Diskrete und Geometrische Algorithmen

3. Übung am 9.11.2020

Richard Weiss Florian Schager Christian Sallinger Fabian Zehetgruber
Paul Winkler Christian Göth

Aufgabe 13. Gegeben sei ein zusammenhängender ungerichteter Graph $G = (V, E)$ mit einer geraden Anzahl an Knoten. Zeigen Sie, dass es einen (nicht notwendigerweise zusammenhängenden) Untergraph mit Knotenmenge V gibt (also einen Graph $G' = (V, E')$ mit $E' \subseteq E$), in dem alle Knotengrade ungerade sind. (Hinweis: beweisen Sie die Behauptung für Bäume und begründen Sie, warum diese Annahme reicht.)

Lösung. Der Grad eines Knoten $v \in V$ ist definert als

$$\text{grad}(x) = |\{\{x, y\} : \{x, y\} \in E\}|.$$

Wir beweisen die Aussage mit Induktion nach $n := |E| = |V - 1|$.

IA($n = 1$):
Es gibt genau eine Kante, welche die beiden Knoten verbindet. Also gilt die Aussage bereits für $E' = E$.

IS($n \mapsto n + 2$):
Betrachte einen beliebigen Baum $G = (V, E)$ mit einer geraden Anzahl an Knoten. O.B.d.A. $\exists\, v \in V$: $\text{grad}(v) = 2k$, $k \in \mathbb{N}$, sonst wäre die Aussage schon erfüllt. Wir betrachten Menge aller Kanten aus E, die an v angrenzen.

$$\{x_1, \ldots, x_{2k}\} = \{x \in E : \{x, v\} \in V\}$$

Löscht man diese Kanten und den Knoten v aus G, so zerfällt der neue Graph in Zusammenhangskomponenten $G'_1, \ldots, G'_{2k}$. Für $i = 1, \ldots, 2k$, ergänzen wir G'_i um den Knoten v und Kante $\{x_i, v\}$ zum „Ast“ G_i.

G besteht aus einer geraden Anzahl von Knoten und somit aus einer ungeraden Anzahl an Kanten. Also gibt es also mindestens einen (von v ausgehenden) Ast G_i, der eine gerade Kanten-Zahl hat. (Wenn alle Äste (gerade viele) ungerade Kanten-Zahl hätten, wäre die Gesamt-Kanten-Zahl gerade mal ungerade, also gerade!) „$G \setminus G_i$“ hat dann eine ungerade Kanten-Zahl, weil „$G = G_i + G \setminus G_i$“ ja eine ungerade Kanten-Zahl hat.

Wir löschen von G nun die Kante $\{x_i, v\}$ (des Asts G_i (mit gerader Kanten-Zahl)). Dann erhalten wir zwei Zusammenhangskomponenten mit ungerader Kantenanzahl $\leq n$. Die sind wieder Bäume. Somit können wir die Induktionsvoraussetzung auf Beide anwenden. Wir vereinigen die resultierenden Graphen und erhalten $G' = (V, E')$.

Sei $G = (V, E)$ ein beliebiger zusammenhängender Graph mit gerader Knotenanzahl. Ein Baum ist genau ein minimal zusammenhängender Graph. Wir finden einen solchen Teilgraphen („Spannbaum“) $G_0 = (V, E_0)$ mit $E_0 \subseteq E$. Auf den wenden wir das oben gezeigte an.

□

Aufgabe 14. Sei $A[1,\ldots,n]$ ein Feld mit n verschiedenen Zahlen. Das Paar (i,j) wird Inversion genannt, wenn $i<j$ und $A[i]>A[j]$ gilt.

(a) Welches Feld mit Elementen der Menge $\{1,\ldots,n\}$ besitzt die meisten Inversionen und wie viele Inversionen sind in diesem Feld enthalten?

(b) Welche Beziehung gibt es zwischen der Anzahl von Inversionen im Eingabefeld und der Laufzeit von Insertion-Sort (Einfügesortieren)?

(c) Geben Sie einen Algorithmus an, der die Anzahl von Inversionen in einer Permutation von n Elementen bestimmt und dessen Laufzeit im schlechtesten Fall $\Theta(n\log n)$ ist. (Hinweis: Modifizieren Sie Merge-Sort (Sortieren durch Verschmelzen) in passender Weise)

Lösung. Wenn wir die Werte des Datenbereichs auf $\{1,\ldots,n\}$ einschränken, bezeichnet die Anzahl der Inversionen genau die Anzahl der Fehlstände, der durch $A[1,\ldots,n]$ induzierten Permutation.

