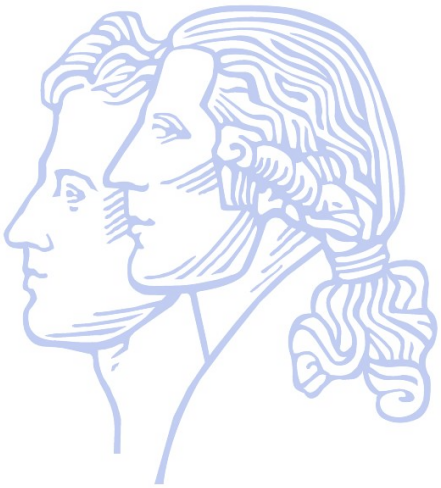
Übungen

Theoretische Informatik 3

Niels Lohmann

Sommersemester 2003



nlohmann@informatik.hu-berlin.de
http://www.informatik.hu-berlin.de/~nlohmann

Inhaltsverzeichnis

Serie 1

Institut für Informatik
Algorithmen und Komplexität
Dr. Till Nierhoff

SS 2003
15. April 2003

Theoretische Informatik III

1. Serie

Abgabe bis zum 22. April 2003

Aufgabe 1 [8 Punkte]

Geben Sie ein verständlich kommentiertes RAM-Programm an, das folgenden Algorithmus emuliert:

Eingabe n

For $i = 2$ To n
prim$[i] \leftarrow 1$
EndFor

For $i = 2$ To n
For $j = 2$ To n
If $i \cdot j \leq n$ Then prim$[ij] \leftarrow 0$
EndFor
EndFor

Ausgabe prim$[n]$

Aufgabe 2 [4+4+4+0 Punkte]

Sind die folgenden Booleschen Formeln F für jedes $n \geq 2$ erfüllbar? Begründen Sie ihre Antwort.

a) Sei $1 \leq k \leq n$.

$$F_k = \overline{x_k} \bigwedge_{\substack{i,j \\ 1 \leq i < j \leq n}} (x_i \vee x_j)$$

b) $F = F_1 \wedge F_2$, wobei F_1 und F_2 wie in a) definiert sind.

c)

$$F = (\bigvee_{1 \leq i \leq n} x_i) \wedge (\bigvee_{1 \leq i \leq n} \overline{x_i}) \wedge (x_n \vee \overline{x_1}) \wedge \bigwedge_{1 \leq i < n} (x_i \vee \overline{x_{i+1}})$$

d) (mündlich)
Das Problem SAT ist wie folgt definiert:
Eingabe: eine Boolesche Formel $F(x_1, \ldots, x_n)$
Gefragt: Ist F erfüllbar?
Finden Sie einen möglichst schnellen Algorithmus (gemessen an der Anzahl n der Variablen) für das SAT-Problem.

Aufgabe 1

1	LOAD #2	Startwert der Laufvariable
2	STORE 2	Laufvariable wird in Register 2 zwischengespeichert
3	SUB 1	$i - n$
4	IF AKKU $>$ 0 GOTO 14	Test der FOR-Schleife
5	LOAD #5	Register der 1. Zelle des Arrays prim
6	ADD 2	Array nach Laufvariable erhöhen
7	SUB #1	eine Zelle zurück
8	STORE 4	Zellenposition in Register 4 zwischenspeichern
9	LOAD #1	eine 1 in den Akkumulator laden
10	STORE *4	diese 1 in die Zelle des Arrays speichern
11	LOAD 2	Laufvariable von Register 2 lesen
12	ADD #1	Laufvariable wird um 1 erhöht
13	GOTO 2	zurück zum Test der FOR-Schleife
14	LOAD #2	Startwert der Laufvariable i
16	STORE 2	Laufvariable i wird in Register 2 zwischengespeichert
17	SUB 1	i-n
18	IF AKKU $>$ 0 GOTO 42	Test der äußeren FOR-Schleife
19	LOAD #2	Startwert der Laufvariable j
20	STORE 3	Laufvariable j wird in Register 3 zwischengespeichert
21	SUB 1	j-n
22	IF AKKU $>$ 0	GOTO 39 Test der inneren FOR-Schleife
23	LOAD 2	Laufvariable i von Register 2 lesen
24	MUL 3	Akkumulator mit Laufvariable j multiplizieren
25	STORE 4	Ergebnis in Register 4 zwischenspeichern
26	LOAD 1	n aus Register 1 laden
27	SUB 4	davon Inhalt von Register 4 abziehen
28	IF AKKU $\geq$ 0 GOTO 30	Test: wenn größer als 0 ist, nach 30 springen
29	GOTO 36	Test: ansonsten nach 36 springen
30	LOAD #5	Register der 1. Zelle des Arrays prim
31	ADD 4	Inhalt von Register 4 ($i \cdot j$) addieren
32	SUB #1	eine Zelle zurück
33	STORE 4	Zellenposition in Register 4 zwischenspeichern
34	LOAD #0	0 in den Akkumulator laden
35	STORE *4	diese 0 in die Zelle mit Index $i \cdot j$ speichern
36	LOAD 3	Laufvariable j von Register 3 lesen
37	ADD #1	Laufvariable j wird um 1 erhöht
38	GOTO 20	zurück zum Test der inneren FOR-Schleife
39	LOAD 2	Laufvariable i von Register 2 lesen
40	ADD #1	Laufvariable i wird um 1 erhöht
41	GOTO 16	zurück zum Test der äußeren FOR-Schleife
42	LOAD #5	Register der 1. Zelle des Arrays prim
43	ADD 1	Eingabezahl addieren
44	SUB #1	eine Zelle zurück
45	STORE 4	Zellenposition in Register 4 zwischenspeichern
46	LOAD *4	Zahl aus dieser Zelle lesen
47	STORE 1	Endergebnis im Register 1 speichern
48	END	Feierabend

Für das RAM-Programm werden die Register wie folgt benutzt:

Register	Verwendungszweck
1	Eingabe n bzw. Ausgabe des Programmes
2	Laufvariable i
3	Laufvariable j
4	Zwischenergebnisse (z.B. Indizes, Registeradressen)
5	Feldinhalt prim[1] (nicht benutzt)
6	Feldinhalt prim[2]
7	Feldinhalt prim[3]
$\vdots$	$\vdots$

Aufgabe 2

Teilaufgabe a

Die Formel

$$F_k = \overline{x_k} \bigwedge_{\substack{i,j \\ 1 \leq i < j \leq n}} (x_1 \vee x_j)$$

ist mit folgender Belegung für alle $n \geq 2$ erfüllbar:

Sei $a_k = 0$ und $a_i = 1$ für alle $i \neq k$.

Mit $a_k = 0$ ist $\overline{a_k}$ erfüllt. In der folgenden $\wedge$-Kette sind durch die Beschränkung $1 \leq i \leq n$ jeweils zwei unterschiedliche Variablen durch $\vee$ verknüpft. Da laut Voraussetzung nur $a_k = 0$ ist, muss mindestens eine der disjunktiv verknüpften Variablen mit 1 belegt sein, sodass alle Disjunktionen und somit die gesamte Formel erfüllt ist.

Teilaufgabe b

Die Formel $F = F_1 \wedge F_2$ ist nicht für jedes $n \geq 2$ erfüllbar, da es beispielsweise für $n = 2$ keine Belegung gibt, um

$$F = F_1 \wedge F_2 = (\overline{x_1} \wedge (x_1 \vee x_2)) \wedge (\overline{x_2} \wedge (x_1 \vee x_2))$$

zu erfüllen:

$a_1 a_2$	$\overline{a_1} \wedge (a_1 \vee a_2)$	$\overline{a_2} \wedge (a_1 \vee a_2)$	F
11	0	0	0
10	0	1	0
01	1	0	0
00	0	0	0

Teilaufgabe c

Die Formel

$$F = (\bigvee_{1 \leq i \leq n} x_i) \wedge (\bigvee_{1 \leq i \leq n} \overline{x_i}) \wedge (x_n \vee \overline{x_1}) \wedge \bigwedge_{1 \leq i < n} (x_i \vee \overline{x_{i+1}})$$

ist nicht für jedes $n \geq 2$ erfüllbar, da es beispielsweise für $n = 2$ keine Belegung gibt, um

$$F = (x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (x_2 \vee \overline{x_1}) \wedge (x_1 \vee \overline{x_2})$$

zu erfüllen:

$a_1 a_2$	$a_1 \vee a_2$	$\overline{a_1} \vee \overline{a_2}$	$a_2 \vee \overline{a_1}$	$a_1 \vee \overline{a_2}$	F
11	1	0	1	1	0
10	1	1	0	1	0
01	1	1	1	0	0
00	0	1	1	1	0

Serie 2

Institut für Informatik
Algorithmen und Komplexität
Dr. Till Nierhoff

SS 2003
22. April 2003

Theoretische Informatik III

2. Serie

Abgabe bis zum 29. April 2003

Aufgabe 3 [4+6 Punkte]

Ein *Monom* ist eine Konjunktion von Literalen. Eine Boolesche Formel ist in *disjunktiver Normalform (DNF)*, wenn sie eine Disjunktion von Monomen ist, also z.B. $(x_1 \wedge \overline{x_2}) \vee (x_3 \wedge x_4)$.
Dnf-Sat ist die Sprache aller Formeln in disjunktiver Normalform, die erfüllbar sind.

a) Beschreiben Sie einen Algorithmus, der Dnf-Sat entscheidet, und zeigen Sie dessen Korrektheit.

b) Zeigen Sie, daß Dnf-Sat in $\mathcal{P}$ liegt.

