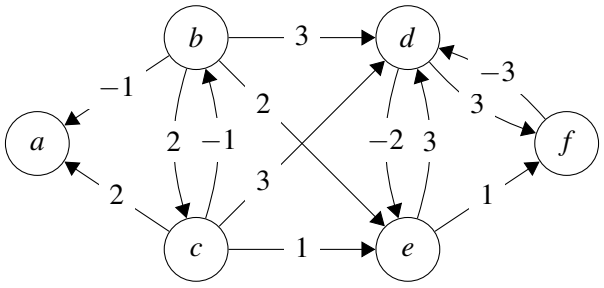
Name: Matrikelnummer:

Klausur Algorithmen I, 14.3.2011 Blatt 1 von 2?

Lösungsvorschlag

Aufgabe 1. Kleinaufgaben [20 Punkte]

a. Enthält folgender gerichteter gewichteter Graph einen kürzesten Weg von b nach f?



Falls ja, geben Sie einen solchen kürzesten Weg an. Falls nein, begründen Sie kurz, warum kein solcher kürzester Weg existiert. [2 Punkte]

Lösung

Es gibt **keinen** kürzesten Weg von b nach f. **Begründung:** Es gibt einen Weg von b nach f der einen negativen Kreis enthält, nämlich $b \to e \to f \to d \to e \to f$.

Lösungsende

b. Betrachten Sie noch einmal den gerichteten gewichteten Graph aus Teilaufgabe **a.** Enthält er einen kürzesten Weg von c nach a? Falls ja, geben Sie einen solchen kürzesten Weg an. Falls nein, begründen Sie kurz, warum kein solcher kürzester Weg existiert. [1 Punkt]

Lösung

Es gibt einen kürzesten Weg von c nach a, nämlich $c \to b \to a$.

Lösungsende

c. Sei A eine $(n \times n)$-Matrix über $\mathbb{R}$ und $n, m \in \mathbb{N}_{>0}$. Kann A^m mit höchstens $O(\log m)$ Matrixmultiplikationen berechnet werden? Begründen Sie kurz. [2 Punkte]

Lösung

Ja, das geht. Man verwende dazu den Algorithmus zur schnelle Potenzierung von Zahlen aus der Vorlesung und wende diesen auf Matrizen an.

Lösungsende

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

d. Zeigen Sie oder widerlegen Sie, dass $5^{\log_3 n} = O(n^2)$ gilt. [2 Punkte]

Lösung

Die Behauptung gilt. Es ist nämlich $5^{\log_3 n} = 5^{\frac{\log_5 n}{\log_5 3}} = \left(5^{\log_5 n}\right)^{1/\log_5 3} = n^{\log_3 5}$. Weiter gilt $\log_3 5 < 2$, woraus die Behauptung folgt.

Lösungsende

e. Zeigen Sie oder widerlegen Sie, dass $2^n = \Omega(2^{n/2})$ gilt. [2 Punkte]

Lösung

Die Behauptung gilt. Es ist nämlich $2^n \geq \sqrt{2}^n = 2^{n/2}$ für alle $n \geq 1$. **Bemerkung:** Man wählt also das c aus der Definition der Ω-Notation als 1.

Lösungsende

f. Gegeben seien zwei rekursive Algorithmen A und B, wobei B in jedem Rekursionsschritt nicht nur sich selbst sondern auch A aufruft. Die Laufzeiten von A und B seien durch die Rekurrenzen

$$\begin{aligned} T_A(n) &= 5n + aT_A(n/3), \qquad T_A(1) = 4 \quad \text{bzw.} \\ T_B(n) &= T_A(n) + bT_B(n/3), \quad T_B(1) = 2 \end{aligned}$$

beschrieben mit $n = 3^k$ und $k \in \mathbb{N}_{>0}$. Geben Sie Werte für a und b aus $\mathbb{N}_{>0}$ an, so dass die Laufzeit von B in $\Theta(n \log n)$ liegt für $n = 3^k$. Begründen Sie Ihre Wahl von a und b kurz. [3 Punkte]

Lösung

$a = 2$ und $b = 3$. Dann braucht A nach Master-Theorem nämlich $\Theta(n)$ Zeit, also $T_B(n) = \Theta(n) + 3T(n/3)$ für $n > 1$. D.h. B braucht dann $\Theta(n \log n)$, wiederum nach Master-Theorem.