(a) Betrachte das Feld $A[1,\ldots,n]=[n,\ldots,1]$. Klarerweise gilt $\forall\, i,j=1,\ldots,n : i<j : A[i]>A[j]$.

$$\begin{aligned}\implies |\{(i,j):1\le i<j\le n, A[i]>A[j]\}| &= |\{(i,j):1\le i<j\le n\}|\\ &= \left|\sum_{j=1}^{n}\{(i,j):1\le i<j\}\right| = \sum_{j=2}^{n}(j-1) = \sum_{j=1}^{n-1} j = \frac{(n-1)n}{2} = \binom{n}{2}\end{aligned}$$

(b) Für jedes $j=2,\ldots,n$ entspricht die Anzahl der inneren Schleifendurchläufe genau der Anzahl aller $i<j$ mit $A[i]>A[j]$. Insgesamt ist die Anzahl der inneren Schleifendurchläufe also genau die Anzahl der Inversionen des Datenfelds.

(c) Wir Modifizieren den Algorithmus „Merge-Sort“ in passender Weise. Man erhält, zusätzlich zum sortierten Datenfeld, die Anzahl der Fehlstände, mit Laufzeit $\Theta(n\log n)$ im Worst-Case.

```
1 : Prozedur Inversions-Verschmelzen(A, B)
2 :   Sei C ein neues Datenfeld der Länge A.Länge + B.Länge
3 :   i := 1
4 :   j := 1
5 :   n := 0
5 :   Für k := 1, ..., A.Länge + B.Länge
6 :     Falls j > B.Länge oder (i ≤ A.Länge und A[i] ≤ B[j]) dann
7 :       C[k] := A[i]
8 :         i := i + 1
9 :     Sonst
10 :      C[k] := B[j]
11 :      j := j + 1
12 :      Falls i ≤ A.Länge
  :      ⟹ B[j] < A[i] ≤ A[i + 1] ≤ ··· ≤ A[A.Länge]
13 :        n := n + (A.Länge − i + 1)
14 :      Ende Falls
15 :    Ende Falls
16 :  Ende Für
17 :  Antworte C, n
18 : Ende Prozedur
```

Streng genommen, ist die Zeile 12 redundant. Die Prozedur wird dadurch nur besser lesbar.

```
1 : Prozedur VInversionszähler(A)
2 :   Falls A.Länge = 1
3 :     Antworte A, 0
4 :   Sonst
5 :     m := ⌈A.Länge / 2⌉
6 :     L := A[1, ..., m]
7 :     R := A[m + 1, ..., A.Länge]
8 :     L', n_L := VInversionszähler(L)
9 :     R', n_R := VInversionszähler(R)
10 :    A', n_A := Inversions-Verschmelzen(L', R')
11 :    n := n_L + n_R + n_A
12 :    Antworte A', n
13 :  Ende Falls
14 : Ende Prozedur
```

□

Aufgabe 15. Natural Merge-Sort ist eine Variante von Merge-Sort, die bereits vorsortierte Teilfolgen (sogenannte runs) ausnutzt. Ein run ist eine Teilfolge aufeinanderfolgender Glieder $x_i, x_{i+1}, \ldots, x_{i+k}$ mit $x_i \leq x_{i+1} \leq \cdots \leq x_{i+k}$. Die Basis für den Verschmelzen-Vorgang bilden hier nicht die rekursiv oder iterativ gewonnenen Zweiergruppen, sondern die runs. Im ersten Durchlauf des Algorithmus bestimmt man die runs, anschließend fügt man die runs mittels VERSCHMELZEN zusammen.

(a) Sortieren Sie folgende Liste mittels Natural Merge-Sort:

$$2, 4, 3, 1, 7, 6, 8, 9, 0, 5 \tag{1}$$

(b) Schreiben Sie einen Pseudocode für Natural Merge-Sort.

(c) Machen Sie eine Best- sowie eine Worst-Case-Analyse für das Laufzeitverhalten von Natural Merge-Sort bei einem Eingabefeld der Größe n.

Lösung.

