Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Moritz Kobitzsch, Dennis Schieferdecker

2. Übungsblatt zu Algorithmen II im WS 2011/2012

http://algo2.iti.kit.edu/AlgorithmenII.php
{kobitzsch,sanders,schieferdecker}@kit.edu

Musterlösungen

Aufgabe 1 (*Kleinaufgaben: A* Suche*)

a) Sei $\text{pot}(\cdot)$ eine gültige Potentialfunktion für die 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 Dijkstras Algorithmus für die gleiche Anfrage? Begründen Sie warum nicht oder geben Sie ein Beispiel an.

Musterlösung:

a) Es ist zu überprüfen, ob

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

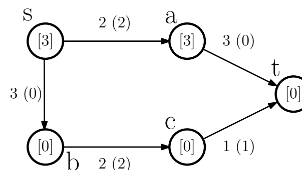
gilt.

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, ts, während A* die Knoten in der Reihenfolge s, b, a, c, t scannt.s



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

Aufgabe 2 (*Rechnen: Monotone ganzzahlige* Priority Queues)

Bei einer Ausführung von *Dijkstras 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 3 (*Analyse: Laufzeit von Dijkstras Algorithmus*)

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

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

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

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

   (ob eine Datenstruktur mit diesen Eigenschaften existiert und Dijkstras 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 Dijkstras 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?

c) Geben Sie eine Klasse von Graphen an, für die die Anzahl an `deleteMin` Operationen von einem beliebigen Knoten zu allen erreichbaren Knoten linear von der Pfadlänge $\mu(s, \cdot)$ abhängt, gegeben dass $m = \Omega(n)$.

Musterlösung:

a) Für die durchschnittliche Laufzeit von Dijkstras Algorithmus mit einem *binary heap* gilt:

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

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

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

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

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

b) Allgemein gilt für die Laufzeit von Dijkstras Algorithmus:

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

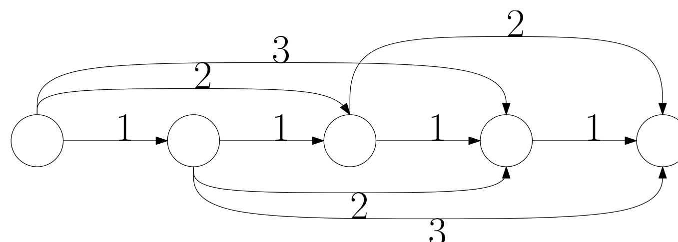
Mit den angegebenen Laufzeiten eingesetzt ergibt sich:

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

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

c) 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 $\log n$ Nachfolgern j eine Kante mit Gewicht größer oder gleich der Distanz zwischen i und j auf der Kette ein.



Beispiel mit 5 Knoten

Aufgabe 4 (*Entwurf:* All Pairs Shortest Paths)

Sie sind von der Finanzaufsichtsbehörde beauftragt worden, einen Algorithmus zu entwickeln, der Irregularitäten im Devisenhandel möglichst zeitnah aufdecken kann. Zu diesem Zweck erhalten Sie die aktuellen direkten Wechselkurse $w_{i,j}$ von Währung i nach Währung j für alle gehandelten Währungen. Dabei bedeutet z.B. $w_{i,j} = 4$, dass man für 1 Einheit aus Währung i genau 4 Einheiten aus Währung j erhält. Eine Unregelmäßigkeit tritt dann auf, wenn eine Möglichkeit existiert, eine Währung i in eine Währung j über mehrere Zwischenwechsel zu tauschen, so dass der Ertrag der Wechsel weniger als die Hälfte des direkten Wechsels von Währung i nach j erzielt. Auf Rückfrage versichert Ihnen Ihr Auftraggeber außerdem, dass eine geldgenerierende Schleife (leider) nicht auftreten kann.

a) Formulieren Sie das Problem als graphentheoretisches Problem. D.h. bilden Sie die gegebenen Informationen auf Knoten und Kanten eines Graphen ab und interpretieren Sie die gestellte Aufgabe als Problem auf dem von Ihnen definierten Graphen.

b) Beschreiben Sie einen Algorithmus, der das Problem löst.

c) Erweitern Sie Ihren Algorithmus, so dass er auch die Folge an Wechseln ausgeben kann, die eine Unregelmäßigkeit verursacht.

d) Ihr Algorithmus muss k Währungen überwachen. Geben Sie eine Laufzeit für Ihren Algorithmus an, die nur von k abhängt.

