3. Musterlösung

Problem 1: HEAPSORT **

Algorithmus 1 : HEAPSORT (A)

Eingabe : Array A der Länge n
Ausgabe : Aufsteigend sortiertes Array A

```
1 MAKEHEAP(A)
2 Für i = n, ..., 2
3 |  Vertausche A[1] und A[i]
4 |  HEAP-Größe(A) ← HEAP-Größe(A) − 1
5 |  HEAPIFY (A, 1)
```

Man betrachte die obere Hälfte des Baumes, also diejenigen Knoten mit Tiefe $\leq \lfloor \log n \rfloor/2$. Diese obere Hälfte beherbergt $2^{\lfloor \lfloor \log n \rfloor/2 \rfloor} - 1 \leq (2^{\lfloor \log n \rfloor})^{1/2} \leq \sqrt{n}$ Knoten. Ferner sei $\mathcal{H}$ die Menge der $n/2$ größten Elemente.

(a) Was ist die Laufzeit von HEAPSORT, angewendet auf ein Array A der Länge n, das bereits aufsteigend sortiert ist? Wie verhält es sich, wenn A schon absteigend sortiert ist?

Lösung. Aufsteigend sortiert: Am Anfang befinden sich die $n/2$ größten Elemente in den Blättern. Durch MAKEHEAP werden mindestens die Hälfte davon, also $n/4$, weiter nach oben getauscht. Die Menge dieser Elemente sei $\mathcal{G}$. Betrachten wir nun die ersten $n/2$ Durchläufe der Schleife in Zeile $2-5$. Während dieser Zeit werden die $n/2$ größten Elemente aus dem HEAP entfernt. Die $n/4$ Elemente aus $\mathcal{G}$ müssen in dieser Zeit (oder zum Teil auch schon im Verlauf von MAKEHEAP) also durch HEAPIFY nach und bis nach $A[1]$ aufsteigen. Dazu sind mindestens $n/4 \cdot ((\log n) - 1) = \Omega(n \log n)$ Vergleiche notwendig. Die Laufzeit von HEAPSORT ist also hier in $\Omega(n \log n)$.

Absteigend sortiert: Nach MAKEHEAP befinden sich die $n/2$ kleinsten Elemente in den Blättern und werden nun in Zeile 3 (für $n \geq i \geq n/2$), beginnend beim kleinsten, ganz nach oben getauscht. Da sie in dieser Zeit nicht aus dem HEAP entfernt werden, müssen aus Platzmangel mindestens $n/2 - \sqrt{n}$ von ihnen tiefer als $(\log n)/2$ sinken. Dadurch ergibt sich ein Gesamtaufwand von mindestens $(n/2 - \sqrt{n}) \cdot (\log n)/2 = \Omega(n \log n)$. □

(b) Zeigen Sie, dass die Laufzeit von HEAPSORT in $\Omega(n \log n)$ ist.

Lösung. HEAPSORT ist ein Sortieralgorithmus. Nach dem Satz aus der Vorlesung kann ein Sortieralgorithmus nicht schneller als $\Omega(n \log n)$ sein, somit auch nicht HEAPSORT. Dies gilt aber nur für den Worst-case. Wir sind hier aber an einer Aussage über den Best-case interessiert.

Unser Ziel ist es, das Argument für den Fall der aufsteigend sortierten Folge zu verallgemeinern. Dazu werden wir beweisen, dass sich in einem HEAP sehr viele ($\Omega(n)$) der größeren

Hälfte der Werte oberhalb der Blätter befinden.

Betrachten wir also die Menge $\mathcal{H}$ im fertigen HEAP nach MAKEHEAP. Wenn sich von diesen mindestens die Hälfte (also $n/4$) oberhalb der Blätter befindet, dann sind mindestens $n/4-\sqrt{n}$ Elemente tiefer als $(\log n)/2$ aber keine Blätter. Die ersten $n/2$ Aufrufe von HEAPIFY haben also zusammen einen Aufwand von mindestens $(n/4 - \sqrt{n}) \cdot (\log n)/2 = \Omega(n \log n)$.

