



---

Wintersemester 2017/2018

---

## Lineare und kombinatorische Optimierung

### Übungsblatt 3

#### Gruppenübungen

##### Aufgabe 1. (Planare Graphen)

(0 Punkte)

$G = (V, E)$  sei einfach, zusammenhängend und planar. Beweisen Sie folgende Behauptungen:

- (a) Falls  $|V| = n \geq 3$ , dann hat  $G$  höchstens  $3n - 6$  Kanten und  $2n - 4$  Flächen.
- (b)  $G$  hat einen Knoten, dessen Grad kleiner als sechs ist.

Hinweis: Verwenden Sie zum Beweis der Behauptungen die Formel von Euler  $n - e + f = 2$ , wobei  $n$  die Anzahl der Knoten,  $e$  die Anzahl der Kanten und  $f$  die Anzahl der Flächen darstellt.

##### Lösung:

- (a) Jede Fläche wird durch mindestens drei Kanten erzeugt und jede Kante grenzt an höchstens zwei Flächen. Also ist die Anzahl der Kanten

$$e \geq 3 \cdot \frac{f}{2}.$$

Aus Eulers Formel folgt dann

$$e + 2 = n + f \leq n + \frac{2}{3}e \quad \text{bzw.} \quad e \leq 3n - 6.$$

Ferner folgt aus Eulers Formel

$$n + f = e + 2 \geq 3 \cdot \frac{f}{2} + 2 \quad \text{d.h.} \quad f \leq 2n - 4.$$

Anders gesagt: Die Anzahl der Kanten und Flächen in  $G$  wächst höchstens linear mit der Anzahl der Knoten.

- (b) Wenn  $G$  nur einen Knoten enthält, ist sein Grad Null. Wenn  $G$  aus zwei Knoten besteht, haben beide einen Grad von eins. Betrachte im Folgenden Graphen mit drei oder mehr Knoten. Angenommen, der Grad eines jeden Knotens ist mindestens sechs, dann gilt

$$\sum_{v \in V} \deg(v) \geq 6n.$$

Es gilt jedoch auch

$$\sum_{v \in V} \deg(v) = 2e,$$

da jede Kanten mit zwei Knoten verbunden ist. Folglich gilt  $2e \geq 6n$  bzw.  $e \geq 3n$ . Dies ist jedoch ein Widerspruch zu Teilaufgabe (a).

### Aufgabe 2. (Durchlaufen einer Hinderniswelt)

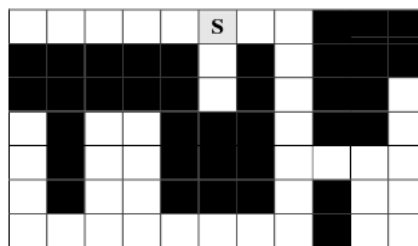
(0 Punkte)

Eine Hinderniswelt lässt sich durch ein Gitter beschreiben, wobei betretbare Zellen weiß und nicht betretbare Zellen schwarz dargestellt werden. Zwei betretbare Zellen sind benachbart, falls sie sich entweder direkt übereinander oder nebeneinander befinden. Damit lässt sich eine Hinderniswelt auch als Graph auffassen, wobei die Knoten gerade die betretbaren Zellen sind.

In der folgenden Abbildung ist eine kleine Hinderniswelt mit dem entsprechenden Graphen zu sehen:



Folgende Hinderniswelt wird bei  $S$  beginnend mit Breitensuche bzw. Tiefensuche durchlaufen.



Tragen Sie in den Zellen die Besuchszeitpunkte an. Gehen Sie dabei davon aus, dass für eine Zelle die benachbarten Zellen immer in der Reihenfolge N, O, S, W (oberhalb, rechts, unterhalb, links) betrachtet werden. Die einzelnen Schritte des Algorithmus müssen nicht mit angegeben werden.

