Algorithmen und Datenstrukturen

Übungsgruppe 14

Utz Pöhlmann
4poehlma@informatik.uni-hamburg.de
6663579

Louis Kobras
4kobras@informatik.uni-hamburg.de
6658699

Rene Ogniwek
reneogniwek@gmx.net
6428103

Steffi Kaussow
s.kaussow@gmail.com
6414862

23. November 2015

Punkte für den Hausaufgabenteil:

4.1	4.2	4.3	Σ

1 Zettel vom 09. 10. – Abgabe: 23. 11.

1.1 Übungsaufgabe 4.1

[| 3]

1.1.1 Aufgabe 4.1.1

Eingabe: Eine endliche Menge U und eine Größe $s(u) \in \mathbb{N}$, eine Kapazität $K \in \mathbb{N}$ und eine Zahl $n \in \mathbb{N}$.
Frage: Gibt es eine Partitionierung von U in paarweise disjunkten Teilmengen $U_1, U_2, \ldots, U_n$ (d.h. die U_i enthalten zusammen alle Elemente aus U und jedes Element aus U ist in genau einem der U_i) derart, dass $\Sigma_{u \in U_i} s(u) \leq K$ für jedes i gilt?

Es kann eine Variante des Binary Sort Algorithmus angewandt werden:

```
PARTITION(SET s, INT c)
1  if sum(s) < c
2        return true
3  elseif max(s) > c
4        return false
5  else
6        Partition(s[0; (s.size/2)], c)
7        Partition(s[(s.size/2); s.size], c)
```

Ist die gesamte Größe von s bereits kleiner gleich K, kann der Algorithmus sofort terminieren und `true` zurückgeben. Andernfalls wird U halbiert und für die Teilmengen wird erneut geprüft. Dies wird rekursiv fortgeführt, bis eine Ebene erreicht wurde, auf der die Größe aller Teilmengen kleiner gleich K ist.
Da bereits zu Beginn geprüft wird, ob ein einzelnes Element größer als die Kapazität ist, ist gewährleistet, dass bei rekursiven Aufruf der Algorithmus spätestens dann terminiert, wenn jede Teilmenge genau die Mächtigkeit 1 hat. Ist ein einzelnes Element von U bereits größer als K, so ist das Problem nicht lösbar und der Algorithmus terminiert, `false` zurückgebend.
Durch die jeweilige Halbierung wird die Funktion höchstens $\log n$ mal rekursiv aufgerufen. $sum(s)$ und $max(s)$ haben jeweils eine lineare Laufzeit aus $\mathcal{O}(n)$. Damit werden insgesamt $a \cdot b \cdot n \cdot \log n + f(n) + g(n)$[1] Schritte zur vollständigen Berechnung benötigt, die obere Schranke liegt also in $\mathcal{O}(n \cdot \log n)$. Dies ist nicht polynomiell und liegt somit in *NP*.

1.1.2 Aufgabe 4.1.2

Eingabe: Ein endliches Alphabet Σ, eine endliche Menge $S \subset \Sigma*$ von Worten und eine Zahl $n \in \mathbb{N}$.
Frage: Gibt es ei Wort $w \in \Sigma*$ mit $|w| \geq n$ derart, dass w ein Teilwort von jedem $x \in S$ ist?

Es wird das kürzeste Wort $x \in S$ gewählt (Laufzeit $\mathcal{O}(S.length)$). Nun wird ein Teilwort mit der Länge n aus x gewählt und geprüft, ob es in allen anderen Wörtern aus S auch existiert (Laufzeit $\mathcal{O}(S.length)$). Ist dies der Fall, wird `true` ausgegeben, sonst wird das nächste Teilwort aus x mit der Länge n gewählt (Laufzeit $\mathcal{O}(k); k \in \mathbb{N}$). Wurde erfolglos über x iteriert, bricht der Algorithmus ab und gibt `false` zurück. Die Laufzeit ist konstant in $\mathcal{O}(a); a \in \mathbb{N}$.
Die Gesamtlaufzeit liegt damit bei $\mathcal{O}(a + k + (2 \cdot S.length))$.