Lösungsende

g. Geben Sie in Θ-Notation an, welches Worst-Case-Laufzeitverhalten ein vergleichsbasierter Sortieralgorithmus bestenfalls haben kann. Nennen Sie ein Beispiel für einen vergleichsbasierten Sortieralgorithmus, der in diesem Sinne optimal und zusätzlich stabil ist. [2 Punkte]

Lösung

$\Theta(n \log n)$ Zeit für Listen der Länge n. Beispiel für Algorithmus: Mergesort

Lösungsende

(weitere Teilaufgaben auf dem nächsten Blatt)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

h. Gegeben seien n Bausteine mit Höhen $h_1, \ldots, h_n \in \mathbb{N}_{>0}$. Die Bausteine sollen auf zwei Stapel verteilt werden. Ziel ist es, die Gesamthöhe des höheren Stapels zu minimieren.

Dieses Problem soll optimal gelöst werden. Dazu schlägt ein Student folgenden Ansatz vor:

> Berechne die Lösung für die ersten $k+1$ Steine rekursiv:
>
> - Berechne die Lösung für die ersten k Steine. Falls $k = 0$ starte mit zwei leeren Stapeln.
> - Platziere den $(k+1)$-ten Stein auf dem kleineren Stapel, oder auf dem ersten Stapel falls beide gleich hoch sind.

Liefert dieser Ansatz immer eine optimale Lösung? Wenn ja, begründen Sie kurz. Wenn nein, geben Sie ein Gegenbeispiel an. [3 Punkte]

Lösung

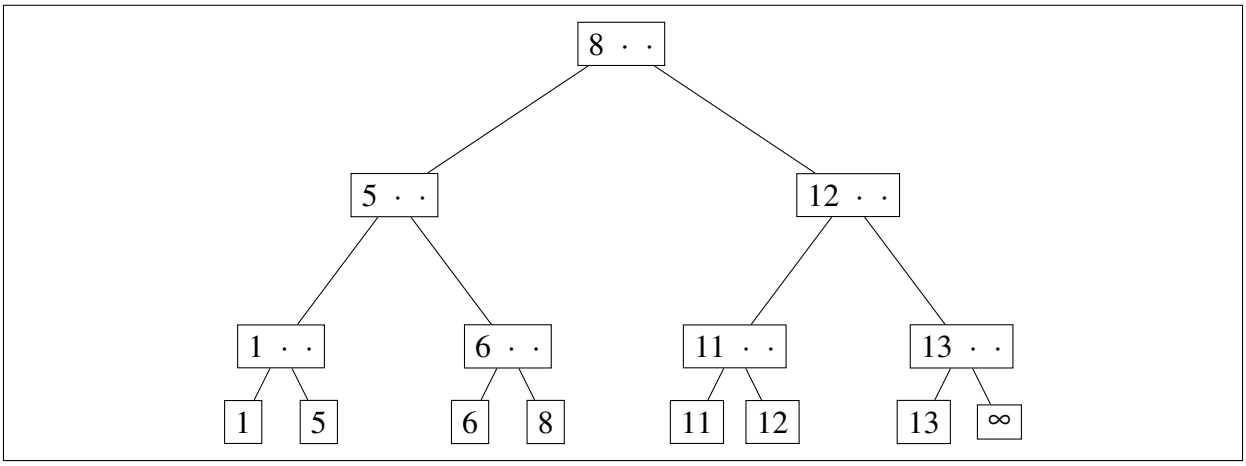
Nein, dieser Ansatz liefert nicht immer eine optimale Lösung. Betrachte das Problem mit drei Steinen der Größen $1, 1, 2$ (in der Reihenfolge). Der vorgeschlagene Ansatz legt den ersten Stein auf Stapel 1, den zweiten Stein auf Stapel 2 und den dritten Stein wieder auf Stapel 1. Damit hat der höhere Stapel die Höhe 3. Es wäre aber mit den ersten beiden Steinen auf Stapel 1 und dem dritten Stein auf Stapel 2 auch die Höhe 2 für den höheren Stapel möglich gewesen. Damit ist gezeigt, dass mit diesem Ansatz nicht immer die Optimallösung gefunden wird.

