Name:
Vorname:
Matrikelnummer:

Klausur-ID:

Lösungsvorschlag

Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dennis Hofheinz

5. Oktober 2016

Klausur Algorithmen I

Aufgabe 1.	Kleinaufgaben	11 Punkte
Aufgabe 2.	Sortieren	15 Punkte
Aufgabe 3.	Verlosungen	11 Punkte
Aufgabe 4.	Pfade in Graphen	15 Punkte
Aufgabe 5.	Minimale Spannbäume	8 Punkte

Bitte beachten Sie:

- Bringen Sie den Aufkleber mit Ihrem Namen, Ihrer Matrikelnummer und Ihrer Klausur-ID oben links auf dem Deckblatt an.
- Merken Sie sich Ihre Klausur-ID und schreiben Sie auf **alle Blätter** der Klausur und Zusatzblätter Ihre Klausur-ID und Ihren Namen.
- Die Klausur enthält 20 Blätter. Die Bearbeitungszeit beträgt 120 Minuten.
- Die durch Übungsblätter gewonnenen Bonuspunkte werden erst nach Erreichen der Bestehensgrenze hinzugezählt. Die Anzahl der Bonuspunkte entscheidet nicht über das Bestehen der Klausur.
- Als Hilfsmittel ist ein beidseitig handbeschriebenes DIN-A4 Blatt zugelassen.
- Bitte kennzeichnen Sie deutlich, welche Aufgabe gewertet werden soll. Bei mehreren angegebenen Möglichkeiten wird jeweils die schlechteste Alternative gewertet.

Aufgabe	1	2	3	4	5	Summe
max. Punkte	11	15	11	15	8	60
Punkte						

Bonuspunkte:	Summe:	**Note:**

Lösungsvorschlag

Aufgabe 1. Kleinaufgaben [11 Punkte]

Bearbeiten Sie die folgenden Aufgaben. Begründen Sie Ihre Antworten jeweils kurz. *Reine Ja/Nein-Antworten ohne Begründung geben keine Punkte.*

a. Ein Algorithmus besitzt *polynomielle Zeitkomplexität*, wenn die Laufzeit im Worst Case durch ein Polynom in der Eingabegröße beschränkt werden kann. Für die meisten Sortieralgorithmen hat sich eine Zeitkomplexität von $O(n \log n)$ ergeben. Offensichtlich ist $n \log n$ kein Polynom. Haben diese Sortieralgorithmen dennoch polynomielle Zeitkomplexität? Begründen Sie Ihre Antwort kurz. [1 Punkt]

Lösung

Ja, diese Sortieralgorithmen haben polynomielle Zeitkomplexität: Für alle $n \in \mathbb{N}$ gilt $n > \log n$ und somit $n^2 > n \log n$. Da n^2 ein Polynom in n ist, wird die Laufzeit also durch ein Polynom beschränkt. Die nicht-polynomielle Form $n \log n$ ist allerdings eine schärfere Laufzeitschranke, weshalb man diese üblicherweise angibt.

b. Was ist der Unterschied zwischen Linear Programs (LP) und Integer Linear Programs (ILP)? Welche von beiden sind im Allgemeinen einfacher zu lösen? Begründen Sie Ihre Antwort kurz. [1 Punkt]

Lösung

Ein Integer Linear Program hat im Vergleich zu einem Linear Program die Zusatzbedingung, dass alle Variablen (für Mixed Integer Programs: einige Variablen) ganzzahlig sein müssen. Während Linear Programs in Polynomialzeit gelöst werden können, gilt dies für Integer Linear Programs im Allgemeinen nicht, diese sind also schwieriger zu lösen.

c. Beweisen oder widerlegen Sie: $2^{\log_{10}(5n)} \in O(n)$. [2 Punkte]

Lösung

Es gilt

$$2^{\log_{10}(5n)} = 2^{\log_{10} 5 + \log_{10} n} = 2^{\log_{10} 5} \cdot 2^{\log_2 n \cdot \log_{10} 2} = 2^{\log_{10} 5} \cdot \left(2^{\log_2 n}\right)^{\log_{10} 2} = 2^{\log_{10} 5} \cdot n^{\log_{10} 2}.$$

Da $2^{\log_{10} 5}$ konstant ist und $\log_{10} 2 < 1$ gilt, folgt damit $2^{\log_{10}(5n)} \in O(n)$.

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

d. Sie möchten Zahlen aus $\mathbb{Q}^+$, die durch Zähler und Nenner repräsentiert vorliegen, sortieren, und dabei eine Implementierung Ihrer Kollegen möglichst unverändert und ohne weitere Vorberechnungen wiederverwenden. Einer Ihrer Kollegen hat bereits LSD-Radix-Sort implementiert, ein anderer InsertionSort, und ein dritter QuickSelect. Welche Implementierung wählen Sie, und warum? [2 Punkte]

Lösung

QuickSelect ist kein Sortieralgorithmus, sondern ein Algorithmus zum Finden eines Quantils. Durch iteratives Finden des jeweils nächstgrößeren Elements mittels QuickSelect ließe sich eine Liste zwar auch sortieren, aber dies wäre ineffizient.
LSD-RadixSort ist ein Sortieralgortihmus zum Sortieren ganzzahliger (oder zumindest diskreter) Elemente. Da die Zahlen aber aus $\mathbb{Q}^+$ stammen, ist dieser hier nicht direkt anwendbar.
InsertionSort ist als einziger vergleichsbasierter Sortieralgorithmus anwendbar und wird daher gewählt.

e. Beweisen oder widerlegen Sie: Sei für eine Konstante $k > 1$

$$T(n) = \begin{cases} 1, & \text{falls } n = 1 \\ (k-1) \cdot T(n/k) + k \log_2 n, & \text{sonst.} \end{cases}$$

Dann gilt $T(n) \in \mathrm{O}(n)$. [2 Punkte]

Lösung

Hinweis: Hier hätte genau genommen T als

$$T(n) = \begin{cases} 1, & \text{falls } n = 1 \\ (k-1) \cdot T(\lceil n/k \rceil) + k \log_2 n, & \text{sonst} \end{cases}$$

definiert werden müssen, damit sicher der Basisfall $n = 1$ irgendwann erreicht wird.

