Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. F. Kuhn

Klausur Algorithmen und Datenstrukturen

27. August 2020, 10:00 -13:00

Name: .....

Matrikel Nr.: .....

Unterschrift: .....

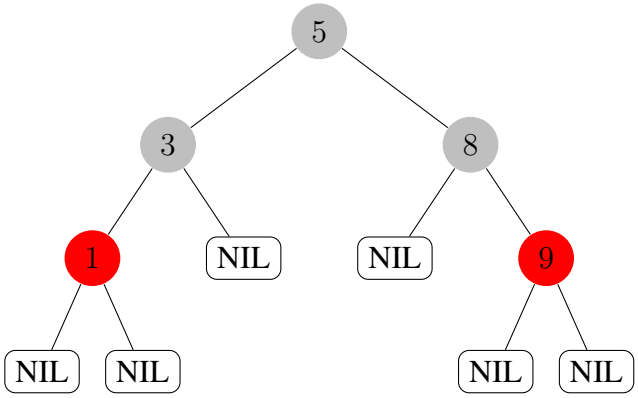
Erst öffnen, wenn die Klausuraufsicht die Erlaubnis erteilt!

- Schreiben Sie Ihren **Namen** und Ihre **Matrikelnummer** an die vorgesehenen Stellen.
- **Unterschreiben Sie** diese Seite um zu bestätigen, dass Sie die Fragen ohne unerlaubte Hilfsmittel beantwortet haben und die Klausuraufsicht über (gesundheitliche) Probleme informiert haben.
- Dies ist eine **Open Book** Klausur, weshalb alle gedruckten und handgeschriebenen Materialien erlaubt sind. **Elektronische Hilfsmittel** sind **nicht** erlaubt.
- Schreiben Sie **lesbar** und nur mit **dokumentenechten** Stiften. Benutzen Sie **keine rote** Farbe und **keinen Bleistift**!
- Es wird **nur eine Lösung pro Aufgabe** gewertet. Vergewissern Sie sich, dass Sie zusätzliche Lösungen durchstreichen, andernfalls wird die schlechteste Lösung gewertet.
- Detaillierte Schritte können Ihnen zu **Teilpunkten** verhelfen, falls das Endergebnis falsch ist.
- Die Schlüsselwörter **Zeigen Sie...**, **Beweisen Sie...**, **Begründen Sie...** oder **Leiten Sie ... her** zeigen an, dass Sie Ihre Antwort sorgfältig und gegebenenfalls formal begründen müssen.
- Die Schlüsselwörter **Geben Sie ... an, Zeichnen Sie ...** zeigen an, dass Sie lediglich die geforderte Antwort oder Zeichnung und keine Begründung liefern müssen (falls nicht explizit verlangt).
- Die folgenden Regeln gelten **überall**, außer sie werden explizit außer Kraft gesetzt.
- Bei Laufzeitfragen ist nur die **asymptotische Laufzeit** notwendig.
- Wenn Sie einen Algorithmus angeben sollen, so können Sie Pseudocode angeben. Eine **ausreichend detaillierte** Beschreibung der Funktionsweise Ihres Algorithmus genügt jedoch.
- Algorithmen aus der Vorlesung **können als Blackbox** verwendet werden.
- **Lesen Sie jede Aufgabe sorgfältig** durch und stellen Sie sicher, dass Sie diese verstanden haben!
- Falls Sie eine Frage zur Aufgabenstellung haben, geben Sie der Klausuraufsicht ein **Handzeichen**.
- Schreiben Sie Ihren Namen auf **alle Blätter**!
- Füllen Sie die **Erklärung über Ihren gesundheitlichen Status** aus.

Aufgabe	1	2	3	4	5	6	7	8	**Total**
Maximum	18	19	17	10	17	12	15	12	120
Punkte									

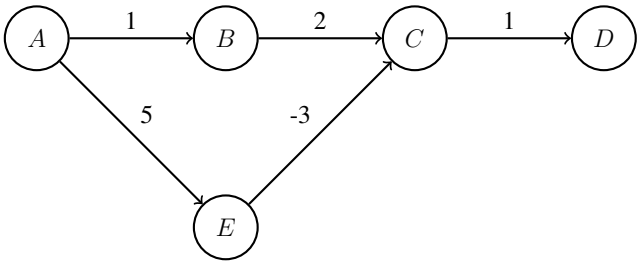
Aufgabe 1: Kurze Fragen *(18 Punkte)*

(a) Führen Sie auf folgendem Rot-Schwarz Baum die Operation `delete`(5) aus. Zeichnen Sie den resultierenden Baum. Sie können neben die Knoten die Farben schreiben. *(4 Punkte)*



(b) Betrachten Sie den folgenden *gerichteten, gewichteten* Graphen G. Führen Sie den Bellman-Ford Algorithmus auf G mit Startknoten A aus. Geben Sie die berechneten Distanzen aller Knoten *nach jeder* Iteration der äußeren Schleife an. *(4 Punkte)*

Hinweis: Die Iteration über die Kanten in der inneren Schleife ist nicht eindeutig. Wir fordern in dieser Aufgabe, dass Sie über die Kanten in alphabetischer Reihenfolge des Ausgangsknotens der Kante iterieren, d.h., die Kante (A, E) wird vor der Kante (B, C) iteriert.

