Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Sebastian Lamm, Demian Hespe

4. Übungsblatt zu Algorithmen II im WS 2018/2019

http://algo2.iti.kit.edu/AlgorithmenII_WS18.php
{sanders, lamm, hespe}@kit.edu

Musterlösungen

Aufgabe 1 (*Analyse: Matchings*)

a) Zeigen oder widerlegen Sie. Existiert in einem bipartiten Matching ein maximaler alternierender Pfad, dessen erste und letzte Kante nicht Teil des Matchings sind, so hat das zugehörige Flussnetzwerk einen augmentierenden Pfad.

b) Gegeben ein bipartiter Graph $G = (L \cup R, E)$ mit $|L| = |R| = n$. Bezeichne $neigh(S) \subseteq R$ die Nachbarn der Knoten aus $S \subseteq L$. Zeigen Sie, gdw. ein perfektes Matching für G existiert, gilt für alle $S \subseteq L$: $|neigh(S)| \geq |S|$.

c) An einem Tanzkurs wollen n Männer und n Frauen teilnehmen. Jeder Teilnehmer kennt genau k Teilnehmer des anderen Geschlechts (alle Bekanntschaften sind beidseitig). Ist es möglich eine Paarung zu finden, in der jeder mit einem Bekannten tanzt?

Der Tanzkurs dauert genau k Wochen. Ist es möglich für jede Woche Paarungen zu bilden, so dass immer zwei Bekannte zusammen tanzen, sich aber keine Paarung wiederholt?

Musterlösung:

a) *(Wiederholung der Modellierung zur Referenz)*
Seien L, R die zu matchenden Mengen und E die möglichen Matchings.

Im zugehörigen Flussnetzwerk sei o.b.d.A. Quelle s mit jedem Knoten aus L und alle Knoten aus R mit Senke t verbunden. Die möglichen Matchings E seien als Kanten von L nach R gerichtet. Alle Kanten haben Gewicht 1. Eine Kante aus E ist gematcht, wenn ein Fluss über sie fließt.

Gegeben ein beliebiges Matching, so enthält der Residualgraph des Flussnetzwerks nur Kanten von L nach R die nicht Teil des Matchings sind, und nur Kanten von R nach L die Teil des Matchings sind.

(Beweis)
Existiere nach Aufgabenstellung ein Matching mit einem maximalen alternierenden Pfad mit nicht gematchten Kanten am Anfang und Ende. Dann startet dieser Pfad im Residualgraphen des Flussnetzwerks bei einem Knoten $l \in L$ und endet bei einem Knoten $r \in R$. Knoten l und r sind nicht benutzt (kein Fluss über Kante (s, l) und (r, t)), da sonst der alternierende Pfad nicht maximal wäre. Damit existiert ein augmentierender Pfad von s über den alternierenden Pfad nach t mit Fluss 1.

b) Die zu beweisende Aussage ist unter dem Namen *Halls Hochzeitstheorem* bekannt.

$\Rightarrow$: (G hat perfektes Matching $\rightarrow$ f.a. $S \subseteq L : |neigh(S)| \geq |S|$)

In einem perfekten Matching ist jeder Knoten aus L mit einem Knoten aus R gematcht. Betrachte beliebiges $S \subseteq L$. Würde gelten $|neigh(S)| < |S|$, könnte ein Knoten aus S nicht gematcht werden (*pigeon hole principle*). Widerspruch!

$\Leftarrow$: (f.a. $S \subseteq L : |neigh(S)| \geq |S| \rightarrow G$ hat perfektes Matching)

Beweis durch Induktion über n. Sei $n = 1$. Es exisitiert trivialerweise ein perfektes Matching. Sei $n > 1$. Wähle $l \in L$ und $r \in neigh(L)$ beliebig. Falls die Voraussetzung auch für $G \setminus \{l, r\}$ erfüllt ist, existiert ein perfektes Matching für $G \setminus \{l, r\}$ (nach Induktion) und zusammen mit (l, r) ein perfektes Matching für G.
Ansonsten existiert $S \subset L$ mit $|neigh(S)| = |S|$. Teile G in zwei disjunkte Teilgraphen G_1, bestehend aus S und $neigh(S)$, und G_2, bestehend aus $L \setminus S$ und $R \setminus neigh(S)$, auf. Finde für jeden ein perfektes Matching:

G1: Da für alle $S' \subseteq S \subseteq L$ gilt, $neigh(S') \subseteq neigh(S)$, folgt direkt $|neigh(S')| \geq |S'|$. Damit folgt, S und $neigh(S)$ besitzen ein perfektes Matching (nach Induktion).