Aufgabe 4 [4+6 Punkte]

Ein Graph heißt *bipartit*, wenn sich seine Knoten derart in zwei Klassen einteilen lassen, dass keine Kanten zwischen Knoten gleicher Klassen verlaufen.
Sei Bipartit die Sprache aller bipartiten Graphen.

a) Beschreiben Sie einen Algorithmus, der Bipartit entscheidet, und zeigen Sie dessen Korrektheit.

b) Zeigen Sie, daß Bipartit in $\mathcal{P}$ liegt.

Aufgabe 5 [mündlich]

Eine *Klausel* ist eine Disjunktion von Literalen. Eine Boolesche Formel ist in *konjunktiver Normalform (CNF)*, wenn sie eine Konjunktion von Klauseln ist, also z.B. $(x_1 \vee \overline{x_2}) \wedge (x_3 \vee x_4)$.
k-Sat ist die Sprache der erfüllbaren Formeln in CNF, in denen jede Klausel genau k Literale enthält.
Finden Sie eine 3-Sat-Formel, so daß es keine 2-Sat Formel auf der gleichen Variablenmenge gibt, die genau dieselben erfüllenden Belegungen hat.

Aufgabe 3

Teilaufgabe a

Folgender Algorithmus entscheidet die Sprache DNF-SAT:

```
ALGORITHMUS DNF-SAT:
Eingabe: Formel F, Anzahl der Variablen n;

BEGIN
1    Truth[1..n] : int;
2    skip : boolean;
3    foreach Monom m in F do
4      skip ← false;
5      initialize Truth[] to -1;
6      foreach Literal X_i in m
7        if (X_i negiert)
8          if ((Truth[i] = 0) or (Truth[i] = -1))
9            Truth[i] ← 0;
10           else
11             skip ← true;
12         else
13           if ((Truth[i] = 1) or (Truth[i] = -1))
14             Truth[i] ← 1;
15           else
16             skip ← true;
17         if (skip = false)
18           Ausgabe: 1;
19       endfor;
20     endfor;
21     Ausgabe: 0;
END
```

Der Algorithmus geht wie folgt vor:

Eine boolsche Formel in disjunktiver Normalform ist genau dann erfüllbar, wenn eines der Monome erfüllbar ist. Der Algorithmus DNF-SAT überprüft sequentiell alle Monome der eingegeben Formel nach Literalen der Form $x_i \wedge \overline{x_i}$, die das Monom unerfüllbar machen.

Dazu untersucht es die Literale nach Negation und baut im Array Truth[] eine Belegung auf, die das jeweilige Monom erfüllt. Sobald für eine Variable eine widersprüchliche Belegung ermittelt, wird das nächste Monom untersucht.

Der Algorithmus gibt eine positive Ausgabe, sobald eine Belegung gefunden wird, die ein Monom erfüllt. Wurden hingegen alle Monome untersucht, ohne eine erfüllende Belegung zu finden, wird eine negative Ausgabe gegeben.

Teilaufgabe b

Die Laufzeit des Algorithmus wird weitestgehend von den zwei FOR-Schleifen bestimmt. Die Laufzeit der äußeren Schleife wird von der Anzahl der in der Formel enthaltenen Monome bestimmt, während die Laufzeit der inneren Schleife von der in den Monomen auftretenden Literalen bestimmt ist.

Sei nun n die Anzahl der Literale, die in der eingegeben Formel F vorkommen. Dementsprechend kann F nicht mehr als n Monome enthalten. Im worst case hat der Algorithmus also die Laufzeit $\mathcal{O}(n^2)$.

Die Funktion $t(n) = n^2$ beschränkt eine k-DTM M, die DNF-SAT berechnet. Also ist DNF-SAT $\in \mathcal{P}$.

Aufgabe 4

Teilaufgabe a

Folgender Algorithmus entscheidet die Sprache BIPARTIT:

```
ALGORITHMUS BIPARTIT
Eingabe: Graph G=([v],E)

BEGIN
1    Part[1..n] : int;
2    initialize Part[] to 0;
3    foreach v ∈ V
4      if (Part[v] = 0) or (Part[v] = 1)
5        Part[v] ← 1;
6        forall u ∈ Γ(v)
7          if (Part[u] = 1)
8            Ausgabe: 0;
9          else
10            Part[u] ← 2;
11       endfor;
12     else
13       Part[v] ← 2;
14       forall u ∈ Γ(v)
15         if (Part[u] = 2)
16           Ausgabe: 0;
```

```
17          else
18            Part[u] ← 1;
19        endfor;
20    endfor;
21    Ausgabe: 1;
END.
```

Der Algorithmus geht wie folgt vor:

Ein Graph ist bipartit, wenn sich seine Knoten derart in zwei Klassen einteilen lassen, dass keine Kanten zwischen Knoten gleicher Klassen verlaufen. Der Algorithmus ordnet einem Knoten v eine Klasse zu, indem er im Array Part[] an der Stelle v den Wert 1 oder 2 speichert.

Der Algorithmus besucht nach und nach alle Knoten des Eingabegraphen und ordnet ihnen eine Klasse zu. Anschließend wird allen Nachbarn die entsprechend andere Klasse zugeordnet. Tritt dabei ein Konflikt auf, bricht der Algorithmus sofort mit einer negativen Ausgabe ab. Werden hingegegen alle Knoten besucht, ohne dass ein Konflikt auftritt, wird mit einer positiven Ausgabe beendet.

Teilaufgabe b

Die Laufzeit des Algorithmus wird weitestgehend von den zwei FOR-Schleifen bestimmt. Die Laufzeit der äußeren Schleife wird von der Anzahl der Knoten des Eingabegraphen bestimmt, während die Laufzeit der inneren Schleife von den Nachbarn dieses Graphen bestimmt ist.

Sei nun n die Anzahl der Knoten, die im Eingabegraphen G enthalten sind. Dementsprechend kann jeder Knoten maximal $n-1$ Nachbarn haben, sodass der Algorithmus im worst case eine Laufzeit von $\mathcal{O}((n \cdot (n-1)) = \mathcal{O}(n^2 - n)$ hat.

Die Funktion $t(n) = n^2 - n$ beschränkt eine k-DTM M, die DNF-SAT berechnet. Also ist BIPARTIT $\in \mathcal{P}$.

Serie 3

Institut für Informatik
Algorithmen und Komplexität
Dr. Till Nierhoff

SS 2003
29. April 2003

Theoretische Informatik III

3. Serie

Abgabe bis zum 6. Mai 2003

Aufgabe 6 [4+6 Punkte]

Gegeben sei ein Graph $G = (V, E)$. Eine Menge $V' \subseteq V$ heißt *Knotenüberdeckung* von G, wenn jede Kante $(u, v) \in E$ wenigstens einen Knoten aus V' enthält.
Mit VertexCover (Entscheidungsversion) bezeichnet man das Problem, zu entscheiden, ob es – gegeben einen Graphen G und eine Zahl k – eine Knotenüberdeckung von G mit höchstens k Knoten gibt.

a) Formulieren Sie analog zur Vorlesung die Optimierungs- und die Suchversion des Problems VertexCover.

b) Zeigen Sie, daß für jede dieser drei Versionen von VertexCover gilt: Wenn sie in P liegt, liegen auch die beiden anderen Versionen in P.

Aufgabe 7 [4+6 Punkte]

Beim Problem BinPacking geht es darum, Objekte in möglichst wenige Behälter gegebener Größe zu packen. Die Entscheidungsversion ist wie folgt definiert: Gegeben die Zahlen $a_1, \ldots, a_n, b, k \in \mathbb{N}$. Frage: Können n Objekte der Größen $a_1, \ldots, a_n$ in k Behälter der Größe b gepackt werden?

a) Formulieren Sie die Optimierungs- und die Suchversion des Problems BinPacking.

b) Zeigen Sie, daß für jede dieser drei Versionen von BinPacking gilt: Wenn sie in P liegt, liegen auch die beiden anderen Versionen in P.

Aufgabe 8 [mündlich]

Mit Prim bezeichnet man das Problem, zu entscheiden, ob eine binär kodierte natürliche Zahl n eine Primzahl ist oder nicht. Betrachten Sie den folgenden Algorithmus:

> Eingabe: n.
> Überprüfe für alle natürlichen Zahlen i mit $2 \leq i \leq \sqrt{n}$, ob die Zahl n durch i teilbar ist.
> Falls dies in keinem Fall zutrifft, ist n eine Primzahl; sonst nicht.

Ist dieser Algorithmus *polynomiell*?

Aufgabe 6

Teilaufgabe a

Mit VertexCover (Optimierungsversion) bezeichnet man das Problem, gegeben einen Graphen G, die minimale Anzahl von Knoten k zu finden, für die es eine Knotenüberdeckung von G mit k Knoten gibt.

Mit VertexCover (Suchversion) bezeichnet man das Problem, gegeben einen Graphen G, die Knotenüberdeckung $V' \subseteq V$ zu finden, deren Knotenanzahl k minimal ist.