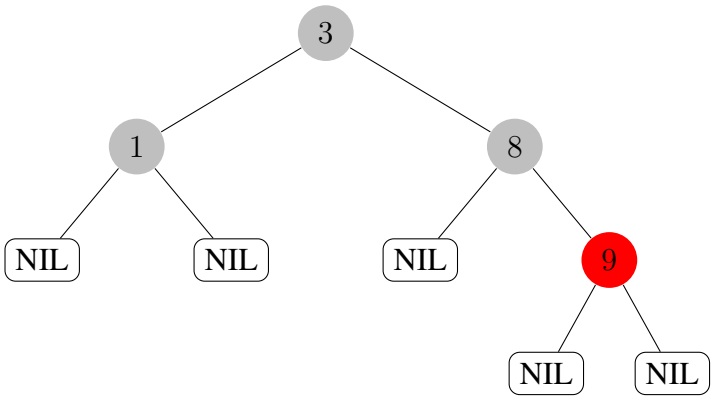


(c) Zeichnen Sie einen *ungerichteten, gewichteten* Graphen $G = (V, E, w)$ und markieren Sie einen Startknoten $s \in V$ so, dass sich jeder *"Shortest Path Tree"* mit Wurzel s in G von jedem *minimalen Spannbaum* von G unterscheidet. *(4 Punkte)*

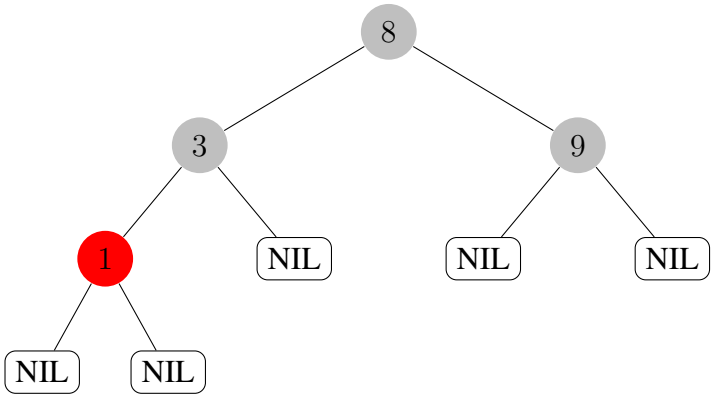
(d) Gegeben sei ein *ungerichteter, ungewichteter* (nicht notwendig zusammenhängender) Graph $G = (V, E)$. Wir möchten testen, ob dieser Graph *fast* ein Spannbaum ist. Darunter verstehen wir, dass G aus einem Spannbaum und höchstens c vielen zusätzlichen Kanten besteht, wobei $c \in O(1)$ eine gegebene Konstante ist. Beschreiben Sie einen Algorithmus der diesen Test in $\mathcal{O}(|V|)$ Zeit durchführt und begründen Sie die Laufzeit. *(6 Punkte)*

Musterlösung

(a) Wir suchen zunächst den Vorgänger v der Wurzel (alternativ Nachfolger). Dies ist der Knoten mit Schlüssel 3. Wir ersetzen den Schlüssel der Wurzel durch 3 und löschen v. Der Knoten mit Schlüssel 1 wird dabei schwarz gefärbt.



Wählt man den Nachfolger statt den Vorgänger, erhält man folgenden Baum:

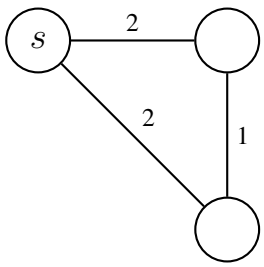


(b) Nach 1. Iteration: $\delta(A, A) = 0$, $\delta(A, B) = 1$, $\delta(A, C) = 2$, $\delta(A, D) = 4$, $\delta(A, E) = 5$.

Nach 2. Iteration: $\delta(A, A) = 0$, $\delta(A, B) = 1$, $\delta(A, C) = 2$, $\delta(A, D) = 3$, $\delta(A, E) = 5$.