Nehmen wir dagegen an, dass mindestens $n/4$ Elemente aus $\mathcal{H}$ Blätter sind, dann haben alle Vorfahren eines solchen Blattes aufgrund der HEAP-Eigenschaft einen größeren Wert. Schon in der Schicht direkt oberhalb der Blätter befinden sich demnach mindestens $n/8$ Elemente aus $\mathcal{H}$. Für diese ist der Aufwand von HEAPIFY zusammen wiederum mindestens $(n/8 - \sqrt{n}) \cdot (\log n)/2 = \Omega(n \log n)$.

Folglich ist der Aufwand von HEAPSORT in jedem Fall in $\Omega(n \log n)$. □

(c) Geben Sie eine $O(\log n)$-Implementation der Prozedur HEAP-INCREASEKEY(A, i, k) an, welche $A[i]$ auf $\max(A[i], k)$ setzt und dann die HEAP-Struktur entsprechend aufrechterhält.

Lösung. Folgender Algorithmus implementiert das Geforderte. Die Korrektheit, sowie die Laufzeit ergeben sich direkt aus SIFT-UP.

Algorithmus 2 : HEAP-INCREASEKEY(A, i, k)

Eingabe : Array A als HEAP, Index i, Wert k
Ausgabe : Array A (Wert $A[i]$ aktualisiert) als HEAP
$A[i] \leftarrow \max(k, A[i])$
SIFT-UP (A, i)

□

(d) Geben sie einen $O(n \log k)$ Algorithmus an, welcher k sortierte Listen in eine sortierte Liste verschmilzt, wobei n die Gesamtzahl aller Elemente ist. Benutzen sie dazu einen HEAP.

Lösung. Wir erweitern die HEAP-Datenstruktur aus der Vorlesung so, dass die Elemente des HEAPs Paare aus einem *Schlüssel* und einem *Wert* sind. Der Schlüssel $A[i]$.Key ist relevant für die HEAP-Struktur, dort speichern wir das Element wie bisher. Der Wert $A[i]$.Value beinhaltet ein zusätzliche Informtion des Elements, in diesem Fall der Index der Liste aus dem dieses Element stammt. Das i-te Element der j-ten Liste sei $L[j][i]$: Weiterhin speichert CurrentElement$[j]$ den Index des nächsten Elements, welches aus der Liste j entnommen werden sollte:

Algorithmus 3 : SORTEDLISTS-MERGE($L[1], \dots, L[k]$)

Eingabe : Sortierte Listen $(L[1], \dots, L[k]), |\bigcup_i L[i]| = n$
Ausgabe : Sortierte Liste B mit allen Elementen

```
1  A ← MAKEHEAP{}
2  Für j = 1, ..., k
3  |  INSERT(A, L[j][1])
4  |  CurrentElement[j] ← 2
5  Für i=1, ..., n
6  |  B[i] ← A[1].Key
7  |  usedlist ← A[1].Value
8  |  DELETE(A, 1)
9  |  HEAPIFY (A, 1)
10 |  Wenn L[usedlist][CurrentElement[usedlist]] exisitert
11 |  |  INSERT(A, L[usedlist][CurrentElement[usedlist]])
```

Korrektheit: In der Hauptschleife des Algorithmus gilt folgende Invariante: Der HEAP A beinhaltet stets die größten Elemente jeder Liste. Das größte Element in A wird dann in B geschrieben und durch das nächstgrößte Element der entsprechenden Liste ersetzt, sofern vorhanden.

Laufzeit:

- Zeile **1** MAKEHEAP in $O(k)$
- Zeilen **2-4** INSERT in $O(k \log k)$
- Zeilen **5-11** Es wird $O(n)$ mal DELETE, HEAPIFY und INSERT in einem HEAP der Größe k durchgeführt $\Rightarrow O(n \log k)$

Somit liegt die Gesamtlaufzeit in $O(n \log k)$. □

Problem 2: Bäume *

Definition 1. *Ein* Baum *ist ein Graph, in dem zwischen je zwei Knoten genau ein Pfad existiert.*

(a) In einem Baum mit n Knoten gibt es genau $n-1$ Kanten.

Lösung. Wir beweisen die Aussage durch Induktion über die Anzahl n der Knoten eines Baumes.

- Induktionsanfang ($n = 1$):
  Der Graph besteht aus einem einzigen Knoten und hat $n - 1 = 1 - 1 = 0$ Kanten, da Selbstschleifen wegen der Einfachheit des Graphen nicht erlaubt sind.

