Lehrstuhl für Informatik 1
Prof. Dr. Berthold Vöcking
Alexander Skopalik
Matthias Englert

WS 2007/08
7.1.2008

Übung zur Vorlesung
Berechenbarkeit und Komplexität

Lösung Blatt 8

Aufgabe 8.1: **(5+5)**

Der euklidische Algorithmus berechnet den größten gemeinsamen Teiler von zwei natürlichen Zahlen a und b. Die in der Aufgabenstellung angegebenen Varianten beruhen auf dem folgenden Lemma.

Lemma 1 *Falls $b \neq 0$, dann gilt*

$$ggT(a,b) = ggT(a-b,b) \quad (1)$$
$$ggT(a,b) = ggT(a \bmod b, b). \quad (2)$$

Wir wollen die Laufzeiten der beiden Varianten miteinander vergleichen und zeigen, dass Variante 1 im logarithmischen Kostenmaß exponentielle Laufzeit benötigt, wohingegen Variante 2 mit polynomieller Zeit auskommt. Uns genügt also eine obere Laufzeitschranke für Variante 2 und eine untere für Variante 1.

Wir analysieren nun zunächst die zweite Variante des Euklidischen Algorithmus, die auf Gleichung (2) beruht. Dabei interessiert uns im Wesentlichen, wie viele Modulo-Operationen ausgeführt werden und wieviel jede einzelne dieser Operationen kostet.

Satz 1 *Die zweite Variante des Euklidischen Algorithmus berechnet den größten gemeinsamen Teiler von a und b mit höchstens $O(\log(a+b))$ Modulo-Operationen.*

Bevor wir den Satz beweisen, schätzen wir noch die Kosten jeder einzelnen Modulo-Operation ab. Eine Modulo-Operation kostet auf einer RAM höchstens $O((\log a + \log b)^2) = O(\log^2 ab)$ bzgl. des logarithmischen Kostenmaßes. Es folgt also eine Gesamtlaufzeit von

$$O(\log^2(ab)) \cdot O(\log(a+b)) = O(\log^3(ab))$$

für die zweite Variante des Euklidischen Algorithmus.

Beweis: Sei $x := \max\{a,b\}$ für die ursprünglichen Werte von a und b und o.B.d.A $a \geq b$. Es gilt $r = a \mod b \leq \frac{a}{2}$. Da in jeder Runde $a := b$ und $b := r$ gesetzt werden, sind nach zwei Runden sowohl a als auch b nur noch höchstens halb so groß wie x. Nach $2 \cdot k$ Runden gilt also $a, b \leq \frac{x}{2^k}$. Die Anzahl der Iterationen beträgt somit maximal $2 \cdot \log_2 x$.

Die erste Variante des Euklidischen Algorithmus unterscheidet sich von der zweiten nur bezüglich der Berechnung des Restes r. Dieser wird nicht mehr direkt durch eine Division der Zahlen a und b berechnet, sondern duch eine wiederholte Subtraktion der Zahlen. Um aber a mod b mit Hilfe der Subtraktion berechnen zu können, benötigen wir $\lceil a/b \rceil$ Subtraktionsoperationen. Es folgt unmittelbar:

Satz 2 *Die erste Variante des Euklidischen Algorithmus berechnet den größten gemeinsamen Teiler von a und b mit $\Omega(a/b)$ Subtraktionsoperationen.*

Im Allgemeinen werden sogar mehr Subtraktionen benötigt (höchstens $O(a/b \cdot \log(a+b))$), aber diese Abschätzung genügt uns bereits, um zu zeigen, dass Variante 1 im Gegensatz zu Variante 2 tatsächlich exponentielle Laufzeit benötigt. Wir stellen also fest:

Korollar 1 *Die Eingabelänge beträgt $O(\log a + \log b) = O(\log ab)$. Die erste Variante hat dann eine exponentielle Laufzeit bzgl. der Eingabelänge, die zweite eine polynomielle.*

Aufgabe 8.2: (**2+3**)

(a) Wir schreiben die Funktion als eine Liste von N Zahlen aus dem Bereich $\{1, \ldots, k\}$. Die Zahlen werden dabei mit einer festen Anzahl an Bits ($\lceil \log_2 k \rceil$) binär kodiert. Wir benötigen also keine Trennzeichen, wenn wir die Auflistung als Wort schreiben. Die Gesamtlänge ist dann genau $N\lceil \log_2 k \rceil$.

