch von normalen *Priority Queues* unterstützt werden, allerdings weniger effizient.

b) In einem *Pairing Heap* wird eine Menge von Bäumen gehalten. Eine `merge`-Operation ist also in konstanter Zeit möglich, indem die jeweiligen Listen vereint werden und der `minPtr` auf das Minimum beider Wälder gesetzt wird. Im *Binary Heap* funktioniert dieses Vorgehen nicht. In der weit verbreiteten Arrayimplementierung gibt es mehrere mögliche Vorgehensweisen. Wenn die Anzahl der Elemente gegeben ist durch n_1 bzw. n_2, $n_1 < n_2$, ergeben sich folgende Laufzeiten:

   (a) Einfügen der kleineren Menge an Elementen: $O(n_1 \cdot \log(n_1 + n_2))$
   (b) Kompletter Neuaufbau: $O(n_1 + n_2)$

   Je nach Verteilung der Elemente können beide Möglichkeiten sinnvoll sein.

Aufgabe 2 (*Analyse: Laufzeitverhalten*)

a) Beweisen Sie allgemein für adressierbare *Priority Queues* die untere Laufzeitschranke von $\Omega(\log n)$ für `deleteMin` unter der Voraussetzung, dass `insert` konstante Laufzeit benötigt.

b) Warum muss diese untere Laufzeitschranke nicht gelten, wenn `insert` mehr Zeit benötigen darf?

Musterlösung:

a) Mit einer Laufzeit von $O(f(n))$ für `deleteMin` ließe sich in Zeit $O(nf(n)) + n \cdot O(1)$ vergleichsbasiert sortieren, indem man zuerst alle Elemente einfügt und dann eines nach dem anderen in aufsteigender Reihenfolge entnimmt. Für `deleteMin` in sublogarithmischer Zeit wäre das ein Widerspruch zur bekannten unteren Schranke für vergleichsbasiertes Sortieren von $\Omega(n \log n)$.

b) `insert` könnte nach jedem Aufruf eine aufsteigend sortierte Liste aller Elemente hinterlassen, mit deren Hilfe sich alle folgenden `min`- und `deleteMin`-Operationen in konstanter Zeit beantworten ließen.

Aufgabe 3 (*Analyse:* best-case *Verhalten*)

a) Geben Sie einen Zustand eines *Fibonacci Heaps* an, für den die nächsten n `deleteMin`-Operationen jeweils konstante Laufzeit benötigen (nicht amortisiert). Begründen Sie Ihre Antwort. Gehen Sie davon aus, dass zwischen den `deleteMin`-Operationen keine anderen Operationen ausgeführt werden.

b) Geben Sie einen Algorithmus an, welcher den von Ihnen angegebene Zustand für beliebige n erzeugt. Beweisen Sie die Korrektheit des Algorithmus.

Musterlösung:

a) Gegeben sei ein *Fibonacci Heap* der Größe n, bestehend aus einem einzelnen Baum. Dieser Baum speichere n Knoten in Form einer verketteten Liste. Die nachfolgenden n `deleteMin`-Operationen führen dann jeweils eine `Cut`-Operation auf den direkten Nachfolgern des jeweiligen Wurzelknotens aus. Da der Wurzelknoten nur einen Nachfolger hat, wird pro `deleteMin`-Operation nur eine `Cut`-Operation ausgeführt. Diese `Cut`-Operationen benötigen jeweils nur konstante Laufzeit. Da der *Fibonacci Heap* vor jeder `deleteMin`-Operation nur aus einem Baum besteht, führt die `deleteMin`-Operation keine `union`-Operation aus. Eine potentiell folgende `union`-Operation auf dem neuen Baum kann durch das Token des alten Wurzelknotens bezahlt werden. Es müssen also auch keine neuen Token für folgende Operationen bezahlt werden.

b) *Behauptung 1*: Die Operation `CreateFibonacciList` erzeugt einen *Fibonacci Heap* F der Größe n, bestehend aus einem einzelnen Baum, repräsentiert durch eine verkettete Liste.

```
1:  function CreateFibonacciList(n ∈ ℕ⁺)
2:      F ← empty fibonacci heap
3:      for i ← n down to 1 do
4:          F.insert(i)
5:          x ← F.insert(i+1)
6:          F.insert(i)
7:          F.deleteMin
8:          F.decreaseKey(x, 0)
9:          F.deleteMin
10:     end for
11: return F
12: end function
```

Beweis: Die Behauptung gilt für $n = 0$, in diesem Fall gibt die Operation `CreateFibonacciList` einen leeren *Fibonacci Heap* zurück.

Invariante 1: Sei $n > 0$ beliebig aber fest. Nach $0 < x \leq n$ Schleifendurchläufen der Zeilen 4-9 besteht F aus einem einzelnen Baum in Form einer verketteten Liste $n - x + 1, \ldots, n$.

Nehmen wir an, dass *Invariante 1* für beliebige aber feste $n > 0$ gilt. Unter dieser Voraussetzung besteht F nach $x = n$ Ausführungen der Zeilen 4-9 aus einem einzelnen Baum in Form einer verketteten Liste $1, \ldots, n$ und *Behauptung 1* gilt somit für $n \in \mathbb{N}_0^+$. Wir beweisen nun *Invariante 1* durch Induktion über die Schleifendurchläufe $x \in \{0, \ldots, n\}$. Sei hierzu $n > 0$ beliebig aber fest.

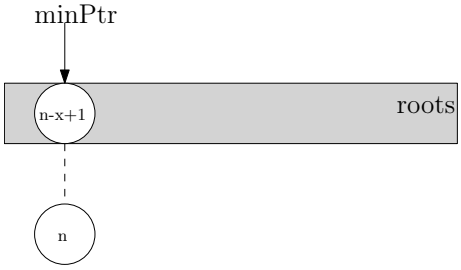
Induktionsanfang: $x = 1$: Nach Ausführung der Zeilen 4-6 enthält F die Elemente "n", "n+1"und "n". Die Zeilen 7-10 entfernen die Elemente "n" und "n+1". Somit enthält F nach dem ersten Schleifendurchlauf das Element "n" und die Invariante gilt für $x = 1$.

Induktionsschritt ($x \to x + 1$): Nach Induktionsvoraussetzung besteht F aus einem Baum, repräsentiert durch eine verkettete Liste, der Form $n - x + 1, \ldots, n$. Die Abbildungen auf der nächsten Seite führen nun die Codezeilen 4-9 aus und zeigen den Zustand von F nach $x + 1$ Iterationen. F besteht nun aus einem Baum, repräsentiert durch eine verkettete Liste, der Form $n - x, \ldots, n$. Somit gilt die Invariante auch nach $x + 1$ Iterationen und der Induktionsschritt $x \to x + 1$ ist abgeschlossen.