G2: Für $L \setminus S$ und $R \setminus neigh(S)$ ist zunächst die Voraussetung zu prüfen: Wähle $T \subseteq (L \setminus S)$ beliebig. Dann gilt $|neigh_{\text{in } G2}(T)| = |neigh_{\text{in } G}(T \cup S)| - |neigh_{\text{in } G}(S)| \geq |T \cup S| - |S| = |T|$. Die Voraussetzung ist erfüllt und damit existiert auch in diesem Teilgraphen ein perfektes Matching (nach Induktion) und damit für G.

c) Angenommen es gäbe kein perfektes Matching. Dann existiert nach Teilaufgabe b) eine Teilmenge M an Männern, die weniger als $|M|$ Frauen kennt. Dies ist nicht möglich, da jeder Mann genau k Bekanntschaften hat und jede der $|M|$ Frauen maximal k Bekanntschaften in M hat: Man kann $k \cdot |M|$ 'ausgehende' Bekanntschaften der Männer nicht auf weniger als $k \cdot |M|$ 'eingehende' Bekanntschaften der Frauen aufteilen.

Per Konstruktion sind k verschiedene Paarungen mit jeweils anderen Bekannten möglich: Nach der ersten Woche streiche die Bekanntschaften, die schon miteinander getanzt haben. Danach hat jeder nur noch $k - 1$ Bekanntschaften. Für diese existiert ein perfektes Matching. Fahre in den folgenden Wochen entsprechend fort.

Aufgabe 2 (*Schlechter Zufall*)

Gegeben sei ein randomisierter Algorithmus badBit, der keine Eingabe liest und als Ausgabe zufällig mit Wahrscheinlichkeit p die Zahl 0 und mit Wahrscheinlichkeit $q = 1 - p$ die Zahl 1 liefert. Es sei $0 < p < 1$; der konkrete Wert von p sei aber unbekannt.
Entwerfen Sie einen randomisierten Algorithmus fairBit, der keine Eingabe liest und als Ausgabe immer zufällig eine der Zahlen 0 und 1 mit Wahrscheinlichkeit 1/2 liefert.
Was können Sie über die Laufzeit Ihres Algorithmus sagen?

Musterlösung:
Algorithmusidee:

```
1: function fairBit
2:     repeat
3:         x ← badBit()
4:         y ← badBit()
5:     until x ≠ y
6:     return x
7: end function
```

Korrektheit:
$\mathbb{P}(x = 1 | x \neq y) = \frac{\mathbb{P}(x=1 \ \wedge \ y=0)}{\mathbb{P}(x \neq y)} = \frac{pq}{2pq} = \frac{1}{2}$

Laufzeit:
Im Folgenden benutzen wir folgende Überlegung (für $z \in \mathbb{R}$ mit $0 < |z| < 1$):

$$\text{sei} \qquad R = \sum_{i=1}^{\infty} i \cdot z^{i-1} = \sum_{i=0}^{\infty} (i+1) \cdot z^i$$

$$\text{dann} \qquad Rz = \sum_{i=1}^{\infty} i \cdot z^i$$

$$\text{also} \qquad R(1-z) = R - Rz = \sum_{i=0}^{\infty} z^i = \frac{1}{1-z}$$

$$\text{also} \qquad R = \frac{1}{(1-z)^2}$$

Nun ist im Algorithmus die Wahrscheinlichkeit für $x = y$ gleich $z = p^2 + q^2$, also $1 - z = 2pq$. Wegen $0 < p < 1$ ist $0 < |z| < 1$. Daher ergibt sich für den Erwartungswert der Laufzeit:

$$\sum_{i=1}^{\infty} i \cdot 2 \cdot (p^2 + q^2)^{i-1} 2pq = 4pq \sum_{i=1}^{\infty} i \cdot z^{i-1} = 4pq \frac{1}{4p^2q^2} = \frac{1}{pq}$$