1.2 Übungsaufgabe 4.2

[| 7]

[1] $a, b, f(n), g(n)$ sind Faktoren bzw. Konstanten, die in $sum(s)$ und $max(s)$ entstehen können, allerdings vernachlässigbar sind

1.2.1 Aufgabe 4.2.1

Zegen Sie, dass P und NP jeweils gegenüber Konkatenation abgeschlossen sind (d.h. mit $L_1, L_2, \in P$ ist auch $L_1 \cdot L_2 \in P$ und entsprechend für NP).

Für *P*: Seien $L_1, L_2 \in P, w_1 \in L_1, w_2 \in L_2, w_3 = w_1 \cdot w_2$. Sei außerdem $w_3 = b_1 b_2 b_3 b_4 ... b_n; w_3 \in L_3; L_3 = L_1 \cdot L_2$.

Wähle $w_1 = b_1 b_2 b_3 \dots b_i$ und $w_2 = b_{i+1} b_{i+2} ... b_n$

Solange $w_1 \notin L_1 \wedge w_2 \notin L_2$ gilt, erhöhe i um 1 (als Startwert für i gilt 0.) Dies teilt das Wort in zwei Teilworte und wir wissen nun, wo wir w_3 teilen müssen, um w_1 und w_2 zu erhalten. Bei $i = 0$ gilt $w_1 = \lambda$, bei $i = n$ gilt $w_2 = \lambda$. Nun wird w_1 wie vorher in L_1 in Polynomialzeit gelöst. Ebenso w_2 in L_2.
Die Zerlegung benötigt im schlimmsten Fall $n + 1$ Ausführungen, da es $n + 1$ mögliche Zerteilungen von w_3 gibt. Somit beträgt die Gesamtlaufzeit die Zeit, die w_1 in L_1 benötigt (ein Polynom) plus die Zeit, die w_2 in L_2 benötigt (auch ein Polynom) mal die Zeit, die das Zerschneiden benötigt $(n + 1)$, also: $(Polynom(L_1) + Polynom(L_2)) * (n + 1)$. Dies ist allerdings wieder ein Polynom.

Für *NP*: Seien $L_1, L_w \in NP, w_1 \in L_1, w_2 \in L_2 und w_3 = w_1 \cdot w_2 und w_3 = b_1 b_2 b_3 b_4 ... b_n und w_3 \in L_3 und L_3 = L_1 \cdot L_2$.

Wähle $w_1 = b_1 b_2 b_3 \dots b_i$ und $w_2 = b_{i+1} b_{i+2} \dots b_n$.

Solange $w_1 \notin L_1$ und $w_2 \notin L_2$ gilt, erhöhe i um 1 (als Startwert für i gilt 0.)
Dies teilt das Wort in zwei Teilworte und wir wissen nun, wo wir w_3 teilen müssen, um w_1 und w_2 zu erhalten. Bei $i = 0$ gilt $w_1 = \lambda$, bei $i = n$ gilt $w_2 = \lambda$. Nun wird w_1 wie vorher in L_1 in Nichtpolynomialzeit gelöst. Ebenso w_2 in L_2.
Die Zerlegung benötigt im schlimmsten Fall $n + 1$ Ausführungen, da es $n + 1$ mögliche Zerteilungen von w_3 gibt. Somit beträgt die Gesamtlaufzeit die Zeit, die w_1 in L_1 benötigt (ein Nichtpolynom) + die Zeit, die w_2 in L_2 benötigt (auch ein Nichtpolynom) $*$ die Zeit, die das Zerschneiden benötigt $(n + 1)$, also: $(Nichtpolynom(L_1) + Nichtpolynom(L_2)) * (n + 1)$ Dies ist allerdings wieder ein Nichtpolynom.