- Induktionsvoraussetzung ($1 \leq k \leq n$):
  Jeder Baum mit k Knoten habe genau $k - 1$ Kanten.

- Induktionsschluss ($n \to n + 1$):
  Sei $G = (V, E)$ ein Baum mit $n+1$ Knoten. Betrachte einen Endknoten v eines maximalen Pfades $P = (w, \dots, u, v)$ in G. Falls der Grad von v größer gleich 2 ist, existiert eine Kante $\{v, u'\} \neq \{v, u\} \in E$. Der Pfad $P' := P \cup \{v, u'\}$ wäre länger als P, was der

Maximalität von P widerspricht oder $u' \in P$, wodurch es zwei Pfade von v nach u' gäbe, im Widerspruch zur Baumeigenschaft. Also hat v Grad 1.
Wir betrachten die Graphen $G' = (V \setminus \{v\}, E \setminus \{u, v\})$. Offensichtlich ist G' ein Baum mit n Knoten und hat nach Induktionsvoraussetzung genau $n-1$ Kanten. Der ursprüngliche Graph G ensteht aus G' durch Hinzunahme der Kante $\{u, v\}$. Er enthält somit $n+1$ Knoten und $(n-1)+1 = n$ Kanten. □

(b) Ein minimaler aufspannender Teilgraph ist immer ein Baum

Lösung. Wir führen einen Widerspruchsbeweis.
Sei $G := (V, E)$ ein Graph. Nehmen wir an, dass $T := (V_T, E_T)$ ein minimaler aufspannender Teilgraph von G sei, der kein Baum ist. Es existieren somit zwei Knoten $v, w \in V_T$, zwischen denen mindestens zwei verschiedene Pfade (als Kantenmengen betrachtet) $P_1, P_2 \subset E_T$ existieren.
Da P_1 und P_2 verschieden sind, existiert mindestens eine Kante $e = \{u, u'\} \in P_2 \setminus P_1$. Sei $T' := (V_T, E_T \setminus e)$.
Der Teilgraph T' enthält weniger Kanten als T und spannt G auf, wie wir im folgenden zeigen.
Zu nächst führen wir folgende Notation ein: Es bezeichne $P_{u,v}$ ein Teilpfad zwischen u und v in T. Für je zwei Punkte $u_1, u_2 \in V$ gilt:

- Falls der Pfad von u_1 nach u_2 in T die Kante e nicht enthält, dann ändert sich nichts durch das Löschen von e.

- Falls der Pfad P von u_1 nach u_2 in T die Kante e enthält, dann hat P o.B.d.A folgende Form (siehe Abbildung 1): $P_{u_1,u}, \{u, u'\}, P_{u',u_2}$. Wir ersetzen P durch $P' := P_{u_1,u}, P_{u,v}, P_1, P_{w,u'}, P_{u',u_2}$.

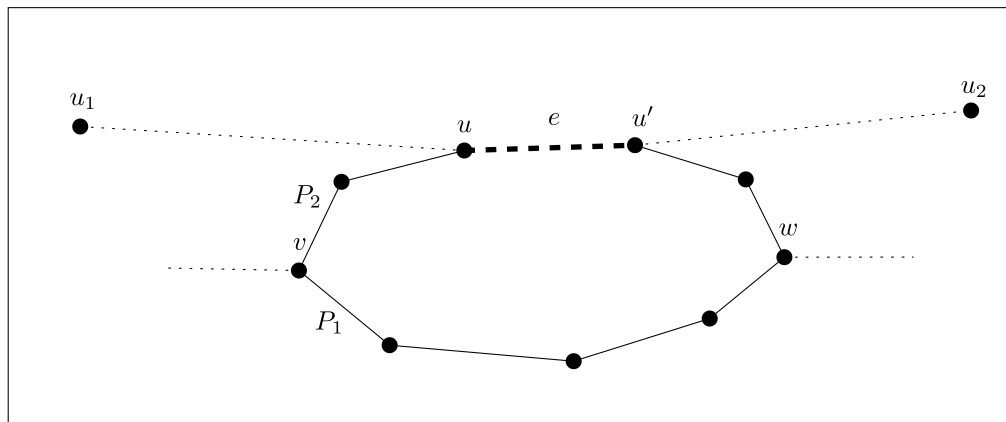


Abbildung 1: Alternative Pfade für e enthaltende Pfade.