Aufgabe 3 (*Entwurf+Analyse: Qualitätskontrolle)*)

Sie sind beauftragt worden, einen Algorithmus für die Abteilung zur Qualitätskontrolle zu entwerfen, der die Validierung der wöchentlichen Resultate übernimmt.
Am Ende jeder Woche erhalten Sie dafür eine Liste L der in dieser Woche produzierten Bauteile mit den drei Angaben: (Typ; ID des Bauteils selbst; ID des Bauteils, in dem es verbaut wurde). Zudem erhalten Sie eine Liste D mit den IDs aller Bauelemente, die in derselben Woche von der Qualitätskontrolle als defekt markiert worden sind. Beide Listen sind potentiell zu groß für den Hauptspeicher und enthalten die Daten in unsortierter Reihenfolge.
Ihre Aufgabe ist es zu überprüfen, ob alle Bauteile, die selbst defekte Bauteile enthalten, als defekt markiert worden sind und falls nicht, dies zu korrigieren. Zu Ihrer Übersicht steht Ihnen ein Schema zur Verfügung, aus dem man ablesen kann, welche Bauteiltypen in welchen anderen verbaut werden. Dieses Schema passt in den Hauptspeicher.

a) Erweitern Sie Liste L um die Angabe aus Liste D, ob das jeweilige Bauteil defekt ist. Sie können davon ausgehen, dass diese Information keinen zusätzlichen Speicher benötigt.

b) Definieren Sie eine totale Ordnung $\prec_b$ auf den Bauteilen, die sich für jedes Bauteil aus lokalen Informationen und dem Bauteilschema berechnen lässt. Die Ordnung soll dabei erfüllen, dass in einer nach $\prec_b$ sortierten Liste L jedes Bauteil nach allen in ihm verbauten Bauteilen steht.

c) Verwenden Sie die sortierte und um Angaben zu Defekten erweiterte Liste L, um alle Bauteile zu identifizieren, die defekte Bauteile enthalten.

   Hinweis: Verwalten Sie die als defekt identifizierten aber noch zu betrachtenden Bauteile in einer geeigneten Datenstruktur.

Musterlösung:

a) Sortiere L und D nach der ID der Bauteile und scanne beide Listen. Erhöhe dazu jeweils den Index der Liste, der auf das Bauteil mit der kleineren ID zeigt. Zeigen beide auf die gleiche ID, setze das Defekt-Flag und erhöhe beide Indizes.

b) Erstelle aus den Abhängigkeiten der Bauteiltypen einen DAG. Dies ist möglich, da kein Bauteil in sich selbst verbaut worden sein kann. Sortiere L nach der durch den DAG induzierten Teilordnung, verwende die ID der Bauteile als Sortierkriterium bei Unvergleichbarkeit.

c) Verwalte defekte Bauteile in einer *Ausschlußliste* in Form einer externen Prioritätswarteschlange (Schlüssel ist die ID des Bauteils, kleinere IDs haben höhere Priorität). Durchlaufe die sortierte Liste L. Falls beim Durchlaufen der Liste L ein defektes Bautteil t_0 gefunden wird und das defekte Bauteil in einem anderen Bauelement e_0 verbaut wurde, so füge das Bauelement e_0 in die Prioritätswarteschlange ein. Wird beim Durchlaufen der Liste L ein Bauteil t_1 gefunden, welches in der Prioritätswarteschlange die höchste Priorität hat, so wird das Bauteil t_1 ebenfalls als defekt markiert. Ist das Bauteil t_1 in einem anderen Bauelement e_1 verbaut, so füge das Bauelement e_1 in die Prioritätswarteschlange ein.