Lösungsende

i. Gegeben sei die sortierte Sequenz $1, 5, 6, 8, 11, 12, 13$. Geben Sie für diese Sequenz einen zulässigen $(2,4)$-Baum an, der eine maximale Anzahl innere Knoten enthält. Vervollständigen Sie dazu die unten angegebene Zeichnung – die Blätter des $(2,4)$-Baumes sind bereits eingezeichnet. [3 Punkte]

1	5	6	8	11	12	13	∞

Lösung



Lösungsende

Lösungsvorschlag

Aufgabe 2. Graphrepräsentationen [7 Punkte]

Gegeben sei ein Graph $G = (V,E)$ mit $V := \{1,2,\ldots,8\}$. Die Kanten seien durch die Adjazenz-Matrix $A \in \{0,1\}^{8\times 8}$ beschrieben:

$$A := \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

a. Geben Sie den Graphen dargestellt als Adjazenzfeld an. Benutzen Sie dafür die vorgegebene Zeichnung. [3 Punkte]

Tragen Sie **hier** die **Lösung** ein:

	1	2	3	4	5	6	7	8	9
V									

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E															

Kopie der obigen Zeichnung zum **Rechnen:**

	1	2	3	4	5	6	7	8	9
V									

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E															

Lösung

Eine mögliche Lösung:

V	1	4	6	8	9	11	14	15	16

E	2	4	7	3	8	4	8	7	1	7	2	5	8	8	2

Lösungsende

(Teilaufgaben **b.** und **c.** auf dem nächsten Blatt)

Lösungsvorschlag

Fortsetzung von Aufgabe 2

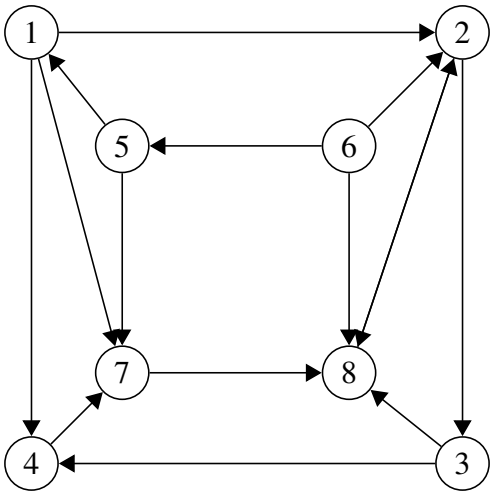
Kopie der Adjazenz-Matrix vom **vorherigen** Blatt:

$$A := \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

b. Zeichnen Sie den Graphen. [2 Punkte]

Lösung

Eine mögliche Darstellung des Graphen ist



Lösungsende

c. Handelt es sich bei dem Graphen um einen DAG? Begründen Sie kurz. [2 Punkte]

Lösung

Es handelt sich bei dem Graphen nicht um einen DAG, da der Graph einen Kreis enthält ($2 \to 8 \to 2$).

Lösungsende

Lösungsvorschlag

Aufgabe 3. Verkettete Listen [8 Punkte]

Betrachten Sie eine Realisierung zyklischer einfach verketteter Listen, bei der die Listenglieder und die Liste durch die folgenden zwei Klassen beschrieben sind:

```
1: class ListItem of Element
2:     e : Element
3:     next : Handle
4: end class
```

```
1: class List of Element
2:     head := (⊥, address of head) : ListItem of Element
3: end class
```

Die leere Liste wird dabei dargestellt, indem das Dummyglied *head* auf sich selbst verweist, also $head.next = head$.

Ziel dieser Aufgabe ist es einen **iterativen** Algorithmus anzugeben, der in $O(n)$ Zeit aus einer gegebenen Liste $\langle e_1, e_2, \ldots, e_n \rangle$ die Liste $\langle e_n, e_{n-1}, \ldots, e_1 \rangle$ erzeugt, die Liste also umdreht. Es soll aber **keine** neue Liste erzeugt werden. Vielmehr soll die Eingabeliste selbst umgedreht werden.

a. Geben Sie zunächst eine aussagekräftige Schleifeninvariante an, aus der die Korrektheit Ihres Algorithmus folgt. [3 Punkte]