Man wendet das Master-Theorem an: Da $\log_2 n < n$ für alle $n \in \mathbb{N}$ gilt, folgt $T(n) < T'(n) := (k-1)T\left(\frac{n}{k}\right) + kn$. Für T' tritt im Master-Theorem der Fall „$d < b$“ ($k - 1 < k$) ein, womit $T'(n) \in \Theta(n)$ folgt, und aus $T(n) < T'(n)$ ergibt sich schließlich $T(n) \in \mathrm{O}(n)$.

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

f. Gegeben sei die Hashfunktion $h(x) = 4x + 7 \mod 10$.
Im Folgenden betrachten wir Hashing mit linearer Suche und Hashing mit doppelt verketteten Listen. Bei Hashing mit linearer Suche werden am Ende der Tabelle zwei Einträge m_1 und m_2 angehängt, die als zusätzliche Pufferplätze dienen. Beim Hashing mit verketteten Listen sollen die Einträge jeweils am Ende der jeweiligen Listen angehängt werden. Die Kästchen, die die Listeneinträge darstellen, sind nicht eingezeichnet und müssen von Ihnen ergänzt werden.
Fügen Sie die Werte $12, 5, 4, 14, 10, 9$ mittels der Hashfunktion h in der angegebenen Reihenfolge ein. Wenn Sie nicht die vorgedruckten Tabellen benutzen, machen Sie kenntlich, in welcher Tabelle Sie welches Verfahren verwendet haben. Als Hilfe geben wir Ihnen hier die Hashwerte der einzufügenden Elemente an:

$$h(12) = 5, \quad h(5) = 7, \quad h(4) = 3, \quad h(14) = 3, \quad h(10) = 7, \quad h(9) = 3.$$

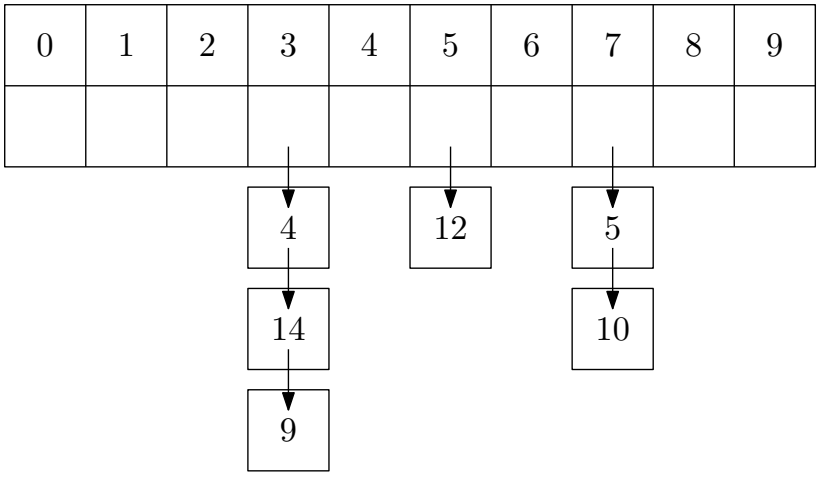
[2 Punkte]

Lösung

Hashing mit linearer Suche:

0	1	2	3	4	5	6	7	8	9	m_1
			4	14	12	9	5	10		

Hashing mit verketteten Listen:



g. Ausgehend von der im vorigen Aufgabenteil **f.** angefertigten Tabelle, löschen Sie aus der

Hashtabelle mit **linearer Suche** den Wert 14. Geben Sie die resultierende Tabelle erneut an.
[1 Punkt]

Lösung

0	1	2	3	4	5	6	7	8	9	m_1
			4	9	12		5	10		

Lösungsvorschlag

Aufgabe 2. Sortieren [15 Punkte]

Betrachten Sie folgenden Algorithmus, der ein Array $A[1...A.\text{length}]$ als Eingabe erhält:

```
1:  Procedure examplesort(A: Array of ℝ)
2:      p = 1
3:      while p ≤ A.length do
#:          // Geben Sie den Zustand in dieser Zeile aus.
4:          if p = 1 then
5:              p = p + 1
6:          else if A[p] ≥ A[p − 1] then
7:              p = p + 1
8:          else
9:              tausche A[p] und A[p − 1]
10:             p = p − 1
```

a. Gegeben sei das Eingabearray $A = [2,1,4,5,3]$. Geben Sie A jeweils *am Anfang* jedes Schleifendurchlaufs wieder (siehe Kommentar im Pseudocode oben), bis der Algorithmus endet. Machen Sie jeweils die Position p durch Umkreisen der entsprechenden Kästchennummer kenntlich (wie im Beispiel).
Hinweis: Die Anzahl der Kästchen entspricht nicht unbedingt der Anzahl der tatsächlich notwendigen Schleifendurchläufe, lassen Sie gegebenenfalls die überflüssigen Kästchen leer. Falls Sie einen Fehler machen, streichen Sie die Kästchen deutlich durch und nehmen Sie den nächsten Block von Kästchen. Tragen Sie in die Überschriften jeweils ein, welche Schleifendurchläufe Sie in der Zeile bearbeitet haben. Falls Sie keinen Fehler machen also gerade 1. bis 3. Schleifendurchlauf, 4. bis 6. Schleifendurchlauf, 7. bis 9. Schleifendurchlauf und so weiter. [3 Punkte]