Der Graph T' ist also ein aufspannender Baum mit einer Kante weniger als T, was in Widerspruch zur Minimimalität von T steht. Daraus schließen wir, dass ein minimaler aufspannender Teilgraph immer ein Baum ist. □

Anmerkung: Fasst man „minimal “als „inklusionsminimal “auf, so funktioniert obiger Beweis analog. Lässt man nun Gewicht aus ganz $\mathbb{R}$ zu, wie in der Vorlesung auch, so betrachte das Gegenbeispiel in Abbildung 2.

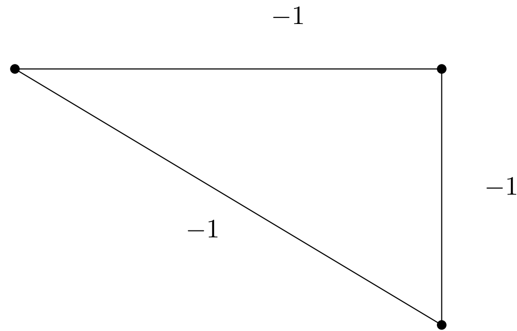


Abbildung 2: Gegenbeispiel mit Gewichten aus $\mathbb{R}$

Problem 3: Vier äquivalente Aussagen **

Zur Erinnerung: Ein Zyklus ist eine Knotenfolge, bei der Anfangs- und Endknoten übereinstimmen und zwischen je zwei aufeinanderfolgenden Knoten eine Kante existiert. Ein Kreis ist ein Zyklus, bei dem keine Knoten außer dem Anfangsknoten mehrmals vorkommen. In jedem Zyklus ist ein Kreis enthalten.

Zeigen Sie, dass die folgenden Aussagen für einen ungerichteten Graphen $G = (V, E)$ äquivalent sind.

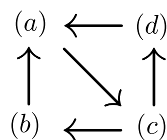
(a) G ist zusammenhängend und enthält keine Kreise.

(b) G ist maximal kreisfrei, d.h. G enthält keinen Kreis, und für je zwei Knoten $u, v \in V$ mit $\{u, v\} \notin E$ enthält der Graph $G' = (V, E \cup \{\{u, v\}\})$ einen Kreis.

(c) Zwischen je zwei Knoten in V gibt es genau einen Pfad in G.

(d) G ist minimal zusammenhängend, d.h. G ist zusammenhängend, und für jede Kante $e \in E$ ist der Graph $G' = (V, E \setminus \{e\})$ nicht zusammenhängend.

Lösung. Falls G nur einen Knoten enthält, dann sind die Aussagen trivialerweise äquivalent. Also bestehe der Graph G im folgenden aus mindestens zwei Knoten. Folgende Implikationen induzieren die Äquivalenz der vier Aussagen:



(a) $(a \Rightarrow c \equiv \neg c \Rightarrow \neg a)$:
Der Graph G enthalte zwei Knoten v und w, zwischen denen 0 oder 2 Pfade existieren. Falls es keinen Weg gibt, dann ist G nicht zusammenhängend. Falls es zwei verschiedene Pfade W_1 und W_2 gibt, dann ergibt die folgende Knotenfolge einen Zyklus: v, W_1, w, W_2, v. Daraus folgt, dass G nicht kreisfrei ist.

(b) $(c \Rightarrow b)$:
Falls G einen Kreis enthält, dann gibt es mindestens zwei Knoten, zwischen denen es mehr als einen Pfad gibt. Betrachte nun zwei beliebige Knoten v und w, so dass $\{w, v\} \notin E$. Zwischen

v und w gibt es genau einen Pfad W in G. In $G' := (V, E \cup \{\{w, v\}\})$ gäbe es dann zwei verschiedene Pfade von v nach w. Daraus folgt, dass G einen Zyklus (v, W, w, v) und somit einen Kreis enthalten müsste.