### Lösung:

Breitensuche:

12	10	8	6	3	S	1	4		
					2		7		
					5		9		25
40		37	34				11		21
39		35	31				13	14	16
38		32	29				15		19
36	33	30	28	27	24	20	17		23

Tiefensuche:

40	39	38	37	36	S	1	2		
					34		3		
					35		4		11
33		25	24				5		10
32		26	23				6	7	8
31		27	22				16		15
30	29	28	21	20	19	18	17		14

### Aufgabe 3. (Verschiedene Sortieralgorithmen)

(0 Punkte)

Sortieren Sie die Folge “ERLANGEN” lexikographisch mit

1. Selectionsort,
2. Bubblesort und
3. Insertionsort (siehe unten).

Geben Sie dabei alle Zwischenschritte an und vergleichen Sie den Sortieraufwand. Versuchen Sie zu verstehen, wie der Algorithmus Insertionsort funktioniert.

---

**Algorithmus 1** Insertionsort

---

**Eingabe:** Ein Array  $a$  der Länge  $n$  mit  $a[i] \in \mathbb{Z}$

**Ausgabe:** Das sortierte Array  $a$  mit  $a[1] \leq a[2] \leq \dots \leq a[n]$

```
1: for  $i = 2, \dots, n$  do
2:    $v \leftarrow a[i]$ 
3:    $j \leftarrow i$ 
4:   while  $j > 1$  und  $a[j-1] > v$  do
5:      $a[j] \leftarrow a[j-1]$ 
6:      $j \leftarrow j-1$ 
7:    $a[j] \leftarrow v$ 
```

---

**Lösung:**

1. Selectionsort (Sortieren durch Auswahl) [Laufzeit:  $\mathcal{O}(n^2)$ ]:

ERLANGEN  
**AR**LENGEN  
**AE**LRNGEN  
**AEER**NGLN  
**AEEG**NRLN  
**AEEG**LRNN  
**AEEG**LNRR  
**AEEG**LNRR

2. Bubblesort [Laufzeit:  $\mathcal{O}(n^2)$ ]:

ERLANGEN  
**EL**RLANGEN  
**ELAR**NGEN  
**ELANR**GEN  
**ELANGR**EN  
**ELANGERN**  
**ELANGENR**  
**EAL**NGENR  
**EALGN**ENR  
**EALGENR**R  
**AEL**GENNR  
**AEGL**ENNR  
**AEGEL**NNR  
**AEGL**NNR

3. Insertionsort [Laufzeit:  $\mathcal{O}(n^2)$ ]: Dieser Algorithmus verwendet eine Methode, die viele Menschen beim Kartenspiel nutzen, um ihre Handkarten zu sortieren:

- Betrachte die Elemente eines nach dem Anderen.
- Füge jedes Element an seinem Platz zwischen den bereits betrachteten ein. Diese bleiben sortiert.
- Das gerade betrachtete Element wird eingefügt, indem alle größeren Elemente einen Platz nach rechts aufrücken.

- Das betrachtete Element wird dann auf dem freien Platz eingefügt.

Das jeweils ausgewählte Element ist im Folgenden hervorgehoben.

ERLANGEN  
**ER**LANGEN  
ER**L**ANGEN  
ELR**A**NGEN  
AELR**N**GEN  
AELNR**G**EN  
AEGLNR**E**N  
AEEGLNR**R**N  
AEEGLNN**R**

## Hausübungen

### Aufgabe 4. (Planare Graphen)

(1+2 Punkte)

$G = (V, E)$  sei einfach, zusammenhängend und planar. Beweisen Sie folgende Behauptungen:

- Für  $|V| = n \geq 3$  ist der durchschnittliche Knotengrad von  $G$  kleiner als sechs.
- Wenn  $G$  keine Kreise der Länge drei enthält und  $n = |V|$ , dann hat  $G$  höchstens  $2n - 4$  Kanten und  $n - 2$  Flächen.

Hinweis: Verwenden Sie für die Beweise die Formel von Euler und Einsichten aus Aufgabe 1.