Lösung

1. bis 3. Schleifendurchlauf:

①	2	3	4	5
2	1	4	5	3

1	②	3	4	5
2	1	4	5	3

①	2	3	4	5
1	2	4	5	3

4. bis 6. Schleifendurchlauf:

1	②	3	4	5
1	2	4	5	3

1	2	③	4	5
1	2	4	5	3

1	2	3	④	5
1	2	4	5	3

7. bis 9. Schleifendurchlauf:

1	2	3	4	⑤
1	2	4	5	3

1	2	3	④	5
1	2	4	3	5

1	2	③	4	5
1	2	3	4	5

10. bis 11. Schleifendurchlauf:

1	2	3	④	5
1	2	3	4	5

1	2	3	4	⑤
1	2	3	4	5

1	2	3	4	5

Lösungsvorschlag

Fortsetzung von Aufgabe 2

```
1: Procedure examplesort(A: Array of ℝ)
2:         p = 1
3:         while p ≤ A.length do
#:             Invariante:_____
4:             if p = 1 oder A[p] ≥ A[p − 1] then
5:                 p = p + 1
6:             else
7:                 tausche A[p] und A[p − 1]
8:                 p = p − 1
```

b. Geben Sie in Zeile # eine Schleifen-Invariante an, die die Korrektheit des Algorithmus impliziert. Beweisen Sie die Invariante, d.h. zeigen Sie, dass die Invariante vor und am Anfang jedes Schleifendurchlaufs gilt.
Wenn Sie mit den Werten von p und A während eines Schleifendurchlaufes argumentieren, verwenden Sie bitte die Bezeichnungen p_{alt} und A_{alt} für die Werte vor der Veränderung im Schleifendurchlauf, und p_{neu} bzw. A_{neu} für die Werte nach der Veränderung.

[4 Punkte]

Lösung

Invariante: das Array $A[1]...A[p_{\text{alt}} - 1]$ liegt in aufsteigend sortierter Reihenfolge vor
Beweis:

- Vor dem 1. Schleifendurchlauf gilt die Invariante trivialerweise, da das Array leer ist.
- Angenommen die Invariante gilt am Anfang des i-ten Schleifendurchlaufs, dann ist zu zeigen, dass sie auch am Anfang des $i+1$-ten Schleifendurchlaufes gilt: Es gelte also nun $A_{\text{alt}}[1]...A_{\text{alt}}[p_{\text{alt}} - 1]$ liegt in aufsteigend sortierter Reihenfolge vor. Wir unterscheiden folgende Fälle:
  1. $p_{\text{alt}} = 1$: dann gilt $p_{\text{neu}} = 2$ und das Array $A_{\text{neu}}[1]...A_{\text{neu}}[2-1]$ besteht nur aus einem Element und ist damit trivialerweise aufsteigend sortiert
  2. $A_{\text{alt}}[p_{\text{alt}}] \geq A_{\text{alt}}[p_{\text{alt}} - 1]$: dann gilt am Anfang des $i+1$-ten Schleifendurchlaufes $A_{\text{neu}}[1]...A_{\text{neu}}[p_{\text{neu}} - 1] = A_{\text{alt}}[1]...A_{\text{alt}}[p_{\text{alt}}]$ liegt in aufsteigend sortierter Reihenfolge vor, denn nach Voraussetzung lag $A_{\text{alt}}[1]...A_{\text{alt}}[p_{\text{alt}} - 1]$ schon in aufsteigend sortierter Reihenfolge vor und weiter gilt $A_{\text{alt}}[p_{\text{alt}} - 1] \leq A_{\text{alt}}[p_{\text{alt}}]$
  3. $A_{\text{alt}}[p_{\text{alt}}] < A_{\text{alt}}[p_{\text{alt}} - 1]$: dann gilt $p_{\text{neu}} = p_{\text{alt}} - 1$ und da nur $A_{\text{alt}}[p_{\text{alt}} - 1]$ und $A_{\text{alt}}[p_{\text{alt}}]$ getauscht werden, liegt $A_{\text{neu}}[1]...A_{\text{neu}}[p_{\text{neu}} - 1] = A_{\text{alt}}[1]...A_{\text{alt}}[p_{\text{alt}} - 2]$ immernoch in aufsteigend sortierter Reihenfolge vor

(weitere Teilaufgaben auf den nächsten Blättern)

c. Was müssten Sie weiter zeigen, damit die Korrektheit des Algorithmus folgt?
Hinweis: Der eigentliche Beweis ist hier nicht gefordert. [1 Punkt]

Lösung

Um die Korrektheit des Algorithmus zu beweisen, müsste noch gezeigt werden, dass die Schleife immer terminiert.