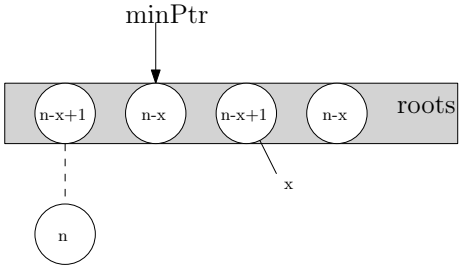
Induktionsschluss: Die Invariante gilt nach jedem Schleifendurchlauf.

Musterlösung:

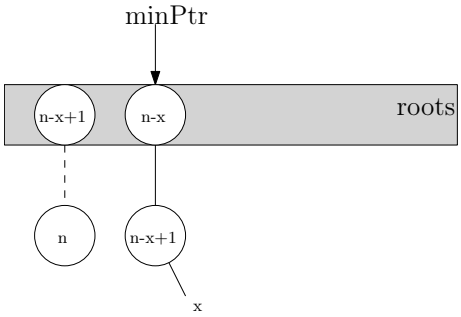
Induktionsvoraussetzung:



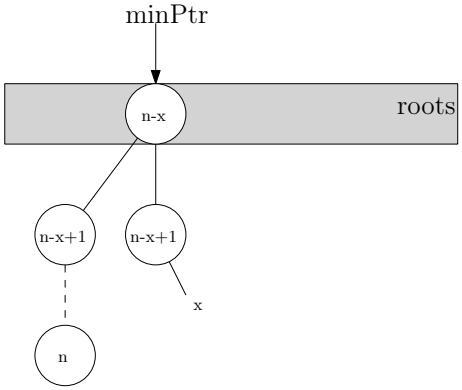
`F.insert`$(n-x)$`,x=F.insert`$(n-x+1)$`,F.insert`$(n-x)$:



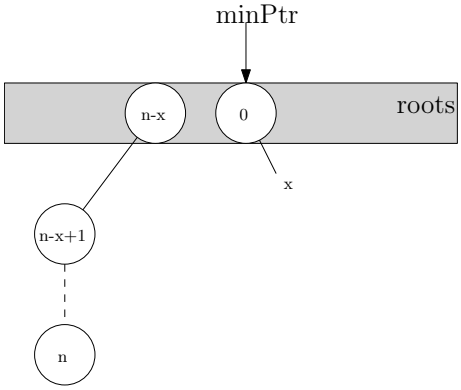
`F.deleteMin` (Zwischenschritt):



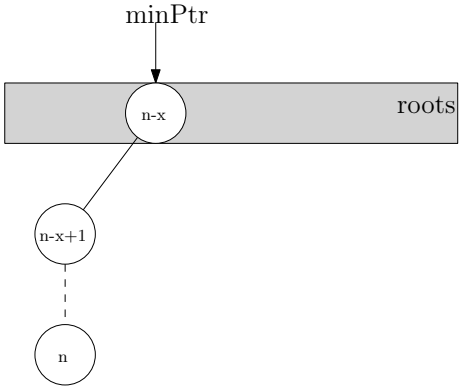
`F.deleteMin`:



`F.decreaseKey(x,0)`:



`F.deleteMin`:



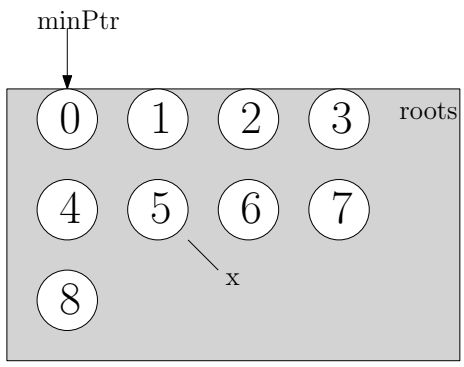
Aufgabe 4 (*Rechnen:* Fibonacci Heaps)

Gegeben sei ein *Fibonacci Heap* mit unten eingezeichnetem Zustand (a).

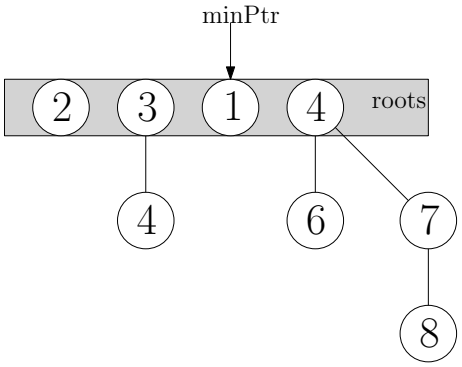
a) Geben Sie eine möglichst kurze Folge von Operationen an, die diesen Zustand erzeugt.

b) Führen Sie anschließend die Operationen `deleteMin()` auf dem Heap aus. Zeichnen Sie den Zustand des Heaps nach jedem Einfügen eines Baum in ein leeres Bucket und nach jeder Union-Operation.

c) Geben Sie eine möglichst kurze Folge von Operationen an, die den unten eingezeichneten Zustand (b) erzeugt. Tipp: der eingezeichnete Zustand lässt sich aus dem Heap in Abbildung (a) nach der `deleteMin`-Operation durch weitere Operationen erzeugen.



(a)



(b)

Musterlösung:

a) Die Folge

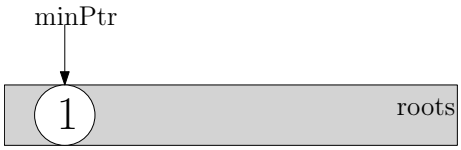
`insert(0)`, `insert(1)`, `insert(2)`, `insert(3)`, `insert(4)`, `insert(5)`, `insert(6)`, `insert(7)`, `insert(8)`

erzeugt den gegebenen Zustand.

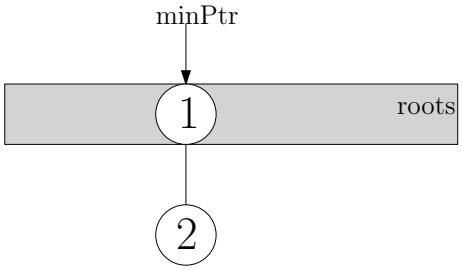
Beachten Sie, dass die Definition des *Fibonacci Heaps* keine Aussage darüber macht, in welcher Reihenfolge Wurzelknoten gespeichert sind. Für diese Lösung und die der nächsten Teilaufgabe nehmen wir eine nach der Reihenfolge der Einfügungeoperationen sortierte Liste von Wurzeln an. Werden Teilbäume abgeschnitten, so werden diese an das Ende der Liste angehängt.