(c) $(c \Rightarrow d)$:
G ist offensichtlich zusammenhängend. Betrachte die Endknoten v und w einer beliebigen Kante $e \in E$. Diese Kante ist der einzige Pfad von v nach w. Durch Entfernen von e würde kein Weg mehr zwischen v und w existieren, d.h. diese Knoten würden zwei verschiedenen Zusammenhangskomponenten angehören. Somit wäre $G' = (V, E \setminus \{e\})$ nicht zusammenhängend. Da e beliebig gewählt wurde folgt die Behauptung.

(d) $(b \Rightarrow a)$:
Wäre G nicht zusammenhängend, so würden mindestens zwei Zusammenhangskomponenten Z_1 und Z_2 existieren. Wähle zwei Knoten $b_1 \in Z_1$ und $b_2 \in Z_2$ und verbinde sie durch eine Kante e. Diese Kante induziert keinen Kreis. Also wäre G nicht maximal kreisfrei. Somit wäre G zusammenhängend und enthält außerdem keine Kreise.

(e) $(d \Rightarrow a)$:
Wir zeigen durch einen Widerspruchsbeweis, dass G keine Kreise enthält. Würde G einen Kreis enthalten, dann könnten wir eine Kante aus diesem Kreis entfernen und der resultierende Graph bliebe immer noch zusammenhängend. Somit ist G zusammenhängend und kreisfrei. □

Problem 4: Schnitte und MSTs I *

Eine Kante e heiße *leicht*, wenn es einen Schnitt gibt, so dass e unter allen Kanten, die diesen Schnitt kreuzen, minimales Gewicht hat. Geben Sie einen gewichteten Graphen an, so dass die Menge der leichten Kanten keinen minimal aufspannenden Baum induziert.

Lösung. Wir betrachten den vollständigen Graphen K_3 mit dem Gewicht 1 für alle Kanten (siehe Abbildung 3).

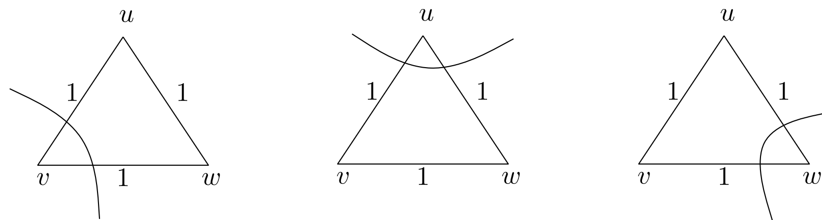


Abbildung 3: Leichte Schnitte

Es gibt genau folgende Schnitte: $(\{u\}, \{v, w\})$, $(\{v\}, \{u, w\})$ und $(\{w\}, \{u, v\})$. Alle drei Kanten sind offensichtlich leicht. Sie induzieren dennoch den Graphen K_3, der kein minimaler aufspannender Graph ist, da er kein Baum ist (Siehe Aufgabe (2b)). □

Problem 5: Schnitte und MSTs II **

(a) Zeigen Sie: Wenn in einem Graph G mit reellen Kantengewichten für jeden Schnitt die den Schnitt kreuzende Kante minimalen Gewichts eindeutig ist, dann hat G einen eindeutigen

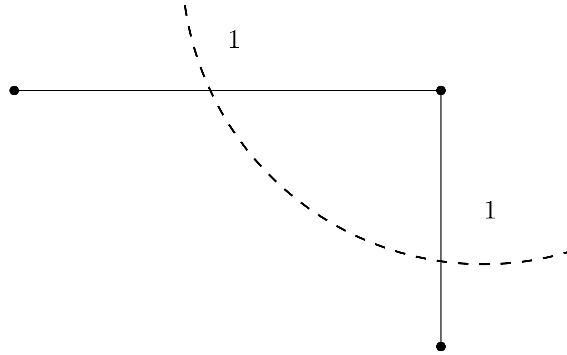


Abbildung 4: Gegenbeispiel zu 5 (b)

aufspannenden Baum minimalen Gewichts.