Nach 3., 4. Iteration: $\delta(A, A) = 0$, $\delta(A, B) = 1$, $\delta(A, C) = 2$, $\delta(A, D) = 3$, $\delta(A, E) = 5$.

(c)



(d) Wir führen eine leicht modifizierte Tiefensuche durch, bei der wir (als zusätzlichen Parameter) einen Zähler der besuchten Kanten mitführen. Jeder Aufruf der Rekursion erhält den

aktuellen Zählerstand und gibt die neue Anzahl besuchter Kanten zurück (aber maximal $|V|+c$). Jedem rekursiven Aufruf auf einen weißen Knoten wird der aktuelle Zählerstand plus eins mitgegeben (initialer Aufruf mit 0!). Nach jedem rekursiven Aufruf wird der Zählerstand auf den Rückgabewert aktualisiert. Der Zähler sorgt dafür, dass wir die Tiefensuche nach spätestens $|V|+c$ besuchten Kanten abbrechen (es werden keine neuen rekursiven Aufrufe mehr gestartet wenn der Zähler diesen Wert erreicht).

Nach dem Traversierungsalgorithmus testen wir noch ob alle Knoten besucht wurden (keine weißen Knoten mehr). Wir geben `True` zurück (G fast ein Spannbaum) falls die Anzahl besuchter Kanten *kleiner gleich* $|V|-1+c$ ist (mehr Kanten kann ein "Fast-Spannbaum" nicht haben) *und* alle Knoten besucht wurden. Sonst geben wir `False` zurück. Da die Graphtraversierung nach $|V|+c$ Iterationen bzw. Rekursionen abbricht, dauert dies höchstens $\mathcal{O}(|V|+c) = \mathcal{O}(|V|)$ viele Zeitschritte. Der nachgelagerte Test ob es noch weiße Knoten gibt dauert ebenfalls höchstens $\mathcal{O}(|V|)$ viele Zeitschritte.

Alternative Lösung mit Breitensuche: Wir führen eine leicht modifizierte Breitensuche durch. Wir führen dazu Markierungen wie bei der Tiefensuche ein: Weiß falls der Knoten noch nicht besucht wurde, schwarz falls er bereits aus der queue entfernt und abgearbeitet wurde, sonst grau (mit den entsprechenden Änderungen im Code). Wir führen außerdem einen Zähler für die Anzahl der besuchten Kanten ein (initial 0) welcher für jeden weißen Nachbarn des aktuellen Knotens inkrementiert wird. Mittels einer Überprüfung in jeder Iteration ob der Zähler den Wert $|V|+c$ schon erreicht hat werden äußere und innere Schleife in diesem Fall abgebrochen. Die Breitensuche gibt dabei die Gesamtanzahl besuchter Kanten zurück (aber maximal $|V|+c$).

Aufgabe 2: Sortieralgorithmen *(19 Punkte)*

Gegeben seien k *sortierte* Arrays $A_1, \ldots, A_k$ mit insgesamt n Elementen. Wir wollen diese Arrays zu einem einzelnen *sortierten* Array A der Länge n zusammenfassen.

(a) Eine Lösungsmöglichkeit ist folgender Algorithmus

Algorithm 1 `sequential_merge`$(A_1, \ldots, A_k)$

$A = A_1$
for $i = 2$ to k **do**
    $A =$ *merge*(A, A_i)
return A

wobei *merge*() die Merge-Operation wie im Merge-Sort Algorithmus ist.

Angenommen k ist ein Teiler von n und alle Arrays haben die Länge n/k. Geben Sie die Laufzeit von `sequential_merge` als Funktion von n und k an. Begründen Sie Ihre Antwort. *(7 Punkte)*

(b) Ein Student schlägt stattdessen vor, alle Elemente auf beliebige Weise in ein Array der Länge n zu schreiben und dieses mit dem Merge-Sort Algorithmus aus der Vorlesung zu sortieren. Ist dieses Vorgehen schneller oder langsamer als `sequential_merge`? Begründen Sie Ihre Antwort. *(3 Punkte)*

Hinweis: Nehmen Sie wie in Teil (a) an, dass alle Arrays die Länge n/k haben.

(c) Wir möchten das gegebene Problem nun in Zeit $\mathcal{O}(n \log k)$ lösen, für beliebige Werte $k \leq n$, unter Verwendung von binären Heaps. Vervollständigen Sie dazu den folgenden Algorithmus `heap_merge` (schreiben Sie den Pseudo-Code, welcher an die Stelle "???" gehört). Begründen Sie die Laufzeit.