Musterlösung:

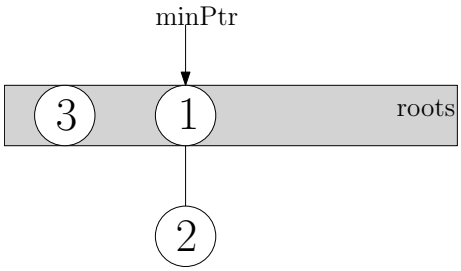
b) h_1 (keine Kollision):



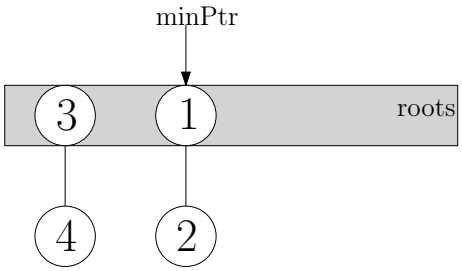
`Union`(h_1, h_2):



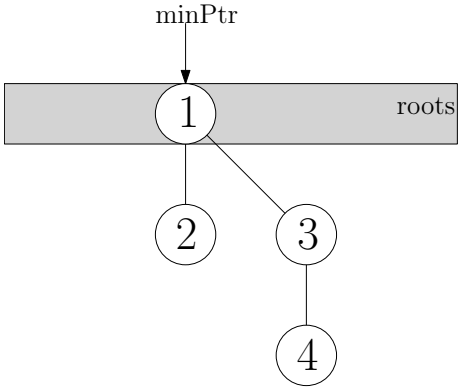
h_3 (keine Kollision):



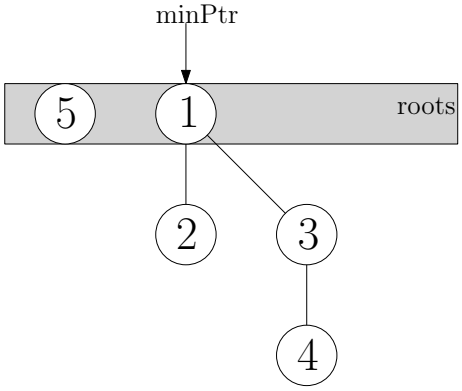
`Union`(h_3, h_4):



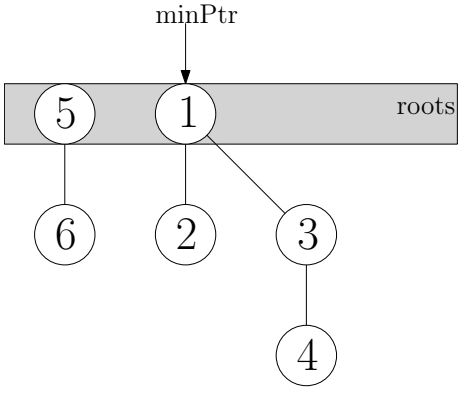
`Union`(h_3, h_1):



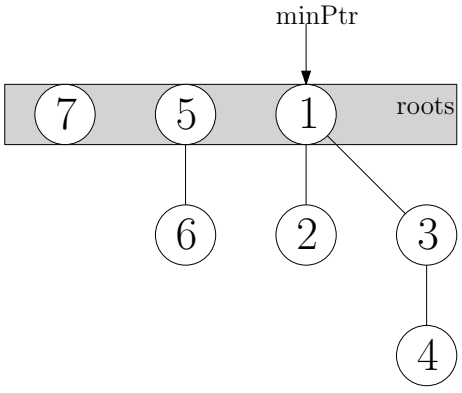
h_5 (keine Kollision):



`Union`(h_5, h_6):

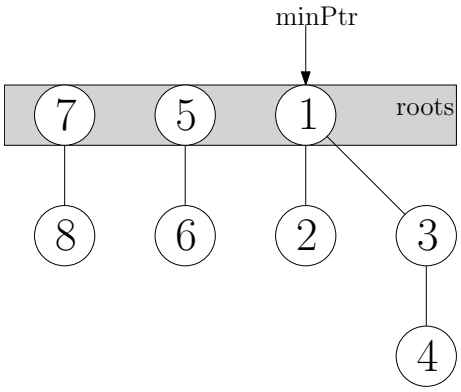


h_7 (keine Kollision):

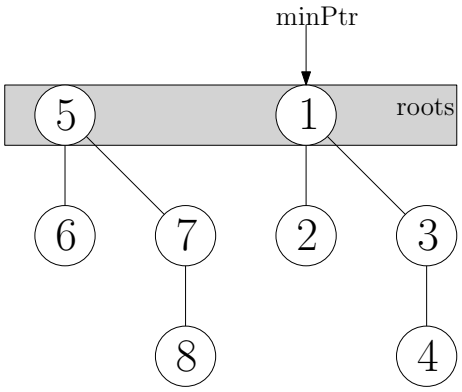


Musterlösung:

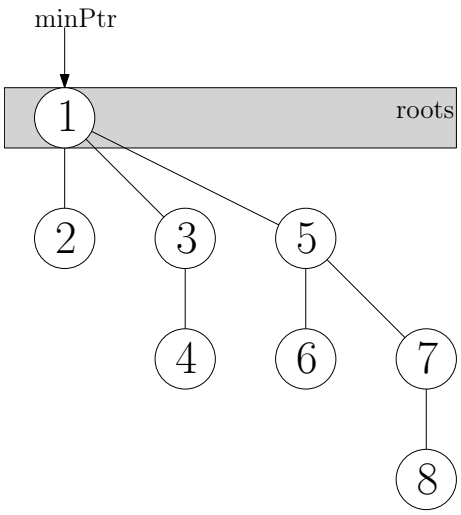
`Union`(h_7, h_8):



`Union`(h_7, h_5):



`Union`(h_5, h_1):



c) Die Folge

`deleteMin()`, `insert(1)`, `decKey(x, 4)`

erzeugt den gegebenen Zustand.

Aufgabe 5 (*Entwurf: Datenstrukturen*)

a) Erweitern Sie die Datenstruktur *Pairing Heap* um die Operation `increaseKey(h: Handle, k: Key)`. Ihre Operation sollte amortisiert $O(\log n)$ Laufzeit benötigen. Geben Sie Pseudocode an. Wie würden Sie bei einem *Binary Heap* vorgehen?

b) Entwerfen Sie eine Datenstruktur welche die Operationen `insert` in $O(\log n)$, Median bestimmen in $O(1)$ und Median entfernen in $O(\log n)$ unterstützt. Eine Beschreibung in Worten ist ausreichend.