d. Geben Sie eine Funktion f an, so dass die Laufzeit des Algorithmus *examplesort* auf einer Eingabe der Länge n in *Best Case* in $\Theta(f)$ ist. Begründen Sie Ihre Antwort kurz. Geben Sie außerdem ein Array der Länge 5 an, so dass der Algorithmus mit dieser Eingabe die optimale Laufzeit erreicht. [2 Punkte]

Lösung

Im Best Case wird nie der **else**-Zweig aufgerufen und daher p in jedem Durchlauf um 1 erhöht. Insgesamt wird die While-Schleife also nur n mal durchlaufen, die Laufzeit des Algorithmus liegt also für $f(n) = n$ in $\Theta(f)$. Ein Beispiel hierfür ist $A = [1,2,3,4,5]$.

e. Geben Sie eine Funktion f an, so dass die Laufzeit des Algorithmus *examplesort* auf einer Eingabe der Länge n im *Worst Case* in $\Theta(f)$ ist. Begründen Sie Ihre Antwort kurz. Geben Sie außerdem ein Array der Länge 5 an, so dass der Algorithmus mit dieser Eingabe die schlechteste Laufzeit erreicht. [2 Punkte]

Lösung

Im Worst-Case muss jedes Element ganz an den Anfang geschoben werden. Dafür sind für das i-te Element $i-1$ Verschiebeoperation nötig, weiter sind i Operationen nötig um danach zum $i+1$-ten Element zu laufen. Damit benötigt der Algorithmus Laufzeit $f(n) = 2\sum_{i=1}^{n}(i-1) \in \Theta(n^2)$. Ein Beispiel hierfür ist $A = [5,4,3,2,1]$.

f. Für welche Art von Eingaben würden Sie Insertionsort zur Sortierung empfehlen und warum? [1 Punkt]

Lösung

Wenn bekannt ist, dass die Eingaben fast sortiert sind, ist Insertionsort empfehlenswert. In diesem Fall liegt die Laufzeit von Insertionsort nahe an $O(n)$ und damit besser als z.B. Mergesort oder Quicksort.
Außerdem ist Insertionsort für kleine Eingaben empfehlenswert, da im Gegensatz zu Algorithmen wie z.B. Mergesort oder Quicksort kein Overhead durch Rekursion anfällt.

g. Betrachten Sie die Algorithmen Insertionsort, Mergesort und Quicksort aus der Vorlesung. Was sind jeweils die Laufzeiten im Best Case und im Worst Case (im O-Kalkül)? [2 Punkte]

Lösung

Laufzeit:

	Best Case	Worst Case
Insertionsort	$\Theta(n)$	$\Theta(n^2)$
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n \log n)$	$\Theta(n^2)$

Lösungsvorschlag

Aufgabe 3. Verlosungen [11 Punkte]

Sie veranstalten eine Verlosung mit n Teilnehmern. Diese läuft wie folgt ab: Jeder Teilnehmer (der durch seinen Namen[1] identifiziert wird) tippt eine natürliche Zahl in $[1,1000]$ und teilt Ihnen diese verdeckt mit. Nachdem dies alle Teilnehmer getan haben, generieren Sie eine natürliche Zufallszahl z ebenfalls aus $[1,1000]$. Insgesamt steht als Preisgeld der Betrag $G \in \mathbb{N}$ zur Verfügung. Derjenige Teilnehmer, dessen Zahl am nächsten an z gelegen ist, bekommt $\frac{1}{2}G$ ausgezahlt. Der Teilnehmer mit der zweitnächsten Zahl bekommt $\frac{1}{4}G$ ausgezahlt, der nächste $\frac{1}{8}G$, und so weiter.[2]
Sie dürfen davon ausgehen, dass keine zwei Teilnehmer so tippen, dass sie denselben Abstand vom von Ihnen gezogenen Wert haben. Außerdem haben keine zwei Teilnehmer denselben Namen. Es gilt weiterhin $G = 2^k$ und $k \geq n$.
Hinweis: Die im folgenden gestellten Probleme lassen sich auch asymptotisch schneller als gefordert lösen. Schnellere Lösungen geben aber keine Bonuspunkte.

a. Sie haben die Tipps von allen Teilnehmern als Paare der Form $(\mathit{Name}, \mathit{Tipp})$ erhalten:

$$(\mathit{Ford}, 250), (\mathit{Cormen}, 700), (\mathit{Karatsuba}, 100), (\mathit{Dijkstra}, 500), (\mathit{Prim}, 800)$$

Sie ziehen $z = 500$, und es steht der Geldbetrag $G = 1024$ Euro zur Verfügung. Welche Auszahlung erhalten die Teilnehmer jeweils? [1 Punkt]

Lösung

Ford	Cormen	Karatsuba	Dijkstra	Prim
128 €	256 €	32 €	512 €	64 €

[1] Namen sind Strings, die nur aus Buchstaben eines endlichen Alphabets bestehen und maximal 100 Zeichen lang sind.

[2] Es ist richtig, dass so niemals der gesamte Betrag G ausgezahlt wird. Das sollte Sie nicht weiter stören.