Lösung

Sei c die Anzahl der Schleifendurchläufe, i verweise auf das aktuell betrachtete Listenelement. Die Invariante lautet dann:

> *Die Liste von i bis (ausschließlich) head repräsentiert die ersten c Elemente der Eingabeliste in umgedrehter Reihenfolge; j verweist auf den Nachfolger des c-ten Elementes der Orginalliste.*

Lösungsende

b. Geben Sie nun – basierend auf Ihrer Invariante aus Teilaufgabe **a.** – Pseudocode für eine Methode *umdrehen* der Klasse *List* an.[1] [5 Punkte]

Lösung

```
 1: method umdrehen
 2:     i := address of head : Handle
 3:     j := head.next : Handle
 4:     while j ≠ head do
 5:         k := j.next : Handle
 6:         j.next := i
 7:         i := j
 8:         j := k
 9:     end while
10: head.next = i
11: end
```

Lösungsende

[1] Grundlage für die Bewertung sind nicht syntaktische Details sondern der iterative Algorithmus.

Lösungsvorschlag

Aufgabe 4. Urlaubsdatenbank [9 Punkte]

In einem Unternehmen habe jeder Mitarbeiter eine eindeutige ID aus $\mathbb{N}$. Gegeben sei eine Datei D (d.h. eine Folge) von n Paaren der Form $(\mathit{MitarbeiterID}, \mathit{UrlaubsTag}) \in \mathbb{N} \times \{1,\ldots,365\}$. Die Datei D beschreibt für ein bestimmtes Jahr, an welchen Tagen die Mitarbeiter Urlaub genommen haben: Für jeden Tag, an dem ein Mitarbeiter in dem Jahr Urlaub genommen hat, enthält D genau ein entsprechendes Paar.

Hinweis: Beachten Sie, dass die IDs der Mitarbeiter beliebig groß sein können.

a. Geben Sie einen Algorithmus an, der in **erwartet** $O(n)$ Zeit die IDs aller Mitarbeiter in D ausgibt, die bereits in der ersten Hälfte des Jahres mindestens 30 Tage Urlaub genommen haben. Jede ID darf dabei nicht mehr als einmal ausgegeben werden. [4 Punkte]

Lösung

- Durchlaufe D und zähle dabei die Anzahl der Paare in D, was n liefert.
- Lege eine Hashtabelle $H : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ an, die verkettete Listen zur Kollisionsauflösung verwendet und $2n$ Slots hat. Wähle dazu zufällig eine Hashfunktion aus einer universellen Familie. **Idee:** H bildet die Mitarbeiter-IDs auf Zähler ab.
- Durchlaufe D erneut. Für jedes dabei betrachtete Paar $(x,d) \in \mathbb{N} \times \{1,\ldots,365\}$ mit $d < 365/2$ schaue in Hashtabelle H, ob es bereits einen Eintrag für Schlüssel x gibt. Falls nein, füge $(x,1)$ in H ein. Falls ja, erhöhe in Eintrag (x,c) das c um eins.
- Durchlaufe alle Einträge in H. Für jeden Eintrag (x,c) mit $c \geq 30$ gib x aus.

Lösungsende

b. Begründen Sie kurz, warum Ihr Algorithmus aus Teilaufgabe **a.** das gewünschte Laufzeitverhalten aufweist. [2 Punkte]

Lösung

- Alle Durchläufe von D und das Initialisieren der Hashtabelle H erfordern jeweils $O(n)$ Zeit.
- Nachschauen in H, ob für gegebenes x ein Eintrag vorhanden ist, kostet jeweils **erwartet** $O(1)$ Zeit, was ja insgesamt n-mal passiert, also zusammen **erwartet** $O(n)$ Zeit.
- Insgesamt wird höchstens n-mal in H eingefügt, was jeweils $O(1)$ Zeit kostet.
- Erhöhen eines Zählers kostet jeweils $O(1)$ Zeit. Dies findet höchstens n-mal statt.

Insgesamt wird erwartet $O(n)$ Zeit benötigt.