Aufgabe 4 (*Analyse: Speicherbandbreite (*)*)

Die Effizienz eines Algorithmus, der auf externem Speicher arbeitet, hängt von der gewählten Blockgröße B in Zusammenspiel mit der maximalen Bandbreite W_{max} und der durchschnittlichen Zugriffszeit T_{seek} des externen Speichers ab.
Bestimmen Sie für die folgenden Fälle die Blockgröße, für die 90% der maximalen Bandbreite ausgereizt werden kann. Sie können davon ausgehen, dass ohne Unterbrechung auf ganze Blöcke in zufälliger Reihenfolge zugegriffen wird. Etwaige Berechnungen können als asynchron angenommen werden. Daher muss für diese keine Zeit berücksichtigt werden.

a) $W_{max} = 144$ MByte$/s$, $T_{seek} = 12ms$ (*Lesen von Festplatte*)

b) $W_{max} = 550$ MByte$/s$, $T_{seek} = 100\mu s$ (*Lesen von SSD*)

c) $W_{max} = 68$ MByte$/s$, $T_{seek} = 60s$ (*Lesen von LTO Streamer*)

Musterlösung:
Ein Block kann in Zeit $T = T_{seek} + B/W_{max}$ eingelesen werden.
Mit der effektiven Bandbreite $W = B/T \stackrel{!}{=} 0.9 \cdot W_{max}$ ergibt sich

$$B = 9 \cdot W_{max} \cdot T_{seek}$$

für die gesuchte Blockgröße. Damit folgt:

a) $B = 15.552$ MByte ≈ 15 MByte

b) $B = 0.495$ MByte ≈ 500 kByte

c) $B = 36\,720$ MByte ≈ 37 GByte

Aufgabe 5 (*Analyse: Externer Stack*)

In der Vorlesung wurde eine Implementierung von *Stack* als externe Datenstruktur vorgestellt. Eine äquivalente Implementierung besitzt folgende Struktur: Im Speicher wird ein Puffer P der Größe $2B$ gehalten – B sei die Blockgröße beim Zugriff auf externen Speicher. Der Puffer ist in Form eines (internen) Stacks organisiert und enthält die neuesten gespeicherten Elemente. Folgende Operationen sind für die externe Datenstruktur definiert:

`pop` Falls P nicht leer, entferne das neueste Element aus P. Ansonsten, lese einen Block ein, um die Hälfte von P zu füllen bevor `pop` auf P ausgeführt wird.

`push` Falls P nicht voll, füge das neue Element direkt zu P. Ansonsten, schreibe die ältere Hälfte von P in den externen Speicher und verschiebe die aktuellere Hälfte an diese Stelle im Speicher. Anschließend führe ein `push` auf P aus.

Für die Analyse können Sie davon ausgehen, dass ein Block B Elemente des Stacks halten kann.

a) Zeigen Sie, dass die Operationen `push` und `pop` amortisiert $O(1/B)$ I/O Operationen benötigen.

b) Warum genügt es nicht, nur einen Puffer mit Größe B zu verwenden?

Musterlösung:

a) Betrachte die minimale Anzahl an Operationen (`push` oder `pop`) bis zur nächsten I/O Operation: Nach einer I/O Operation ist die Hälfte des Puffers leer – bei einem `push` auf den vollen Puffer wurde die Hälfte in den externen Speicher verlagert bzw. bei einem `pop` auf einen leeren Puffer wurde der halbe Puffer aufgefüllt. Von diesem Zustand ausgehend werden mindestens B Operationen einer Art ausgeführt, bevor erneut ein I/O Zugriff erfolgt – entweder, um bei einem `push` Daten in den externen Speicher zu schreiben, da der Puffer voll ist, oder um bei einem `pop` Daten aus dem externen Speicher zu laden, weil der Puffer leer ist.

b) Bei nur einem Puffer der Größe B könnte man sich folgende maximal schlechte Folge an Operationen überlegen: $B + 1$ `push` Operationen, gefolgt von einer Reihe von je 2 `pop` und 2 `push` Operationen. Jeweils die zweite dieser Anweisungen löst eine I/O Operation aus, da das `pop` auf einem leeren und das `push` auf einem vollen Puffer stattfindet. Amortisiert ergeben sich $O(1)$ I/O Operationen.