Musterlösung:

a) Lösche das Element aus der Datenstruktur ($O(\log n)$) und füge es mit dem geänderten Schlüssel wieder ein ($O(1)$):

```
1: function increaseKey(h : Handle, k : Key)
2:     remove(h)
3:     key(h) := k
4:     insert(h)
5: end function
```

Bei einem *Binary Heap* würde man zuerst den Schlüssel anpassen und anschließend eine `siftDown`-Operation ausführen.

b) Aufbau des Datentyps:

- Speicherung des aktuellen Median in v.
- Verwendung einer Maximum *Priority Queue* (MaxPQ) für Elemente kleiner als v und einer Minimum *Priority Queue* (MinPQ) für Elemente größer als v.

Bestimmung des Median:
Frage v direkt ab: $O(1)$.

Löschen des Medians:
Ersetze den aktuellen Median v durch das oberste Element der größeren *Priority Queue* (bei Gleichheit verwende MinPQ): `deleteMin` in $O(\log n)$, z.B. mit *Fibonacci Heap*.

Einfügen eines Elements:
Füge das neue Element –in Abhängigkeit von v– in eine der *Priority Queues* ein: `insert` in $O(1)$. Füge v in die kleinere ein (bei Gleichstand verwende MaxPQ): `insert` in $O(1)$. Ersetze v durch das oberste Element der größeren *Priority Queue* (bei Gleichstand verwende MinPQ): `deleteMin` in $O(\log n)$, z.B. mit *Fibonacci Heap*.

Aufgabe 6 (*Entwurf: Anwendung*)

Für ein großes –und wir meinen ein wirklich großes– Fest sind Sie für die Bar zuständig. Für diese Aufgabe haben Sie Ihren eigenen Barroboter entworfen, der automatisch wunderbare Cocktails mischen kann. Die Zutaten dafür werden in großen Kanistern bereitgestellt. Doch genau hier ist das Problem: In all der Hektik des Abends müssen Sie darauf achten, dass kein Kanister leerläuft. Sie wollen den Abend allerdings auch so gut wie möglich genießen und nicht andauernd die Kanister überprüfen. Um dieses Problem zu umgehen, gibt es nur eine Lösung: Eine Nachfüllanzeige muss her! Leider gab es nur noch Anzeigen, die es erlauben eine einzelne Zeile darzustellen.
Es ist klar, dass auf der Anzeige die am dringensten benötigte Zutat angezeigt werden sollte. Ziel ist es also einen Algorithmus zu entwerfen, der die Anzeige immer aktuell hält.

a) Überlegen Sie sich, welche Datenstruktur Sie als Grundlage für Ihren Algorithmus verwenden wollen, um ihn effizient implementieren zu können. Sie können davon ausgehen, dass die für einen Cocktail benötigten unterschiedlichen Zutaten wesentlich weniger sind als die Gesamtmenge an vorhandenen unterschiedlichen Zutaten.

b) Entwerfen Sie eine Funktion `MixDrink( recipe )`, die auf Ihrer Datenstruktur operiert. Geben Sie Pseudocode an. Sie müssen dabei nur Ihre Datenstruktur aktualisieren. Sonstige Funktionen des Roboters müssen Sie nicht berücksichtigen.

c) Wenn ein Kanister gewechselt wird, muss ihre Datenbasis natürlich auch aktualisiert werden. Beschreiben Sie, welche Auswirkungen das Wechseln auf Ihre Datenstruktur hat.

Musterlösung:

a) Die Zutat, die auf dem Display angezeigt werden soll, ist die, die den geringsten Restvorrat vorweist. Zusätzlich sollen die Mengen aller Zutaten im laufenden Betrieb aktualisiert werden können. Dabei werden Zutaten in ihrer Menge immer reduziert. Die klassische Datenstruktur hierfür ist ein adressierbarer *Heap*.

b) Die Funktion `MixDrink`, wie gefordert, hat nur die Aufgabe die vorhandene Zutatenmenge aktuell zu halten.

```
1: function MixDrink(r : Recipe, q : Queue, d : Display)
2:     for ingredient i ∈ r do
3:         q.decreaseKey(i, amount(i))
4:     end for
5:     d.show(q.min(), amount(q.min()))
6: end function
```

c) Beim Wechsel eines Kanisters muss das Element aus dem *Heap* entfernt werden und mit vollem Kanisterwert wieder eingefügt werden. Dabei ist allerdings zu beachten, dass nicht zwingend das aktuell minimale Element gewechselt wird. Daher sollte eine Folge von Operationen durchgefürt werden. Als erstes sollte der Schlüssel der betroffenen Zutat auf $-\infty$ gesetzt werden. Danach kann ein `deleteMin` gefolgt von einem `insert` ausgeführt werden.

Aufgabe 7 (*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

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

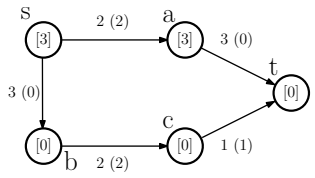
gilt.

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. $u \in V$.

Damit ist $\text{pot}^c(\cdot)$ 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 8 (*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 9 (*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) 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.

b) Geben Sie eine Klasse von Graphen an, für welche die Anzahl an `deleteMin` Operationen in Dijkstra's Algorithmus von einem beliebigen Knoten s zu einem beliebigen erreichbaren Knoten t linear von der minimalen Pfadlänge $\mu(s,t)$ abhängt. Für die Klasse von Graphen muss weiter $m = \Theta(n \log n)$ gelten.

Musterlösung:

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

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

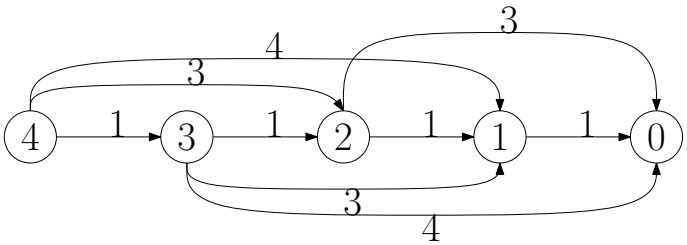
Mit den angegebenen Laufzeiten eingesetzt ergibt sich:

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

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 $m = \Omega(n^2)$.

b) Eine Klasse von Graphen, die diese Anforderungen erfüllt, lässt sich wie folgt konstruieren:

Man bilde eine gerichtete Kette von n Knoten, verbunden durch Kanten mit Gewicht 1. Außerdem füge man von jedem Knoten i zu jeweils $\max(i, \lfloor \log n \rfloor)$ Nachfolgern eine Kante mit Gewicht größer der Distanz zwischen i und dem jeweiligen Nachfolger j auf der Kette ein.