Teilaufgabe b

Der folgende Algorithmus berechnet VertexCover_O, wobei VertexCover_E gegeben ist:

```
ALGORITHMUS VertexCover_O:
Eingabe: Graph G = (V, E);

BEGIN
1    for i = 1 to |V|
2      if VertexCover_E(G, i) = 1
3        Ausgabe: i;
4    endfor;
5    Ausgabe: ∞;
END
```

Der Algorithmus probiert für alle Zahlen 1 bis $|V| = n$ durch, ob es eine Knotenüberdeckung mit dieser Knotenanzahl gibt und gibt beim ersten Erfolg diese Zahl aus. Da jeder zusammenhängende Graph eine Knotenüberdeckung von $|V| = n$ Knoten hat, gibt der Algorithmus stehts eine Zahl $k \in \mathbb{N}$ aus oder ∞, falls es keine Knotenüberdeckung gibt.

Der Algorithmus VertexCover_E wird hierbei höchstens $|V| = n$ mal aufgerufen. Ist $\text{VertexCover}_E \in \mathcal{P}$, so auch VertexCover_O.
Der folgende Algorithmus berechnet VertexCover_S, wobei VertexCover_O gegeben ist:

```
ALGORITHMUS VertexCover_S:
Eingabe: Graph G = (V, E);

BEGIN
1    V' ← V;
2    E' ← E;
```

```
3     o ←VertexCover_O(G);
4     for i = 1 to |V|
5       if VertexCover_O(V'\{i}, E'\{u,v|u = i ∨ v = i}) = o
6         V' ← V'\{i};
7         E' ← E'\{u,v|u = i ∨ v = i};
8     endfor;
9     Ausgabe: V'
END
```

Der Algorithms merkt sich die minimale Anzahl von Knoten einer Knotenüberdeckung und testet sequentiell für jeden Knoten, ob er in deser Knotenüberdeckung vorkommt: stimmt das neue Optimum nicht mehr mit dem alten Optimum o überein, war $i \in V'$ des Optimums – ansonsten wird der Knoten i und alle Kanten, die i enthalten, aus den Mengen V' bzw. E' entfernt, sodass V' endlich nur die Knoten enthält, die zur Knotenüberdeckung gehören.

Dazu wird der Algorithmus VertexCover$_O$ wird höchstens $|V| = n+1$ mal aufgerufen. Ist VertexCover$_O \in \mathcal{P}$, so auch VertexCover$_O$.

Der folgende Algorithmus berechnet VertexCover$_E$, wobei VertexCover$_S$ gegeben ist:

```
ALGORITHMUS VertexCover_E:
Eingabe: Graph G = (V,E), Zahl k;
BEGIN
1     V' ← VertexCover_S(V,E);
2     if |V'| < k
3       Ausgabe: 0;
4     else
5       Ausgabe: 1;
END
```

Der Algorithmus bestimmt mittels VertexCover$_S$ die optimale Lösung V' und kann anschließend mittels $|V'|$ entscheiden, ob es zur Eingabe k eine Knotenüberdeckung existiert oder nicht.

VertexCover$_S$ wird genau einmal aufgerufen. Ist VertexCover$_S \in \mathcal{P}$, so auch VertexCover$_E$.

Also gilt: VertexCover$_E \in \mathcal{P} \Leftrightarrow$ VertexCover$_S \in \mathcal{P} \Leftrightarrow$ VertexCover$_S \in \mathcal{P}$

Aufgabe 7

Teilaufgabe a

Mit BinPacking (Optimierungsversion) bezeichnet man das Problem, gegeben die Zahlen $a_1, \dots a_n, b$, die minimale Anzahl von Behältern k der Größe b zu finden, sodass die n Objekte in die Behälter gepackt werden können.

Mit BinPacking (Suchversion) bezeichnet man das Problem, gegeben gegeben die Zahlen $a_1, \dots a_n, b$, die optimale Aufteilung in k Behälter zu finden, sodass k minimal ist und die n Objekte in die Behälter gepackt werden können.

Teilaufgabe b

Der folgende Algorithmus berechnet BinPacking_O, wobei BinPacking_E gegeben ist:

```
ALGORITHMUS BinPacking_O:
Eingabe: a_1, ..., a_n, b;

BEGIN
1    for k = 1 to n
2      if BinPacking_E(a_1, ..., a_n, b, k) = 1
3        Ausgabe: k;
4    endfor;
5    Ausgabe: ∞;
END
```

Der Algorithmus probiert für alle Zahlen $k = 1$ bis n durch, ob es möglich ist, die Objekte $a_1, \dots, a_n$ in k Behälter der Größe b zu packen. Sobald der Entscheidungsalgorithmus BinPacking_E eine Lösung k gefunden hat, wird diese als optimale Lösung ausgegeben. Wird hingegen keine Lösung gefunden, wird ∞ ausgegeben.

Der Algorithmus BinPacking_E wird hierbei höchstens n mal aufgerufen.
Ist $\text{BinPacking}_E \in \mathcal{P}$, so auch BinPacking_O.
Der folgende Algorithmus berechnet BinPacking_S, wobei BinPacking_O gegeben ist:[1]

```
ALGORITHMUS BinPacking_S:
Eingabe: a_1, ..., a_n, b;

BEGIN
1    M = {a_1, ..., a_n};
```

[1] Im folgenden werden Mengen benutzt, die jedoch gleiche Elemente enthalten können. Die „Menge“ $M \backslash \{a_i\}$ soll als Liste $a_1, \dots, a_{i-1}, a_{i+1}, \dots a_n$ verstanden werden.

```
2     k ←BinPacking_O(a_1, ..., a_n, b);
3     Bin_i ← ∅   ∀i ∈ {1, ..., k};
4     for b = 1 to o
5       for i = 1 to n
6         for j = 1 to n
7           if i ≠ j
8             m = a_i + a_j;
9             if BinPacking_O((M\{a_i, a_j}) ∪ {m}, b) ≠ k
10              Bin_b ← Bin_b ∪ {a_i, a_j};
11              M ← M\{a_i, a_j};
12        endfor;
13      endfor;
14    endfor;
15    Ausgabe: Bin_i   i ∈ {1, ..., k};
END
```

Der Algorithmus nimmt sich sequentiell zwei Objekte a_i und a_j und verschmelzt diese zu einem Objekt m. Wenn durch diese Verschmelzung keine Veränderung an der minimalen Behälterzahl k (bestimmt duch den Optimierungsalgorithmus) auftritt, lagen diese Elemente a_i und a_j in dem selben Behälter und werden im Algorithmus in Bin_b einsortiert und anschließend aus der Objektliste entfernt.

Bei n Elementen gibt es maximal $\binom{n}{2}$ Zweiermengen. Des Weiteren gibt es für n Elemente maximal n Behälter, sodass der Algorithmus BinPacking_O ungefähr n^3 mal aufruft. Ist $\text{BinPacking}_O \in \mathcal{P}$, so auch BinPacking_S.

Der folgende Algorithmus berechnet BinPacking_E, wobei BinPacking_S gegeben ist:

```
ALGORITHMUS BinPacking_E:
Eingabe: a_1, ..., a_n, b, Zahl k;
BEGIN
1     Bins ← BinPacking_S(a_1, ..., a_n, b);
2     if |Bins| < k
3       Ausgabe: 0;
4     else
5       Ausgabe: 1;
END
```

Der Algorithmus bestimmt mittels BinPacking_S die optimale Lösung $Bins$ und kann anschließend mittels $|Bins|$ entscheiden, ob es eine Lösung mit k Behältern der Größe b existiert, in die die Objekte $a_1, \ldots, a_n, b$ gepackt werden können oder nicht.

BinPacking_S wird genau einmal aufgerufen. Ist $\text{BinPacking}_S \in \mathcal{P}$, so auch BinPacking_E.

Also gilt: $\text{BinPacking}_E \in \mathcal{P} \Leftrightarrow \text{BinPacking}_S \in \mathcal{P} \Leftrightarrow \text{BinPacking}_S \in \mathcal{P}$

Serie 4

Institut für Informatik
Algorithmen und Komplexität
Dr. Till Nierhoff

SS 2003
6. Mai 2003

Theoretische Informatik III

4. Serie

Abgabe bis zum 13. Mai 2003

Aufgabe 9 [10 Punkte]

Gegeben einen Graphen G und $k \in \mathbb{N}$, ist beim Problem Clique zu entscheiden, ob G einen vollständigen Teilgraphen (auch Clique genannt) auf k Knoten enthält. Gegeben einen Graphen G und $k \in \mathbb{N}$, ist beim Problem IndependentSet zu entscheiden, ob G eine stabile Menge (auch unabhängige Menge genannt) auf k Knoten enthält.
Zeigen Sie, dass folgendes gilt:

a) Clique $\leq_p$ IndependentSet

b) IndependentSet $\leq_p$ VertexCover

Aufgabe 10 [10 Punkte]

Zeigen Sie, dass gilt: 3Sat $\leq_p$ Clique.
Hinweis: Betrachten Sie eine Reduktion, die jedem Vorkommen eines Literals einen Knoten zuordnet und zwei Knoten dann mit einer Kante verbindet, wenn Sie aus verschiedenen Klauseln stammen und kompatibel sind.

Aufgabe 11 [mündlich]

Vollziehen Sie die Reduktion 3Sat $\leq_p$ DHC an einem Beispiel nach.
Betrachten Sie dazu folgende 3Sat-Instanz:

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$$

a) Finden Sie eine erfüllende Belegung.

b) Vollziehen Sie die Reduktion aus der Vorlesung nach, indem sie die zugehörige DHC-Instanz konstruieren.

c) Markieren Sie in der DHC-Instanz den Hamilton-Kreis, der Ihrer erfüllenden Belegung entspricht.