Hinweis: $\log ab = \log a + \log b$.

Musterlösung:

a) Modellierung des Problems als Graph:

Knoten entsprechen Währungen, Kanten erlaubten Geldwechseln. Kantengewichte geben den Logarithmus des Wechselkurses an.

Es ergibt sich ein vollständiger Graph mit k Knoten. Der Graph kann negative Kantengewichte aber –laut Vorgabe– keine negativen Zyklen enthalten. Gesucht sind kürzeste Verbindungen von jedem Knoten zu jedem anderen.

b) Führe eine *All-To-All* Suche durch und prüfe für je zwei Währungen i, j, ob der berechnete kürzeste Abstand kleiner als $\log 0.5 w_{i,j}$ ist. Trifft dies zu, melde eine Unregelmäßigkeit.

c) Es müssen zusätzlich *parent* Zeiger gespeichert werden. Über diese kann man rückwärts von der Zielwährung die Folge an Wechseln rekonstruieren.

d) Der Graph hat $n = k$ Knoten und $m = k(k-1)/2$ Kanten (vollständiger Graph). Der verwendete Algorithmus braucht allgemein $O(nm + n^2 \log n)$ Laufzeit. Eingesetzt ergibt sich eine Laufzeit von $O(k^3)$.

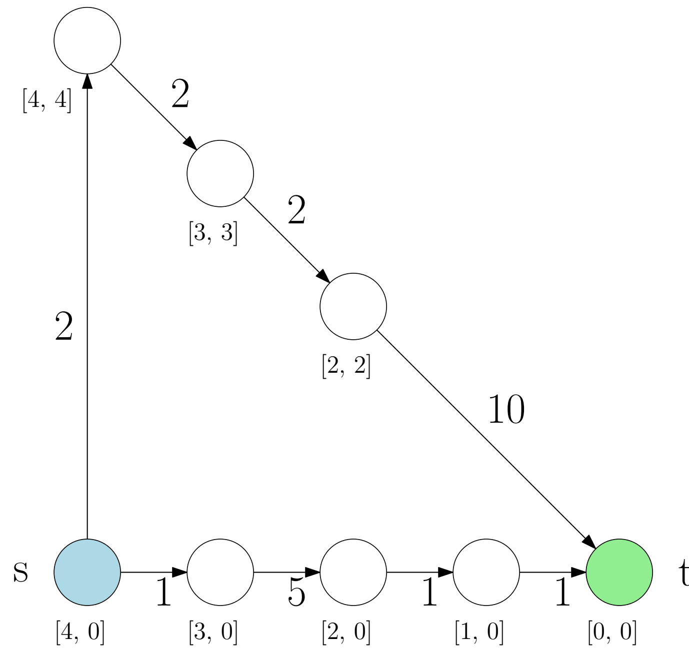
Aufgabe 5 (*Rechnen: A* Suche*)

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 die Manhatten-Distanz ($\hat{=}$ Einsnorm $||\cdot||_1$) als Abschätzung für die Entfernung zum Ziel.
Hinweis: $||\cdot||_1 : ||(x_1, y_1), (x_2, y_2)||_1 = y_2 - y_1 + x_2 - x_1$.

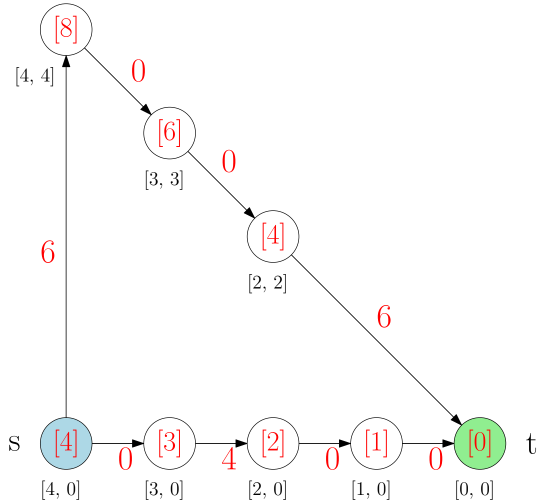
b) Tragen Sie die reduzierten Kantengewichte in den Graphen ein.