Algorithm 2 `heap_merge`$(A_1, \ldots, A_k)$

$H =$ *create_binary_heap*() ▷ creates an empty binary heap
for $i = 1$ to k **do**
    key $= A_i[0]$
    H.*insert*$((i, 0),$ *key*$)$
$A =$ Array of length n ▷ allocate an array of length n
for $j = 0$ to $n - 1$ **do**
    ???
return A

Hinweis: H verwaltet Daten der Form (i, ℓ), wobei ℓ eine Position im Array A_i angibt. *(9 Punkte)*

Musterlösung

(a) Die Kosten von *merge*(A, A_i) betragen $\mathcal{O}(|A| + |A_i|)$. Initial hat A die Länge n/k und wächst in jedem Schleifendurchlauf um n/k. Die Gesamtkosten betragen also

$$\mathcal{O}\left(\sum_{i=2}^{k} i \cdot \frac{n}{k}\right) = \mathcal{O}\left(\frac{n}{k}\sum_{i=2}^{k} i\right) = \mathcal{O}\left(\frac{n}{k} \cdot k^2\right) = \mathcal{O}(n \cdot k)\ .$$

(b) Das Vorgehen des Studenten dauert $\mathcal{O}(n \log n)$. Für $k = o(\log n)$ ist dieses Vorgehen asymptotisch langsamer als `sequential_merge`, für $k = \omega(\log n)$ schneller und für $k = \Theta(\log n)$ gleich schnell.

(c) **Algorithm 3** `heap_merge`$(A_1, \ldots, A_k)$

```
H = create_binary_heap()                    ▷ creates an empty binary heap
for i = 1 to k do
    key = A_i[0]
    H.insert((i, 0), key)
A = Array of length n                       ▷ allocate an array of length n
for j = 0 to n − 1 do
    (x, y) = H.deleteMin()
    A[j] = A_x[y]
    if y ≤ |A_x| − 2 then
        key = A_x[y + 1]
        H.insert([x, y + 1], key)
Return A
```

In der ersten Schleife wird ein Heap mit k Elementen gebildet. Dies kostet $\mathcal{O}(k \log k)$. Danach werden n delete-min und $\leq n$ insert Operationen auf dem Heap ausgeführt. Der Heap hat dabei stets die Größe $\leq k$. Die Gesamtkosten betagen also $\mathcal{O}(n \log k)$.

Aufgabe 3: O-Notation *(17 Punkte)*

Geben Sie an, ob die folgenden Aussagen wahr oder falsch sind. Beweisen oder widerlegen Sie die Aussagen anhand der Mengendefinition der Landau-Notation (d.h. insbesondere ohne Verwendung von Grenzwerten).

(a) $n^2 - 3n \in \Omega(n^2)$ *(4 Punkte)*

(b) $(\log n)^2 \in \mathcal{O}(\log(n^3))$ *(4 Punkte)*

(c) $n^2 \in \mathcal{O}\Big(\sum_{i=1}^{n} i\Big)$ *(4 Punkte)*

(d) Falls $f(n) \in o(g(n))$ und $h(n)$ monoton steigt, so gilt $h(f(n)) \in \mathcal{O}(h(g(n)))$ *(5 Punkte)*

Musterlösung

(a) Wahr. Wähle $c = \frac{1}{2}$ und $n_0 = 6$. Für $n \geq n_0$ gilt dann $n^2 \geq 6n$ und daher

$$n^2 - 3n = \tfrac{1}{2} \cdot n^2 + \tfrac{1}{2}(n^2 - 6n) \geq \tfrac{1}{2} \cdot n^2.$$

(b) Falsch. Für $c > 0$ gilt

$$\begin{array}{rccc} & (\log n)^2 & \leq & c \log(n^3) \\ \Leftrightarrow & (\log n)^2 & \leq & 3c \log n \\ \Leftrightarrow & \log n & \leq & 3c \\ \Leftrightarrow & n & \leq & 2^{3c} \end{array}$$

Für gegebenes $c, n_0 > 0$ wähle also $n := \max\{2^{3c} + 1, n_0\}$. Dann ist $n \geq n_0$, aber $(\log n)^2 > c \log(n^3)$.

(c) Wahr. In $\sum_{i=1}^{n} i$ sind mindestens $n/2$ Summanden $\geq n/2$, d.h. es gilt (für $n \geq 1$)

$$4 \sum_{i=1}^{n} i \geq 4 \cdot \frac{n}{2} \cdot \frac{n}{2} = n^2 \,.$$

Alternativ mit Gaußscher Summenformel.

(d) Wahr. Sei $c \leq 1$. Aus $f(n) \in o(g(n))$ folgt, dass es ein n_0 gibt, so dass für alle $n \geq n_0$ gilt $f(n) \leq c \cdot g(n) \leq g(n)$ und somit $h(f(n)) \leq h(g(n))$. Daher gilt $h(f(n)) \in \mathcal{O}(h(g(n)))$.

