Universität Bremen

FB3: Mathematik/Informatik

Prof. Dr. Daniel Neuen
Jens Schlöter

Wintersemester 2023/2024

Algorithmentheorie

Übungsblatt: Aufgaben zur Klausurvorbereitung

Hinweis: Auf diesem Übungsblatt befindet sich eine Sammlung von Aufgaben, die vom Stil und von der Schwierigkeit her auf einer Klausur vorkommen könnten. Die Sammlung deckt auch noch nicht abgehaltene Vorlesungen mit ab.

Übung 1

[Vorlesung 2] Zeigt oder widerlegt die folgenden Aussagen:

(a) $n^4 + 3n^2 + 7n \in \mathcal{O}(2^{(\log_2(n))^2})$.

(b) Für alle Funktionen $f, g\colon \mathbb{N} \to \mathbb{N}$ gilt $f + g \in \mathcal{O}(\max\{f, g\})$. Hier ist $f + g$ die Kurzschreibweise für die Funktion $h\colon \mathbb{N} \to \mathbb{N}$ mit $h(n) = f(n) + g(n)$ und $\max\{f, g\}$ ist die Kurzschreibweise für die Funktion $h'\colon \mathbb{N} \to \mathbb{N}$ mit $h'(n) = \max\{f(n), g(n)\}$.

(c) $3^n \cdot n^2 \in \Omega(n!)$.

Lösung:

(a) Die Aussage stimmt. Wir müssen zeigen, dass es ein $c \in \mathbb{R}_+$ und ein $n_0 \in \mathbb{N}$ gibt, so dass $n^4 + 3n^2 + 7n \leq c \cdot 2^{(\log_2(n))^2}$ für alle $n \geq n_0$ gilt.

Wir beobachten zunächst, dass $11n^4 \geq n^4 + 3n^2 + 7n$ für alle $n \geq 1$ gilt. Wenn wir ein $n_0 \geq 1$ wählen, dann reicht es also $11n^4 \leq c \cdot 2^{(\log_2(n))^2}$ für ein $c \in \mathbb{R}_+$ und alle $n \geq n_0$ zu zeigen.

Wähle zunächst $c = 11$. Dann gilt

$$\begin{aligned} & 11n^4 \leq c \cdot 2^{(\log_2(n))^2} \\ \Leftrightarrow \quad & 11n^4 \leq 11 \cdot 2^{(\log_2(n))^2} \\ \Leftrightarrow \quad & n^4 \leq 2^{(\log_2(n))^2} \\ \Leftrightarrow \quad & \log_2(n^4) \leq (\log_2(n))^2 \\ \Leftrightarrow \quad & 4 \cdot \log_2(n) \leq (\log_2(n))^2. \end{aligned}$$

Anschließend können wir beobachten, dass $\log_2(n) \geq 4$ für $n \geq 16$. Für $n \geq 16$ folgt also

$$(\log_2(n))^2 = \log_2(n) \cdot \log_2(n) \geq 4 \cdot \log_2(n).$$

Wenn wir also $c = 11$ und $n_0 = 16$ wählen, dann gilt $n^4 + 3n^2 + 7n \leq 11n^4 \leq c \cdot 2^{(\log_2(n))^2}$.

(b) Die Aussage stimmt. Seien $f, g\colon \mathbb{N} \to \mathbb{N}$ beliebig. Es gilt $f(n) \leq \max\{f(n), g(n)\}$ und $g(n) \leq \max\{f(n), g(n)\}$ für alle $n \geq 0$. Daraus folgt, für alle $n \geq 0$, dass $f(n) + g(n) \leq 2 \cdot \max\{f(n), g(n)\}$. Wir können also $n_0 = 0$ und $c = 2$ wählen, dann gilt $f(n) + g(n) \leq c \cdot \max\{f(n), g(n)\}$ für alle $n \geq n_0$. Per Definition gilt dann auch $f + g \in \mathcal{O}(\max\{f, g\})$.

(c) Die Aussage ist falsch. Um $3^n \cdot n^2 \in \Omega(n!)$ zu widerlegen, müssen wir zeigen, dass es für alle $c \in \mathbb{R}_+$ und $n_0 \in \mathbb{N}$ ein $n \geq n_0$ gibt, so dass $3^n \cdot n^2 < c \cdot n!$.

Wir beobachten zunächst, dass $3^n < n!$ für alle $n \geq 7$ gilt. Die Aussage lässt sich per Induktion über n zeigen:

- **IA**: $3^7 = 3^4 \cdot 3^3 = 81 \cdot 27 \leq 120 \cdot 42 = (1 \cdot 2 \cdot 3 \cdot 4 \cdot 5) \cdot (6 \cdot 7) = 7!$.
- **IV**: Nehme an, die Aussage $3^n < n!$ gilt für ein $n \geq 7$.
- **IS**: Betrachte $n + 1$. Es gilt $3^{n+1} = 3^n \cdot 3 < n! \cdot 3 < n! \cdot (n + 1) = (n + 1)!$. Dabei gilt die erste Ungleichung per IV und die zweite Ungleichung gilt, da $n + 1 > 3$.