Aufgabe 6 (*Entwurf+Analyse: Telekommunikationsgesellschaft*)

Eine Telekommunikationsgesellschaft beauftragt Sie eine Anwendung zu schreiben, die monatlich die k Kunden bestimmt, bei denen sich die Rechnung im Vergleich zum Vormonat am meisten verändert hat. Diese Kunden will sich die Telekommunikationsgesellschaft noch einmal genau anschauen, um ihnen eventuell einen neuen Vertrag anzubieten.
Die zu bearbeitenden Daten werden Ihnen auf (langsamen) Bandspeichern zur Verfügung gestellt. Sie erhalten eine Liste mit den aneinandergefügten Datensätzen jeder Zweigstelle ihres Auftraggebers für den aktuellen Monat. Außerdem haben Sie eine entsprechende Liste für den Vormonat zur Verfügung. Gespeichert sind jeweils Tupel $(Kundennummer, Kosten)$.

a) Geben Sie einen Algorithmus an, der die geforderte Aufgabe erfüllt. Geben Sie außerdem die Laufzeit Ihres Algorithmus an und begründen Sie diese. Sie können davon ausgehen, dass die k zu bestimmenden Kunden in den Hauptspeicher passen.

b) Seien nun die k zu bestimmenden Kunden zu groß, um im Hauptspeicher gehalten zu werden. Ändern Sie Ihren Algorithmus so ab, dass er mit der erhöhten Datenmenge zurecht kommt. Geben Sie die Laufzeit Ihres neuen Algorithmus an und begründen Sie diese.

Hinweis: Diese Aufgabe war ursprünglich für die Klausur vorgesehen.

Musterlösung:

a) In einem ersten Schritt werden beide Listen nach aufsteigender Kundennummer sortiert mit externem MergeSort. Anschließend scannt der Algorithmus linear über beide sortierte Listen M, N. Zu diesem Zweck wird für jede Liste ein Zeiger i_M bzw. i_N mit der aktuellen Position gespeichert und ein Teil der Liste mit Größe B im Speicher gehalten, der bei Bedarf durch den folgenden ersetzt wird. Beide Zeiger starten am Anfang der jeweiligen Liste. Stimmen die Kundennummern von $M[i_M]$ und $N[i_N]$ überein, wird die Differenz ihrer Kosten gebildet. Diese wird in einer lokalen Prioritätswarteschlange PQ gespeichert. Hat PQ nach dem Einfügen mehr als k Elemente, so wird das kleinste entfernt. Stimmen die Kundennummern M_{i_M} und N_{i_N} nicht überein, wird der Zeiger um eins erhöht, der auf den Eintrag mit der kleineren Kundennummer zeigt. Nachdem die kürzere von beiden Listen vollständig abgearbeitet wurde, enthält PQ die gesuchten Elemente.

   Die Laufzeit wird von den benötigten I/O Operationen dominiert. Sei $n := |N| + |M|$ und H die Größe des Hauptspeichers. Sortieren benötigt $\Theta(\frac{n}{B} log_{\frac{H}{B}} \frac{n}{H})$, der lineare Scan über beide Listen $O(n/B)$ I/O Operationen. Der gesamte Algorithmus ist also vom Sortieren dominiert.

   Für die Blockgröße B beim linearen Scan gilt: $B < (S - \text{maximaler Speicher für PQ})/2$, wobei S den verfügbaren Hauptspeicher angibt. Die Blockgröße beim MergeSort kann größer gewählt werden, da die PQ zu diesem Zeitpunkt noch leer ist. Dies ändert die Anzahl I/O Operationen aber nur um einen konstanten Faktor.

b) Verwende eine externe Prioritätswarteschlange statt einer internen. Diese benötigt bis zu $O(\frac{n}{B} \log_{\frac{H}{B}} \frac{n}{H})$ I/O Operationen, falls jeder Wert eingefügt werden muss (Werte löschen ist amortisiert kostenlos). Dies entspricht der Anzahl, die für das initiale Sortieren benötigt wird. Die Laufzeit ändert sich also nicht.

   Für die Blockgrößen gilt die gleiche Argumentation wie in der ersten Teilaufgabe.