1.2.2 Aufgabe 4.2.2

Geben Sie unter der Annahme $P = NP$ einen Algorithmus an, der in polynomieller Zeit zu einer aussagenlogischen Formel eine erfüllende Belegung bestimmt, sofern eine existiert. Begründen Sie kurz, warum Ihr Algorithmus das Gewünschtete leistet.

Brute Force:
Wir setzen für jedes Literal 1 ein und testen.
Erfüllt die Belegung nicht, ersetzen wir eine 1 durch eine 0 und prüfen dann alle Belegungen, die aus $(n - 1)$ 1 und einer 0 bestehen. Wenn die Belegung wieder nicht erfüllt, wird eine weitere 1 durch eine 0 ersetzt, solange, bis die Belegung aus 0^n besteht oder eine erfüllende Belegung gefunden wurde. Handelt es sich bei 0^n auch nicht um eine erfüllende Belegung, so existiert keine.

Dieser Algorithmus benötigt im schlimmsten Fall $\Sigma_{k=0}^{n} \binom{n}{k} = 2^n$ viele Schritte. Dies liegt in NP und damit unter der Annahme $P = NP$ auch in P. Damit ist es in polynomieller Zeit lösbar.

1.2.3 Aufgabe 4.2.3

Zeigen Sie, dass unter der Annahme $P = NP$ jedes $J \in P$ außer $L = \emptyset$ und $L = \Sigma^*$ NP-vollständig ist. Warum gilt diese Aussage nicht für die beiden ausgeschlossenen Fälle?

Sei $P = NP$. Damit sind alle Probleme aus NP auch in Polynomialzeit lösbar. Zudem ist jedes Problem aus P in Polynomialzeit auf jedes andere Problem aus P reduzierbar (, da es sich bei allen Problemen in P um Polynome handelt,) was genau der Definition von NP entspricht. Allerdings sind in P nur endliche Teilmengen von Σ^+, denn $\emptyset$ ist nicht entscheidbar, ebenso wie unendliche Sprachen (Σ^*). Somit sind sie auch nicht in P. Da jedoch $P = NP$ gilt, liegt der Rest aus P ebenfalls in NPC. Somit ist, wenn $P = NP$ gilt, jedes L aus P auch in NP, außer der Leeren Menge und dem unendlichen Σ^*.

1.3 Übungsaufgabe 4.3

[| 6]

1.3.1 Aufgabe 4.3.1

Sei `2-SAT` die Menge aller (sinnvoll codierten) aussagenlogischen Formeln, die mindestens zwei erfüllende Belegungen haben. Zeigen Sie, dass `2-SAT` NP-vollständig ist.

Ein Algorithmus für `2-SAT`:
Als Algorithmus für `2-SAT` kann eine modifizierte Version des Brute Force Algorithmus' aus 1.2.2 dienen:
Es wird jede mögliche Belegung für eine aussagenlogische Formel durchprobiert und geschaut, ob sie erfüllend ist. Wenn sie erfüllend ist, wird ein Zähler um 1 inkrementiert. Sollte dieser Zähler die Zahl 2 erreichen, wird `"true"` ausgegeben, sonst wird die nächste Belegung probiert. Hat der Zähler, nachdem alle Belegungen durchprobiert wurden, noch nicht die 2 erreicht, wird `"false"` ausgegeben.
Dieser Algorithmus läuft im schlimmsten Fall in $\Sigma_{k=0}^{n}\binom{n}{k} = 2^n$ vielen Schritten.
Dies liegt in NP und zeigt daher `2-SAT` $\in NP$.