Seien also $c \in \mathbb{R}_+$ und $n_0 \in \mathbb{N}$ beliebig. Wähle ein $n > \max\{n_0, 10, \frac{27}{c} + 3\}$, dann gilt:

$$
\begin{aligned}
3^n \cdot n^2 &= n^2 \cdot 27 \cdot 3^{n-3} \\
&< n^2 \cdot 27 \cdot (n-3)! && \text{//Wegen } n - 3 \geq 7 \\
&= \frac{n^2 \cdot 27 \cdot n!}{n \cdot (n-1) \cdot (n-2)} \\
&= \frac{n^2 \cdot 27 \cdot n!}{2n - 3n^2 + n^3} = \frac{27 \cdot n!}{\frac{2}{n} - 3 + n} \\
&\leq \frac{27 \cdot n!}{n - 3} \\
&\leq \frac{27 \cdot n!}{\frac{27}{c} + 3 - 3} && \text{//Wegen } n \geq \frac{27}{c} + 3 \\
&\leq c \cdot n!
\end{aligned}
$$

Übung 2

[Vorlesung 3] Gebt für folgende Rekursionsgleichungen eine geschlossene Form in Θ-Notation an und zeigt die Korrektheit per Induktion. Für beide Rekursionsgleichungen könnt ihr annehmen, dass $T(1) = 1$.

(a) $T(n) = T(\frac{n}{2}) + 3$.

(b) $T(n) = 8 \cdot T(\frac{n}{2}) + n^2$.

Anmerkung: In der Klausur würde das Mastertheorem auf der Klausur angegeben werden.

Lösung:

(a) **Ermitteln der geschlossenen Form:** Wir finden die geschlossene Form via Mastertheorem. In diesem Fall sind die Parameter des Theorems $a = 1$, $b = 2$ und $f(n) = 3$.

Der erste Fall des Mastertheorem lässt sich nicht anwenden, da $n^{\log_b(a-\epsilon)} = n^{\log_2(1-\epsilon)}$ für jedes $\epsilon > 0$ einen negativen Exponenten hat (Für $x < 1$ gilt $\log_2(x) < 0$) und damit $f(n) = 3 \notin \mathcal{O}(n^{\log_b(a-\epsilon)})$.

Der zweite Fall des Mastertheorem lässt sich anwenden, da $n^{\log_b(a)} = n^{\log_2(1)} = n^0 = 1$ und somit $f(n) = 3 \in \Theta(1) = \Theta(n^{\log_b(a)})$. Laut Mastertheorem ist die geschlossene Form also:

$$\Theta(n^{\log_b(a)} \cdot \log(n)) = \Theta(\log(n)) = \Theta(\log(2) \cdot \log_2(n)) = \Theta(\log_2(n)).$$

Korrektheitsbeweis: Wir zeigen (i) $T(n) \in \mathcal{O}(\log_2(n))$ und (ii) $T(n) \in \Omega(\log_2(n))$.

Für (i) müssen wir zeigen, dass es ein $n_0 \in \mathbb{N}$ und ein $c \in \mathbb{R}_+$ gibt, so dass $T(n) \leq c \cdot \log_2(n)$ für alle $n \geq n_0$.

Wir wählen $n_0 = 2$ und $c = 4$ und zeigen die Aussage via Induktion über $n \geq 2$.

IA: Es gilt $T(n_0) = T(2) = T\left(\frac{2}{2}\right) + 3 = 4 = 4 \cdot \log_2(2) = c \log_2(n_0)$.

IV: Wir nehmen an, dass $T(m) \leq c \cdot \log_2(m)$ für alle $m < n$.

IS: Betrachte $T(n)$. Es gilt:

$$\begin{aligned}
T(n) &= T\left(\frac{n}{2}\right) + 3 && \text{//Definition } T \\
&\leq c \cdot \log_2\left(\frac{n}{2}\right) + 3 && \text{//IV} \\
&= c \cdot (\log_2(n) - 1) + 3 && \text{//Es gilt } \log_2(n/2) = \log_2(n) - 1 \\
&= c \cdot \log_2(n) - c + 3 \\
&\leq c \cdot \log_2(n) && \text{//} c = 4 \geq 3
\end{aligned}$$

Für (ii) müssen wir zeigen, dass es ein $n_0 \in \mathbb{N}$ und ein $c \in \mathbb{R}_+$ gibt, so dass $T(n) \geq c \cdot \log_2(n)$ für alle $n \geq n_0$.

Wir wählen $n_0 = 2$ und $c = 1$ und zeigen die Aussage via Induktion über $n \geq 2$.

IA: Es gilt $T(n_0) = T(2) = T\left(\frac{2}{2}\right) + 3 \geq 1 = 1 \cdot \log_2(2) = c \log_2(n_0)$.

IV: Wir nehmen an, dass $T(m) \geq c \cdot \log_2(m)$ für alle $m < n$.

IS: Betrachte $T(n)$. Es gilt:

$$\begin{aligned}
T(n) &= T\left(\frac{n}{2}\right) + 3 && \text{//Definition } T \\
&\geq c \cdot \log_2\left(\frac{n}{2}\right) + 3 && \text{//IV} \\
&= c \cdot (\log_2(n) - 1) + 3 && \text{//Es gilt } \log_2(n/2) = \log_2(n) - 1 \\
&= c \cdot \log_2(n) - c + 3 \\
&\geq c \cdot \log_2(n) && \text{//} c = 1 \leq 3
\end{aligned}$$

(b) **Ermitteln der geschlossenen Form:** Wir finden die geschlossene Form via Mastertheorem. In diesem Fall sind die Parameter des Theorems $a = 8$, $b = 2$ und $f(n) = n^2$.

Hier lässt sich direkt der erste Fall des Mastertheorems anwenden. Für $\epsilon = 4$ gilt $n^{\log_b(a-\epsilon)} = n^{\log_2(4)} = n^2$ und damit $n^2 \in \mathcal{O}(n^2) = \mathcal{O}(n^{\log_b(a-\epsilon)})$. Laut Mastertheorem ist die geschlossene Form also

$$\Theta(n^{\log_b(a)}) = \Theta(n^{\log_2(8)}) = \Theta(n^3).$$

Korrektheitsbeweis: Wir zeigen (i) $T(n) \in \mathcal{O}(n^3)$ und (ii) $T(n) \in \Omega(n^3)$.