### Lösung:

- Aus Aufgabe 1 folgt mit  $n \geq 3$

$$\frac{\sum_{v \in V} \deg(v)}{|V|} = \frac{2e}{n} \leq \frac{6n - 12}{n} < 6.$$

- Jede Fläche wird durch mindestens vier Kanten erzeugt. Jede Kante grenzt an höchstens zwei Flächen. Also ist die Anzahl Kanten

$$e \geq 4 \cdot \frac{f}{2} = 2f.$$

Aus Eulers Formel folgt dann

$$e + 2 = n + f \leq n + \frac{1}{2}e \quad \text{bzw.} \quad e \leq 2n - 4.$$

Ferner folgt aus Eulers Formel

$$n + f = e + 2 \geq 2 \cdot f + 2, \quad \text{bzw.} \quad f \leq n - 2.$$

### Aufgabe 5. (Breitensuche und bipartite Graphen)

(4 Punkte)

Modifizieren Sie den Algorithmus zur Breitensuche so, dass damit entschieden werden kann, ob ein Graph bipartit ist oder nicht. Schreiben Sie ihren Algorithmus in Pseudocode auf.

### Lösung:

Gegeben sei ein Graph  $G = (V, E)$ . Wir wollen testen, ob  $G$  bipartit ist, d.h. ob  $V = V_1 \cup V_2$  mit  $V_1 \cap V_2 = \emptyset$  und  $E(V_1) = E(V_2) = \emptyset$ . Wir starten die Breitensuche mit einem beliebigen Knoten und durchsuchen den Graphen. Hierbei wird jedem Knoten ein Wert 1 oder 2 zugeordnet (1 = Knoten befindet sich in  $V_1$ , 2 = Knoten befindet sich in  $V_2$ ). Jedem Knoten wird ein anderer Wert als sein Vorgängerknoten zugeordnet. Trifft man auf einen markierten Knoten und muss diesem eine andere als die schon zugewiesene Markierung zuordnen, ist der Graph nicht bipartit. Bricht die Breitensuche ab, bevor alle Knoten besucht wurden, müssen evtl. weitere Zusammenhangskomponenten mit demselben

Verfahren durchsucht werden.

Als Pseudocode:

1. Wähle beliebigen Startknoten  $s \in V$ .
2.  $R = \{s\}$ ,  $Q = (s)$
3.  $mark(s) = 1$ ,  $mark(v) = 0$  für  $v \in V \setminus \{s\}$
4. Solange  $Q \neq \emptyset$ :
  - (A) Wähle erstes  $v \in Q$
  - (B) Für alle Kanten  $e = \{v, w\} \in E$ :
    - (i) Falls  $mark(w) = 0$  und  $mark(v) = 1$ : setze  $mark(w) = 2$ ,  $Q = Q \cup (w)$ ,  $R = R \cup \{s\}$
    - (ii) Falls  $mark(w) = 0$  und  $mark(v) = 2$ : setze  $mark(w) = 1$ ,  $Q = Q \cup (w)$ ,  $R = R \cup \{s\}$
    - (iii) Falls  $mark(w) \neq 0$  und  $mark(v) = mark(w)$ : STOP,  $G$  ist nicht bipartit
  - (C)  $Q = Q \setminus (v)$
5. STOP:  $G$  (bzw. Zusammenhangskomponente von  $s$ ) ist bipartit

Ist  $|R| \neq |V|$ , muss das Verfahren noch mal für einen Startknoten  $s \notin R$  durchgeführt werden.

**Aufgabe 6.** (Quicksort)

(6+2 Punkte)

1. Beweisen Sie folgenden Satz: Die durchschnittliche Laufzeit des Quicksort beträgt  $\mathcal{O}(n \log n)$ .
2. Beschreiben Sie allgemein die Gestalt von zu sortierenden Schlüsselfolgen, so dass Quicksort den maximalen Aufwand  $\mathcal{O}(n^2)$  bzw. den minimalen Aufwand  $\mathcal{O}(n \log n)$  benötigt und konstruieren sie aus den ersten zehn natürlichen Zahlen jeweils eine Beispielfolge.