Lösung. Wir nehmen an, dass die Voraussetzung erfüllt ist, aber G zwei verschiedene MSTs T_1 und T_2 hat. Sei $e_1 \in E(T_1) \setminus E(T_2)$ eine Kante von T_1, die nicht in T_2 ist. Diese partitioniert T_1 in zwei disjunkte Teilbäume. Wir fassen diese als Schnitt $(S, V \setminus S)$ auf. Unter allen Kanten, die den Schnitt kreuzen, hat e_1 minimales Gewicht (sonst wäre T_1 nicht minimal). Andererseits induziert e_1 in T_2 einen Kreis. Mindestens eine weitere der Kanten dieses Kreises kreuzt den Schnitt. Diese Kante sei e_2. Wegen $c(e_1) < c(e_2)$ ist $T_2 \setminus \{e_2\} \cup \{e_1\}$ ein Baum mit echt kleinerem Gewicht als T_2, ein Widerspruch. □

(b) Gilt die Umkehrung von (a)?

Lösung. Nein, als Gegenbeispiel siehe Abbildung 4. □

(c) Zeigen oder widerlegen Sie: Wenn in einem Graph G mit reellen Kantengewichten alle Kanten paarweise verschiedene Gewichte haben, dann hat G einen eindeutigen aufspannenden Baum minimalen Gewichts.

Lösung. Folgt aus (a): Wenn alle Gewichte verschieden sind, dann gibt es unter den Kanten, die einen Schnitt kreuzen, stets genau eine mit minimalem Gewicht. □

(d) Zeigen oder widerlegen Sie die Umkehrung der Aussage in (c).

Lösung. Auch diese Umkehrung gilt nicht, siehe als Gegenbeispiel wie in Teilaufgabe (b) Abbildung 4. □

Problem 6: Matroide **

Beweisen Sie, dass die Menge aller Teilmengen von Kanten eines Graphen, die eine Menge von Bäumen induzieren, ein Matroid ergibt.

Lösung. Erinnerung: Für eine Grundmenge E ist $(E, \mathcal{U})$ mit $\mathcal{U} \subset \mathcal{P}(E)$ ein Matroid falls:

- $\emptyset \in \mathcal{U}$
- $I \subset J,\ J \in \mathcal{U} \Rightarrow I \in \mathcal{U}$
- $I \in \mathcal{U},\ J \in \mathcal{U},\ |I| < |J| \Rightarrow \exists x \in J \setminus I : I \cup \{x\} \in \mathcal{U}.$

Wir betrachten die Kantenmenge E eines Graphen $G = (V, E)$ als Grundmenge. Sei $\mathcal{U} \subset \mathcal{P}(E)$ die Menge aller Teilmengen von Kanten, die eine Menge von Bäumen induzieren. Wir zeigen, dass $(E, \mathcal{U})$ ein Matroid ist.

- Die leere Kantenmenge induziert $|V|$ Bäume, die jeweils aus einem Knoten bestehen. Daraus folgt: $\emptyset \in \mathcal{U}$.
- Sei $I \subset J,\ J \in \mathcal{U}$. Die Teilmenge I entsteht aus J durch Weglassen von Kanten. Das Weglassen von Kanten aus einem Wald liefert weiterhin einen Wald. Also gilt: $I \in \mathcal{U}$.
- Sei $I \in \mathcal{U},\ J \in \mathcal{U},\ |I| < |J|$. Sei $zus(A)$ die Anzahl der Zusammenhangskomponenten, die von einer Kantenmenge A induziert werden. Es gilt: $zus(I) > zus(J) \geq zus(I \cup J)$, d.h. es gibt mindestens eine Kante $e \in J \setminus I$, die zwei verschiedene Zusammenhangskomponenten von I in $I \cup J$ verbindet. Offensichtlich induziert $I \cup \{e\}$ einen Wald. Daraus folgt: $I \cup \{e\} \in \mathcal{U}$. □

Problem 7: Einmaschinenscheduling **

Eine Reihe von Aufträgen $A_1, \ldots A_n$ müssen von einer einzigen Maschine abgearbeitet werden, es kann also zu jedem Zeitpunkt höchstens ein Auftrag bearbeitet werden. Alle Aufträge benötigen die gleiche Bearbeitungszeit. Jeder Auftrag A_i hat eine Deadline $D_i \in \mathbb{R}$ bis zu der er fertig sein muss.

(a) Zeigen Sie, dass die Menge aller Teilmengen von Aufträgen, die rechtzeitig bearbeitet werden können, ein Matroid ist.