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 3

b. Entwerfen Sie einen Algorithmus, der nachdem Sie die Zufallszahl z generiert haben, im Worst Case mit asymptotischer Zeitkomplexität $O(n \log n)$ die Auszahlung **für jeden Teilnehmer** berechnet, wobei n die Anzahl der Teilnehmer ist. Ihr Algorithmus erhält als Eingabe eine Liste von Paaren der Form

$$(Name, Tipp)$$

und soll eine Liste von Paaren der Form

$$(Name, Betrag)$$

ausgeben, wobei *Betrag* dem Auszahlungsbetrag an den entsprechenden Spieler entspricht. Die Reihenfolge der ausgegebenen Paare ist unerheblich.
Hinweis: Sie dürfen Algorithmen, die aus der Vorlesung bekannt sind, als implementiert annehmen und für Ihre Lösung verwenden. [3 Punkte]

Lösung

Sei die gezogene Zahl z und die Menge der abgebenen Tipps $T = \{(Name_1, k_1), (Name_2, k_2), \ldots (Name_n, k_n)\}$. Zunächst generieren wir daraus die Menge der Abstände von z als $T' = \{(Name_i, |k_i - z|) : i \in \{1, \ldots, n\}\}$.
Wir sortieren nun diese Menge z.B. mittels Mergesort aufsteigend nach dem zweiten Element der Paare. Damit entspricht das erste Paar dem Teilnehmer, der am nächsten an z lag, das zweite dem Zweitnächsten, usw. Wir berechnen nun den *Betrag* für den i-ten Eintrag in der Liste als $\frac{1}{2^i} \cdot G$ und ersetzen den zweiten Eintrag im Tupel jeweils durch den Betrag. Schließlich iterieren wir nochmal über die Liste und geben die entsprechenden $(Name, Betrag)$-Paare aus.

c. Begründen Sie, weshalb Ihr Algorithmus aus **b.** die geforderte Zeitkomplexität erreicht. [1 Punkt]

Lösung

Initial müssen wir nur einmal alle Tupel iterieren, was in $O(n)$ vonstatten geht. Im zweiten Schritt benötigt Mergesort $O(n \log n)$ Zeit, um alle Tupel zu sortieren. Abschließend muss noch zweimal in $O(n)$ iteriert werden. Insgesamt ergibt sich also $O(n + n \log n + 2n) = O(n \log n)$ Zeitkomplexität.
Hinweis: Mittels ganzzahligem Sortieren nach den Abständen hätten sich hier noch schnellere Algorithmen entwerfen lassen - das war hier aber nicht gefordert.

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 3

d. Nun veranstalten Sie nicht eine solche Verlosung, sondern k Verlosungen parallel. Nicht jeder Teilnehmer nimmt an jeder Verlosung teil, und die Verlosungen sind vollkommen voneinander unabhängig, d.h. pro Verlosung gibt eine Teilmenge aller Teilnehmer jeweils eine Zahl ab, und Sie ziehen auch eine Zufallszahl pro Verlosung. Ein Teilnehmer, der an einer Verlosung nicht teilnimmt, erhält für diese auch keine Auszahlung. Für die i-te Verlosung liegen die Tipps nun als Liste von Tripeln der Form

$$(i, Name, Tipp)$$

vor. Erweitern Sie Ihren Algorithmus so, dass er je die **Gesamtauszahlung für jeden Teilnehmer** in erwarteter Zeitkomplexität $O(kn \log n)$ berechnet.[3] Ihr Algorithmus soll schließlich eine Liste von Paaren der Form

$$(Name, Gesamtbetrag)$$

ausgeben, wobei der Gesamtbetrag hier der Summe der Beträge aller Auszahlungen an die entsprechende Person entspricht. Die Reihenfolge der ausgegebenen Paare ist unerheblich.
Hinweis: Sie dürfen Algorithmen, die aus der Vorlesung bekannt sind, als implementiert annehmen und für Ihre Lösung verwenden. [4 Punkte]

Lösung

Zunächst müssen die Tripel nach Verlosungen sortiert werden. Dazu bietet sich zum Beispiel Bucket Sort (mit Buckets 1 bis k) nach der 1ten Komponente an. Anschließend führen wir für jede der k Verlosungen die in Lösungsteil **b.** beschriebene Prozedur durch, ohne jedoch die entsprechenden Tupel auszugeben. Stattdessen werden die ansonsten ausgegebenen Paare in k Listen $R_1, \ldots R_k$ gespeichert.
Aus diesen Listen müssen nun noch die Gesamtauszahlungen berechnet werden. Dazu gehen wir wie folgt vor: Wir initialisieren eine Hashtabelle mit n Einträgen und ziehen aus einer universellen Familie eine Hashfunktion, um die Namen der Teilnehmer in den Bereich $[1, n]$ hashen zu können. (Da die Länge der Namen beschränkt ist, ist die Auswertung der Hashfunktion in konstanter Zeit möglich.) Nun iterieren wir alle Paare in den Listen $R_1, \ldots R_k$. Für jedes Paar $(Name, Betrag)$ prüfen wir dabei, ob für den Namen bereits ein Wert in der Hashtabelle eingetragen ist. Wenn ja, dann addieren wir die aktuelle Auszahlung zu diesem Wert in der Hashtabelle hinzu, andernfalls fügen wir $(Name, Betrag)$ in die Hashtabelle ein. Nachdem wir über alle Paare iteriert haben, steht in der Hashtabelle für jeden Namen die Gesamtauszahlung. Wir iterieren einmal abschließend alle Einträge in der Hashtabelle und geben die entsprechenden $(Name, Gesamtbetrag)$-Paare aus.

e. Begründen Sie, weshalb Ihr Algorithmus aus **d.** die geforderte Zeitkomplexität erreicht. [2 Punkte]