Wir wissen aus der Vorlesung: `SAT` $\in NPC$
Reduktion von `SAT` auf `2-SAT`:
Angenommen Automat A löst `2-SAT`, dann hat A einen Unterautomaten B, der ausgibt, ob eine beliebige aussagenlogische Formel erfüllbar ist. A gibt nun `"true"` aus, wenn B für eine Formel mindestens 2 mal `"true"` ausgegeben hat.
Dieser Automat B wäre allerdings genau die Lösung zu unserem `SAT`-Problem, welches laut der Vorlesung ja sogar NPC ist. Daher kann dieser Automat B nicht existieren, woraus folgt, dass A auch nicht existieren kann.
Dies zeigt `2-SAT` $\in NPH$.
Somit muss `2-SAT` in der Schnittmenge von NP und NPH sein, also gilt `2-SAT` $\in NPC$.

1.3.2 Aufgabe 4.3.2

Die folgende Beschreibung verallgemeinert das Spiel *Minesweeper* auf einen ungerichteten Graphen: Sei G ein ungerichteter Graph. Jeder Knoten von G enthält entweder eine einzelne *Mine* oder ist leer. Der Spieler kann einen Knoten wählen. Ist es eine Mine, hat er verloren. Ist der Knoten leer, dann wird dieser mit der Anzahl der direkt benachbarten Knoten beschriftet, die eine Mine enthalten. Der Spieler gewinnt, wenn alle leeren Knoten gewählt wurden. Das Problem ist nun folgendes: Gegeben ein Graph zuzüglich einiger beschrifteter Knoten, ist es möglich Minen so auf die verbleibenden Knoten abzulegen, dass jeder Knoten v, der mit k beschriftet ist, genau k direkt benachbarte Knoten besitzt, die eine Mine enthalten? Formulieren Sie dieses Problem sinnvoll als formale Sprache (und füllen Sie dadurch die Lücken in obiger Beschreibung) und zeigen

Sie dann, dass dieses Problem *NP*-vollständig ist.

Eingabe: Ein ungerichteter Graph $G = (V, E)$ und eine Funktion f, die jedem Knoten v aus V einen Wert $k \in \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ zuweist. Dieser Wert kann sich von Knoten zu Knoten unterscheiden.
Frage: Gibt es eine Verteilung von Minen U, sodass jeder Knoten v aus V genau k Nachbarknoten hat, die eine Mine besitzen?

Algorithmus: Wir gehen jeden Knoten durch und belegen - sofern die Zahl auf dem Knoten noch nicht der Anzahl an Nachbarknoten mit einer Mine entspricht - einen beliebigen Nachbarknoten mit einer Mine, bis die Zahl stimmt. Sobald wir an einem Knoten ankommen, der bereits mehr Nachbarknoten mit Minen besitzt, als seine Zahl groß ist, gehen wir einen Knoten zurück und merken uns die bisher gewählte Konfiguration als *falsch* und wählen eine andere. Sollten alle möglichen Konfigurationen - an der Anzahl $\binom{8}{k}$, wobei k der Wert des Knotens ist - als *falsch* markiert worden sein, wird einen Knoten zurück gegangen, die dortige aktuelle Konfiguration als *falsch* markiert und mit dieser neuen Kombination fortgefahren. Wobei jetzt sämtliche Konfigurationen der Folgeknoten wieder möglich sind.

Wird auf diese Weise bis zum ersten Knoten zurückgekehrt und werden dort sämtliche Konfigurationen als *falsch* markiert, gibt der Algorithmus `"false"` aus. Sollte allerdings jede Mine verteilt sein, ohne dass es zu einem Konflikt mit der Zahl auf einem der Knoten kommt, wird `"true"` ausgegeben.
Dies hat eine maximale Laufzeit von $\prod_{i=1}^{n} \binom{8}{k_i} \approx \binom{8}{k}^n \leq \binom{8}{4}^n = 70^n$, wenn für jeden Knoten alle Kombinationen durchlaufen werden müssen, wobei $\binom{8}{k}$ konstant und höchstens $70 = \binom{8}{4}$ ist.
Dies liegt in *NP*.

Reduktion: Wir können es nicht sinnvoll auf ein anderes Problem überführen, damit ist es offensichtlich schwerer als alle Probleme in NP und somit in NPH. Damit ist es auch in *NPC*.