Aufgabe 4: Hashing mit offener Adressierung *(10 Punkte)*

Wir betrachten Hashtabellen mit offener Adressierung und zwei Methoden zur Auflösung von Kollisionen: *Doppel-Hashing* und *Cuckoo-Hashing*. Sei m die Größe der Hashtabelle. Wir definieren

$$\begin{aligned} h_1(x) &:= (5 \cdot x) \mod m, \\ h_2(x) &:= 1 + \big(2x \mod (m-1)\big), \\ h_3(x) &:= (3 \cdot x - 2) \mod m. \end{aligned}$$

(a) Sei $h_d(x, i) := (h_1(x) + i \cdot h_2(x)) \bmod m$. Fügen Sie die Schlüssel 13, 14, 2, 3, 11 der Reihe nach in eine Hashtabelle der Größe $m := 11$ ein. Benutzen Sie dafür h_d und *Doppelhashing* zur Kollisionsauflösung. *(5 Punkte)*

(b) Fügen Sie die Werte 3, 10, 7 der Reihe nach in eine Hashtabelle der Größe $m := 7$ ein. Benutzen Sie *Cuckoo-Hashing* mit den Funktionen h_1 und h_3 zur Kollisionsauflösung. Geben Sie den Zwischenstand der Tabelle nach dem Einfügen jeden Wertes an (d.h. es sind drei Tabellen anzugeben). *(5 Punkte)*

Hinweis: Die Lösungen sind in die Tabellen auf dem Lösungsblatt dieser Aufgabe einzutragen.

Musterlösung

Aufgabenteil (a):

Tabellenzustand nach Einfügen aller Werte:

3			11	14					2	13
0	1	2	3	4	5	6	7	8	9	10

Aufgabenteil (b):

Nach Einfügen von 3:

	3					
0	1	2	3	4	5	6

Nach Einfügen von 10:

3	10					
0	1	2	3	4	5	6

Nach Einfügen von 7:

10	3				7	
0	1	2	3	4	5	6

Aufgabe 5: Graphen *(17 Punkte)*

Gegeben ein *gerichteter, ungewichteter* Graph $G = (V, E)$ definiert man $G^2 = (V, E^2)$ durch

$(u, v) \in E^2$ *genau dann, wenn* $u \neq v$ *und es in* G *einen (gerichteten) Pfad von* u *nach* v *der Länge höchstens 2 gibt.*

(a) Beschreiben Sie einen Algorithmus mit Laufzeit $\mathcal{O}(|E| \cdot |V|)$, der aus der Adjazenzlisten-Repräsentation von G die Adjazenzlisten-Repräsentation von G^2 berechnet. D.h., v muss genau dann in der Adjazenzliste von u enthalten sein, wenn es einen Pfad von u nach v der Länge höchstens 2 gibt. Wir nehmen hier erstmal an, dass v in diesem Fall auch **mehrfach** in der Liste vorkommen darf. Begründen Sie die Laufzeit. *(6 Punkte)*

(b) Beschreiben Sie einen Algorithmus mit Laufzeit $\mathcal{O}(|E| \cdot |V|)$, der aus der Adjazenzlisten-Repräsentation von G die Adjazenzlisten-Repräsentation von G^2 **ohne Mehrfachvorkommen** eines Knotens in einer Adjazenzliste berechnet. Begründen Sie die Laufzeit.

Hinweis: Falls Sie eine Hashtabelle verwenden, dürfen Sie annehmen, dass Hinzufügen und Nachschauen von Werten $\mathcal{O}(1)$ Zeit benötigt. *(4 Punkte)*

(c) Beschreiben Sie, wie man aus der Adjazenzmatrix von G in Zeit $\mathcal{O}(|V|^3)$ die Adjazenzmatrix von G^2 berechnet. Erklären Sie die Laufzeit. *(7 Punkte)*

Musterlösung

(a) Man *ergänzt* die Adjazenzliste A von G wie folgt

Algorithm 4 `square-adj-list`$(A[0..n-1])$ ▷ *Assume nodes are numbered* $0 \ldots n-1$

```
initialize L_u as empty list for each u ∈ V
allocate array A' of length n
for each node u ∈ V do
    for each v in the adjacency list A[u] do
        for each w in the adjacency list A[v] do
            Add w to L
    append list A[u] to L_u
    A'[u] = L_u
return A'
```

In den ersten beiden Schleifen hat man insgesamt $\mathcal{O}(\max\{|V|, |E|\})$ Durchläufe da *alle Iterationen* der zweiten Schleife der Anzahl der Kanten entspricht und man mindestens Anzahl Knoten viele Iterationen in der ersten Schleife hat. In der dritten Schleife hat man höchstens $\min\{|V|, |E|\}$ viele Iterationen, da ein Knoten nur maximal so viele Nachbarn haben kann. Wegen $\max\{x, y\} \cdot \min\{x, y\} = x \cdot y$ ist die Gesamtlaufzeit der Schleifen daher $\mathcal{O}(|E| \cdot |V|)$. (zusammenhängen von Listen geht in $\mathcal{O}(1)$).