Aufgabe 9

Teilaufgabe a

Zu zeigen: Clique $\leq_{\mathcal{P}}$ IndependentSet.

Beweis
Sei die Clique-Instanz $\langle G, k\rangle$ mit $G = (V, E)$ gegeben. Dazu wird der Graph $G' = (V, \overline{E})$ mit invertierter Knotenmenge gebildet, d.h. $\{u, v\} \in E \Leftrightarrow \{u, v\} \notin \overline{E}$.

Es gilt: $G = (V, E)$ enthält einen Teilgraphen $K = (V', E')$ mit $|V'| = k$ Knoten

- genau dann, wenn $\forall u, v \in V'$ $(u \neq v)$ gilt: $\{u, v\} \in E$
- genau dann, wenn $\forall u, v \in V'$ $(u \neq v)$ gilt: $\{u, v\} \notin \overline{E}$
- genau dann, wenn $V' \subseteq V$ mit $|V'| = k$ zu G' Independent-Set der Größe k ist.

Bei $|V|$ Knoten enthält E höchstens $\binom{V}{2}$ Kanten. Die Funktion f, die die eingegebe Clique-Instanz $\langle G, k\rangle$ wie beschrieben in die IndependentSet-Instanz $\langle G', k\rangle$ überführt, ist polynomiell. Dabei soll f außerdem alle Eingaben $x \notin$ Clique auf $x_0 \notin$ IndependentSet abbilden.

Somit gilt: Clique $\leq_{\mathcal{P}}$ IndependentSet □

Teilaufgabe b

Zu zeigen: IndependentSet $\leq_{\mathcal{P}}$ VertexCover.

Beweis
Sei I ein IndependentSet der Größe i des Graphen $G = (V, E)$ gegeben. Sei $V' := V \backslash I$ und $k := |V| - i$.

Es gilt: $G = (V, E)$ enthält Independent-Set I der Größe i

- genau dann, wenn $\forall u, v \in I$ gilt: $\{u, v\} \notin E$
- genau dann, wenn $\forall \{u, v\} \in E$ gilt: $u \notin I \vee v \notin I$
- genau dann, wenn $\forall \{u, v\} \in E$ gilt: $u \in V \backslash I \vee v \in V \backslash I$
- genau dann, wenn $V \backslash I$ ist Vertex-Cover der Größe k von G.

Die Funktion f, die die eingegebe IndependentSet-Instanz $\langle G, i\rangle$ wie beschrieben in die VertexCover-Instanz $\langle G, k\rangle$ überführt, ist polynomiell. Dabei soll f außerdem alle Eingaben $x \notin$ IndependentSet auf $x_0 \notin$ VertexCover abbilden.

Somit gilt: IndependentSet $\leq_{\mathcal{P}}$ VertexCover □

Aufgabe 10

Zu zeigen: 3SAT $\leq_{\mathcal{P}}$ CLIQUE.

Beweis

Sei $F = \bigwedge_{i=1}^{m}(z_{i,1}, z_{i,2}, z_{i,3})$ eine beliebige 3SAT-Formel gegeben, die o.B.d.A. genau drei Literale pro Klausel enthält. Dabei ist $z_{i,j} \in \{x_1, x_2, \ldots\} \cup \{\overline{x_1}, \overline{x_2}, \ldots\}$.

Dieser Formel wird nun wie folgt ein Graph $G = (V, E)$ und $k \in \mathbb{N}$ zugeordnet:

- $V := \{(1,1), (1,2), (1,3), \ldots, (m,1), (m,2), (m,3)\}$, d.h. jedem Vorkommen eines Literales $z_{i,j}$ wird ein Knoten (i,j) zugeordnet.
- $E := \{\{(i,j),(p,q)\} \mid i \neq p \wedge z_{i,j} \neq \overline{z_{p,q}}\}$, d.h. es verlaufen nur Kanten zwischen Knoten, die aus unterschiedlicher Klauseln stammen und die kompatibel, d.h. vereinbar zueinander sind.
- $l := m$

Es gilt: F ist erfüllbar durch eine Belegung B

- genau dann, wenn es in jeder Klausel ein Literal gibt, dass durch die Belegung B den Wert w annimmt, z.B. $z_{1,j_1}, z_{2,j_2}, \ldots, z_{m,j_m}$.
- genau dann, wenn es Literale $z_{1,j_1}, z_{2,j_2}, \ldots, z_{m,j_m}$ gibt, die paarweise kompatibel sind, d.h. $z_{i,j} \neq \overline{z_{i,j}}$.
- genau dann, wenn es Knoten $(1,j_1), (2,j_2), \ldots, (m,j_m)$ in G gibt, die paarweise verbunden sind.
- genau dann, wenn es eine Clique der Größe $k = m$ in G gibt.

Es gibt bei m Klauseln $3m$ Knoten und maximal $\binom{3m}{2}$ Kanten. Die Funktion f, die die 3SAT-Instanz F (mit m Klauseln) wie beschrieben in die CLIQUE-Instanz $\langle G, k\rangle$ überführt, ist polynomiell. Des Weiteren soll f Eingaben $x \notin$ 3SAT auf $x_0 \notin$ CLIQUE abbilden.

Somit gilt: 3SAT $\leq_{\mathcal{P}}$ CLIQUE □

Serie 5

Institut für Informatik
Algorithmen und Komplexität
Dr. Till Nierhoff

SS 2003
13. Mai 2003

Theoretische Informatik III

5. Serie

Abgabe bis zum 20. Mai 2003

Aufgabe 12 [10 Punkte]

Partition ist das Problem, zu einer Menge $a_1, \ldots, a_n \in \mathbb{N}$ eine Unterteilung in zwei Teilmengen zu finden, so dass deren Summe jeweils gleich ist. Part ist die Sprache aller Instanzen, für die eine solche Unterteilung existiert:

$$\textsc{Part} = \left\{(a_1, \ldots, a_n) \mid \exists S \subseteq [n] : \textstyle\sum_{i \in S} a_i = \sum_{i \in [n] \setminus S} a_i\right\}$$

*Knapsack** ist eine spezielle Version des Knapsack-Problems, bei der die Gewichtsschranke gleich der Nutzenschranke ist:

$$\textsc{Kp}^* = \{(a_1, \ldots, a_n, g_1, \ldots, g_n, G, A) \in \textsc{Kp} \mid G = A\}$$

Zeigen Sie, dass $\textsc{Kp}^* \leq_p \textsc{Part}$.

Aufgabe 13 [10 Punkte]

Gegeben sei eine $m \times n$-Matrix A aus ganzzahligen Koeffizienten und ein m-Vektor b. Wir betrachten das Ungleichungssystem $Ax \geq b$, d. h.

$$\forall i \in [m] : \sum_{j=1}^{n} a_{i,j} \cdot x_j \geq b_i. \tag{1}$$

Das Problem *0/1 Integer Linear Programming* besteht darin, für dieses Ungleichungssystem zu entscheiden, ob es einen Lösungsvektor x gibt, dessen Komponenten jeweils 0 oder 1 sind. 0/1-Ilp ist also die Sprache der (A, b), für die es ein $x \in \{0, 1\}^n$ gibt, so dass (1) gilt.
Zeigen Sie, dass gilt: VertexCover $\leq_p$ 0/1-Ilp.
Hinweis: Ordnen Sie jedem Knoten eine Variable zu.

Aufgabe 14 [mündlich]

Eine *Euler-Tour* ist ein Weg durch einen Graphen, der jede Kante genau einmal enthält und dessen Anfangs- und Endknoten identisch sind. Euler sei die Sprache aller Graphen, für die es eine Euler-Tour gibt.
Zeigen Sie, dass Euler $\in P$.

Aufgabe 12

Zu zeigen: KP* $\leq_{\mathcal{P}}$ PART.

Beweis
Sei $(a_1, \ldots, a_n, A)$ eine Eingabe für KP*. Daraus kann in polynomieller Zeit die Eingabe $(a_1, \ldots, a_n, S-A+1, A+1)$ für PART konstruiert werden, wobei S die Summe aller a_i ist.

Falls I eine Lösung für KP* ist ($\sum_{i \in I} a_i = A$), erhalten wir mit $I \cup \{S - A + 1\}$ eine Lösung für PART, da

$$\sum_{i \in I} a_i + S - A + 1 = A + S - A + 1 = S + 1 = \sum_{1 \leq i \leq n} a_i + 1 = \sum_{i \notin I} a_i + A + 1$$

Die Summe aller Zahlen in der Eingabe für PART beträgt $2S + 2$. Eine Lösung für PART muss also so aussehen, dass jeder Teil sich zu $S + 1$ aufsummiert. Damit müssen die Zahlen $S - A + 1$ und $A + 1$ in verschiedenen Teilmengen sein. Die Zahlen, die $S - A + 1$ zu $S + 1$ ergänzen, haben die Summe A und bilden eine Lösung für KP*.

Somit ergeben sich zwei Teilmengen: in der ersten ist die Lösung von KP*, zusammen mit der Zahl $S - A + 1$, während die andere alle Zahlen enthält, die nicht zur Lösung von KP* gehören, zusammen mit der Zahl $A + 1$.

Somit gilt: KP* $\leq_{\mathcal{P}}$ PART □

Aufgabe 13

Zu zeigen: VERTEXCOVER $\leq_{\mathcal{P}}$ 0/1-ILP.

