Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Hans-Peter Lehmann, Daniel Seemaier

4. Übungsblatt zu Algorithmen II im WS 2021/2022

http://algo2.iti.kit.edu/AlgorithmenII_WS21.php
{sanders, hans-peter.lehmann, daniel.seemaier}@kit.edu

Musterlösungen

Aufgabe 1 (*Kleinaufgaben: Parallele Algorithmen*)

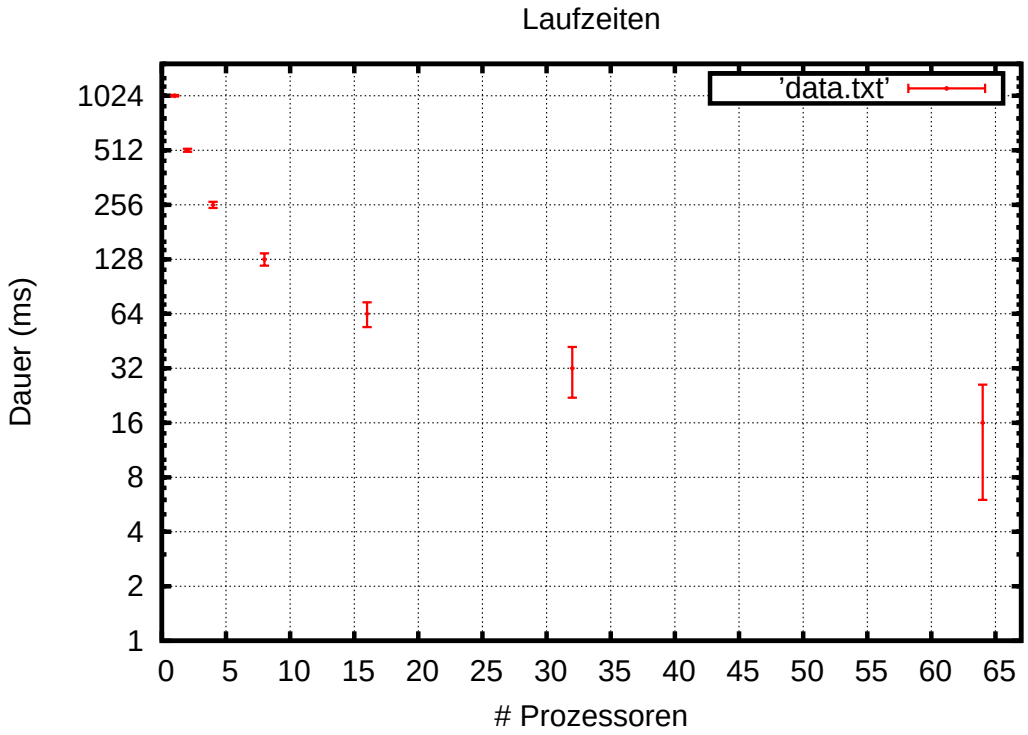
a) Gegeben sei ein paralleler vergleichsbasierter Sortieralgorithmus zum Sortieren von n komplexen Objekten auf p Prozessoren mit einer Laufzeit von

$$T(p) := \Theta(\frac{n^2 \log^2 n}{p^2}).$$

Geben Sie den absoluten *speed-up* und die *efficiency* an.

b) Wie muss in der vorherigen Teilaufgabe die Prozessorzahl p mit der Eingabegröße n wachsen, damit der absolute *speed-up* konstant bleibt?

c) Sie haben für einen parallelen Algorithmus folgende Laufzeiten bei unterschiedlicher Prozessorzahl gemessen. Was können Sie über die Skalierung dieses Algorithmus aussagen?