Für (i) müssen wir zeigen, dass es ein $n_0 \in \mathbb{N}$ und ein $c \in \mathbb{R}_+$ gibt, so dass $T(n) \leq c \cdot n^3$ für alle $n \geq n_0$. Wir zeigen stattdessen die stärkere Aussage, dass $T(n) \leq c \cdot n^3 - n^2$ für alle $n \geq n_0$. Dies impliziert die originale Aussage, da $c \cdot n^3 - n^2 \leq c \cdot n^3$ für alle $n \geq 0$.

Wir wählen $n_0 = 1$ und $c = 2$ und zeigen die Aussage per Induktion über $n \geq n_0$.

IA: Es gilt $T(n_0) = T(1) = 1 = 2 - 1 = c \cdot n_0^3 - n_0^2$.

IV: Wir nehmen an, dass $T(m) \leq c \cdot m^3 - m^2$ für alle $m < n$.

IS: Betrachte $T(n)$. Es gilt:

$$\begin{aligned} T(n) &= 8 \cdot T\left(\frac{n}{2}\right) + n^2 && \text{//Definition } T \\ &\leq 8 \cdot \left(c \cdot \frac{n^3}{2^3} - \frac{n^2}{2^2}\right) + n^2 && \text{//IV} \\ &= c \cdot n^3 - 2n^2 + n^2 \\ &= c \cdot n^3 - n^2 \end{aligned}$$

Für (ii) müssen wir zeigen, dass es ein $n_0 \in \mathbb{N}$ und ein $c \in \mathbb{R}_+$ gibt, so dass $T(n) \geq c \cdot n^3$ für alle $n \geq n_0$.

Wir wählen $n_0 = 1$ und $c = 1$ und zeigen die Aussage per Induktion über $n \geq n_0$.

IA: Es gilt $T(n_0) = T(1) = 1 = c \cdot n_0^3$.

IV: Wir nehmen an, dass $T(m) \geq c \cdot m^3$ für alle $m < n$.

IS: Betrachte $T(n)$. Es gilt:

$$\begin{aligned} T(n) &= 8 \cdot T\left(\frac{n}{2}\right) + n^2 && \text{//Definition } T \\ &\geq 8 \cdot c \cdot \frac{n^3}{2^3} + n^2 && \text{//IV} \\ &= c \cdot n^3 + n^2 \\ &\geq c \cdot n^3 \end{aligned}$$

Übung 3

[**Vorlesung 4**] Gegeben seien zwei Arrays A und B mit jeweils n positiven natürlichen Zahlen. Ziel ist es, die Elemente in den beiden Arrays so zu permutieren, dass $\prod_{i=1}^{n} A[i]^{B[i]}$ maximiert wird. Gebt einen Algorithmus an, der dieses Problem in Laufzeit $\mathcal{O}(n \log n)$ löst. Zeigt, dass euer Algorithmus die optimale Lösung für dieses Problem findet und die gewünschte Laufzeit erzielt.

Lösung: Der Algorithmus sortiert die beiden gegebenen Arrays A und B mittels Mergesort nach aufsteigenden Werten.

Laufzeit : Das Sortieren der Arrays mittels Mergesort benötigt jeweils eine Laufzeit von $\mathcal{O}(n \log n)$. Damit liegt auch die Gesamtlaufzeit in $\mathcal{O}(n \log n)$.

Korrektheit: Die Terminierung folgt aus der Terminierung von Mergesort. Es bleibt zu zeigen, dass der Wert $\prod_{i=1}^{n} A[i]^{B[i]}$ von der berechneten Lösung tatsächlich maximiert wird. Beweis per Widerspruch:

- Nehme an, die Permutationen A und B des Algorithmus sind nicht optimal.
- Dann gibt es optimale Permutationen A^* und B^* mit $\prod_{i=1}^{n} A[i]^{B[i]} < \prod_{i=1}^{n} A^*[i]^{B^*[i]}$.
- Wir können ohne Beschränkung der Allgemeinheit davon ausgehen, dass A und A^* die gleiche Permutation verwenden. Der Grund hierfür ist, dass die Reihenfolge der Faktoren $A^*[i]^{B^*[i]}$ für die Zielfunktion egal ist. Wir können diese Faktoren also so anordnen, dass A^* die gleicher Permutation wie A verwendet, ohne das Produkt der Lösung A^*, B^* zu verkleinern.
- Unter allen sochen optimalen Lösungen, wähle die Lösung in der B^* bis zu einem maximalen Index $\ell - 1$ mit B übereinstimmt.
- Da B aufsteigend sortiert ist, muss also $B^*[\ell] > B[\ell]$ gelten. In B^* muss der Wert $B[\ell]$ stattdessen an einem Index $\ell' > \ell$ vorkommen. Betrachte die alternative Lösung A', B' mit $A' = A = A^*$ und
$$B'[i] = \begin{cases} B^*[\ell'] & \text{falls } i = \ell \\ B^*[\ell] & \text{falls } i = \ell' \\ B^*[i] & \text{sonst.} \end{cases}$$
Dann gilt $A'[\ell]^{B'[\ell]} \cdot A'[\ell']^{B'[\ell']} = A^*[\ell]^{B^*[\ell']} \cdot A^*[\ell']^{B^*[\ell]} \geq A^*[\ell]^{B^*[\ell]} \cdot A^*[\ell']^{B^*[\ell']}$, wobei die Ungleichung aus $A^*[\ell] \leq A^*[\ell']$ und $B^*[\ell] < B^*[\ell']$.
- Da die Faktoren der Lösung A', B' an allen andern Indizes (abgesehen von ℓ und ℓ') genau gleich sind wie in der Lösung A^*, B^* folgt, dass $\prod_{i=1}^{n} A'[i]^{B'[i]} \geq \prod_{i=1}^{n} A^*[i]^{B^*[i]}$.
- Damit ist A', B' auch optimal, ein Widerspruch dazu, dass A^*, B^* bis zu einem maximalen Index mit A, B übereinstimmt.