Beispiel mit 5 Knoten

Für die Anzahl an Kanten gilt nun:

$$m \coloneqq \sum_{i=1}^{n-1} \lfloor \log n \rfloor - \sum_{i=1}^{\lfloor \log n \rfloor} i = \sum_{i=1}^{n-1} \lfloor \log n \rfloor - \frac{\lfloor \log n \rfloor \left(\lfloor \log n \rfloor + 1 \right)}{2} \in \Theta\left(n \log n - \log^2 n \right) \in \Theta\left(n \log n \right)$$

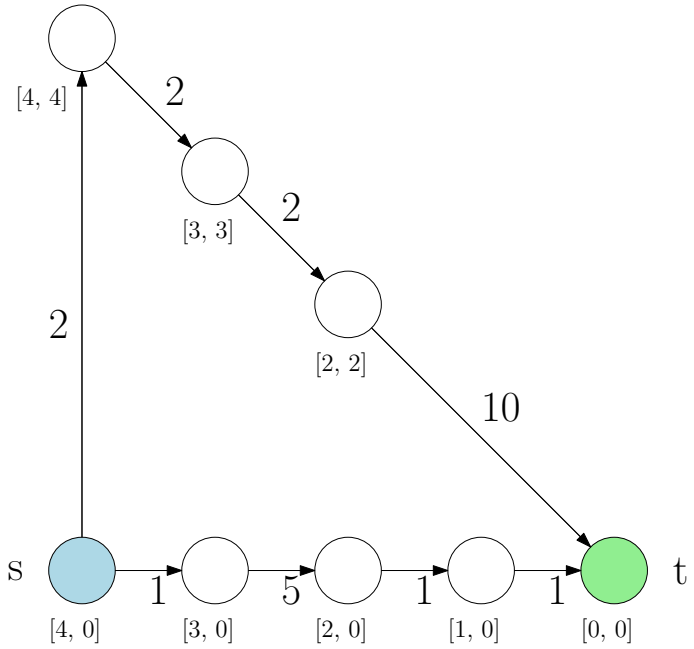
Aufgabe 10 (*Rechnen: A* Suche*)

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

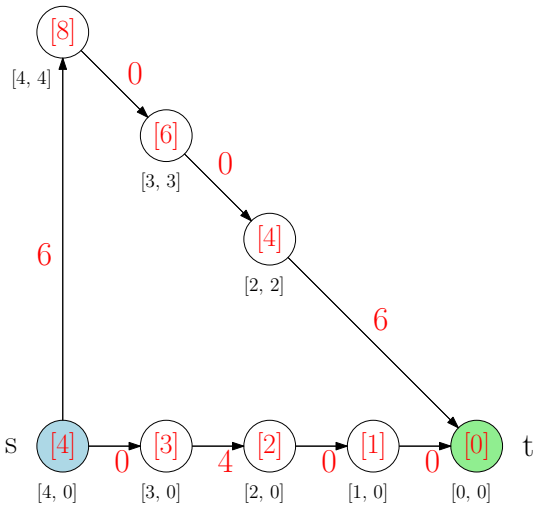
b) Tragen Sie die reduzierten Kantengewichte in den Graphen ein.

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



Musterlösung:

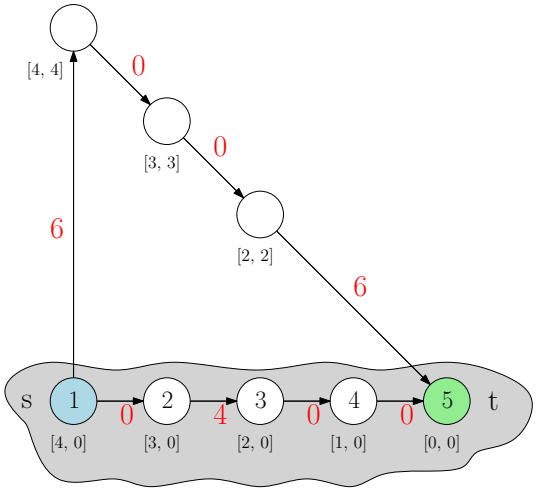
a) Knotenpotentiale pot(·) in Knoten eingetragen; Kantengewichte $c(\cdot)$ durch reduzierte Gewichte $\bar{c}(\cdot) : \bar{c}(u,v) = c(u,v) + \text{pot}(v) - \text{pot}(u)$ ersetzt:



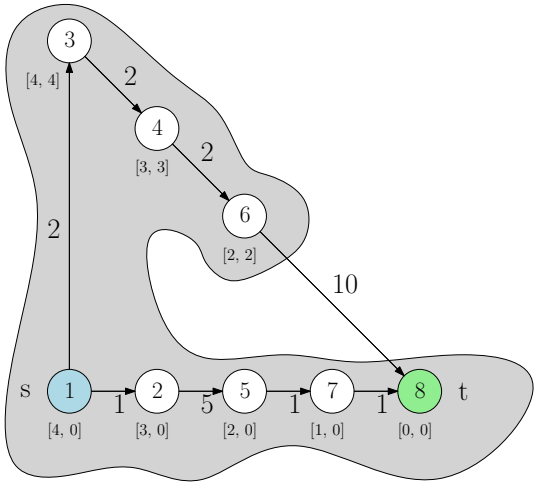
b) Siehe vorherige Teilaufgabe.

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:

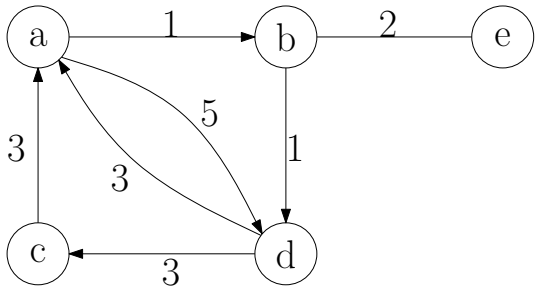


Aufgabe 11 (*Einführung+Analyse: Bidirektionaler Dijkstra*)