Musterlösung:

a) Die besten sequentiellen vergleichsbasierten Sortierverfahren benötigen $T_{seq} = \Theta(n \log n)$ Laufzeit. Damit ergibt sich für den absoluten *speed-up*

$$S(p) = \frac{T_{seq}}{T(p)} = \frac{\Theta\left(n \log n\right)}{\Theta\left(\frac{n^2 \cdot \log^2 n}{p^2}\right)} = \Theta\left(\frac{p^2}{n \log n}\right).$$

Mit der Definition der *efficiency* ergibt sich

$$E(p) = \frac{S(p)}{p} = \frac{\Theta\left(\frac{p^2}{n \log n}\right)}{p} = \Theta\left(\frac{p}{n \log n}\right).$$

b) Für einen konstanten *speed-up* unabhängig von der Prozessoranzahl muss gelten

$$S(p) = \Theta(\frac{p^2}{n \log n}) \stackrel{!}{=} \Theta(1)$$

und damit muss die Anzahl Prozessoren mit der Eingabegröße nach $p = \Theta(\sqrt{n \log n})$ wachsen.

c) Ignoriert man die Fehlerbalken, so weisst der Algorithmus eine Halbierung der Laufzeit bei Verdopplung der Prozessoren auf. Der relative *speed-up* ist also

$$S_{rel}(p) = \frac{T(1)}{T(p)} = p.$$

Der absolute *speed-up* skaliert auch linear mit p, über den genauen Faktor und über die Skalierung mit n kann man keine Aussage treffen. Die *efficiency* ist unabhängig von p.

Aufgabe 2 (*Entwurf+Analyse:* `findif`-*Anweisung*)

Gegeben sei ein Array $a[\cdot]$ im verteilten Speicher der n Objekte hält. Gesucht ist ein Algorithmus, der eine parallele `findif` Anweisung auf $a[\cdot]$ ausführt. Die Anweisung sortiert die Elemente von $a[\cdot]$ anhand eines Prädikats $pred(\cdot)$, so dass Elemente, die das Prädikat erfüllen, vorne stehen. Die relative Ordnung der Elemente untereinander soll dabei erhalten bleiben.

Bsp.: `findif( {1,4,9,7,3},` *is_bigger_than_3* `) = {4,9,7,1,3}`

a) Beschreiben Sie einen Algorithmus, der eine parallele `findif` Anweisung auf $a[\cdot]$ möglichst schnell ausführt. Sie haben $p = n$ Prozessoren zur Verfügung.

b) Untersuchen Sie die Laufzeit der Anweisung für den Fall, dass $p = n$ Prozessoren zur Verfügung stehen und das Prädikat in $T(n) = O(1),\ O(\log n)$ bzw. $O(n)$ ausgewertet werden kann.

c) Wie verhalten sich die Laufzeiten, wenn Sie nur noch $p < n$ Prozessoren zur Verfügung haben?

Musterlösung:

a) Prozessor p_i prüft Bedingung $pred(a[i])$ und schreibt das Resultat als 0 (falsch) oder 1 (wahr) in ein neues Array $s[\cdot]$ an Stelle i. Anschließend wird die Präfixsumme über $s[\cdot]$ gebildet. Aus diesen Werten kann jeder Prozessor p_i die neue Position für sein Datenelement $a[i]$ ableiten:

- $a[i] \rightarrow a[s[i] - 1]$, wenn $a[i]$ das Prädikat erfüllt,
- $a[i] \rightarrow a[s[n-1] + i - s[i]]$, wenn $a[i]$ das Prädikat nicht erfüllt.

Prozessoren werden von 0 durchnummeriert.

b) Die Ausführungszeit beträgt $O(T(n))$ für die Berechnung von $s[\cdot]$, $O(\log n)$ für die Berechnung der Präfixsumme und $O(1)$ für die Umsortierung. Gesamt ergeben sich die Laufzeiten in Abhängigkeit von $T(n)$ zu $O(\log n)$, $O(\log n)$ bzw. $O(n)$, wobei $p = n$ gilt.

c) Stehen weniger als n Prozessoren zur Verfügung, muss man die Arbeit für jeweils n/p Objekte von einem Prozessor ausführen lassen. Es ergeben sich folgende Laufzeiten für die drei Schritte $O(n/p \cdot T(n))$, $O(n/p + \log p)$ und $O(n/p)$. Insgesamt ergeben sich die Laufzeiten wieder in Abhängigkeit von $T(n)$ zu $O(n/p + \log p)$, $O(n/p \cdot \log p)$ bzw. $O(n^2/p + \log p)$.