**Lösung:**

1. Sei  $\alpha_n$  die mittlere Laufzeit für das Sortieren eines Arrays der Länge  $n$ . Wir gehen davon aus, dass die Stelle  $k$ , an denen das Array in zwei Hälften geteilt wird, gleichverteilt ist. Dann gilt (mit einer Konstanten  $\gamma \in \mathbb{N}$ ):

$$\begin{aligned}\alpha_n &= \frac{1}{n} \sum_{k=1}^n (\alpha_{k-1} + \alpha_{n-k}) + \gamma \cdot n \\ &= \gamma n + \frac{2}{n} \sum_{k=0}^{n-1} \alpha_k \\ &= \gamma n + \frac{2}{n} \alpha_{n-1} + \frac{n-1}{n} (\alpha_{n-1} - \gamma(n-1)) \\ &= \frac{n+1}{n} \alpha_{n-1} + \gamma n - \gamma \frac{(n-1)^2}{n} \\ &= \frac{n+1}{n} \alpha_{n-1} + 2\gamma - \frac{\gamma}{n} \\ &\leq \frac{n+1}{n} \alpha_{n-1} + 2\gamma.\end{aligned}$$

Nun lässt sich per Induktion zeigen, dass für eine Folge mit  $a_1 = b_1$  und  $a_n = b_n + c_n a_{n-1}$  gilt

$a_n = \sum_{i=1}^n (\prod_{j=i+1}^n c_j) b_i$ . Angewandt auf  $\alpha_n$  erhalten wir:

$$\begin{aligned}\alpha_n &= \sum_{i=1}^n \left( \prod_{j=i+1}^n \frac{j+1}{j} \right) 2\gamma \\ &= 2\gamma \sum_{i=1}^n \frac{n+1}{i+1} \\ &= 2\gamma(n+1) \sum_{i=2}^{n+1} \frac{1}{i} \\ &\leq 2\gamma(n+1) \int_1^{n+1} \frac{1}{x} dx \\ &= 2\gamma(n+1) \ln(n+1) \in \mathcal{O}(n \log n).\end{aligned}$$

2. Damit Quicksort den maximalen Aufwand von  $\mathcal{O}(n^2)$  hat, müssen die zu sortierenden Listen unbalanciert sein, d.h. in jedem Aufruf von Quicksort ist das Pivotelement entweder das kleinste oder das größte Element des (Teil-)Arrays. Dann wird in jedem Durchgang die maximale Anzahl an Elementen verglichen. Dies ist der Fall bei geeignet permutierten Eingabedaten passend zur Wahl des Pivotelements.

Beispielfolgen für den worst case wären (2,3,4,5,6,7,8,9,10,1) oder (10,9,8,7,6,5,4,3,2,1) bei Wahl des letzten Elements als Pivot. Dabei ist eine der beiden zu untersuchenden Listen der rekursiven Aufrufe immer leer, während die andere genau ein Element weniger enthält als im vorherigen Rekursionsaufruf. Eine gutartige Folge wäre (1,4,3,2,8,6,7,10,9,5).

#### Aufgabe 7. (Heapsort)

(4 Punkte)

Sortieren Sie die Zahlenfolge (3, 1, 4, 1, 5, 9, 2) aufsteigend mit Heapsort.

#### Lösung:

Mit Heapsort funktioniert die Sortierung folgendermaßen:

Zunächst führen wir den ersten Teil des Algorithmus durch:

- (1) For  $i = \lfloor \frac{n}{2} \rfloor$  To 1 Do
- (2)     HEAPIFY( $a, i, n$ ).
- (3) End For

$n = 7$ , demnach ist  $i = 3$  im ersten Schritt:

$i = 3$  HEAPIFY( $a, 3, 7$ ):  $f = 3, s = 6, r = 7$ .  $a[f] = a[3] = 4 \leq a[s] = a[6] = 9 \Rightarrow$  tausche  $a[f]$  und  $a[s]$ ,  
 $f = s = 6, s = 2 \cdot f = 12 \Rightarrow a = (3, 1, 9, 1, 5, 4, 2)$

$i = 2$  HEAPIFY( $a, 2, 7$ ):  $f = 2, s = 4, r = 7$ .  $a[s] = a[4] = 1 \leq a[s+1] = a[5] = 5 \Rightarrow$  Setze  $s = s + 1 = 5$   
 $a[f] = a[2] = 1 \leq a[s] = a[5] = 5 \Rightarrow$  tausche  $a[f]$  und  $a[s]$ ,  $f = s = 5, s = 2 \cdot f = 10 \Rightarrow a =$   
 $(3, 5, 9, 1, 1, 4, 2)$ .