Lösung. Eine Menge von Aufträgen, die rechtzeitig bearbeitet werden können, heiße *zulässig.* Sei o.B.d.A. die Bearbeitungszeit eines Auftrages auf 1 normiert und $D_1 \leq D_2 \leq \cdots \leq D_n$. Da die Bearbeitungszeit für j Aufträge j beträgt, ist eine Teilmenge $\{A_{i_1}, \ldots, A_{i_k}\}$ von k paarweise verschiedenen Aufträgen $(i_1 \neq \cdots \neq i_k \in \{1, \ldots, n\})$ genau dann zulässig, wenn gilt:

$$\forall j \leq k \colon D_{i_j} \geq j \qquad (*)$$

- Eine leere Menge von Aufträgen kann stets rechtzeitig bearbeitet werden.
- Eine Teilmenge einer zulässigen Menge von Aufträgen kann ebenfalls rechtzeitig bearbeitet werden.
- Seien $\mathcal{I} := \{A_{i_1}, \ldots, A_{i_k}\}$ und $\mathcal{J} := \{A_{j_1}, \ldots, A_{j_\ell}\}$ zwei zulässige Mengen mit $k < \ell$. Seien k' bzw. ℓ' der minimale Index mit $i_{k'+1} = j_{\ell'+1}$ und $i_{k'+2} = j_{\ell'+2}$ und ... und $i_k = j_\ell$, d.h. die Aufträge nach k' bzw. ℓ' sind paarweise gleich und $A_{i_{k'}}$ und $A_{j_{\ell'}}$ sind verschieden.
  Behauptung: $\mathcal{I}' := \{A_{i_1}, \ldots, A_{i_{k'}}, A_{j_{\ell'}}, A_{i_{k'+1}}, \ldots, A_{i_k}\}$ ist zulässig.
  Beweis: Für $\nu \leq k'$ gilt $D_{i_\nu} \geq \nu$ da $\mathcal{I}$ zulässig ist. Da $\ell > k$ ist, gilt auch $\ell' > k'$, also $\ell' \geq k' + 1$. Daraus folgt $D_{i_{\ell'}} \geq \ell' \geq k' + 1$, da $\mathcal{J}$ zulässig ist. Außerdem gilt für $1 \leq \mu \leq k - k'$, dass $D_{i_{k'+\mu}} = D_{j_{\ell'+\mu}} \geq \ell' + \mu \geq k' + 1 + \mu$ ist. Also ist $\mathcal{I}'$ zulässig.

Verbal: Falls $A_{i_k} \neq A_{j_\ell}$ dann kann A_{j_ℓ} an $\mathcal{I}$ angehängt werden, da $|\mathcal{J}| > |\mathcal{I}|$ und $\mathcal{J}$ zulässig. Falls $A_{i_k} = A_{j_\ell}$ dann laufe in $\mathcal{I}$ und $\mathcal{J}$ solange parallel rückwärts, bis der erste Unterschied $A_{i_{k'}} \neq A_{j_{\ell'}}$ auftritt. Dann, füge $A_{j_{\ell'}}$ nach $A_{i_{k'}}$ in $\mathcal{I}$ ein. Da die Elemente $A_{j_{\ell'}}$ bis A_{j_ℓ} in $\mathcal{J}$ zulässig sind, sind sie es auch in $\mathcal{I}$. □

(b) Wird ein Auftrag nicht rechtzeitig fertig, so muss eine Strafe P_i bezahlt werden, deren Höhe vom jeweiligen Auftrag abhängt. In welcher Reihenfolge sollten die Aufträge abgearbeitet werden, damit die Gesamtstrafe minimiert wird?

Lösung. Die optimale Reihenfolge sieht so aus, dass zuerst eine Menge von Aufträgen abgearbeitet wird, deren Gesamtstrafe maximal ist. Damit wird die Gesamtstrafe minimiert, die durch die folgenden, nicht rechtzeitig bearbeiteten Aufträge erzeugt wird.

Also führt folgender Greedy-Ansatz zum Ziel:

1. Sortiere die Aufträge in nicht-aufsteigender Reihenfolge ihrer Strafen.
2. Setze $S := \emptyset$.
3. Für jeden Auftrag A in obiger Reihenfolge
4. Wenn $S \cup \{A\}$ zulässig ist
5. Setze $S := S \cup \{A\}$.

□