Übung 4

[Vorlesung 5] Gegeben sind ein einfacher und ungerichteter Graph $G = (V, E)$ sowie zwei Knoten $s, t \in V$. Gebt einen Algorithmus an, der in Laufzeit $\mathcal{O}(|V| + |E|)$ entscheidet, ob es einen Weg von s nach t mit ungerader Länge gibt. Begründet warum euer Algorithmus korrekt ist und die gewünschte Laufzeit erzielt.

Hinweis: Wir haben die Aufgabe geändert. Es wird jetzt nach einem Weg statt einem Pfad gesucht. Die Lösung für Pfade wird etwas komplizierter.

Lösung: Wir erstellen den Hilfsgraphen $G' = (V', E')$ mit $V' = V \cup \{\bar{v} \mid v \in V\}$ und $E' = \{\{v, \bar{u}\} \mid \{v, u\} \in E\} \cup \{\{\bar{v}, u\} \mid \{v, u\} \in E\}$. Wir fügen also für jeden originalen Knoten v eine Kopie $\bar{v}$ hinzu und ersetzen jede originale Kante $\{v, u\}$ mit zwei Kanten $\{v, \bar{u}\}$ und $\{u, \bar{v}\}$. Dieser Graph hat $2 \cdot |E|$ Kanten und $2 \cdot |V|$ Knoten und kann somit in Laufzeit $\mathcal{O}(|E| + |V|)$ erstellt werden.

Wenn es im originalen Graph G einen Weg $P = (v_1, \ldots, v_k)$ mit $v_1 = s$, $v_k = t$ und k gerade gibt (wenn k gerade ist, dann ist die Anzahl der Kanten in P ungerade), dann können wir einen ungeraden s-$\bar{t}$-Weg P' in G' erzeugen, indem wir jeden zweiten Knoten v_i in P (angefangen mit v_2) durch die Kopie $\bar{v}_i$ ersetzen.

Andersrum können wir einen s-$\bar{t}$-Weg P' in G' in einen s-t-Weg P in G übersetzen, indem wir jeden zweiten Knoten $\bar{v}$ auf diesem Weg durch das Original v ersetzen. Per Konstruktion von G' muss jeder zweite Knoten in P' eine Kopie sein. Außerdem gilt, dass jeder s-$\bar{t}$-Weg in G' ungerade Länge hat. Das folgt daraus, dass G' ein bipartiter Graph zwischen den Mengen V und $\{\bar{v} \mid v \in V\}$ ist. Weil P und P' gleich lang sind, hat P dann also auch ungerade Länge.

Es gibt also genau dann einen s-t-Weg ungerader Länge in G, wenn es einen s-$\bar{t}$-Weg in G' gibt.

Unser Algorithmus kann also einfach den Graph G' erstellen und mittels Breitensuche entscheiden, ob es einen s-$\bar{t}$-Weg in G' gibt. Wenn ja, dann gibt der Algorithmus true zurück und ansonsten false.

Laufzeit: Das Erstellen von G' hat Laufzeit $\mathcal{O}(|E| + |V|)$, wie oben bereits argumentiert wurde. Die Breitensuche auf G' hat Laufzeit $\mathcal{O}(|E'| + |V'|) \subseteq \mathcal{O}(|E| + |V|)$. Die Gesamtlaufzeit ist also in $\mathcal{O}(|E| + |V|)$.

Korrektheit: Die Terminierung folgt aus der Terminierung der Breitensuche. Die Korrektheit folgt aus der Korrektheit der Breitensuche und daraus, dass es einen ungeraden s-t-Weg in G gibt gdw. es einen s-$\bar{t}$-Weg in G' gibt. Letzteres haben wir oben bereits begründet.

Übung 5

[Vorlesungen 6 und 7] Gegeben sei ein zusammenhängender Graph $G = (V, E)$ mit der Kostenfunktion $c\colon E \to \mathbb{R}$. Sei T ein MST von G. Für welche der folgenden Kostenfunktionen ist T immer noch ein MST von G? Begründet Eure Antworten.

(i) $c_1\colon E \to \mathbb{R}$ mit $c_1(e) = c(e) + 17$ für alle $e \in E$.

(ii) $c_2\colon E \to \mathbb{R}$ mit $c_2(e) = 17 \cdot c(e)$ für alle $e \in E$.

(iii) $c_3\colon E \to \mathbb{R}$ mit $c_3(e) = c(e) \cdot c(e)$ für alle $e \in E$.

Lösung:

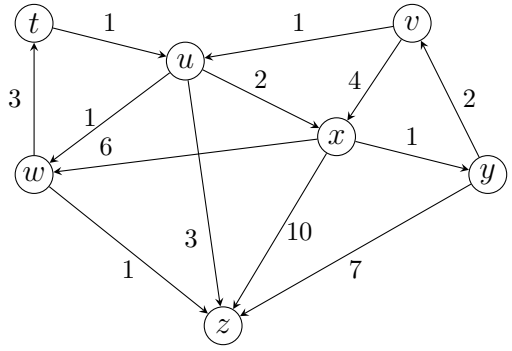
(i) T ist dann auch ein MST für c_1. Für jeden beliebigen Spannbaum T' von G gilt, dass $c_1(T') = \sum_{e \in T'} c_1(e) = \sum_{e \in T'} c(e) + 17 = c(T') + (n-1) \cdot 17$, da jeder Spannbaum genau $n-1$ Kanten hat. Weil der additive Term $(n-1) \cdot 17$ in den c_1-Kosten aller Spannbäume vorkommt, können wir beobachten, dass T' die Kosten $c_1(T') = c(T') + 17(n-1)$ minimiert gdw. T' die Kosten $c(T')$ minimiert.