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



Musterlösung:

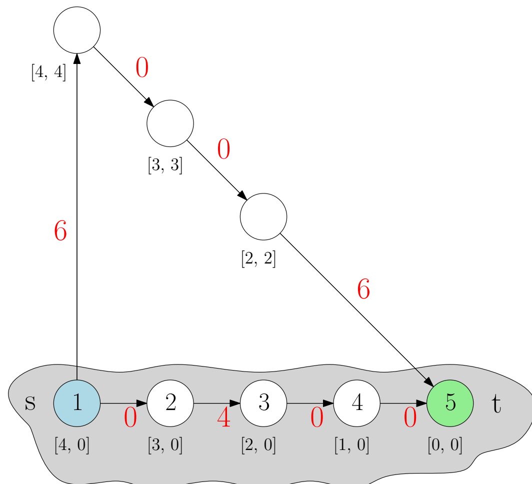
a) Knotenpotentiale pot(·) in Knoten eingetragen; Kantengewichte $c(\cdot)$ durch reduzierte Gewichte $\bar{c}(\cdot) : \bar{c}(u,v) = c(u,v) + \text{pot}(v) - \text{pot}(u)$ ersetzt:



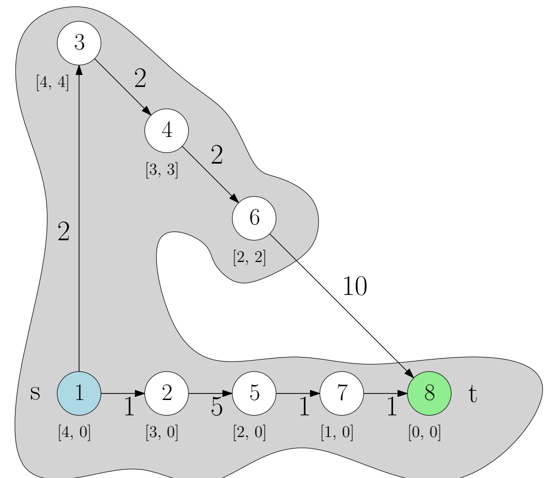
b) Siehe vorherige Teilaufgabe.

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:

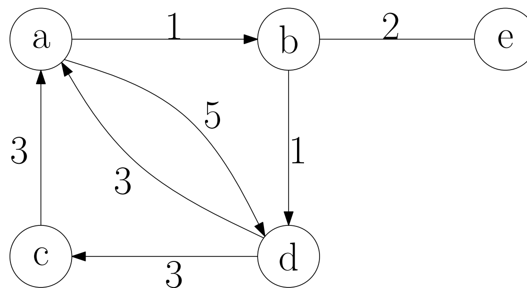


Aufgabe 6 (*Einführung+Analyse: Bidirektionaler Dijkstra*)

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

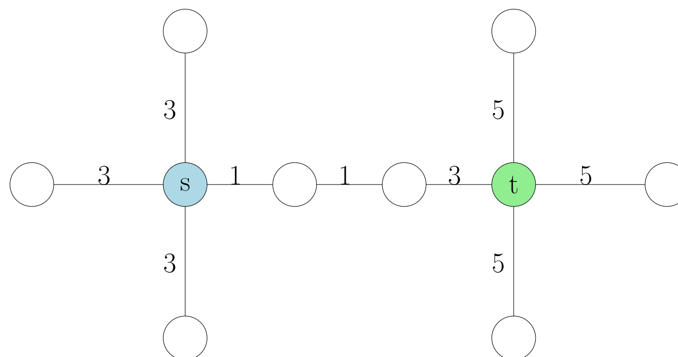
Zur Wiederholung:
Gegeben sei –wie üblich– ein gerichteter Graph $G = (V,\ E)$ mit $|V| = n$ und $|E| = m$, sowie eine Kantengewichtungsfunktion $c : E \to \mathbb{R}_0^+$. Gesucht ist der kürzeste Pfad $p = \langle s, \ldots, t\rangle$ zwischen zwei Punkten $s,\ t \in V$.
Eine bidirektionale Suche löst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

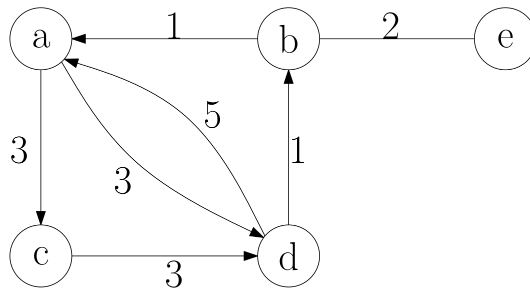


c) Zeigen Sie, dass die Abbruchbedingung korrekt ist.