Aufgabe 3 (*Entwurf+Analyse: Assoziative Operationen*)

Gegeben sei ein Array A im gemeinsamen Speicher bestehend aus n Objekten vom Typ X. Auf den Objekten sei eine Operator $\odot$ definiert. Es sei nach dem Ergebnis von $\odot_{i=1}^{n} a_i$ gesucht.

a) Sei $X = \mathbb{R}^2$ und der Operator definiert als

$$(x_1, x_2) \odot (y_1, y_2) := (x_1 y_1, x_2 + y_2)$$

Zeigen Sie, dass der Operator $\odot$ assoziativ ist.

b) Beschreiben Sie einen schnellen parallelen Algorithmus, der $\odot_{i=1}^{n} a_i$ berechnet und geben Sie dessen Laufzeit $T(n, p)$ an.

c) Nun sei $\odot$ wie folgt definiert: X beschreibe die Menge an möglichen Zeichenketten über dem Alphabet $\{(,)\}$. Die Operation $x \odot y$ verknüpfe beide Zeichenketten und schiebe alle öffnenden Klammern nach links, alle schließenden Klammern nach rechts (Bsp.: $()(() \odot (()) = ((((()))))$.

Können Sie den selben Lösungsansatz wie in der vorherigen Teilaufgabe verwenden? Falls nein, geben Sie einen neuen parallelen Algorithmus an. Wie lange dauert die Ausführung?

Musterlösung:

a) Der Operator ist offensichtlich assoziativ, da beide Koordinaten unabhängig voneinander sind und die ausgeführte Addition bzw. Multiplikation auf reelen Zahlen assoziativ ist.

b) Der Operator $\odot$ ist assoziativ und in konstanter Zeit ausführbar. Daher kann $\odot_{i=1}^{n} a_i$ mit Hilfe des Reduktionsschemas in $T(n, p) = O(n/p + \log p)$ berechnet werden. Für $p = n$ liegt die Ausführungszeit in $T(n, p) = O(\log n)$.

c) Auch hier ist $\odot$ assoziativ, benötigt aber Zeit proportional zur Länge beider Operanden (ein Scan über beide Operanden liefert die Anzahl öffnender und schließender Klammern, die anschließend geschrieben werden können). Das Reduktionsschema ist weiterhin anwendbar, die Laufzeit erhöht sich allerdings insgesamt zu $T(n, p) = O\left(\sum_{i=0}^{\log p} \left(2^i \frac{n}{p} + 1\right)\right) = O(p \cdot n/p + \log p)$ bzw. zu $T(n, p) = O(n)$ für $p = n$.

Ein schnellerer Lösungsansatz verwendet paralleles Sortieren. In diesem Fall ist die Laufzeit $T(n, p) = O(\frac{n \log n}{p} + \log^2 p)$ bzw. $T(n, p) = O(\log^3 n)$ für $p = n$.

Ein alternativer Ansatz wäre es, zunächst in einem Vorverarbeitungsschritt aus jedem Objekt ein Tupel zu erzeugen, das die Anzahl der öffnenden und schließenden Klammern enthält. Dann kann eine normale Reduktion durchgeführt werden, bei der eine komponentenweise Addition als Operation verwendet wird.

Aufgabe 4 (*Kleinaufgaben: Eigenschaften von Flüssen*)

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

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

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

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

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

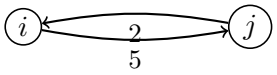
Entfernt man e' aus E und reduziert $c(e) := c(e) - c(e')$, ändert sich der maximale Fluss nicht, d. h. man kann entgegengesetzte Kanten a-priori (für beliebige s und t) gegeneinander aufrechnen.

Musterlösung:

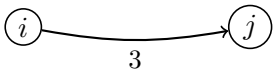
a) Ein Pfad in einem Graphen kann höchstens n unterschiedliche Knoten haben. Betrachte nun einen beliebigen augmentierenden Pfad in G_f. Für jede Kante auf diesem Weg wächst die Distanz für s höchstens um eins. Folglich kann ein augmentierender Pfad nur $d(s) \leq n - 1$ bedeuten.