(ii) T ist dann auch ein MST für c_2. Für jeden beliebigen Spannbaum T' von G gilt, dass $c_2(T') = \sum_{e \in T'} c_2(e) = \sum_{e \in T'} 17 \cdot c(e) = 17 \cdot c(T')$. Es folgt, dass T' die Kosten $c_2(T') = 17 \cdot c(T')$ minimiert gdw. T' die Kosten $c(T')$ minimiert.

(iii) T ist nicht zwangsläufig ein MST für c_3. Betrachtet als Gegenbeispiel den Graph G, der aus einem einfachen Kreis mit den drei Kanten e_1, e_2, e_3 und Kantenkosten $c(e_1) = -2$ und $c(e_2) = c(e_3) = 1$ besteht. Jeder MST für die Kantenkosten c muss e_1 enthalten, da e_1 eindeutig minimal in einem Schnitt ist. Für die Kantenkosten c_1 gilt stattdessen $c_1(e_1) = 4$ und $c_1(e_2) = c_2(e_2) = 1$. D.h., kein MST für die Kantekosten c_1 kann e_1 enthalten, da e_1 eindeutig maximal auf einem Kreis ist.

Übung 6

[Vorlesung 8] Ermittelt für den abgebildeten Graphen mit Hilfe des Algorithmus von Dijkstra die kürzesten Wege (und deren Länge) von Startknoten x zu allen anderen Knoten im Graphen. Gebt in nachvollziehbarer Art und Weise die Zwischenschritte an.



Lösung: Wir wenden den Algorithmus von Dijskstra an. Die folgende Tabelle zeigt die Menge der eingefrorenen S, die Distanzlabel der Knoten und die Vorgänger nach jeder Iteration. Die kürzesten Wege und deren Längen können aus der letzten Zeile abgelesen werden:

	Menge S	x	y	z	v	u	w	t
	$\emptyset$	0	∞	∞	∞	∞	∞	∞
1.	$\{x\}$	0	1 (pred: x)	10 (pred: x)	∞	∞	∞	∞
2.	$\{x, y\}$	0	1 (pred: x)	8 (pred: y)	3 (pred: y)	∞	∞	∞
3.	$\{x, y, v\}$	0	1 (pred: x)	8 (pred: y)	3 (pred: y)	4 (pred: v)	∞	∞
4.	$\{x, y, v, u\}$	0	1 (pred: x)	7 (pred: u)	3 (pred: y)	4 (pred: v)	5 (pred: u)	∞
5.	$\{x, y, v, u, w\}$	0	1 (pred: x)	6 (pred: w)	3 (pred: y)	4 (pred: v)	5 (pred: u)	8 (pred: w)
6.	V	0	1 (pred: x)	6 (pred: w)	3 (pred: y)	4 (pred: v)	5 (pred: u)	8 (pred: w)

Übung 7

[Vorlesung 9] Gegeben sei eine Menge W von natürlichen Zahlen. Für jedes $w \in W$ habt ihr eine unbegrenzte Anzahl an Münzen mit dem Wert w zur Verfügung. Zusätzlich ist eine natürliche Zahl M gegeben, die als Wechselgeld ausgegeben werden soll.

(a) Gebt ein dynamisches Programm an, welches die minimale Anzahl an Münzen berechnet, die benötigt wird, um den Betrag M auszugeben. Es kann vorkommen, dass es nicht möglich ist den Betrag auszugeben, in dem Fall soll ∞ zurückgegeben werden. Die Laufzeit des Algorithmus soll $\mathcal{O}(M \cdot |W|)$ sein. Gebt dabei insbesondere die induktive Definition der DP-Tabelle an.

(b) Begründet warum euer Algorithmus korrekt ist und die gewünschte Laufzeit erzielt.

(c) Erklärt, wie aus der gefüllten DP-Tabelle berechnet werden kann, welche Münzen ausgegeben werden sollen, um die Anzahl der Münzen zu minimieren.

Lösung:

(a) Wir erstellen eine eindimensionale DP-Tabelle T, welche einen Eintrag für jedes $i \in \{0, \ldots, M\}$ hat.

Der Eintrag $T[i]$ für ein $i \in \{0, \ldots, M\}$ soll am Ende die minimale Anzahl an Münzen speichern, die benötigt werden um i als Wechselgeld auszugeben.

Sei $w_{min} = \min_{w \in W} w$. Die Basisfälle für die Berechnung unserer Tabelle sind:

- $T[0] = 0$, weil der Geldbetrag 0 mit 0 Münzen ausgegeben werden kann.
- $T[i] = \infty$ für alle $0 < i < w_{min}$, weil solche Geldbeträge i größer als der kleinste Münzwert sind und somit nicht ausgegeben werden können.

Für alle $i \in \{w_{min}, \ldots, M\}$ verwenden wir die folgende induktive Definition:

$$T[i] = \min_{w \in W \colon w \leq i} 1 + T[i - w].$$

Mit Hilfe der induktiven Definition kann T dann mittels Tabularization oder Memoization berechnet werden. Der Eintrag $T[M]$ enthält dann die minimale Anzahl an Münzen, die benötigt werden, um M auszugeben.