Sei $G = (V, E)$ ein Graph und $V' \subseteq V$ eine Knotenüberdeckung von G gegeben. Daraus wird wie folgt eine $m \times n$-Matrix A gebildet, wobei $m = |E|$ und $n = |V|$.

Für jede Kante $\{u, v\}$ wird eine Zeile in der Matrix A wie folgt gefüllt: an der Spalte des Knotens u und v wird eine 1, in alle anderen Spalten eine 0 geschrieben.

Aus der Knotenüberdeckung V' wird wie folgt der Vektor $\vec{x}$ gebildet:

$$x_i = \begin{cases} 1, & i \in V' \\ 0, & \text{sonst} \end{cases}$$

Der Vektor $\vec{b}$ besteht aus n Einsen.

Die Konstruktion der Matrix, sowie der beiden Vektoren ist sicherlich in polynomieller Zeit machbar.

$V' \subseteq V$ ist Knotenüberdeckung von $G = (V, E)$

- genau dann, wenn jede Kante $\{u, v\} \in E$ wenigstens einen Knoten aus V' enthält
- genau dann, wenn in der Matrix A in der Zeile der Kante $\{u, v\}$ in der Spalte u und v eine 1, ansonsten nur Nullen stehen und der Vektor $\vec{x}$ an der in der Zeile u oder v eine 1 enthält
- genau dann, wenn für jede Zeile i gilt: $\sum_{j=1}^{n} a_{i,j} \cdot x_j \geq b_i$
- genau dann, wenn gilt: $Ax \geq b$
- genau dann, wenn $(A, b) \in 0/1\text{-ILP}$

Somit gilt: $\text{VERTEXCOVER}^* \leq_{\mathcal{P}} 0/1\text{-ILP}$ □

Beispiel

Gegeben Graph $G = (V, E)$ mit $V = [5]$ und $E = \{\{1, 2\}, \{1, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}\}$. Für die Knotenüberdeckung $V' = \{1, 4\}$ ergibt sich:

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}, \vec{x} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \vec{b} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Es gilt: $Ax \geq b$.

Serie 6

Institut für Informatik
Algorithmen und Komplexität
Dr. Till Nierhoff

SS 2003
20. Mai 2003

Theoretische Informatik III

6. Serie

Abgabe bis zum 27. Mai 2003

Aufgabe 15 [4+6 Punkte]

Seien $x_1, \ldots, x_n$ Boole'sche Variablen.

a) Lösen sie Aufgabe b) für den Spezialfall $i = 1$.

b) Geben Sie eine CNF-Formel $\varphi(x_1, \ldots, x_n)$ an, die polynomielle Länge in n hat und genau dann erfüllt ist, wenn genau i der Variablen $x_1, \ldots, x_n$ wahr sind.

Aufgabe 16 [10 Punkte]

Sei N' eine NTM, die polynomiell zeitbeschränkt ist. Zeigen Sie, dass es dann auch eine – ebenfalls polynomiell zeitbeschränkte – NTM N mit $L(N) = L(N')$ gibt, deren Berechnung in eine (nichtdeterministische) Ratephase und eine darauffolgende, deterministische Prüfphase unterteilt ist. Dabei ist die Konfiguration zu Beginn der Prüfphase von der Gestalt

$$\cdots \square r \square q_0 w \square \cdots ,$$

wobei $r \in \{0,1\}^*$ eine geratene $\{0,1\}$-Folge ist, q_0 ein ausgezeichneter Zustand und w die Eingabe.

Aufgabe 17 [mündlich]

Eine andere Möglichkeit, Laufzeit für eine nichtdeterministische Turing-Maschine N zu definieren, ist:

$$t_N(x) = \begin{cases} \text{Länge einer längsten akzeptierenden Berechnung für x,} & \text{falls } x \in L(N) \\ 1, & \text{sonst} \end{cases}$$

Ändert sich die Klasse NP, falls man diese Definition der Laufzeit zugrundelegt?

Aufgabe 15

Seien $x_1, \ldots, x_n$ Boole'sche Variablen. Für beliebiges i werden zunächst zwei Mengen von Klauseln gebildet:

- K_1 ist die Menge aller Klauseln mit jeweils $i+1$ verschiedenen negierten Variablen.
- K_2 ist die Menge aller Klauseln mit jeweils $n-(i-1)$ verschiedenen, nicht negierten Variablen.

Sei $F(x_1, \ldots, x_n) = \bigwedge K_1 \wedge \bigwedge K_2$

Zu zeigen:

1. F ist CNF-Formel
2. F hat polynomielle Länge in n
3. F ist genau dann erfüllt, wenn genau i der Variablen wahr sind

Beweis

1. F ist eine Konjuktion von Klauseln.
2. K_1 enthält $\binom{n}{i+1}$ Klauseln mit je $i+1$ Literalen.
   K_2 enthält $\binom{n}{n-(i-1)}$ Klauseln mit je $n-(i-1)$ Literalen.

   Da $\binom{n}{i+1} \leq n^{i+1}$ und $\binom{n}{n-(i-1)} = \binom{n}{n-(n-(i-1))} = \binom{n}{i-1} \leq n^{i-1}$ hat F in n polynomiell viele Klauseln mit jeweils $i+1$ bzw. $n-(i-1)$ vielen Literalen.
3. K_1 ist genau dann erfüllbar, wenn höchstens i der Variablen wahr sind. Wenn mehr als i Variablen wahr sind, gibt es in K_1 mindestens eine Klausel, die nicht erfüllt werden kann.
   K_2 ist genau dann erfüllbar, wenn mindestens i der Variablen wahr sind. Wenn weniger als i Variablen wahr sind, gibt es in K_2 mindestens eine Klausel, die nicht erfüllt werden kann.
   Aus beiden Aussagen folgt: F ist genau dann erfüllbar, wenn genau i Varialben wahr sind.

Anmerkung

Der Fall $i = n$ soll durch die Formel $F(x_1, \ldots, x_n) = \bigwedge_{j=1}^{n} x_j$ abgedeckt werden, da hier die Konstruktion von K_1 mit „i+1 verschiedenen Variablen“ nicht möglich ist. Offensichtlich ist F genau dann erfüllbar, wenn alle (bzw. i) Variablen wahr sind.

Aufgabe 16

Bei einer nichtdeterministischen Turingmaschine kann es zu jedem Zustand und gelesenem Zeichen mehrere Nachfolgezustände geben, aus denen einer nichtdeterministisch ausgewählt wird. Bei der Eingabe w durchläuft die Turinmaschine N' einen Pfad auf einen Konfigurationsbaum, dessen Wurzel die Startkonfiguration $q_0 w$ ist. Sie akzeptiert die Eingabe w, wenn sie auf diesem Pfad eine Konfiguration mit einem Endzustand erreicht oder die polynomielle Zeitbeschränkung erreicht.

Die Turingmaschine N geht wie folgt vor:

- Ratephase: N schreibt nichtdeterministisch $r \in \{0,1\}^*$ links neben die Eingabe und geht dann in die Prüfphase über.
- Prüfphase: N rekonstruiert den Konfigurationsbaum von N' und interpretiert die geratene $\{0,1\}$-Folge r als Gödelzahlen der von N' eingenommen Konfigurationen. So kann N in jedem Schritt deterministisch ihre Nachfolgekonfiguration ermitteln und w akzeptieren, falls die letzte in r codierte Konfiguration einen Endzustand enthält oder die polynomielle Zeitschranke überschritten wird.

Die Turingmaschine N ist dabei immer noch polynomiell zeitbeschränkt:

- die Ratephase ist per Definition linear
- die Konstruktion des Konfigurationsbaumes von N' ist polynomiell, da N' polynomiell zeitbeschränkt ist
- die Auswahl der Nachfolgekonfiguration in jedem Schritt ist in $|w|$ quadratisch, da N in jedem Schritt den Kopf vom gelesenen Zeichen w_i zu den Ratebits r und zurück bewegen muss

Serie 7

Institut für Informatik
Algorithmen und Komplexität
Dr. Till Nierhoff

SS 2003
27. Mai 2003

Theoretische Informatik III

7. Serie

Abgabe bis zum 3. Juni 2003

Aufgabe 18 [10 Punkte]

3Sat* ist die Sprache aller Formeln in konjunktiver Normalform, die erfüllbar sind und deren Klauseln aus jeweils drei *verschiedenen* Literalen bestehen.
Zeigen Sie, dass 3Sat* NP-vollständig ist.

Aufgabe 19 [10 Punkte]

Das Problem *Task Scheduling* ist wie folgt definiert: Gegeben eine Zahl $k \in \mathbb{N}$ und n Aufträge mit den Ausführungszeiten $t_1, \ldots, t_n \in \mathbb{N}$, dem Schlusstermin $d \in \mathbb{N}$ und den Verlusten $v_1, \ldots, v_n \in \mathbb{N}$. Dabei steht v_i für den Verlust, der entsteht, wenn Auftrag i nicht zum Termin d fertig ist. Die Fragestellung ist: Gibt es einen Ausführungsplan beginnend zum Zeitpunkt 0, der zu einem Gesamtverlust von höchstens k führt?
Formal: Gibt es eine Permutation $(i_1, \ldots, i_n)$ der Zahlen $1, \ldots, n$, so dass

$$\sum_{j=1}^{n} Verlust_j(i_1, \ldots, i_n) \leq k?$$

Dabei ist

$$Verlust_j(i_1, \ldots, i_n) := \begin{cases} v_{i_j}, & \text{falls } t_{i_1} + t_{i_2} + \cdots + t_{i_j} > d \\ 0, & \text{sonst.} \end{cases}$$

TaskScheduling ist die Sprache aller *Task-Scheduling*-Instanzen, für die es eine solche Partition gibt.
Zeigen Sie, dass TaskScheduling NP-vollständig ist.
Hinweis: Als Ausgangsproblem für die Reduktion können sie z.B. Part verwenden.