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

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



so ergibt sich



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

Aufgabe 5 (*Rechnen: Segmentierung mit Flüßen*)

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

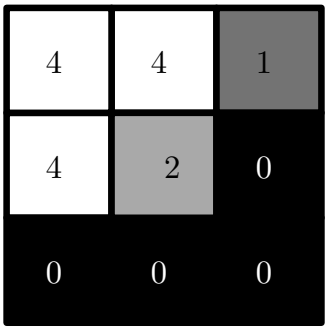
$$c(e=(u,v)) = \begin{cases} p_v(v) & u = s \\ p_h(u) & v = t \\ f(u,v) & \textit{sonst} \end{cases}$$

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.

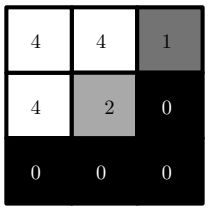
b) Führen Sie einen augmenting Path Algorithmus auf dem entstandenen Graphen aus.

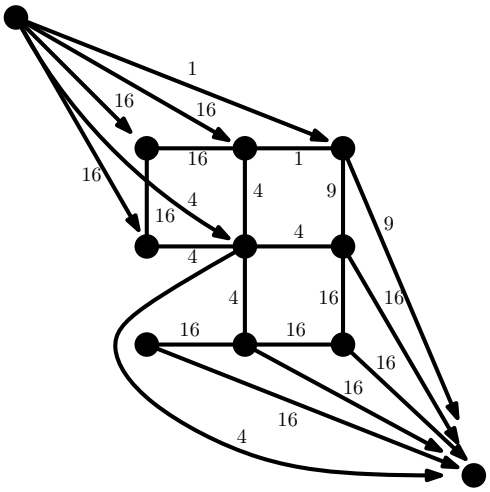
c) Wie würde die Segmentierung in Vorder- und Hintergrund im Bild als Ergebnis aussehen?



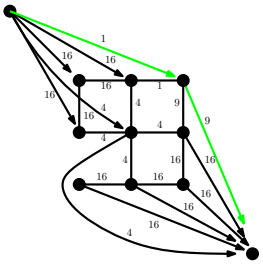
Musterlösung:

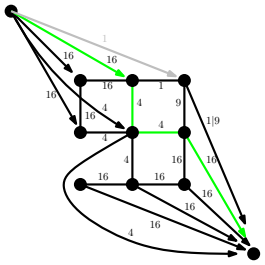
a) Als Transformation ergibt sich folgender Graph. Kanten ohne Kapazität wurden weggelassen.

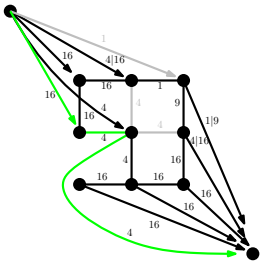


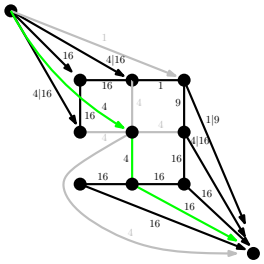


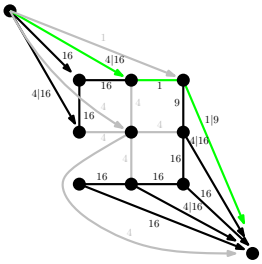
b) Der Algorithmus wird skizziert durch folgende Schritte:

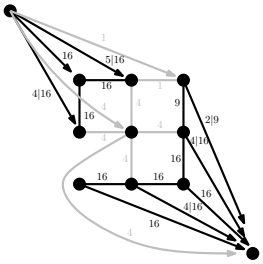


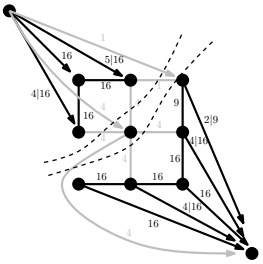




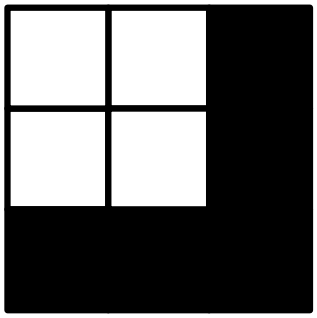


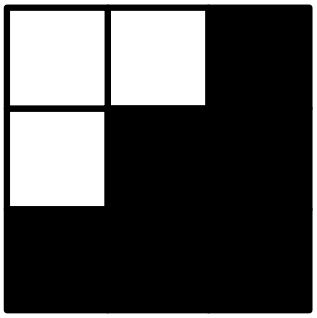






c) Die beiden gleichwertigen Lösungen nach unserer Modellierung sind:





Aufgabe 6 (*Analyse+Entwurf+Rechnen: Grenzüberwachung*)

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

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

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

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

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

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

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

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

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

Musterlösung:

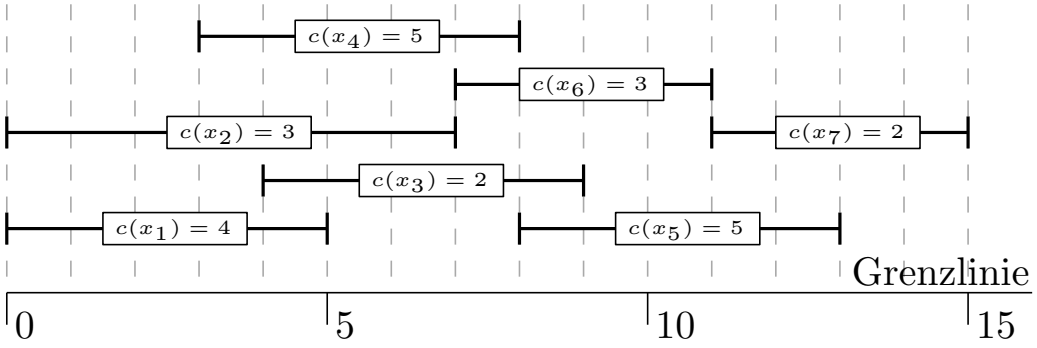
a) Man definiert einen neuen Flussgraph $G' = (V', E')$. Für jeden Knoten $v \in V$ fügt man zwei Knoten v_{in} und v_{out} sowie eine Kante (v_{in}, v_{out}) mit Kapazität $c(v_{in}, v_{out}) = c(v)$ in G' ein. Für jede Kante $(u, v) \in E$ fügt man eine neue Kante (u_{out}, v_{in}) in E' ein. Die Kapazität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

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

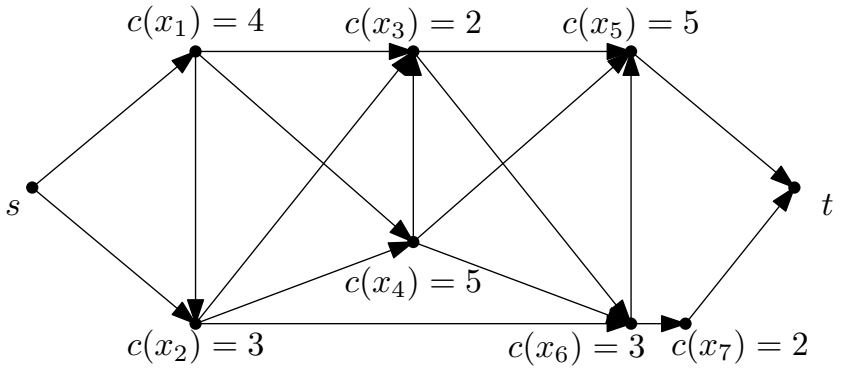
b) Das Flussnetz kann mit Hilfe von Knotenkapazitäten–wie in der letzten Teilaufgabe besprochen–konstruiert werden. Für jeden Sensorknoten x_i fügt man eine Knoten i mit Kapazität $c(i)$ gleich der Batteriekapazität des Sensorknotens ein. Außerdem fügt man eine Quelle s und eine Senke t ein. Anschließend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

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

c) Eingezeichnete Überwachungsbereiche und Batteriekapazitäten der Sensorknoten:



Sich ergebendes Flussnetzwerk:



Musterlösung:

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

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

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