(b) Man führt zuerst den Algorithmus aus (a) aus und löscht dann alle Mehrfachvorkommen.

Algorithm 5 `clean-up` ($A'[0..n-1]$) ▷ *Assume nodes are numbered* $0 \ldots n-1$

```
for each node u ∈ V do
    allocate a dictionary/hash table D
    add u to D
    for each v in the adjacency list A'[u] do
        if v ∈ D then
            delete v from A'[u]          ▷ deletion in O(1) by remembering predecessor
        else
            add v to D
return A'
```

Die Laufzeit des Löschalgorithmus beträgt $\mathcal{O}(|V| + |E| + \sum_{u \in V} |L_u|)$. Mit $\sum_{u \in V} |L_u| = \mathcal{O}(|V||E|)$ erhält man als Laufzeit $\mathcal{O}(|V||E|)$.

Man kann auch direkte Adressierung verwenden und statt D ein Array der Länge n nehmen (der Speicherplatz war ja in keinster Weise eingeschränkt). Man muss auch nicht zuerst den Algorithmus aus (a) anwenden und dann Mehrfachvorkommen löschen, sondern kann den Test ob v schon in der $A'[i]$ enthalten ist direkt in den Algorithmus aus (a) einbauen.

(c) *Algebraische Beschreibung:* Sei A die Adjazenzmatrix von G. Wir berechnen $B = A^2 + A$ und setzen alle Einträge ≥ 1 von B auf 1 und setzen die Hautpdiagonale von B auf 0.

Intuitivere Beschreibung: Man berechnet Eintrag (i, j) in B folgendermaßen: Die Einträge auf der Hauptdiagonalen setzt man auf 0. Für $i \neq j$ und $A[i, j] = 1$ (d.h. in G gibt es eine Kante von i nach j) setzt man $B[i, j] = 1$. Für $i \neq j$ und $A[i, j] = 0$ geht man durch alle Nachbarn von i und schaut, ob diese eine Kante nach j haben. In Pseudocode:

Algorithm 6 `square-adj-matrix`($A[0..n-1][0..n-1]$)

```
allocate an n × n matrix B
for i = 0 to n − 1 do
    for j = 0 to n − 1 do
        if i == j then
            B[i][j] = 0
        else if A[i][j] = 1 then
            B[i][j] = 1
        else
            for k = 0 to n − 1 do
                if A[i][k] == 1 and A[k][j] == 1 then
                    B[i][j] = 1
return B
```

Für jeden der n^2 Einträge muss man $\mathcal{O}(n)$ Operationen ausführen. Die Laufzeit beträgt daher $\mathcal{O}(n^3)$.

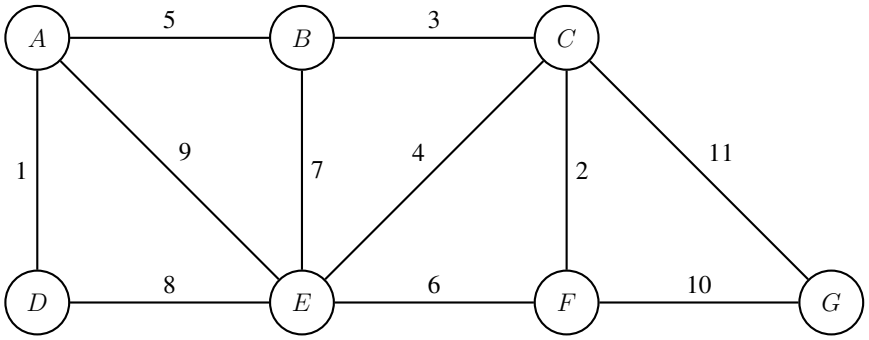
Aufgabe 6: Mystische Funktion *(12 Punkte)*

Betrachten Sie den folgenden Algorithmus in abstraktem Pseudocode. Dieser erhält als Eingabe einen *gewichteten, ungerichteten, zusammenhängenden* Graphen $G = (V, E, w)$.