(a) Wir haben dafür ein Python-Programm geschrieben. Der Output lautet wie folgt.

```
# ------------------------------------------------------------ #

[[2, 4], [3], [1, 7], [6, 8, 9], [0, 5]]

[2, 4]
[2, 3, 4]
[1, 2, 3, 4, 7]
[1, 2, 3, 4, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# ------------------------------------------------------------ #
```

Jetzt machen wir's manuell. Zuerst bestimmen wir die Runs.

$$[2, 4], [3], [1, 7], [6, 8, 9], [0, 5]$$

Nun verschmelzen wir.

$$\begin{aligned}[2, 4], [3], [1, 7], [6, 8, 9], [0, 5] &\rightsquigarrow [2, 3, 4], [1, 7], [6, 8, 9], [0, 5] \\ &\rightsquigarrow [1, 2, 3, 4, 7], [6, 8, 9], [0, 5] \\ &\rightsquigarrow [1, 2, 3, 4, 6, 7, 8, 9], [0, 5] \\ &\rightsquigarrow [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]\end{aligned}$$

(b)

```
1 : Prozedur Runs Verschmelzen(R)
2 :   m := R.Länge
3 :   C := R[1]
4 :   Für i = 2, ..., m
5 :     C := Verschmelzen(C, R[i])
6 :   Ende Für
7 :   Antworte C
8 : Ende Prozedur
```

```
1 : Prozedur Natural Merge Sort(A)
2 :   Sei R ein neuer Datenfeld der Länge 0.
3 :   n := A.Länge
4 :   i := 1
5 :   Solange i ≤ n
6 :     j := 0
7 :     r := [A[i]]
8 :     Solange i + j + 1 ≤ n und A[i + j] ≤ A[i + j + 1]
9 :       r.append(A[i + j + 1])
10 :      j := j + 1
11 :    Ende Solange
12 :    R.append(r)
13 :    i := i + j + 1
14 :  Ende Solange
15 :  Antworte Runs Verschmelzen(R)
16 : Ende Prozedur
```

Man könnte den Algorithmus evtl. noch optimieren, indem man die Runs aufsteigend nach ihrer Länge sortiert. Damit spart man sich beim Verschmelzen etwas Aufwand.

(c) Sei A ein Datenfeld der Länge n. Das Erstellen der Runs gelingt immer in linearem Aufwand.

- Best-Case:
  Sei das Datenfeld bereits aufsteigend sortiert. Der Aufwand liegt also nur in der Erstellung der Runs. Somit erhalten wir linearen Aufwand.
- Worst-Case:
  Sei das Datenfeld absteigend sortiert. Wir erhalten also n Runs der Länge 1. Abgesehen vom Erstellen der Runs, müssen wir die Prozedur Verschmelzen für $i = 2, \ldots, n$, also $(n-1)$-mal, auf die Datenfelder C und $R[i]$ der Länge $i-1$ bzw. 1 anwenden. Verschmelzen hat linearen Aufwand. Wir kommen also insgesamt auf quadratischen Aufwand.

□

Aufgabe 16. Die Fibonacci-Zahlen seien durch die Rekursion $F_n = F_{n-1} + F_{n-2}$ für $n \geq 2$ mit Anfangswerten $F_0 = 0$ und $F_1 = 1$ definiert. Die Fibonacci-Zahlen können effizient mittels folgender auf Matrizenmultiplikation beruhender Formel berechnet werden:

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \quad \text{für } n \geq 1$$

a. Beweisen Sie diese Formel durch vollständige Induktion.

b. Überlegen Sie sich einen Algorithmus, der $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$ in nur logarithmisch vielen Schritten berechnet.

Lösung. Wir nennen die linke Matrix L_n und die rechte R_n.

a. IA($n = 1$):

$$\implies F_2 = F_1 + F_0 = 1 + 0 \implies L_1 = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 = R_1$$

IS($n \mapsto n + 1$):

$$\begin{aligned} \implies R_{n+1} = R_n \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = L_n \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} &= \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} F_{n+1} + F_n & F_{n+1} \\ F_n + F_{n-1} & F_n \end{pmatrix} = \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix} = L_{n+1} \end{aligned}$$

b. Sei $k \in \mathbb{N}$.

$$\implies P_k := \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{2^k} = \underbrace{\left(\left(\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \cdots \right) \right)^2}_{k\text{-mal}} = P_{k-1}^2 \implies P_0 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

Stelle $n \in \mathbb{N}_0$ (eindeutig) binär dar, d.h.

$$n = \sum_{k=0}^{m} a_k 2^k, \quad m = \lfloor \log(n) \rfloor, \quad a_0, \ldots, a_m \in \{0, 1\}.$$

$$\implies F_n := \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \prod_{k=0}^{m} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{a_k 2^k} = \prod_{\substack{k=0 \\ a_k = 1}}^{m} P_k$$

```
1 : Prozedur Fibonacci Matrix(n)
2 :   a := Binärkoeffizienten(n)
3 :   m := a.Länge
4 :   F := ( 1  0 )
          ( 0  1 )
5 :   P := ( 1  1 )
          ( 1  0 )
6 :   Für k = 0, ..., m
7 :     Falls a_k = 1
8 :       F := F · P
9 :     Ende Falls
11 :    Falls k < m
12 :      P := P^2
13 :    Ende Falls
14 :  Ende Für
15 :  Antworte F
16 : Ende Prozedur
```