(b) Eine Permutation auf $\{1, \ldots, N\}$ ist eine bijektive Funktion $f : \{1, \ldots, N\} \to \{1, \ldots, N\}$. Wir können also auf die in Aufgabenteil (a) angegebene Kodierung zurückgreifen. Eine gültige Kodierung hat somit genau eine Länge von $N\lceil \log_2 N \rceil$. Andersgeartete Eingaben können sofort verworfen werden. Die Bijektivität überprüfen wir folgendermaßen: Ein Zähler geht alle Elemente aus $\{1, \ldots, N\}$ der Reihe nach durch. Für jede Zahl i wird geprüft, ob ihre binäre Kodierung im Wort enthalten ist. Falls nicht, so wird das Wort verworfen, ansonsten wird die gefundene Kodierung von i vom Band gelöscht. Taucht im Eingabewort jede binär kodierte Zahl aus dem Bereich $\{1, \ldots, N\}$ einmal auf, so stellt das Wort die gültige Kodierung einer Permutation dar (vorausgesetzt es hat eine Länge von $N\lceil \log_2 N \rceil$). Nach spätestens N Suchläufen auf $N\lceil \log_2 N \rceil$ Bandzellen terminiert der Algorithmus.

Aufgabe 8.3: (**10**)

Wir nehmen also an, dass die Entscheidungvariante des TSP in P ist. Wir konstruieren einen polynomiellen Algorithmus, der eine optimale Lösung des TSP berechnet, und dabei den polynomiellen Algorithmus für die Entscheidungsvariante als Unterprogramm benutzt. Gegeben sei ein gewichteter vollständiger Graph mit n Knoten und dem maximalen Kantengewicht $d_{\max}$. Man mache sich klar, dass eine Kante aus dem Graphen „gelöscht“ werden kann, indem ihre Kosten auf $nd_{\max} + 1$ gesetzt werden. Mit binärer Suche können wir in polynomieller Zeit die Kosten c einer optimalen Tour errechnen. Nun iterieren wir über alle Kanten des Graphen. Für jede Kante k testen wir mit Hilfe des Unterprogramms, ob es immer noch eine TSP-Tour mit Kosten c gibt, wenn die Kante aus dem Graphen entfernt wird. Wenn es nach dem Entfernen von k immer noch eine Tour mit Kosten c gibt, dann gibt es eine optimale Tour, die nicht über k führt, und wir entfernen die Kante k aus dem Graphen. Wenn das Entfernen von k dazu führt, dass es keine Tour mit Kosten c mehr gibt, dann existiert eine optimale Tour, die über k führt, und die Kante k wird nicht gelöscht. Nachdem wir über alle Kanten iteriert haben, sind nur noch die Kanten einer optimalen Tour im Graphen vorhanden, und diese geben wir als optimale Lösung aus.

Aufgabe 8.4: (**10**)

Wir nutzen aus, dass man zwei Elemente x und y zusammenkleben kann, indem man beide löscht und durch ein Element x' mit Gewicht $w(x') = w(x) + w(y)$ ersetzt. Damit wird erzwungen, dass x und y im gleichen Behälter landen. Falls sich dadurch der Wert der Lösung nicht ändert, so gibt es einen Schedul, der bei gleichbleibender Qualität x und y dem gleichen Behälter zuordnet. Beachte, dass die Lösung nicht besser werden kann, und wir den Wert b_0 der optimalen Lösung mit binärer Suche in Polynomialzeit finden können.

Um den optimalen Schedul zu finden, wenn wir einen Algorithmus $\mathcal{A}$ zur Lösung der Entscheidungsvariante haben, iterieren wir über alle Paare (x, y) von Elementen, kleben diesen zusammen, und prüfen mit Hilfe von $\mathcal{A}$, ob sich der Wert der Lösung dadurch verändert hat. Falls ja, müssen x und y in verschiedene Behälter. Falls sich aber der Wert der Lösung nicht verschlechtert hat, können x und y in den gleichen Behälter gelegt werden.

Wenn ein solches Paar (x, y) gefunden ist, kleben wir beide Elemente zusammen, und setzen obiges Verfahren rekursiv fort, bis die maximale Anzahl von zusammengeklebten Elementen erreicht ist.

Wenn es zu einem x kein y gibt, so dass x und y im gleichen Behälter sein dürfen, muss x im optimalen Schedul allein in einem Behälter sein. Der so erhaltene Schedul ordnet jedem Behälter höchstens ein Element zu. Zerlegt man dieses wieder in seine einzelnen Elemente, aus denen es zusammengeklebt worden ist, erhält man den genauen Schedul.

Da in jedem Rekursionsschritt die Eingabe um ein Element verringert wird, und ein Schritt nur $n-1$ viele Aufrufe von $\mathcal{A}$ benötigt, folgt die polynomielle Laufzeit dieses Verfahrens.