Lösungsende

(Teilaufgaben **c.** und **d.** auf dem nächsten Blatt)

Lösungsvorschlag

Fortsetzung von Aufgabe 4

c. Geben Sie einen Algorithmus an, der an Hand von D in **deterministisch** $O(n)$ Zeit einen Tag des Jahres ermittelt, an dem eine maximale Anzahl Mitarbeiter Urlaub genommen hat. [2 Punkte]

Lösung

- Alloziere ein Array $A[1..365]$ von Zählern, d.h. von ganzen Zahlen, die mit 0 initialisiert werden.
- Durchlaufe D und erhöhe für jedes Paar (x,d) den Zähler $A[d]$ um eins.
- Durchlaufe A und behalte jeweils den Index $d_{\max}$ des größten bisher gesehenen Wertes $A[d_{\max}]$.
- Gib $d_{\max}$ aus.

Lösungsende

d. Begründen Sie kurz, warum Ihr Algorithmus aus Teilaufgabe **c.** das gewünschte Laufzeitverhalten aufweist. [1 Punkte]

Lösung

- Allozieren und initialisieren von A kostet $O(1)$ Zeit, da 365 nicht von Eingabe abhängt.
- Durchlaufen von D kostet $O(n)$ Zeit.
- Durchlaufen von A kostet wieder $O(1)$ Zeit.

Insgesamt wird $O(n)$ Zeit benötigt.

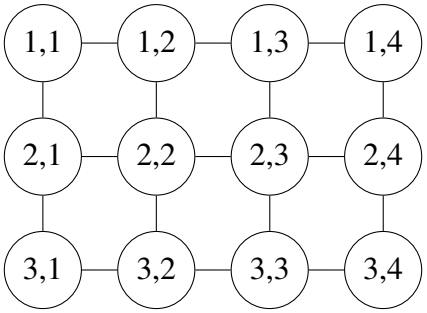
Lösungsende

Lösungsvorschlag

Aufgabe 5. Kürzeste Wege [7 Punkte]

a. Gegeben sei ein regelmäßiges $(a \times b)$-Gitter, bei dem zwischen direkt nebeneinander und direkt übereinander liegenden Knoten je eine horizontale bzw. vertikale Kante existiert. Alle diese Kanten sind ungerichtet und haben das Gewicht 1.

Beispiel: 3×4 Gitter:



Der Knoten an i-ter Stelle von oben und j-ter Stelle von links heißt (i, j) wie im Beispiel.

Geben Sie einen Algorithmus an, der einen kürzesten Weg von einem Knoten (i_1, j_1) zu einem Knoten (i_2, j_2) berechnet und ausgibt. Ihr Algorithmus soll eine Laufzeit in O(Weglänge) besitzen. [3 Punkte]

Lösung

Ein kürzester Weg muss nicht gesucht werden sondern kann explizit angegeben werden. Sei $\Delta i = 1$ falls $i_2 \geq i_1$, sonst $\Delta i = -1$ und sei $\Delta j = 1$ falls $j_2 \geq j_1$, sonst $\Delta j = -1$. Ein kürzester Weg ist $\langle (i_1, j_1), (i_1 + \Delta i, j_1), (i_1 + 2\Delta i, j_1), \ldots, (i_2, j_1), (i_2, j_1 + \Delta j), (i_2, j_1 + 2\Delta j), \ldots, (i_2, j_2) \rangle$. Dieser kann direkt mittels zwei Schleifen ausgegeben werden.

Lösungsende

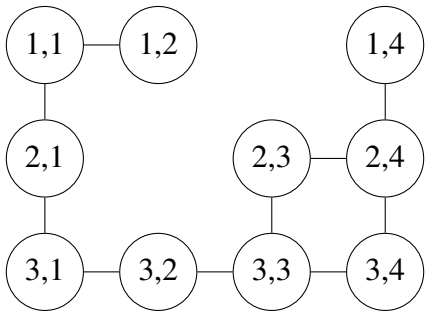
(Teilaufgabe **b.** auf dem nächsten Blatt)

Lösungsvorschlag

Fortsetzung von Aufgabe 5

b. Gegeben sei ein regelmäßiges $(a \times b)$-Gitter wie in Teilaufgabe **a.**, bei dem nun aber einzelne Knoten fehlen.

Beispiel: 3×4 Gitter mit fehlenden Knoten $(2,2)$ und $(1,3)$:



Beschreiben Sie einen Algorithmus, der einen kürzesten Weg von einem existierenden Knoten (i_1, j_1) zu einem existierenden Knoten (i_2, j_2) berechnet und ausgibt. Falls kein Weg zwischen diesen Knoten existiert soll der Algorithmus "nicht verbunden" ausgeben.