(b) **Korrektheit:** Die Korrektheit der Basisfälle wurde in (a) bereits begründet. Es bleibt also der induktive Fall mit $i \in \{w_{min}, \ldots, M\}$:

$$T[i] = \min_{w \in W \colon w \leq i} 1 + T[i - w].$$

Die erste Beobachtung ist hier, dass keine Münzen mit Wert $w > i$ verwendet werden können um den Betrag i auszugeben. Auf der anderen Seite muss mindestens eine Münze mit Wert $w \in W$ und $w \leq i$ verwendet werden, um den Betrag i auszugeben. Wenn eine Münze mit Wert $w \leq i$ gewählt wird, dann erhöht dies die Anzahl der verwendeten Münzen um eins und zusätzlich muss noch der Betrag $i - w$ ausgegeben werden. Unter den Annahmen, dass die Münze mit Wert w gewählt wird und der Eintrag $T[i-w]$ richtig

berechnet wurde, ist die optimale Anzahl an Münzen für den Betrag i also $1+T[i-w]$. Der Minimum-Operator sorgt dafür, dass unter allen Münzen die Münze gewählt wird, welche zur minimalen Anzahl an Rückgabemünzen führt. Da der induktive Fall nur Beträge $i \geq w_{\min}$ betrachtet, ist das Minimum wohldefiniert.

Dies impliziert auch, dass der Algorithmus terminiert wenn Tabularization oder Memoization für die Berechnung von T verwendet werden.

Laufzeit: Die DP-Tabelle T hat insgesamt $M+1$ Einträge. Die Berechnung eines Eintrags kann mit Laufzeit $\mathcal{O}(|W|)$ durchgeführt werden, da der Minimum-Operator eine Laufzeit von $\mathcal{O}(|W|)$ benötigt und alle weiteren Operationen konstante Laufzeit haben. Da der Algorithmus abgesehen von der Berechnung von T keine weiteren Operationen durchführt, ist die Laufzeit also in $\mathcal{O}(M \cdot |W|)$.

(c) Die auszugebenen Münzen können mit Hilfe von T via Backtracking berechnet werden:

Algorithmus 1 : Backtracking

```
1 A ← An empty list;
2 i ← M;
3 if T[i] = ∞ then return Failure;
4 while i > 0 do
5 |   for w ∈ W do
6 |   |   if w ≤ i and T[i] = 1 + T[i − w] then
7 |   |   |   Append w to list A;
8 |   |   |   i ← i − w;
9 return A;
```

Hinweis: In der Klausur wäre für eine solche Aufgabe nicht zwingend Pseudocode erforderlich. Ihr könnt das Backtracking genauso gut schriftlich erklären.

Übung 8

[Vorlesung 10] Ihr arbeitet bei einem Fahrrad-Hersteller und wollt einen Produktionsplan für die nächsten n Monate erstellen. Für jeden Monat i, kennt ihr den Bedarf d_i an Fahrrädern. Ihr wisst also im voraus, wie viele Fahrräder ihr verkaufen werdet. Ihr müsst nun sicherstellen, dass ihr ausreichend Fahrräder produziert, um den Bedarf in jedem Monat zu decken.

In jedem Monat können eure Angestellten m Fahrräder produzieren, ohne dass zusätzliche Kosten entstehen. Für jedes zusätzlich produzierte Fahrrad zahlt ihr einen Stückpreis von c Geldeinheiten. Falls am Ende eines Monats noch unverkaufte Fahrräder übrig sind, müsst ihr zusätzliche Lagerungskosten bezahlen. Die Lagerungskosten hängen von der Anzahl der zu lagernden Fahrräder ab und werden durch die Funktion $h(j)$ beschrieben, die für jedes $j \in \{1, \ldots, \sum_{i=1}^{n} d_i\}$ angibt, wie viel es kostet j Fahrräder zu lagern. Dabei gilt $h(j) \geq 0$ für alle $1 \leq j \leq \sum_{i=1}^{n} d_i$ und $h(j) \leq h(j+1)$ für alle $1 \leq j \leq (\sum_{i=1}^{n} d_i) - 1$

Gebt ein dynamisches Programm an, das einen Produktionsplan berechnet, der alle Bedarfe erfüllt und die zusätzlichen Kosten minimiert. Die Laufzeit des Algorithmus soll polynomiell in n und $(\sum_{i=1}^{n} d_i)$ sein. Gebt insbesondere auch die induktive Definition der DP-Tabelle an. Begründet warum euer Algorithmus die optimale Lösung findet und die gewünschte Laufzeit erzielt.

Lösung: Wir verwenden eine zweidimensionale DP-Tabelle T mit Einträgen $T[i,j]$ für alle $i \in \{1,\dots,n\}$ und $j \in \{0,\dots,D\}$ für $D = \sum_{i=1}^{n} d_i$.

Der Eintrag $T[i,j]$ soll am Ende die minimalen Zusatzkosten enthalten, die nötig sind um die Bedarfe der ersten i Monate zu decken und am Ende von Monat i noch j Fahrräder übrig zu haben. Die Lagerkosten für die j zusätzlichen Fahrräder sollen dabei mit enthalten sein. Wir beschränken uns hier auf $j \leq D$, weil zusätzliche übrige Fahrräder nie nötig sind um alle Bedarfe zu decken.

Am Ende enthält dann $T[n,0]$ die optimalen Zusatzkosten (im letzten Monat macht es keinen Sinn noch Fahrräder übrig zu behalten).