Aufgabe 18

Zu zeigen: 3SAT* ist $\mathcal{NP}$-vollständig.

Es gilt: B ist $\mathcal{NP}$-vollständig, falls

1. $B \in \mathcal{NP}$
2. $A \in \mathcal{NP} \Rightarrow A \leq_{\mathcal{P}} B$

Zu 1.

Sei F eine Boole'sche Formel. Die Sprache 3SAT* liegt in $\mathcal{NP}$, da sie von einer RVTM entschieden werden kann:

- rate nichtdeterministisch eine Belegung β
- prüfe, ob β die Formel F erfüllt

Dieser Test, zusammen mit der Überprüfung, ob jede Klausel drei unterschiedliche Literale enthält, ist polynomiell machbar.

Es gilt: 3SAT$^* \in \mathcal{NP}$

Zu 2.

Die $\mathcal{NP}$-Härte wird durch die Reduktion 3SAT $\leq_{\mathcal{P}}$ 3SAT* gezeigt:

Sei $F(x_1, \ldots, x_n)$ eine eine Boole'sche Formel in konjunktiver Normalform, deren Klauseln höchstens drei Literale enthalten. Da nur Klauseln der mit drei unterschiedlichen Literalen den Einschränkungen von 3SAT* genügen, müssen die restlichen Klauseln umgeformt werden.

Dazu werden in jenen Klauseln, die ein Literal mehrfach enthalten, neue Variablen y_i eingefügt, die jeweils ein mehrfaches Vorkommen ersetzen. So werden Klauseln der Form $(x \vee x \vee x)$ durch $(x \vee y_1 \vee y_2)$ ersetzt. Damit sich die Erfüllbarkeit von F durch die neuen Variablen nicht verändert, außerdem werden drei neue Klauseln $(x \vee \overline{y_1} \vee y_2)$, $(x \vee y_1 \vee \overline{y_2})$ und $(x \vee \overline{y_1} \vee \overline{y_2})$ eingeführt.

Die daraus entstehende Formel $F'(x_1, \ldots, x_n, y_1, \ldots, y_m)$ enthält nun nur noch Klauseln mit unterschiedlichen Literalen. Es bleibt nun, Klauseln mit weniger als drei Literalen auf drei Literale „aufzufüllen“: Klauseln der Form x werden in $(x \vee z_1 \vee z_2)$ umgeformt. Anschließend werden wieder Klauseln $(x \vee \overline{z_1} \vee z_2)$, $(x \vee z_1 \vee \overline{z_2})$ und $(x \vee \overline{z_1} \vee \overline{z_2})$ eingeführt.

Die nun entstandene Formel $F''(x_1, \ldots, x_n, y_1, \ldots, y_m, z_1 \ldots, z_o)$ enthält nun nur noch Klauseln mit genau drei unterschiedlichen Literalen.

Eine Reduktion f die von 3SAT-Instanzen auf 3SAT*-Instanzen abbildet führt eben jene Umformungen durch. Diese hängen nur von der Anzahl der Klauseln und den enthaltenen Literalen ab und sind polynomiell machbar (im

worst case enthält $f(F)$ nach den Umformungen viermal so viele Klauseln wie F).

Eingaben $x \notin$ 3SAT soll f auf $x_0 \notin$ 3SAT* abbilden.

Es gilt: 3SAT $\leq_{\mathcal{P}}$ 3SAT* und durch die Transitivität der $\leq_{\mathcal{P}}$-Relation gilt: CNF-SAT $\leq_{\mathcal{P}}$ 3SAT*.

Somit gilt: 3SAT* ist $\mathcal{NP}$-vollständig. □

Aufgabe 19

Zu zeigen: TASKSCHEDULING ist $\mathcal{NP}$-vollständig.

1. TASKSCHEDULING$\in \mathcal{NP}$, da eine nichtdeterministische Turingmaschine eine Permutation der Aufträge raten kann und anschließend durch Ausrechnen des Gesamtverlustes überprüfen kann, ob die geratene Reihenfolge korrekt war. Dies ist in polynomieller Zeit machbar.

2. Sei $(g_1, \ldots, g_n, a_1, \ldots, g_n, G, A)$ eine KP-Instanz mit

   - Gewichten g_i
   - Nutzen a_i
   - Gewichtsschranke G
   - Nutzenminimum A

   Daraus kann eine TASKSCHEDULING-Instanz gewonnen werden, indem die KP-Parameter wie folgt interpretiert werden:

   - Ausführungszeiten $t_i := g_i$
   - Verluste $v_i := a_i$
   - Schlusstermin $d := G$
   - Maximalverlust $k := \left(\sum_{i=1}^{n} a_i\right) - A$

   Sei f die Reduktion, die diese Umformung, die sicherliche polynomiell ist, vollzieht. Außerdem soll f alle $x \notin$ KP auf $x_0 \notin$ TASKSCHEDULING abbilden.

   Es gilt: $(g_1, \ldots, g_n, a_1, \ldots, g_n, G, A) \in$ KP

   - genau dann, wenn es eine Auswahl $O \subseteq [n]$ aus den Objekten $1, \ldots, n$ gibt sodass gilt: $\sum_{i \in O} g_i \leq G$ und $\sum_{i \in O} a_i \geq A$ (sei $O = \{i_1, \ldots, i_j\}$)
   - genau dann, wenn $\sum_{i \in O} g_i \leq G$ und $\sum_{i \notin O} a_i \leq (\sum_{i=1}^{n} a_i) - A$
   - genau dann, wenn $\sum_{i \in O} d_i \leq d$ und $\sum_{i \notin O} v_i \leq k$

- genau dann, wenn $Verlust_j(i_1, \ldots, i_j) = 0$ und $Verlust_m(i_1, \ldots, i_j, l_1, \ldots, l_m) \leq k$ (sei $\{l_1, \ldots, l_m\} := [n] \backslash O$)
- genau dann, für die Permutation $(i_1, \ldots, i_j, l_1, \ldots, l_m)$ der Objekte $1, \ldots, n$ gilt: $Verlust_m(i_1, \ldots, i_j, l_1, \ldots, l_m) \leq k$
- genau dann, wenn $(d_1, \ldots, d_n, v_1, \ldots, v_n, d, k) \in$ TaskScheduling

Es gilt: Kp $\leq_{\mathcal{P}}$ TaskScheduling und durch die Transitivität der $\leq_{\mathcal{P}}$-Relation gilt: CNF-Sat $\leq_{\mathcal{P}}$ TaskScheduling.

Somit gilt: TaskScheduling ist $\mathcal{NP}$-vollständig. □

Serie 8

Institut für Informatik
Algorithmen und Komplexität
Dr. Till Nierhoff

SS 2003
3. Juni 2003

Theoretische Informatik III

8. Serie

Abgabe bis zum 10. Juni 2003

Aufgabe 20 [10 Punkte]

Betrachten Sie folgenden Appoximationsalgorithmus:

> Eingabe: Graph G
>
> Finde Tiefensuchbäume auf allen Komponenten von G
>
> Gib die Menge S aller Knoten von G aus, die nicht Blatt eines gefundenen Suchbaumes sind.

Zeigen Sie, dass dies ein korrekter Approximationsalgorithmus für eine *minimale Knotenüberdeckung* mit Güte 2 ist.
Hinweis für die Güte: Zeigen Sie, dass G ein Matching auf $|S|/2$ Knoten hat.

Aufgabe 21 [10 Punkte]

Gegeben ist ein gerichteter Graph G. Das Problem AzyklischerSubgraph besteht darin, einen Subgraphen von G mit maximal vielen Kanten zu finden, der keine gerichteten Kreise enthält (d.h. aus jedem Knoten des Kreises zeigt eine Kante hinaus und eine hinein). Finden Sie einen Approximationsalgorithmus für AzyklischerSubgraph, der Güte 2 hat.
Hinweis: Knoten nummerieren und entweder die Kanten nehmen, die aufwärts gehen oder die, die abwärtsgehen, je nachdem, wovon es mehr gibt.

Aufgabe 22 [mündlich]

Im letzten Semester haben Sie den Algorithmus Kp_Dp kennengelernt, der Kp in der Zeit $O(n \cdot G)$ löst. Gilt deshalb $P = NP$?