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:

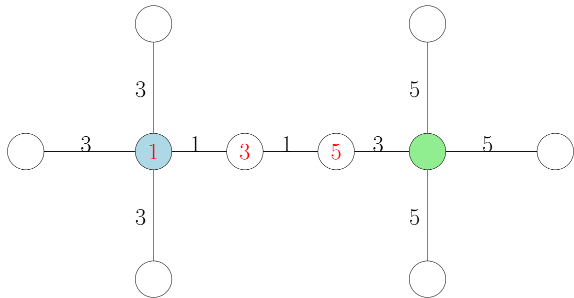
a) Rückwärtsgraph G^r:



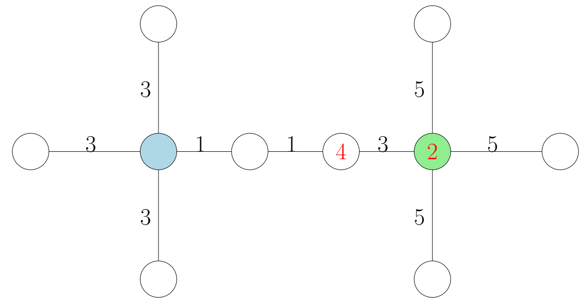
Es sind einfach alle Pfeile umgedreht worden.

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:



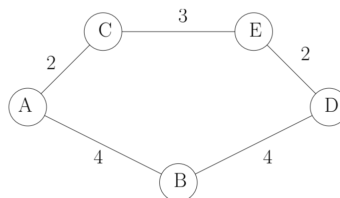
Rückwärtssuche:



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.

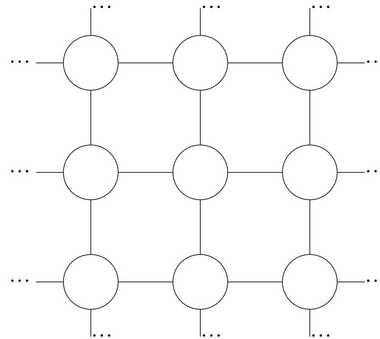
d) Der abgebildete Graph ist ein mögliches Beispiel.



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 7 (*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:

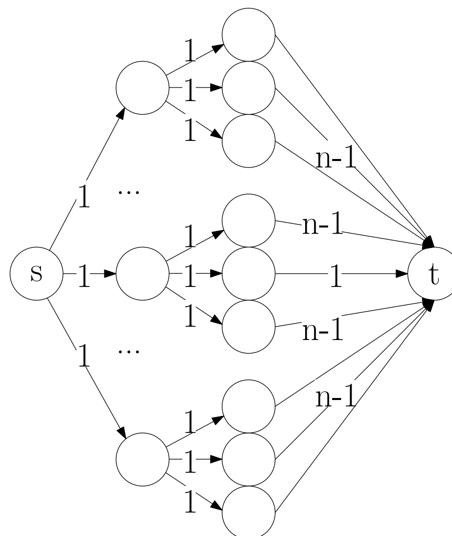
a) Dijkstras Algorithmus scannt Knoten kreisförmig um den Startknoten. Für den Kreisumfang gemessen in Knoten gilt im Gridgraph: $u(r) = 2(r+1) + 2(r-2) = 4r$ (Einsnorm).

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

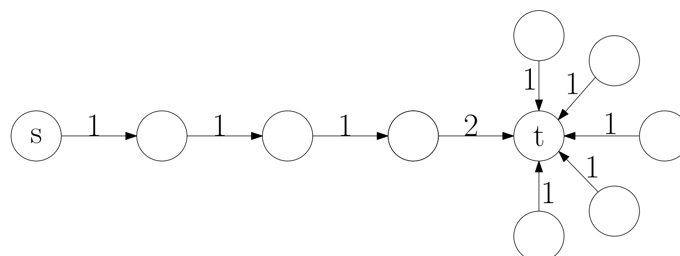
Im Falle der bidirektionalen Suche werden für jede Richtung Kreise mit Radius 1 bis $\lfloor d/2 \rfloor$ vollständig und der Kreis mit Radius $\lfloor d/2 \rfloor + 1$ teilweise abgesucht. Damit werden zwischen $\sum_{r=0}^{d-1} u(r) + 1 = 1 + 4(\lfloor d/2 \rfloor)(\lfloor d/2 \rfloor - 1)/2 + 1$ und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