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

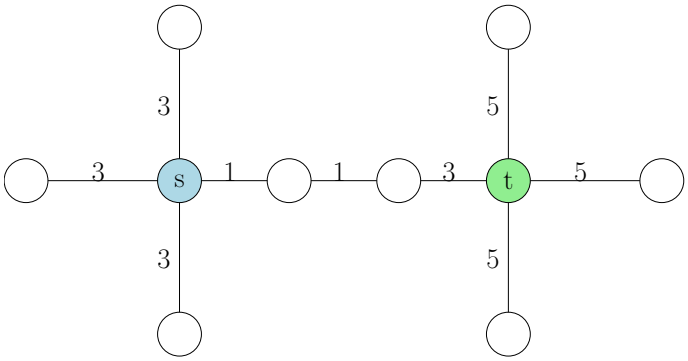
Zur Wiederholung:
Gegeben sei –wie üblich– ein gerichteter Graph $G = (V,\ E)$ mit $|V| = n$ und $|E| = m$, sowie eine Kantengewichtungsfunktion $c : E \to \mathbb{R}_0^+$. Gesucht ist der kürzeste Pfad $p = \langle s, \ldots, t \rangle$ zwischen zwei Punkten $s,\ t \in V$.
Eine bidirektionale Suche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

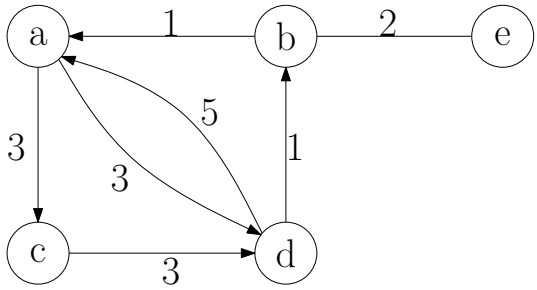


c) Zeigen Sie, dass die Abbruchbedingung korrekt ist.

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:

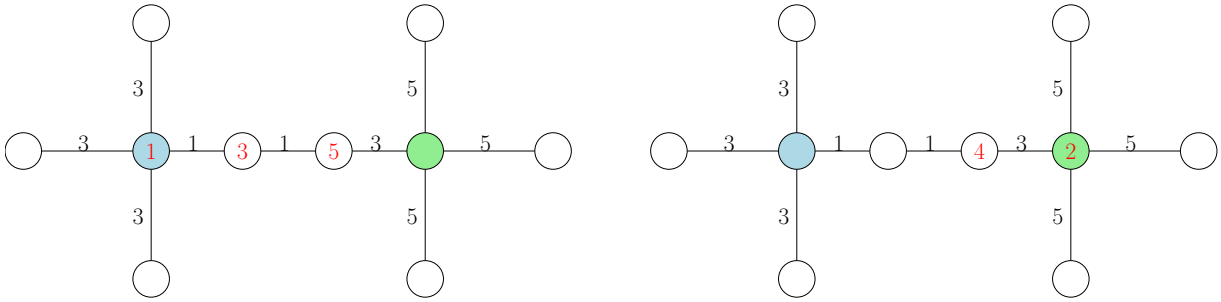
a) Rückwärtsgraph G^r:



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 $c(u,v) = c^r(v,u)$.

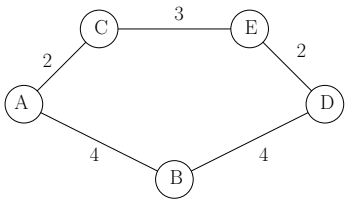
b) Vorwärtssuche: Rückwärtssuche:



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.

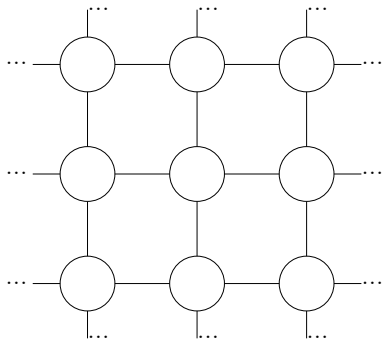
d) Der abgebildete Graph ist ein mögliches Beispiel.



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 12 (*Analyse: Bidirektionaler Dijkstra*)

a) Gegeben sei ein Gittergraph G mit allen Kantengewichten gleich 1. Wieviele Knoten wird eine bidirektionale Suche besuchen in Abhängigkeit von Abstand d zwischen Start und Ziel? Wieviele die unidirektionale Suche?



Beispiel eines Gittergraphen

b) Geben Sie ein Beispiel an, in dem die bidirektionale Suche von s nach t exponentiell weniger Knoten besucht als die unidirektionale Suche.

c) Geben Sie ein Beispiel, in dem die bidirektionale Suche von s nach t mehr Knoten besucht als die unidirektionale Suche.

d) Zeigen Sie, dass die bidirektionale Suche nie mehr als doppelt so viele Knoten besucht als die unidirektionale Suche.

Musterlösung:

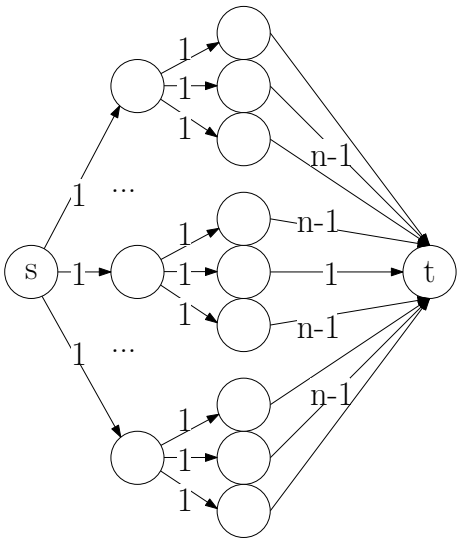
a) Dijkstra's Algorithmus scannt Knoten kreisförmig um den Startknoten. Für den Kreisumfang gemessen in Knoten gilt im Gridgraph: $u(r) = 2(r+1) + 2(r-2) = 4r$ (Einsnorm).

Im Falle der unidirektionalen Suche werden Kreise mit Radius 1 bis $d-1$ vollständig und der Kreis mit Radius d teilweise abgesucht. Damit werden $\sum_{r=0}^{d-1} u(r) + 1 = 1 + 4d(d-1)/2 + 1$ bis $\sum_{r=0}^{d} u(r) = 1 + 4d(d-1)/2 + 4d$ Knoten gescannt. Die zusätzliche +1 entspricht dem Scannen des Startknotens.