Aufgabe 20

Behauptung

Der Algorithmus ist ein Approximationsalgorithmus für eine minimale Knotenüberdeckung der Güte 2. Zu zeigen:

1. der Algorithmus ist polynomiell
2. S ist eine Knotenüberdeckung von G
3. S hat höchstens doppelt so viele Knoten wie die minimale Knotenüberdeckung von G

Beweis

1. Da die Tiefensuche in $\mathcal{O}(m(G)) = \mathcal{O}(n^2)$ machbar ist, ist der Algorithmus polynomiell.
2. Die Suchbäume enthalten alle Knoten des Graphen G, und zwar als Blatt oder als Nicht-Blatt. Die Menge S enthält per Definition alle Nichtblätter. Jedes Blatt eines Baumes hat per Definition nur eine einzige Kante zum Rest des Baumes, nämlich zu einem Nichtblatt. Somit berührt S sowohl die Kanten zwischen den einzelnen Nicht-Blättern, als auch die Kanten zu den Blättern und ist somit sowohl eine Knotenüberdeckung für die Suchbäume, als auch für den gesamten Graph G.
3. Da alle Knoten aus S auf den Suchbäumen Nicht-Blätter waren, haben sie (mit Ausnahme der Wurzeln) mindestens zwei Nachbarn, also Kanten zu entweder Blättern $v \notin S$ oder Nicht-Blättern $w \in S$. Fügt man solche Kanten zu einem Matching M hinzu, sodass jeder Knoten aus S vom Matching berührt wird, so erhält man ein Matching mit mindestens $\frac{|S|}{2}$ Kanten. Dieses Matching ist gleichzeitig ein maximales Matching auf G, da S eine Knotenüberdeckung ist.

   Nach der Vorlesung gilt: $\bigcup M$ ist eine Knotenüberdeckung der Güte 2. □

Aufgabe 21

Algorithmus

Eingabe: Graph $G = (V, E)$ mit $V = [n]$

while $E \neq \emptyset$

  wähle Kante $e = (u, v)$ aus E aus und entferne sie aus E

  if $u < v$

    then $E_1 \leftarrow (u, v)$

    else $E_2 \leftarrow (u, v)$

if $|E_1| > |E_2|$

then Ausgabe: (V, E_1)

else Ausgabe (V, E_2)

Behauptung

Der obige Algorithmus ist ein Approximationsalgorithmus für AzyklischerSubgraph der Güte 2. Zu zeigen (sei $G' = (V, E')$ der ausgegeben Graph):

1. der Algorithmus ist polynomiell
2. G' ist ein Subgraph von G
3. G' ist azyklisch
4. G' hat mindestens halb so viele Kanten wie der maximale azyklische Subgraph von G

Beweis

1. Für eine Kantenmenge E mit m Kanten muss der Algorithmus insgesamt $2m$ Vergleiche durchführen, um zu entscheiden, ob die jeweilige Kante zu E_1 oder zu E_2 hinzugefügt werden soll. Bei maximal $\binom{n}{2}$ Kanten arbeitet der Algorithmus in $\mathcal{O}(n^2)$.
2. G' ist offensichtlich ein Subgraph von G, da $E' \subseteq E$ gilt.
3. Lemma: Die Knoten jedes gerichteten azyklischen Graphen lassen sich so durchnummerieren, dass für jede Kante $(u, v) \in E$ gilt: Nummer von $u <$ Nummer von v. (aus Schöning: „Ideen der Informatik“)

   Wurde vom Algorithmus E_1 gewählt, ist G' offensichtlich azyklisch, da $u < v$ für jede Kante $(u, v) \in E_1$ vorausgesetzt wurde. Wurde vom Algorithmus hingegen E_2 ausgewählt, so kann durch andere Durchnummerierung der Knoten (beispielsweise mit anderem Vorzeichen, also $V' := \{-u \mid u \in V\}$ und $E' := \{(-u, -v) \mid (u, v) \in E\}$) das Lemma erfüllt werden. G' ist auch in diesem Falle azyklisch.
4. Der Algorithmus teilt die Kantenmenge E stets in zwei Teilmengen E_1 und E_2 auf und gibt die größere zurück. Sei nun E^* die Kantenmenge eines azyklischen Subgraphen von G mit maximal vielen Kanten. Es können Kanten $e \in E^*$ existieren, die nicht vom Algorithmus erfasst werden und somit $e \notin E'$ gilt. Diese Kanten gehören dann zur Menge $E'' := E \backslash E'$. Es gilt: $|E'| \geq |E''|$. Es können also maximal $|E'|$ viele Kanten e existieren mit $e \in E^*$ und $e \notin E'$. Für diesen Fall gilt: $|E'| = |E''|$ und $|E'| = \frac{1}{2}|E|$. Da außerdem gilt: $E' \subseteq E^*$ und wir $|E'|$ Kanten e mit $e \in E^*$ und $e \notin E'$ vorausgesetzt haben, gilt: $E' \cup E'' = E^* = E$. Für $|E'| = |E''|$ gilt also $|E'| = \frac{1}{2}|E^*|$.

   Für den allgemeinen Fall $|E'| \geq |E''|$ gilt: $|E'| \geq \frac{1}{2}|E^*|$. Der Algorithmus findet somit einen azyklischen Subgraphen von G, der mindestens halb so viele Kanten wie der maximale azyklische Subgraph von G hat.

□

Serie 9

Institut für Informatik
Algorithmen und Komplexität
Dr. Till Nierhoff

SS 2003
10. Juni 2003

Theoretische Informatik III

9. Serie

Abgabe bis zum 17. Juni 2003

Aufgabe 23 [10 Punkte]

Ein *k-Cut* ist eine Partition der Knoten eines Graphen in k Teile. Unter der *Größe eines k-Cut* versteht man die Anzahl der Kanten, die zwischen *verschiedenen* Teilen des k-Cut verlaufen. Das Optimierungsproblem $\textsc{MaxCut}_k(G)$ sucht für konstantes k nach einem größten k-Cut.
Geben sie einen Approximationsalgorithmus der Güte $1 - \frac{1}{k}$ für $\textsc{MaxCut}_k(G)$ an.

Aufgabe 24 [3+7 Punkte]

Gegeben sei eine Instanz $n, g_1, \ldots, g_n, a_1, \ldots, a_n, A, G$ des Rucksackproblems.

a) Der Greedy-Algorithmus sortiert zuerst die Objekte nach dem Quotienten aus Nutzen und Gewicht (d.h. es gelte $a_1/g_1 \geq a_2/g_2 \geq \ldots \geq a_n/g_n$) und versucht dann nacheinander die Objekte $1, \ldots, n$ in den Rucksack zu packen.
Zeigen Sie, dass der Greedy-Algorithmus kein R-Approximationsalgorithmus für irgendein konstantes R ist.

b) Ein modifizierter Greedy-Algorithmus gebe das Maximum der Lösung des obigen Greedy-Algorithmus und einer Lösung, die nur aus einem Objekt besteht, aus. Zeigen Sie, dass das ein 2-Approximationsalgorithmus ist.

Hinweis zu b): Sei $I \subseteq \{1, \ldots, n\}$ mit Nutzen $a(I)$ die Lösung des Greedy-Algorithmus, und sei $i := \min\{j : j \notin I\}$. Zeigen Sie, dass die optimale Lösung einen Nutzen von höchstens $a(I) + a_i$ hat, und folgern Sie die Aussage daraus.

Aufgabe 23

Algorithmus

Eingabe: Graph $G = (V, E), k$

foreach $u \in V$

bestimme Menge M_i, zu der u die wenigsten Kanten hat, also die Menge M_i, für die gilt: $|\{\{u, v\} \mid v \in M_i\}|$ ist minimal.

$M_i \leftarrow u$

endfor

Ausgabe: $M_1, \ldots, M_k$

Falls nach Ablauf des Algorithmus noch leere Mengen existieren, werden in diese jeweils eine Kante aus anderen Mengen verschoben.

Behauptung

Der obige Algorithmus ist ein Approximationsalgorithmus für $\textsc{MaxCut}_k(G)$ der Güte $1 - \frac{1}{k}$. Zu zeigen:

1. der Algorithmus ist polynomiell
2. $M_1, \ldots M_k$ ist ein $k - Cut$
3. der gefundene k-Cut hat mindestens $1-\frac{1}{k}$ so viele Kanten wie eine optimale Lösung

Beweis

1. Der Algorithmus durchläuft die Schleife bei n Knoten genau n mal. Der Test auf Nachbarschaft muss maximal $\binom{n}{2}$ (Anzahl der Kanten) mal durchgeführt werden.
2. Die Knotenmenge V wird in k Mengen aufgeteilt, was der Definition des k-Cuts genügt.
3. Ein einzusortierende Knoten hat in den Mengen $M_1, \ldots, M_k$ höchstens $\frac{d(v)}{k}$ Nachbarn. Insgesamt haben die Knoten in den Mengen $M_1, \ldots, M_k$ den Grad $\frac{\sum_{v \in V} d(v)}{k}$. Dies entspricht $\frac{\sum_{v \in V} d(v)}{2k}$ Kanten innerhalb der Mengen.

   Diese Kanten werden bei der Größenbestimmung des k-Cut nicht berücksichtigt. Der Algorithmus findet also höchstens

   $$|E| - \frac{\sum_{v \in V} d(v)}{2k} = |E| - \frac{2|E|}{2k} = |E| - \frac{|E|}{k} = |E| \cdot \left(1 - \frac{1}{k}\right)$$

   Kanten. Bei $|E|$ Kanten ist die maximale Größe eines k-Cuts $|E|$. Für die Güte des Algorithmus gilt also mindestens:

   $$\frac{|E| \cdot \left(1 - \frac{1}{k}\right)}{|E|} = 1 - \frac{1}{k}$$

□

Aufgabe 24

Teilaufgabe a

Behauptung: Der Greedy-Algorithmus ist kein R-Approximationsalgorithmus für irgendein konstantes R.