Lösung

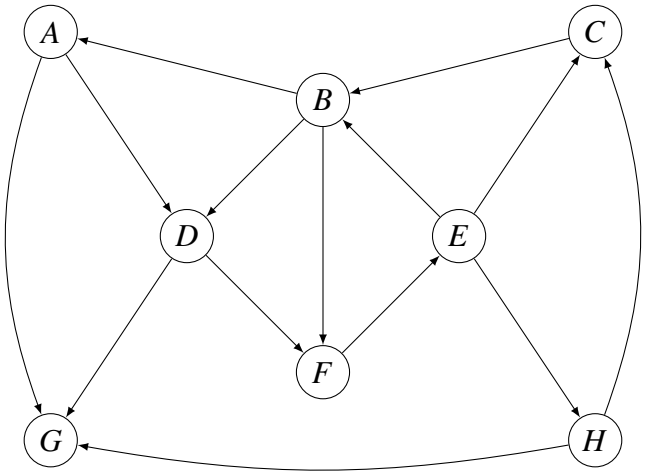
[3] Zur Erinnerung: n ist die Anzahl der Teilnehmer, k die Anzahl der parallelen Verlosungen.

Bucket Sort für k Buckets und $O(kn)$ Teilnehmer hat Laufzeit $O(kn)$. Da in Lösungsteil **b.** für eine Verlosung mit n Teilnehmern die Zeitkomplexität von $O(n \log n)$ gezeigt wurde, lassen sich die Ergebnisse der k Verlosungen in $O(k \cdot n \log n)$ berechnen.
Wir haben ursprünglich insgesamt kn Paare, die wir iterieren. Für jedes dieser Paare müssen wir bis zu zwei Zugriffe (lesen und schreiben) auf die Hashtabelle durchführen, die Dank universellem Hashing ihrerseits erwartet in $O(1)$ liegen. Dieser Schritt ist also erwartet in $O(kn)$. Insgesamt ist der Algorithmus erwartet in $O(kn + kn \log n + kn) = O(kn \log n)$.

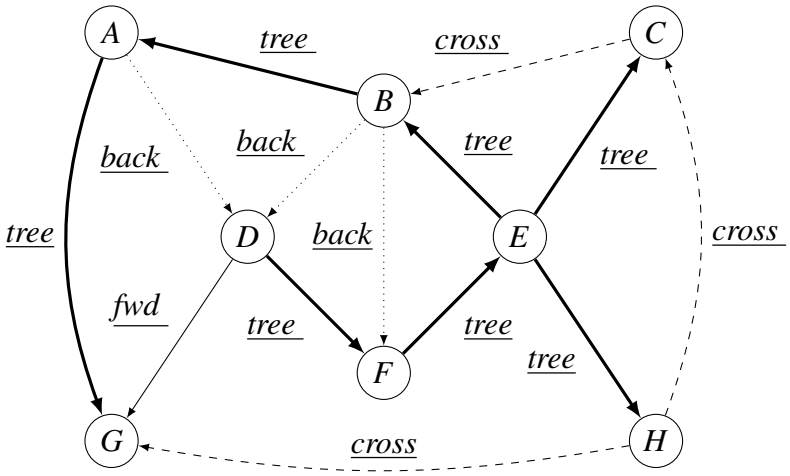
Lösungsvorschlag

Aufgabe 4. Pfade in Graphen [15 Punkte]

Gegeben sei der folgende ungewichtete, *gerichtete* Graph G:



Weiter sei der folgende Graph G' gegeben, in dem Vorwärts- (*fwd*), Rückwärts- (*back*), Kreuz (*cross*) und Baumkanten (*tree*) eingezeichnet sind:



a. Ist der Graph G' das mögliche Ergebnis einer Tiefensuche auf G? Falls ja, geben Sie den jeweiligen Startknoten an, und eine mögliche Reihenfolge der Knoten, in der sie zum ersten Mal vom Suchalgorithmus betrachtet wurden. Falls nein, begründen Sie Ihre Antwort. [2 Punkte]

Lösung

Ja, mit Reihenfolge: D, F, E, B, A, G, C, H.

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

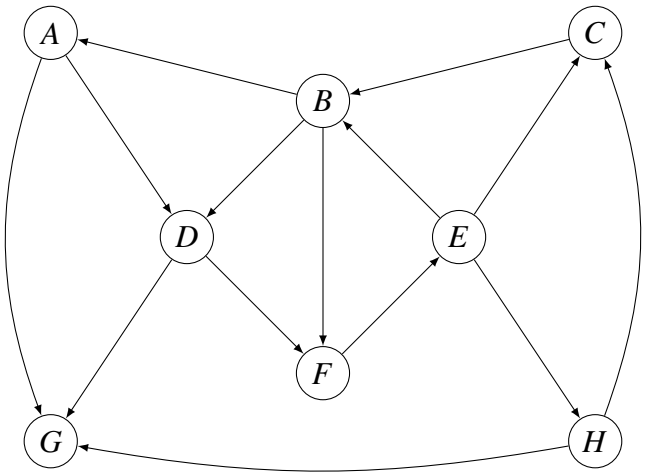
Fortsetzung von Aufgabe 4

b. Ist der Graph G' das mögliche Ergebnis einer Breitensuche auf G? Falls ja, geben Sie den jeweiligen Startknoten an, und eine mögliche Reihenfolge der Knoten, in der sie zum ersten Mal vom Suchalgorithmus betrachtet wurden. Falls nein, begründen Sie Ihre Antwort. [2 Punkte]