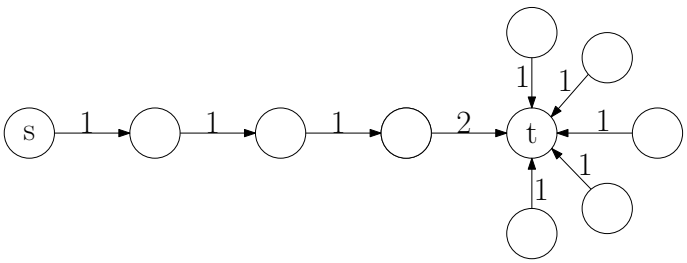
Im Falle der bidirektionalen Suche werden für jede Richtung Kreise mit Radius 1 bis $\lfloor d/2 \rfloor$ vollständig und der Kreis mit Radius $\lfloor d/2 \rfloor + 1$ teilweise abgesucht. Damit werden zwischen $\sum_{r=0}^{d-1} u(r) + 1 = 1 + 4(\lfloor d/2 \rfloor)(\lfloor d/2 \rfloor - 1)/2 + 1$ und $\sum_{r=0}^{d} u(r) = 1 + 4(\lfloor d/2 \rfloor)(\lfloor d/2 \rfloor - 1)/2 + 4(\lfloor d/2 \rfloor + 1)$ Knoten in jeder Richtung gescannt.

Vergleicht man beide Ergebnisse, so stellt man fest, dass die bidirektionale Suche nur ungefähr halb so viele Knoten scannt wie die unidirektionale Suche.

b) Von s wird ein Baum mit allen Kantengewichten gleich 1 aufgespannt, dessen Blätter über jeweils eine Kante mit Gewicht $n-1$ mit t verbunden sind – bis auf ein Blatt, das über eine Kante mit Gewicht 1 mit t verbunden ist. Die unidirektionale Suche besucht alle n Knoten. Die bidirektionale Suche besucht nur $O(\log n)$ Knoten.



c) Der abgebildete Graph ist ein mögliches Beispiel. Unidirektionale Suche scannt 5 Knoten, bidirektionale Suche 9.



d) Sei k die Anzahl Knoten, die von der unidirektionalen Suche besucht werden. Die bidirektionale Suche führt zwei unabhängige unidirektionale Suchen aus. Dabei entspricht die Vorwärtssuche der unidirektionalen Suche. Beide Suchrichtungen wechseln sich ab und es wird mit der Vorwärtsrichtung begonnen. Hat die bidirektionale Suche $2k-1$ Schritte durchgeführt, entfallen davon k auf die Vorwärtsrichtung. Die Vorwärtssuche hat damit die gleichen k Knoten abgesucht wie die unidirektionale Suche, einschließlich des Zielknotens. Da er in der Gegenrichtung auch schon abgesucht wurde (als erster Knoten), kann die Suche beendet werden.

Aufgabe 13 (*Analyse: Äquivalenzrelation*)

Gegeben sei ein gerichteter Graph $G = (V, E)$ und folgende Relation

$$v \overset{*}{\longleftrightarrow} w \qquad \text{gdw.} \qquad \text{es gibt Pfade } \langle v, \ldots, w \rangle \text{ und } \langle w, \ldots, v \rangle \text{ in } G$$

für $v, w \in V$.
Zeigen Sie, dass $\overset{*}{\longleftrightarrow}$ eine Äquivalenzrelation ist.

Musterlösung:
Um nachzuweisen, dass es sich um eine Äquivalenzrelation handelt, müssen *Reflexivität*, *Symmetrie* und *Transitivität* der Relation gezeigt werden:

Reflexivität:
Alle Pfade $\langle v \rangle$ mit $v \in V$ existieren per Definition.

Symmetrie:
Sei $v, w \in V$ mit $v \overset{*}{\longleftrightarrow} w$. Dann gibt es Pfade $\langle v, \ldots, w \rangle$ und $\langle w, \ldots, v \rangle$ in G. Damit gilt $w \overset{*}{\longleftrightarrow} v$.

Transitivität:
Sei $u, v, w \in V$ mit $u \overset{*}{\longleftrightarrow} v$ und $v \overset{*}{\longleftrightarrow} w$. Dann gibt es Pfade $\langle u, \ldots, v \rangle$ und $\langle v, \ldots, w \rangle$ in G und somit auch einen Pfad $\langle u, \ldots, v, \ldots, w \rangle$. Analog zeigt man, dass es einen Pfad $\langle w, \ldots, u \rangle$ in G gibt. Damit folgt $u \overset{*}{\longleftrightarrow} w$. Also ist $\overset{*}{\longleftrightarrow}$ transitiv.

Aufgabe 14 (*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.

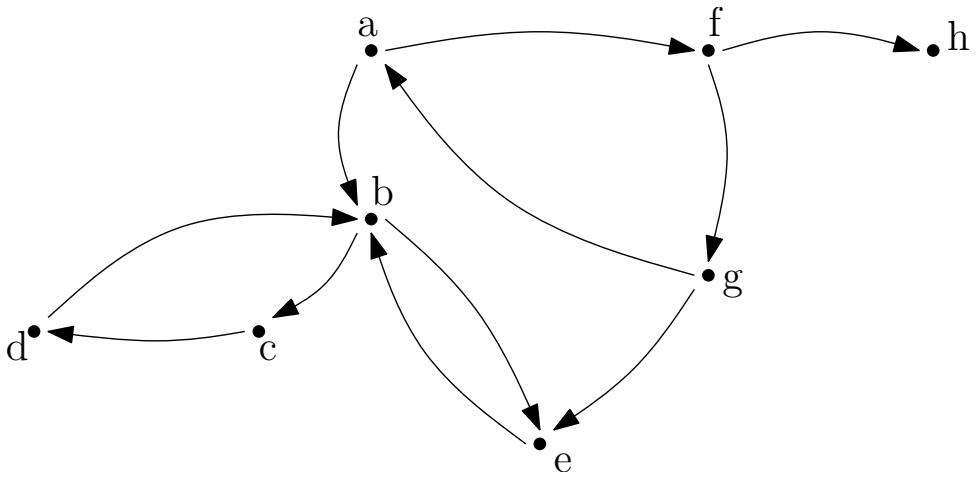
c) Betrachte einen spannenden Baum des Graphen G. Jeder Blattknoten dieses Baumes kann entfernt werden ohne dass der Graph zerfällt.