Beweis durch Kontraposition:
Annahme: Der Greedy-Algorithmus ist ein R-Approximationsalgorithmus ($R \in \mathbb{Q}_+$). Dann findet er für eine beliebige Instanz $I = (g_1, \ldots, g_n, a_1, \ldots, a_n, A, G) \in \textsc{Kp}$ eine Lösung $L \subseteq \{1, \ldots, n\}$, für die gilt: $a(L) \geq \frac{1}{R} \cdot opt(I)$.

Insbesondere muss diese auch für $I' = (R+2, 1, R+1, 1, 1, R+2) \in \textsc{Kp}$ gelten. Da der Algorithmus jedoch auf Grundlage der Quotienten aus Nutzen und Gewicht die Bepackung bestimmt, findet der Algorithmus die Lösung $L' = \{2\}$ mit $a(L') = 1$. Die optimale Lösung wäre allerdings $L^* = \{1\}$ mit $a(L^*) = R + 1$.

Da der Algorithmus laut Annahme ein R-Approximationsalgorithmus ist, muss gelten: $a(L') \geq \frac{1}{R} \cdot a(L^*)$, jedoch ist $1 \ngeq \frac{1}{R} \cdot (R+1)$. Dies ist ein Widerspruch zur Annahme, der Greedy-Algorithmus ist ein R-Approximationsalgorithmus.

Serie 10

Institut für Informatik
Algorithmen und Komplexität
Dr. Till Nierhoff

SS 2003
17. Juni 2003

Theoretische Informatik III

10. Serie

Abgabe bis zum 24. Juni 2003

Aufgabe 25 [5+3+7+5 Punkte]

Gegeben seien n Aufträge mit den Laufzeiten $p_1, \ldots, p_n$, sowie eine Anzahl m von Maschinen, die parallel arbeiten. Das Optimierungsproblem MachineScheduling besteht darin, eine Reihenfolge und Maschinenzuordnung für die Aufträge zu finden, so dass die benötigte Zeit bis zum Abarbeiten des letzten Auftrags minimal ist. Betrachten Sie den folgenden Algorithmus GreedySum:

Eingabe: n, $p_1, \ldots, p_n$, m
For x in $\{1, \ldots, n\}$ do
  Hänge Auftrag x an die Warteschlange der Maschine mit der momentan kleinsten Summe der Laufzeiten der zu bearbeitenden Prozesse
EndFor

a) Zeigen Sie, dass die Entscheidungsversion von MachineScheduling für $m = 2$ NP-vollständig ist.

b) Zeigen Sie, dass die Entscheidungsversion von MachineScheduling für alle $m > 2$ NP-vollständig ist.

c) Zeigen Sie, dass GreedySum eine Approximationsgüte von $2 - \frac{1}{m}$ hat.
Hinweis: Sei C die Zeit, die der Algorithmus benötigt, bis der letzte Auftrag abgearbeitet ist. Sei M eine Maschine, die die volle Zeit C benötigt. Sei i der Auftrag (mit Laufzeit p_i) der als letztes auf M verteilt wird. Sei C_{opt} die von der optimalen Lösung benötigte Zeit. Zeigen Sie, dass

$$m(C - p_i) + p_i \leq \sum_{j=1}^{n} p_j \leq mC_{opt}$$

und leiten Sie daraus das Ergebnis ab.

d) Zeigen Sie, dass es für jedes m ein n und eine Instanz $n, p_1, \ldots, p_n, m$ gibt, so dass GreedySum auf dieser Instanz eine Güte von genau $2 - \frac{1}{m}$ hat.

Aufgabe 25

Teilaufgabe a

Zu zeigen: die Entscheidungsversion von MachineScheduling für $m = 2$ ist $\mathcal{NP}$-vollständig:

Part $\leq_{\mathcal{P}}$ 2-MS Part:

- Eingabe ist eine Menge A mit den Elementen $a_1, \ldots, a_n \in A$
- Ausgabe ist eine Aufteilung der Elemente in 2 Untermengen A_1 und A_2, so dass die jeweiligen Summen der Elemente gleich sind ($\sum A_1 = \sum A_2$).

Sei (A, s) eine Instanz von Part. Sei $t = \frac{\sum s(n)}{2}$ mit $s(1) = p_1$. Dann ist (A, s, t) eine Instanz von 2-MS, wobei t der maximale Zeitaufwand zur Abarbeitung der Aufgaben ist.

($\Rightarrow$) Wenn (A, s) eine „Ja"-Instanz der Entscheidungsversion für Part ist, dann ist $\sum A_1 = \sum A_2 = t$, und (A, s, t) ist auch eine „Ja"-Instanz der Entscheidungsversion für 2-MS.

($\Leftarrow$) Wenn (A, s, t) eine „Ja"-Instanz der Entscheidungsversion des 2-MS, dann gilt: A_1 und A_2 seien die Aufgaben für die Maschinen M_1 und M_2 des 2-MS. Der maximale Zeitaufwand, um A_1 zu bearbeiten ist t, das gilt auch für A_2.
Also ist $\sum A_1 = \sum A_2 = 2t = \sum s(n)$ aus Part. (A, s) ist also eine „Ja"-Instanz für Part.

Da Part $\mathcal{NP}$-vollständig ist und Part $\leq_{\mathcal{P}}$ 2-MS gilt: 2-MS ist $\mathcal{NP}$-vollständig. □

Teilaufgabe b

Zu zeigen: die Entscheidungsversion von MachineScheduling für beliebiges m ist $\mathcal{NP}$-vollständig:

Wir nehmen uns eine Instanz (A, s, t) aus 2-MS und formen sie um in eine m-MS-Instanz (A', s', t) (also zeigen, dass 2-MS $\leq_{\mathcal{P}}$ m-MS gilt), indem wir folgende Substitutionen vornehmen:

- $A' = A \cup \{v\} \cup \{v\} \ldots \cup \{v\}$ mit $m - 2$ Kopien von $\{v\}$, wobei $v \notin A$
- $s(v) = t$ und $s'(x) = s(x) \quad \forall x \in A$

($\Rightarrow$) Wenn (A, s, t) eine „Ja"-Instanz der Entscheidungsversion des 2-MS ist, dann entspreche die Aufteilung A für 2 Maschinen der Aufteilung A' für m Maschinen (mit $A' = A \cup \{v\} \ldots \cup \{v\}$). Die $m - 2$ Aufgaben werden zusätzlich zugewiesen und t wird als Zeitlimit für alle Aufgaben berücksichtigt. (A', s', t) ist dann also eine „Ja"-Instanz

der Entscheidungsversion des m-MS.

($\Leftarrow$) Die ja-Instanz der Entscheidungsversion des m-MS (A', s', t) ist nun wie folgt in eine ja-Instanz der Entscheidungsversion des 2-MS (A, s, t) umzuwandeln:

- man betrachte die Aufgabenunterteilungen $A_1, \ldots, A_n$, keine von ihnen kann gleichzeitig die Aufgaben v und x beinhalten, also gilt $s'(v) + s'(x) = t + s(x) > t$
- $m-2$ der Aufgaben aus A' müssen v enthalten, weil es ja $m-2$ Kopien von v gibt.
- wenn nun A_i und A_j die Aufgabenunterteilungen sind, die übrig bleiben (also $m-(m-2)$), so enthalten diese nur Aufgaben, die unterschiedlich von $\{v\}$ sind.

(A, s, t) ist dann also eine „Ja“-Instanz der Entscheidungsversion des 2-MS.

Da nach Teilaufgabe a) gilt: 2-MS ist $\mathcal{NP}$-vollständig und 2-MS $\leq_{\mathcal{P}}$ m-MS gilt, gilt: m-MS ist $\mathcal{NP}$-vollständig. $\square$

Bewertungen

- Aufgabe 1: alles OK [8/8]
- Aufgabe 2: alles OK [12/12]
- Aufgabe 3: alles OK [10/10]
- Aufgabe 4a: Korrektheitsbeweis fehlt [2/4]
- Aufgabe 4b: alles OK [6/6]
- Aufgabe 6a: alles OK [4/4]
- Aufgabe 6b: Fehler im Such-Algorithmus: Dreiecke werden nicht korrekt erkannt [3/6]
- Aufgabe 7a: alles OK [4/4]
- Aufgabe 7b: Tippfehler im Such-Algorithmus [4/6]
- Aufgabe 9: Missverständnisse [8/10]
- Aufgabe 10: Fragezeichen am „o.B.d.A.“ und Tippfehler [7/10]
- Aufgabe 12: alles OK [10/10]
- Aufgabe 13: fehlt: „x darf höchstens k Einsen enthalten“ [8/10]
- Aufgabe 15: alles OK [10/10]
- Aufgabe 16: Gödelzahlen nicht polynomiell [6/10]
- Aufgabe 18: Umformung der Klausel nicht ausführlich genug erklärt, auffüllen nicht notwendig [6/10]
- Aufgabe 19: Tippfehler $(a_1, \ldots g_n)$, d_i nicht erklärt [6/10]
- Aufgabe 20: nicht schlüssig [1/10]
- Aufgabe 21: nicht schlüssig, Umkehrschluss des Lemmas nicht OK [4/10]
- Aufgabe 23: teilweise ungenau [7/10]
- Aufgabe 24a: alles OK [3/3]

- Aufgabe 24b: nicht gemacht [0/7]
- Aufgabe 25a: unvollständig (habe Text aus Buch übersetzt) [0/5]
- Aufgabe 25b: unverständlich (habe Text aus Buch übersetzt) [0/3]
- Aufgabe 25c: nicht gemacht [0/7]
- Aufgabe 25d: nicht gemacht [0/5]