Lösung

Nein, denn bei einer Breitensuche können keine Vorwärtskanten auftreten.

c. Geben Sie die Repräsentation von G als Adjazenzfeld an. Sie können dafür die Vorlage verwenden.
Zur Erinnerung ist hier noch einmal der Graph abgedruckt: [2 Punkte]



Lösung

Adjazenzfeld:

A	B	C	D	E	F	G	H	
1	3	6	7	9	12	13	13	15

1	2	3	4	5	6	7	8	9	10	11	12	13	14
D	G	A	D	F	B	F	G	B	C	H	E	C	G

d. Nennen Sie einen Vorteil und einen Nachteil von Adjazenzfeldern gegenüber Adjazenzlisten. [1 Punkt]

Lösung

Vorteil von Adjazenzfeldern:	weniger Platz als Adjazenzlisten, weniger Cache-Misses
Nachteil von Adjazenzfeldern:	Einfügen und Löschen von Kanten einfacher

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 4

e. Es geht nun darum, in einem beliebigen ungewichteten, *ungerichteten* Graphen $G = (V, E)$ Pfade zu finden, die aber explizit **nicht unbedingt kürzeste** Pfade sein müssen. Zusätzlich zum Graphen G sei in dieser Aufgabe mit l auch die Länge[4] des längsten einfachen Pfades in G als Eingabe gegeben. Ein *einfacher* Pfad ist ein Pfad, der jeden Knoten maximal einmal enthält. Ihre Aufgabe ist es, einen Algorithmus zu entwickeln, der für Knotenpaare $\{u, v\}$ *irgendeinen* Pfad zwischen u und v findet. Ihr Algorithmus soll dabei in zwei Phasen vorgehen: In der *Vorberechnungsphase* darf Ihr Algorithmus (innerhalb der unten gegebenen Zeit- und Platzschranken) Daten vorberechnen. In dieser Phase ist dem Algorithmus der Graph bekannt, **nicht aber die später folgenden Anfragen**. In der auf die Vorberechnungsphase folgenden *Anfragephase* bekommt Ihr Algorithmus dann eine Reihe von Anfragen, d.h. von Knotenpaaren $\{u, v\}$, und muss (wieder innerhalb der unten gegebenen Schranken) einen Pfad zwischen beiden Knoten finden. Die Nebenbedingungen für die beiden Phasen lauten:

- In der **Vorberechnungsphase** darf Ihr Algorithmus maximal $O(|V| + |E|)$ Zeit und maximal $O(|V|)$ zusätzlichen Speicher verwenden. *Erinnerung:* In dieser Phase sind die Paare $\{u, v\}$, die später angefragt werden, noch nicht bekannt!
- In der **Anfragephase** muss Ihr Algorithmus dann für jedes angefragte Paar $\{u, v\}$ in Zeit $O(l)$ einen *einfachen* Pfad ausgeben, oder aber ausgeben, dass kein Pfad zwischen u und v existiert.

Geben Sie einen solchen Algorithmus an.
Hinweis: Sie dürfen Algorithmen, die aus der Vorlesung bekannt sind, als implementiert annehmen und für Ihre Lösung verwenden. [6 Punkte]

Lösung

Vorberechnung: Wir berechnen mittels wiederholter DFS einen Spannwald auf G. Wir speichern für jeden Knoten einen Zeiger auf seinen Elternknoten im Spannwald sowie die Tiefe des Knotens, d.h. die Länge des Baum-Pfades von der Wurzel zu diesem Knoten.
Anfrage: Seien u und v gegeben. Seien $u.parent$ und $v.parent$ die Elternknoten im berechneten Spannbaum, und seien $u.depth$ und $v.depth$ die gespeicherte Baumtiefe. Wir führen aus:

1: $p1, p2$: Lists of Vertices
2: **while** $u \neq v$ **do**
3: **if** $u.depth > v.depth$ **then**
4: $p1.push\_back(u)$
5: $u = u.parent$
6: **else**
7: $p2.push\_back(v)$
8: $v = v.parent$
9: **endif**
10: **done**
11: Gebe $p1$, u und $reverse(p2)$ aus

[4] Das heißt die Anzahl der Kanten auf dem Pfad.

Wir laufen also iterativ immer entweder von u oder von v aus einen Schritt weiter den Baum hinauf, bis beide Läufe sich treffen. Dabei laufen wir immer von dem Knoten aus, der weiter von der Wurzel entfernt ist. Auf diese Art und Weise stellen wir sicher, dass die Suche tatsächlich beim ersten Knoten, den u und v auf ihren Wegen zur Wurzel gemeinsam haben, abbricht. Andernfalls würde kein *einfacher* Pfad ausgegeben.

f. Begründen Sie, warum Ihr Algorithmus aus **e.** die geforderte Zeitkomplexität erreicht. [2 Punkte]