$i = 1$  HEAPIFY( $a, 1, 7$ ):  $f = 1, s = 2, r = 7$ .  $a[s] = a[2] = 5 \leq a[s+1] = a[3] = 9 \Rightarrow$  Setze  $s = s + 1 = 3$   
 $a[f] = a[1] = 3 \leq a[s] = a[3] = 9 \Rightarrow$  tausche  $a[f]$  und  $a[s]$ ,  $f = s = 3, s = 2 \cdot f = 6 \Rightarrow a =$   
 $(9, 5, 3, 1, 1, 4, 2)$ .  
 $a[f] = a[3] = 3 \leq a[s] = a[6] = 4 \Rightarrow$  tausche  $a[f]$  und  $a[s]$ ,  $f = s = 6, s = 2 \cdot f = 12 \Rightarrow a =$   
 $(9, 5, 4, 1, 1, 3, 2)$ .

Nun steht das größte Element an erster Stelle. Dieses wird nach hinten getauscht und es wird wieder das verbleibende größte Element an die erste Stelle gebracht. Der Rest des Arrays besitzt die sog. Heap-Eigenschaft. Dies wird so lange weitergeführt, bis das Array sortiert ist. Jetzt führen wir den zweiten Teil des Algorithmus durch:

- (4) For  $i = n$  To 2 Do

(5) Tausche  $a[1]$  und  $a[i]$ .

(6) HEAPIFY( $a, 1, i - 1$ ).

(7) End For

$i = 7$  Tausche  $a[1]$  und  $a[7] \Rightarrow a = (2, 5, 4, 1, 1, 3, \mathbf{9})$ .

HEAPIFY( $a, 1, 6$ ):  $f = 1, s = 2, r = 6. a[f] = a[1] = 2 \leq a[s] = a[2] = 5 \Rightarrow$  tausche  $a[f]$  und  $a[s]$ ,  
 $f = s = 2, s = 2 \cdot f = 4 \Rightarrow a = (5, 2, 4, 1, 1, 3, 9)$ .

$a[s] = a[4] = 1 \leq a[s + 1] = a[5] = 1 \Rightarrow$  Setze  $s = s + 1 = 5 \Rightarrow a = (5, 2, 4, 1, 1, 3, 9)$

$i = 6$  Tausche  $a[1]$  und  $a[6] \Rightarrow a = (3, 2, 4, 1, 1, \mathbf{5}, 9)$ .

HEAPIFY( $a, 1, 5$ ):  $f = 1, s = 2, r = 5. a[s] = a[2] = 2 \leq a[s + 1] = a[3] = 4 \Rightarrow$  Setze  $s = s + 1 = 3$   
 $a[f] = a[1] = 3 \leq a[s] = a[3] = 4 \Rightarrow$  tausche  $a[f]$  und  $a[s]$ ,  $f = s = 3, s = 2 \cdot f = 6 \Rightarrow a =$   
 $(4, 2, 3, 1, 1, 5, 9)$ .

$i = 5$  Tausche  $a[1]$  und  $a[5] \Rightarrow a = (1, 2, 3, 1, \mathbf{4}, 5, 9)$ . HEAPIFY( $a, 1, 4$ ):  $f = 1, s = 2, r = 4. a[s] = a[2] =$   
 $2 \leq a[s + 1] = a[3] = 3 \Rightarrow$  Setze  $s = s + 1 = 3$

$a[f] = a[1] = 1 \leq a[s] = a[3] = 3 \Rightarrow$  tausche  $a[f]$  und  $a[s]$ ,  $f = s = 3, s = 2 \cdot f = 6 \Rightarrow a =$   
 $(3, 2, 1, 1, 4, 5, 9)$ .

$i = 4$  Tausche  $a[1]$  und  $a[4] \Rightarrow a = (1, 2, 1, \mathbf{3}, 4, 5, 9)$ .