Um die Binätdarstellung zu umgehen, wird man vielleicht dazu verleitet, folgende Prozedur zu verwenden.

```
1 : Prozedur Matrix Potenzieren(M, n)
2 :   Falls n = 1
3 :     Antworte M
4 :   Sonst
5 :     Antworte Matrix Potenzieren(M, ⌊n/2⌋) · Matrix Potenzieren(M, ⌈n/2⌉)
6 :   Ende Falls
7 : Ende Prozedur
```

Wenn man aber diese Prozedur mit M^{2^k} testet, merkt man, dass der Aufwand dennoch linear ist. Man gewinnt durch das „teilen und herrschen“ hier leider Nichts. (Dazu kommt, dass rekursive Funktionen den stack schnell sehr zumüllen.)

□

Aufgabe 17.

(a) Wie schnell könnte man eine $(kn \times n)$-Matrix A mit einer $(n \times kn)$-Matrix B multiplizieren, d.h. $C = A \cdot B$ berechenen, wenn man Strassens Algorithmus als Unterprogramm verwendet?

(b) Benantworten Sie die gleiche Frage, wenn die Reihenfolge der Eingabematrizen vertauscht ist, man also $\tilde{C} = B \cdot A$ bestimmen möchte.

Lösung.

(a) Man kann die Matrizen als Blockmatrizen auffassen.

$$A = \begin{pmatrix} A_1 \\ \vdots \\ A_k \end{pmatrix}, \quad B = (B_1 \cdots B_k), \quad A_1, \ldots, A_k, B_1, \ldots, B_k \ (n \times n)\text{-Matrizen}$$

Man muss somit, zur Berechnung der $(kn \times kn)$-Matrix C, für k^2 Blöcke der Größe $n \times n$ durch je eine $(n \times n)$-Matrix-Multiplikation durchführen. Dies kann jeweils mit Strassens Algorithmus gemacht werden.

$$C = AB = \begin{pmatrix} A_1 \\ \vdots \\ A_k \end{pmatrix} (B_1 \cdots B_k) = \begin{pmatrix} A_1B_1 & \cdots & A_1B_k \\ \vdots & \ddots & \vdots \\ A_kB_1 & \cdots & A_kB_k \end{pmatrix}$$

Sei n eine Zweierpotenz. Strassens Algorithmus hat den Aufwand $S(n) = \Theta(n^{\log 7})$. Es ergibt sich also eine Laufzeit von $T(n, k) = k^2 S(n) = \Theta(k^2 n^{\log 7})$.

(b) Analog zu oben betrachten wir wieder die $(n \times n)$-Matrix-Multiplikation von $(n \times n)$-Blockmatrizen. Hier benötigen wir k viele $(n \times n)$-Matrix-Multiplikationen und $k - 1$ viele $(n \times n)$-Matrix-Additionen.

$$\tilde{C} = BA = (B_1 \cdots B_k) \begin{pmatrix} A_1 \\ \vdots \\ A_k \end{pmatrix} = \sum_{i=1}^{k} B_i A_i$$

Es ergibt sich also eine Laufzeit von $T(n, k) = kS(n) = \Theta(kn^{\log 7})$.

□

Aufgabe 18. Zeigen Sie, wie man komplexe Zahlen $a + bi$ und $c + di$ mit nur drei Multiplikationen reeller Zahlen multiplizieren kann. Der Algorithmus sollte a, b, c und d als Eingabe bekommen und den Realteil $ac - bd$ sowie den Imaginärteil $ad + bc$ getrennt ausgeben.

Lösung.

1 : **Prozedur** KOMPLEXE MULTIPLIKATION(a, b, c, d)
2 : $P_1 := (a - b)d$
3 : $P_2 := (d - c)a$
4 : $P_3 := (c + d)b$
5 : **Antworte** $P_1 - P_2, P_1 + P_3$
6 : **Ende Prozedur**

Der Algorithmus leistet das Gewünschte, da

$$P_1 - P_2 = (ad - bd) - (ad - ac) = ac - bd, \quad P_1 + P_3 = (ad - bd) + (bc + bd) = ad + bc.$$

□