Lösung

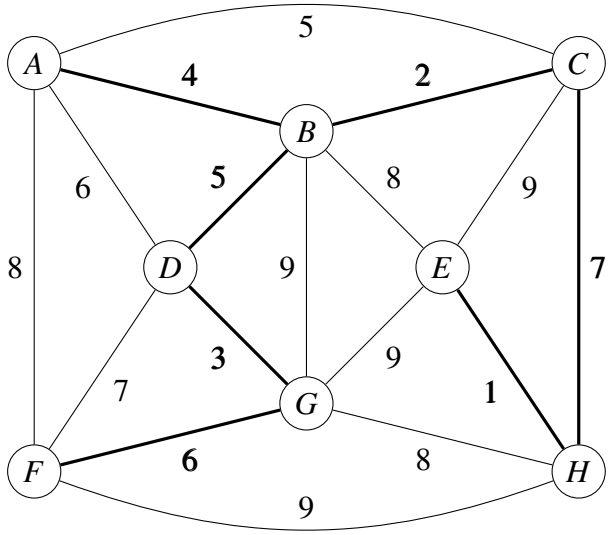
Vorberechnung: Wir brauchen $O(|V|)$ Speicher, die DFS besucht jede Kanten und jeden Knoten maximal einmal und läuft somit in $O(|V|+|E|)$ Zeitkomplexität.
Anfrage: Die Schleife wird insgesamt maximal so oft durchlaufen, wie es Knoten auf dem Weg von v und u zur Wurzel gibt. Diese Zahl ist durch l beschränkt, also wird die Schleife insgesamt maximal $2l$ mal durchlaufen. Die Operationen innerhalb des Schleifenrumpfes laufen alle in $O(1)$, somit ergibt sich eine gesamte Zeitkomplexität von $O(l)$ für die Schleife. Der auszugebende Pfad ist ebenfalls durch l beschränkt, und somit lässt er sich auch in $O(l)$ ausgeben (und invertieren).

Lösungsvorschlag

Aufgabe 5. Minimale Spannbäume [8 Punkte]

a. Berechnen Sie einen minimalen Spannbaum des angegebenen Graphen mit dem Algorithmus von Jarník-Prim. Geben Sie jeweils die Kanten des minimalen Spannbaumes in der Reihenfolge an, in der sie der Algorithmus auswählt. Für Knoten V, W geben Sie die verbindende Kante als $\{V, W\}$ oder (V, W) an. Verwenden Sie den Knoten A als Startknoten. [3 Punkte]

Lösung



Kantenreihenfolge: $\{A,B\}, \{B,C\}, \{B,D\}, \{D,G\}, \{G,F\}, \{C,H\}, \{H,E\}$

b. Nennen und erklären Sie die Eigenschaft, auf der die Korrektheit des Jarník-Prim Algorithmus beruht. [1 Punkt]

Lösung

Schnitteigenschaft (Cut Property): Sei $S \subseteq V$ ein Schnitt und $C = \{\{u,v\} \mid u \in S, v \in V \setminus S\}$. Die leichteste Kante in C kann in einem minimalen Spannbaum verwendet werden.

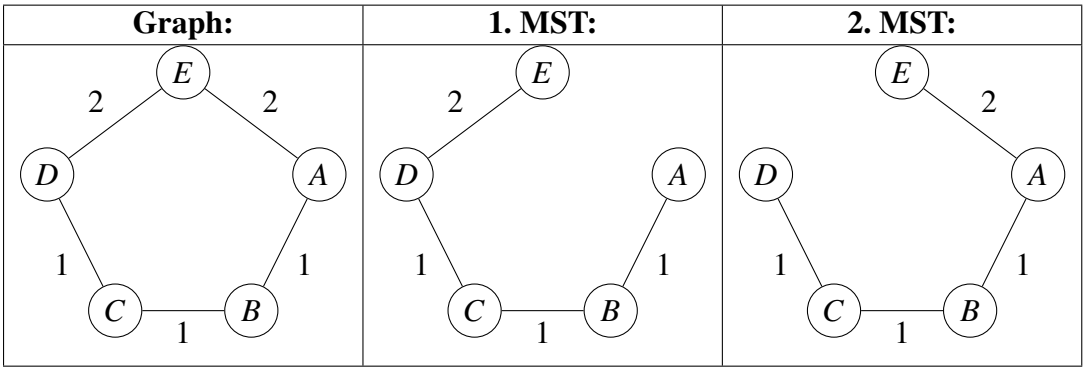
(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 5

c. Ergänzen Sie im folgenden Graphen ungerichtete Kanten mit positiven Kantengewichten, so dass ein zusammenhängender Graph entsteht, der **genau zwei** verschiedene minimale Spannbäume enthält. Geben Sie links den Graphen und rechts jeweils die minimalen Spannbäume an. Falls Sie einen Fehler machen, können Sie die unteren Vorlagen verwenden. Kennzeichnen Sie *deutlich*, welche Lösung die zu wertende ist, indem Sie die andere durchstreichen. [2 Punkte]

Lösung



d. Gegeben sei ein ungerichteter zusammenhängender zyklenfreier Graph $G = (V, E)$ mit positiven Kantengewichten und $n = |V|$ Knoten. *Zyklenfrei* bedeutet, dass der Graph keinen Zyklus, also keinen Pfad der Länge > 1 mit gleichem Start- und Endknoten, enthält.

d.1 Wieviele Kanten $|E|$ hat der Graph G in Abhängigkeit von n?

d.2 Ist der minimale Spannbaum von G eindeutig?

Begründen Sie Ihre Antworten jeweils kurz. [2 Punkte]

Lösung

Der Graph G enthält genau $n-1$ Kanten. Er enthält mindestens $n-1$ Kanten, da er sonst nicht zusammenhängend wäre. Er enthält höchstens $n-1$ Kanten, da er sonst einen Kreis enthalten würde. Insbesondere ist G ein Baum, also entspricht der minimale Spannbaum gerade G und ist somit eindeutig.