Für die Berechnung der Tabelle betrachten wir zunächst die Basisfälle $i = 1$ und $j \in \{0,\dots,D\}$. In diesem Fall müssen mindestens $d_1 + j$ Fahrräder produziert werden, damit der Bedarf des ersten Monats gedeckt ist und anschließend noch j Fahrräder übrig sind. Die ersten m produzierten Fahrräder verursachen dabei keine Kosten. Die übrigen $d_1 + j - m$ Fahrräder verursachen jeweils Kosten c (sofern $d_1 + j - m > 0$). Es gilt also

$$T[1,j] = \max\{c \cdot (d_1 + j - m), 0\} + h(j).$$

Betrachen wir als nächstes den induktiven Fall mit $i > 1$ und $j \in \{0,\dots,D\}$. Für die Lösung von Eintrag $T[i,j]$ werden wieder $d_i + j$ Fahrräder benötigt, allerdings können manche dieser Fahrräder aus dem Überschuss des Vormonats kommen.

Unter der Annahme, dass wir $j' \leq d_i + j$ Fahrräder aus dem Vormonat verwenden, müssen noch $d_i + j - j'$ Fahrräder neu produziert werden. Von diesen Fahrrädern verursachen die ersten m wieder keine Kosten und die übrigen Fahrräder verursachen Stückkosten von c. Die optimalen Kosten um am Ende des Vormonats $i-1$ noch j' Fahrräder übrig zu haben sind $T[i-1,j']$. Die optimalen Kosten um am Ende von Monat i noch j Fahrräder übrig zu haben sind unter der Annahme, dass j' Fahrräder aus dem Vormonat übrig, sind also

$$\max\{c \cdot (d_i + j - j' - m), 0\} + h(j) + T[i-1,j'].$$

Für die induktive Definition bleibt jetzt noch das optimale j' auszuwählen:

$$T[i,j] = \min_{j' \in \{0,\dots,d_i+j\}} \left(\max\{c \cdot (d_i + j - j' - m), 0\} + h(j) + T[i-1,j']\right).$$

Die DP-Tabelle kann via Tabularization oder Memoization berechnet werden. Der zugehörige Produktionsplan kann wie üblich per Backtracking berechnet werden.

Laufzeit: Die Tabelle hat $\mathcal{O}(n \cdot D)$ Einträge. Die Berechnung eines Eintrages kann in Laufzeit $\mathcal{O}(D)$ durchgeführt werden, da der Minimum-Operator eine Laufzeit von $\mathcal{O}(D)$ benötigt und alle anderen Operationen eine konstante Laufzeit haben. Die Berechnung der Tabelle hat also Laufzeit $\mathcal{O}(n \cdot D^2)$. Da die Berechnung der Tabelle die Laufzeit des Algorithmus dominiert, folgt, dass die Laufzeit des Algorithmus polynomiell in n und D ist.

Korrektheit: Die Begründung der Korrektheit ist bereits bei der Definition der Tabelle erfolgt. Terminierung folgt aus der Definition von T sowie der Verwendung der Standardtechniken Tabularization/Memoization und Backtracking.

Übung 9

[Vorlesungen 11 und 12] Du leitest ein Bauunternehmen und wurdest beauftragt ein Haus zu renovieren. Für die Renovierung des Hauses ist eine Liste von Aufgaben zu erledigen, die jeweils von genau einer Person bearbeitet werden müssen. Dafür steht ein Team von Angestellten zur Verfügung, doch leider kann nicht jedes Teammitglied jede Aufgabe bearbeiten. Für

jede Aufgabe werden bestimmte Fähigkeiten benötigt, die nicht unbedingt jedes Teammitglied besitzt. Wegen der aktuell extremen Hitzewelle schafft es jedes Teammitglied an einem Tag höchstens eine Aufgabe zu bearbeiten. Ziel ist es eine Aufgabenverteilung zu finden, welche es erlaubt an einem Tag möglichst viele Aufgaben zu erledigen. Die folgenden Tabellen spezifizieren das Team, die Aufgaben, und die vorhandenen/benötigten Fähigkeiten.

Aufgabe	Benötigte Fähigkeiten
A1	F1
A2	F2, F3
A3	F3, F4
A4	F4, F5
A5	F6

Teammitglied	Vorhandene Fähigkeiten
TM1	F2, F3, F4
TM2	F2, F3, F4
TM3	F1, F3, F4, F5
TM4	F4, F5, F6
TM5	F4, F5

a) Modelliere das Problem als Optimierungsproblem auf Graphen. Zeichne dafür den resultierenden Graphen und beschrifte ihn.

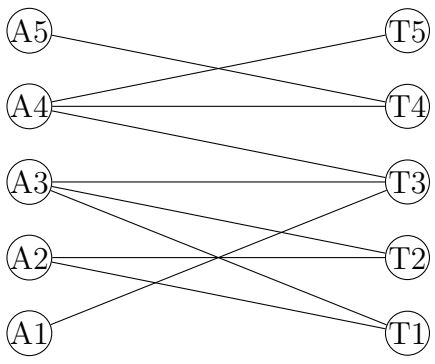
b) Um welches Problem aus der Vorlesung handelt es sich? Gib eine formale Definition an.

c) Betrachte die folgende Aufgabenverteilung: TM2 bearbeitet A2, TM3 bearbeitet A3 und TM4 bearbeitet A4. Ist diese Aufgabenverteilung optimal? Gib ein allgemeines Optimalitätskriterium an, anhand dessen man entscheiden kann ob eine gegebene Aufgabenverteilung optimal ist.

d) Benutze einen aus der Vorlesung bekannten Algorithmus um ausgehend von der Aufgabenverteilung aus Aufgabenteil c) eine optimale Aufgabenverteilung zu finden. Der Algorithmus soll die Aufgabenverteilung aus c) als Startlösung verwenden, darf sie aber im Verlauf seiner Ausführung verändern. Gib dabei in nachvollziehbarer Weise Zwischenschritte und Erläuterungen an.