Algorithm 7 `myst-edge-set`(V, E, w)

for jedes $e \in E$ nach *absteigendem* Gewicht $w(e)$ **do** ▷ *beachte Iterationsreihenfolge!*
  entferne e aus E
  if (V, E) ist nicht zusammenhängend **then**
    füge e zu E hinzu
return E

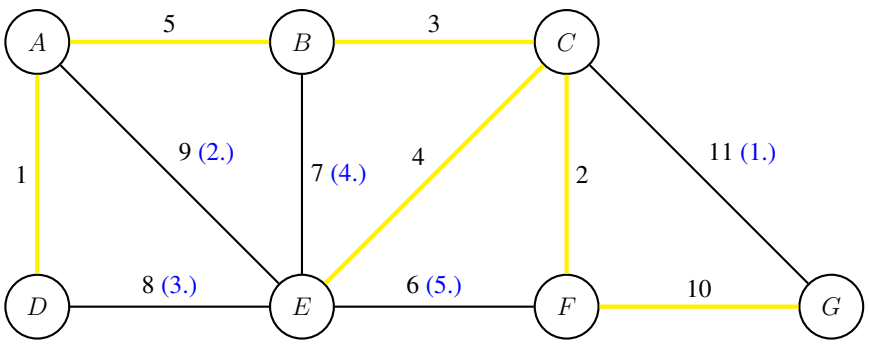
(a) Führen Sie den Algorithmus `myst-edge-set`(V, E, w) auf dem unten stehenden Graphen aus. Nummerieren Sie in dem Graphen die Reihenfolge, in der Kanten von dem Algorithmus gelöscht werden (als Zahl neben der jeweiligen gelöschten Kante). Markieren Sie außerdem alle Kanten, die zurück gegeben werden (umranden oder fett nachzeichnen). *(5 Punkte)*



(b) Welche Ausgabe gibt `myst-edge-set`(V, E, w) zurück? Beweisen Sie Ihre Behauptung. *(7 Punkte)*

Musterlösung

(a)



(b) Die Rückgabe entspricht einem MST. Für den Beweis beobachten wir zunächst folgende Eigenschaften des Algorithmus.

(1.) Der Algorithmus entfernt Kanten nur dann, wenn der Graph dann zusammenhängend bleibt. Damit ist der durch die zurückgegebenen Kanten induzierte Graph am Ende ebenfalls zusammenhängend.

(2.) Die zurückgegebene Kantenmenge hat keinen Zyklus. Denn andernfalls gäbe es einen Schnitt "durch" diesen Zyklus, welcher dann zwei Kanten in der zurückgegebenen Menge enthielte. Der Algorithmus hätte aber eine dieser beiden Kanten entfernt, da die Kantenmenge nach Entfernung noch zusammenhängend ist.

(3.) Sei e eine Kante in der Rückgabemenge. Es gibt einen Schnitt $(S, V \setminus S)$ auf dem e die einzige Kante in der Rückgabemenge ist, da diese zyklenfrei ist (trenne einen "Ast" im Baum ab). Da die Kanten im Schnitt $(S, V \setminus S)$ von G in absteigender Reihenfolge nach Gewicht entfernt wurden, ist e die (eine) leichteste Kante von $(S, V \setminus S)$.

Nach (3.) haben wir nur sichere Kanten in der Rückgabemenge. Die Rückgabemenge ist nach (2.) zyklenfrei und nach (1.) zusammenhängend, ist also ein *Spannbaum*. Eine sichere, zyklenfreie Kantemenge ist immer Teilmenge eines MST (nach Vorlesung). Damit ist der zurückgegebene Spannbaum insbesondere ein minimaler Spannbaum.

Aufgabe 7: Wörter trennen *(15 Punkte)*

Gegeben sei ein Dictionary D, welches Wörter der Länge höchstens $k \in \mathcal{O}(1)$ enthält. Sei T eine Zeichenkette bestehend aus n Zeichen. Wir möchten berechnen, ob sich T in zusammenhängende Zeichenketten zerlegen lässt, sodass jede Zeichenkette einem Wort in D entspricht.

Beispiel: Sei $D = \{$`'Flugzeug', 'Algorithmen', 'Zug', 'super', 'sind'`$\}$. Dann ist die Lösung des obigen Problems für die Eingaben $T_1 =$ `'Algorithmensinddoof'` oder $T_2 =$ `'FlugZugzeug'` jeweils `False`. Für $T_3 =$ `'sindAlgorithmensuper'` ist die Antwort zu obigem Problem `True`.

Hinweise: Gehen Sie davon aus, dass die Konstante k als Teil der Eingabe gegeben ist und dass Sie (bspw. mittels Hashing) in $\mathcal{O}(1)$ Zeitschritten testen können, ob eine Zeichenkette der Länge höchstens k in D enthalten ist. Sie dürfen außerdem annehmen, dass die Zeichen von T in einem Array $T[0..n-1]$ vorliegen.