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 15 (*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

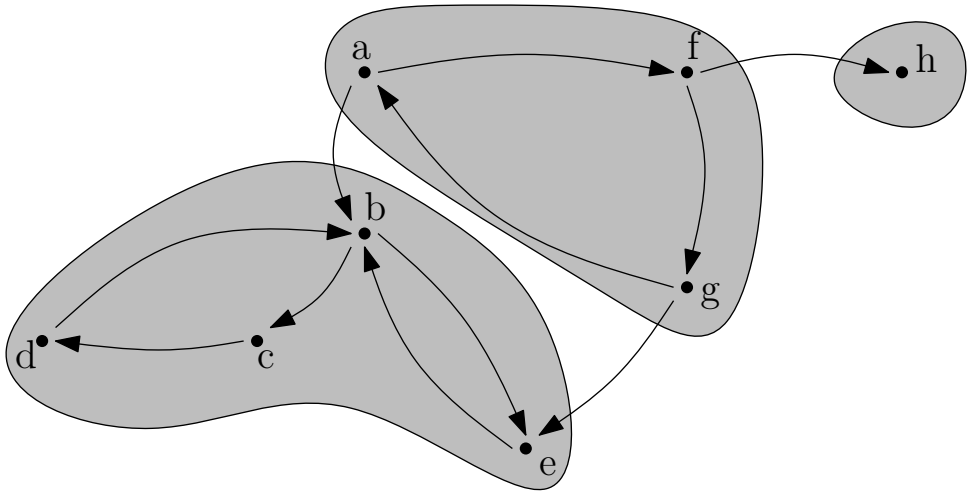
w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	h

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 16 (*Analyse: Kürzeste Wege*)

a) Betrachten Sie eine Suche mit bidirektionalem Dijkstra. Geben Sie eine Familie von Graphen an, bei der ein ausgezeichneter Knoten u eine Anzahl an `decreaseKey` Operationen erfährt, die linear in der Länge des kürzesten Weges für jede mögliche Anfrage ist.

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.

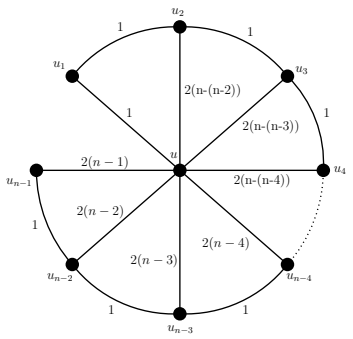
c) Bei der Durchführung von *Dijkstras Algorithmus* können kürzeste Wege gespeichert werden, indem Vorgängerknoten gespeichert werden. Diese werden immer dann geändert, wenn eine bessere vorläufige Distanz gefunden wird. Dieses kann auch auf einem Graphen durchgeführt werden, der negative Kantengewichte enthält. Das Stopkriterium von Dijkstras Algorithmus muss dafür durch ein schwächeres Kriterium ersetzt werden: Es wird gestoppt, sobald keine Verbesserung mehr gefunden werden kann. Zeigen Sie, wenn auf diese Art ein Kreis entsteht, so ist dieser negativ.

d) (*) Dijkstras Algorithmus ist ein Spezialfall des allgemeinen *Labeling Algorithmus*. Der allgemeine Labeling Algorithmus wählt eine beliebige Kante aus, die das Label des Zielknotens verbessert. Geben Sie einen Graphen sowie eine Reihenfolge der Kanten an, so dass bei positiven Kantengewichten eine exponentielle Zahl von Schritten ausgeführt wird.

Hinweis: Achtung, schwierige Knobelaufgabe!

Musterlösung:

a) Der abgebildete Graph mit n Knoten ist ein mögliches Beispiel.



Jeder kürzeste Weg zwischen zwei Knoten geht entlang des äußeren Kreises. Wir bezeichnen den mittleren Knoten mit u und die Knoten auf dem Kreis mit u_1 bis u_{n-1} entsprechend der Abbildung. Betrachten wir den Pfad $u_i, \dots u_j$ mit $i > j$ und kürzester Weglänge $\mu(i,j) = i - j$. O.b.d.A sei $s = u_i$ und $t = u_j$. Die Vorwärtssuche führt zwischen $3(i-j)/4$ und $(i-j)/4$ Schritte entlang des Pfades aus (die Suchrichtungen wechseln sich ab und besuchen ggf. auch Knoten außerhalb des Pfades). Bis auf den ersten Schritt führt jeder dieser Schritte zu einer `decreaseKey` Operation auf Knoten u. Schritte außerhalb des Pfades führen zu keiner `decreaseKey` Operation auf Knoten u.

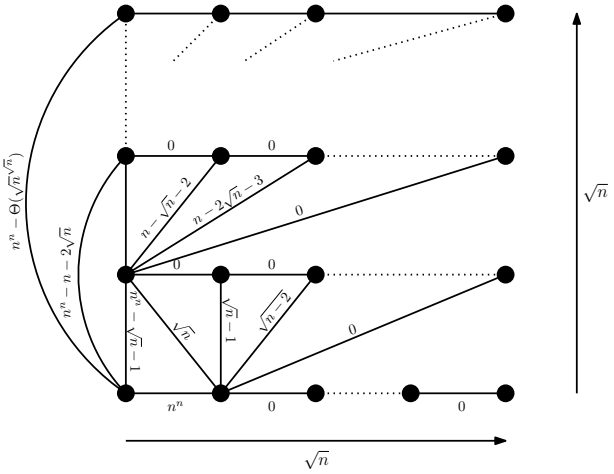
b) Es ist zu zeigen:

- 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 $c((u,v)) + \pi(v) \geq \pi(u)$.

Musterlösung:

c) Nehmen wir an, dass es einen solchen Kreis geben würde und dieser wäre nicht negativ. Betrachten wir nun den Moment, in dem der Kreis zum ersten Mal geschlossen wird. Dies geschehe an Knoten u. Der Kreis sei dabei bezeichnet als $k = \{u = n_1, \ldots, n_k = u\}$. Wenn der Kreis ein positives Gewicht hätte, so wäre $d(s,u) + k \geq d(s,u)$ und somit würden wir den Vorgänger von u nicht auf n_{k-1} setzen.

d) Der abgebildete Graph ist ein mögliches Beispiel.



Im allgemeinen Labeling Algorithmus kann die Ausführungsreihenfolge beliebig schlecht gewählt werden. Der angegebene Graph erlaubt es, immer die komplette untere Reihe von Knoten abzulaufen, und dann dem zweiten Knoten dieser Reihe eine um 1 kleinere Distanz zuzuweisen. Dabei kann jede der $\sqrt{n}$ Zeilen dazu genutzt werden, den ersten Knoten der darunter liegenden Reihe $\sqrt{n}$ mal eine kürzere Distanz zuzuweisen. Rekursiv lässt sich somit eine Bearbeitungsreihenfolge festlegen. Wir starten mit der Bearbeitung der untersten Zeile (in Serie). Wenn wir n Zeilen bearbeiten können, so können wir $n+1$ Zeilen bearbeiten, indem wir $\sqrt{n}$ mal den zweiten Knoten der obersten Reihe der n Reihen eine um 1 kürzere Distanz als bisher bekannt zuweisen und danach die Bearbeitung der n Reihen wiederholen. Somit ergibt sich eine Gesamtbearbeitungszeit von $\sqrt{n}^{\sqrt{n}}$ Schritten.