Lösung:

(a) Wir erstellen einen Graphen, der für jede Aufgabe sowie für jedes Teammitglied einen Knoten hat. Wir fügen eine Kante zwischen einer Aufgabe und einem Teammitglied hinzu gdw. das Teammitglied die Fähigkeiten besitzt, die zur Bearbeitung der Aufgabe benötigt werden. In diesem bipartiten Graph suchen wir dann ein kardinalitätsmaximales Matching. Eine Kante in diesem Matching zwischen einem Teammitglied und einer Aufgabe bedeutet, dass das Teammitglied die Aufgabe bearbeiten soll. Mit einem Kardinalitätsmaximalen Matching finden wir die Zuordnung, welche die Bearbeitung der meisten Aufgaben erlaubt.



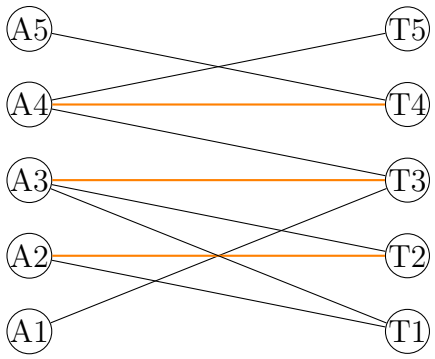
Hinweis: Die Aufgabenstellung kann genauso auch über das maximale Flussproblem gelöst werden. Das Netzwerk kann aus dem obigen Graph via der aus der Vorlesung bekannten Reduktion konstruiert werden. Die Antworten zu den folgenden Teilaufgaben müssen dann entsprechend auf das maximale Flussproblem angepasst werden.

(b) Kardinalitätsmaximales Matching Problem:

Gegeben: Ein bipartiter Graph $G = (V, E)$.

Gesucht: Ein Kardinalitätsmaximales Matching M in G, d.h., ein Matching M, so dass für alle Matchings M' in M gilt: $|M| \geq |M'|$.

(c) Die gegebene Aufgabenverteilung ist in folgendem Graph orange markiert:



Allgemeines Optimalitätskriterium: Ein Matching M ist kardinalitätsmaximal genau dann, wenn es keinen M-augmentierenden Pfad gibt.

Ein M-augmentierender Pfad ist ein Pfad, welcher abwechselnd Kanten innerhalb und außerhalb von M verwendet und dessen Start- und Endknoten zu keiner Kante in M inzident sind.

Optimalität der gegebenen Zuordnung: Für das gegebene Matching existiert der M-augmentierende Pfad $(A1, T3, A3, T1)$. Das Matching ist also nicht optimal.

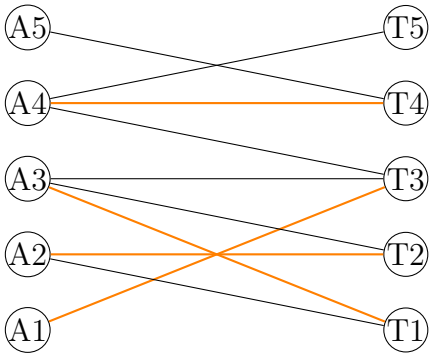
(d) Wir verwenden den Algorithmus aus der Vorlesung für Kardinalitätsmaximale Matchings in bipartite Graphen. Dieser Algorithmus startet mit einem Matching M. Solange ein M-augmentierender Pfad P existiert, aktualisiert der Algorithmus das Matching zu $M = M \Delta E(P)$ wobei $E(P)$ die Kantenmenge von Pfad P ist. Wenn kein M-augmentierender Pfad P mehr existiert, dann gibt der Algorithmus das aktuelle Matching zurück.

Anwendung des Algorithmus:

(i) Das Startmatching ist $M = \{\{A2, T2\}, \{A3, T3\}, \{A4, T4\}\}$.

(ii) Es existiert der M-augmentierende Pfad $(A1, T3, A3, T1)$. Das aktualisierte Matching ist dann

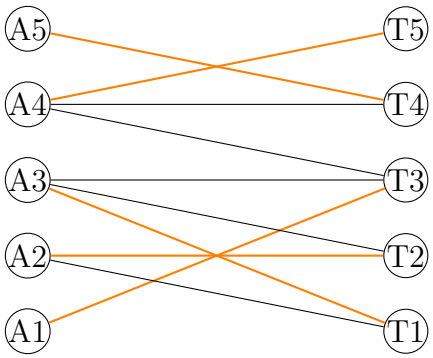
$$M = M \Delta \{\{A1, T3\}, \{A3, T_3)\}, \{A3, T1\}\} = \{\{A2, T2\}, \{A1, T3\}, \{A3, T1\}, \{A4, T4\}\}.$$



Hinweis: In der Klausur müsstert ihr den Graph nicht zwangsläufig jedes mal neu zeichnen, die Angabe der augmentierenden Wege und der aktualisierten Matchings reicht aus.

(iii) Es existiert der M-augmentierende Pfad $(A5, T4, A4, T5)$. Das aktualisierte Matching ist dann

$$\begin{aligned} M &= M\Delta\{\{A5, T4\}, \{A4, T4\}, \{A4, T5\}\} \\ &= \{\{A5, T4\}, \{A3, T1\}, \{A4, T4\}, \{\{A5, T4\}, \{A4, T5\}\}. \end{aligned}$$



(iv) Danach gibt es keinen M-augmentierenden Pfad mehr und der Algorithmus gibt das aktuelle Matching zurück.