(a) Sei $t : \{0, \ldots, n-1\} \to \{$`True, False`$\}$ eine Funktion, sodass $t(i) =$ `True` genau dann, wenn sich die Zeichenkette gegeben durch $T[0..i]$ in zusammenhängende Zeichenketten zerlegen lässt, sodass jede dieser Zeichenketten in D enthalten ist.

Bestimmen Sie zunächst einen rekursiven Zusammenhang für $t(i)$. Das heißt, geben Sie eine Vorschrift wie $t(i)$ mittels $t(j)$ mit $j < i$ berechnet werden kann. Begründen Sie kurz. *(8 Punkte)*

Hinweis: Sei T eine Zeichenkette, die sich zerlegen lässt in Zeichenketten $T = T_1 \ldots T_\ell$ sodass $T_i \in D$. Betrachten Sie $T_1 \ldots T_{\ell-1}$ und T_ℓ gesondert.

(b) Geben Sie einen Algorithmus an, der das obige Problem in $\mathcal{O}(n)$ vielen Zeitschritten mittels *dynamischer Programmierung* löst. Begründen Sie die Laufzeit. *(7 Punkte)*

Musterlösung

(a) Wenn sich eine Zeichenkette T zerlegen lässt in $T = T_1 \ldots T_\ell$ mit $T_i \in D$, dann ist insbesondere $T_\ell \in D$ und hat weniger als k Zeichen. Wir müssen also nur testen ob T ein Suffix mit höchstens k Zeichen hat welches in D ist **und** ob sich das übrig bleibende Präfix ebenfalls in Wörter aus D zerlegen lässt. Die Frage ob $t(i) =$ `True` ist, lässt sich also zurückführen auf die Frage ob es ein Suffix $T[j..i] \in D$ von $T[0..i]$ gibt, welches höchstens aus höchstens k Zeichen besteht und außerdem $t(j-1) =$ `True` ist. Wir formalisieren diese Beobachtung wie folgt

$$t(i) = \bigvee_{j=\max(i-k+1,0)}^{i} \Big(t(j-1) = \texttt{True} \textbf{ and } T[j..i] \in D \Big)$$

Um den Fall abzudecken in dem $T[0..i]$ schon in D ist, setzen wir $t(j) :=$ `True` für $j < 0$.

(b) Ausgehend von der obigen Rekursion, wenden wir die Strategie des dynamischen Programmierens an.

Algorithm 8 `is-decomposable(`i`)` ▷ *D, k, T, `memo` seien global gegeben*

```
if j < 0 then return True
d = False
for j = max(i − k + 1, 0) to i do                    ▷ Schleife inklusive i
    if memo[j−1] = Null then                         ▷ Ergebnis für j−1 liegt noch nicht vor
        memo[j−1] = is-decomposable(j−1)
    d = d or (memo[j−1] and T[j..i] ∈ D)
return d
```

Jeder Aufruf von `is-decomposable(`i`)` dauert $\mathcal{O}(1)$ Zeitschritte (unter Vernachlässigung rekursiver Aufrufe), da wir eine Schleife nur $\mathcal{O}(k) = \mathcal{O}(1)$ Durchläufe hat und der Test $T[j..i] \in D$ in $\mathcal{O}(1)$ Zeit durchgeführt werden kann. Es gibt auch nur $\mathcal{O}(n)$ Aufrufe von `is-decomposable(`i`)`, denn spätestens dann liegt das Ergebnis von `memo`$[i]$ für jedes $i \in \{0, \ldots, n-1\}$ vor.

Aufgabe 8: Textsuche *(12 Punkte)*

Gegeben sei das Muster $P = BACBAB$ und der Text $T = CBACABBACBABACBABA$.

(a) Geben Sie das Array S des Knuth-Morris-Pratt Algorithmus an. *(4 Punkte)*

(b) Finden Sie mittels des Knuth-Morris-Pratt Algorithmus alle Vorkommen von P in T. Die Schritte des Algorithmus sollen klar erkennbar sein. Sie können dazu die Tabelle auf dem Lösungsblatt verwenden. *(8 Punkte)*

Musterlösung

(a) $[-1, 0, 0, 0, 1, 2, 1]$

(b)

C	B	A	C	A	B	B	A	C	B	A	B	A	C	B	A	B	A
B	A	C	B	A	B												
	B	A	C	B	A	B											
				B	A	C	B	A	B								
					B	A	C	B	A	B							
						B	A	C	B	A	B						✓
											B	A	C	B	A	B	✓
																B	A