HEAPIFY( $a, 1, 3$ ):  $f = 1, s = 2, r = 3. a[f] = a[1] = 1 \leq a[s] = a[2] = 2 \Rightarrow$  tausche  $a[f]$  und  $a[s]$ ,  
 $f = s = 2, s = 2 \cdot f = 4 \Rightarrow a = (2, 1, 1, 3, 4, 5, 9)$ .

$i = 3$  Tausche  $a[1]$  und  $a[3] \Rightarrow a = (1, 1, \mathbf{2}, 3, 4, 5, 9)$ .

HEAPIFY( $a, 1, 2$ ):  $f = 1, s = 2, r = 2. a[f] = a[1] = 1 \leq a[s] = a[2] = 1 \Rightarrow$  tausche  $a[f]$  und  $a[s]$ ,  
 $f = s = 2, s = 2 \cdot f = 4 \Rightarrow a = (1, 1, 2, 3, 4, 5, 9)$ .

$i = 2$  Tausche  $a[1]$  und  $a[2] \Rightarrow a = (1, \mathbf{1}, 2, 3, 4, 5, 9)$ .

HEAPIFY( $a, 1, 1$ ):  $f = 1, s = 2, r = 1$ . Hier geschieht nichts, da  $s > r$ .

(8) Gib  $a$  aus.

Das sortierte Array lautet:  $a = (1, 1, 2, 3, 4, 5, 9)$

**Aufgabe 8.** (Programmieraufgabe: Tiefensuche, Quicksort)

(2+2 Punkte)

Laden Sie die Datei `ProgAufgabe03.py` aus Studon herunter und fügen Sie sie in ihr PyCharm ein.

1. Im ersten Teil dieser Programmieraufgabe werden Sie die Tiefensuche auf Graphen implementieren. Ziel ist es, eine Methode zu implementieren, die zu einem gegebenen Graphen alle Zusammenhangskomponenten bestimmt. Wir nehmen hierbei das Graphen-Modul `networkx`, welches am Kopf der Datei importiert wurde, zuhilfe.
  - (a) Zunächst ein paar `networkx`-Erklärungen: Am Fuße der Datei ist ein Testcode, der später ihre Tiefensuche testen wird. Dort sehen Sie, wie man einen Graphen erstellt (nicht notwendig für das Lösen der Aufgabe, aber hilfreich um selber Tests zu entwerfen). Zum Lösen der Aufgabe benötigen Sie die folgenden zwei `networkx`-Methoden. Sei `graph` ein `networkx`-Graph und `v` ein Knoten in `graph`.
    - `graph.nodes()` gibt ihnen eine Liste aller Knoten des Graphen.
    - `graph.neighbors(v)` gibt ihnen eine Liste aller benachbarten Knoten von `v`.
  - (b) Implementieren Sie nun den Tiefensuche-Algorithmus (DFS), den Sie in der Vorlesung kennengelernt haben, in den Methoden `dfs` und `checkNeighbor`.  
WICHTIG: `checkNeighbor` benötigt nicht nur `w` und `next` als Parameter, sondern auch `graph` und `mark`, da dies lokale Variablen in der Methode `dfs` sind.
2. Im zweiten Teil dieser Programmieraufgabe implementieren Sie den Quicksort-Algorithmus, den Sie ebenfalls aus der Vorlesung kennen. Implementieren Sie hierfür die Methode `quicksort`, welche die übergebene Liste `a` im Bereich von `l` bis `r` sortiert. Hier ist ebenfalls wieder ein kurzer Testcode am Fuße der Datei beigefügt.

TIPP: Mit `a[i]` erhalten Sie den  $i$ -ten Eintrag von `a` und mit `a[i] = x` schreiben sie den Wert `x` an die  $i$ -te Stelle.

**Lösung:**

Die Musterlösung der Programmieraufgabe finden Sie im StudOn in der Datei `ProgAufgabe03_Loesung.py`.

Gruppe 1: Abgabe der Hausaufgaben am 14.11.2017 in der Übung.  
Gruppe 2+3: Abgabe der Hausaufgaben am 15.11.2017 in der jeweiligen Übung.