Das Gitter sei in Form eines Adjazenzfeldes gegeben. Ihr Algorithmus soll eine Laufzeit in $O(n)$ besitzen (n := Anzahl existierender Knoten). [4 Punkte]

Lösung

Die Idee ist, die Laufzeit durch eine Breitensuche zu erreichen, die beim Knoten (i_1, j_1) startet. Dabei beachte man, dass die Breitensuche ein Array *parent* der Größe n verwaltet, in dem für jeden Knoten gespeichert wird, von welchem Knoten aus er in der Breitensuche gefunden wurde. Da das Gitter als Adjazenzfeld gegeben ist, ist für jeden Knoten eine Nummer vorhanden, die als Index dienen kann. Die Breitensuche auf einem Graphen mit einheitlichem Kantengewicht garantiert, dass man jeden Knoten über den kürzesten Weg von (i_1, j_1) findet. Die Breitensuche hat einen Aufwand in $O(n)$.
Wenn man (i_2, j_2) gefunden hat, kann man in O(Pfadlänge) Zeit mit Hilfe von *parent* den Pfad rückwärts in ein Array der Größe n schreiben. Nun muss nur noch die Umkehrung der Pfadbeschreibung in O(Pfadlänge) ausgegeben werden.
Falls die Breitensuche endet, ohne dass man (i_2, j_2) gefunden hat, so gibt man "nicht verbunden" aus.

Lösungsende

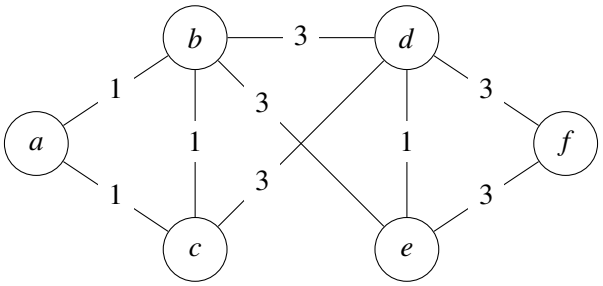
Lösungsvorschlag

Aufgabe 6. Minimale Spannbäume [9 Punkte]

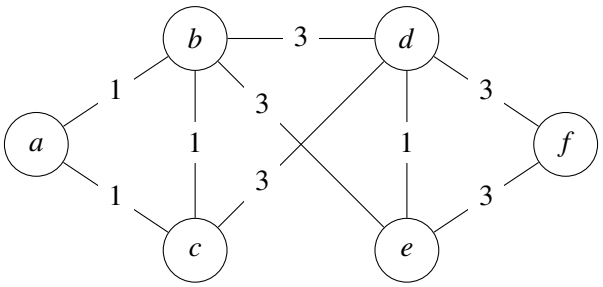
a. Gegeben sei der abgebildete ungerichtete Graph mit Kantengewichten aus $\{1,3\}$. Auf diesem Graph soll der Jarnik-Prim Algorithmus beginnend beim Knoten c ausgeführt werden.

Markieren Sie im Graph die Kanten eines möglichen resultierenden MST. Nummerieren Sie die markierten Kanten außerdem in einer möglichen Reihenfolge, in der diese Kanten vom Jarnik-Prim Algorithmus in den MST aufgenommen werden. [2 Punkte]

Tragen Sie die **Lösung** bitte in **diesen** Graphen ein:



Kopie des obigen Graphen zum **Rechnen:**



Lösung

Kanten eines möglichen MST in möglicher Aufnahmereihenfolge:
$\{a,c\},\{b,c\},\{c,d\},\{d,e\},\{d,f\}$

Lösungsende

(Teilaufgaben **b.** und **c.** auf dem nächsten Blatt)

Lösungsvorschlag

Fortsetzung von Aufgabe 6

b. Wir betrachten nun beliebige zusammenhängende ungerichtete Graphen $G = (V,E)$ mit $V = \{1,\ldots,n\}$ und Kantengewichten aus $\{1,3\}$. Sei G in Form eines **Adjazenzfeldes** gegeben. Geben Sie einen Algorithmus an, der in $O(|E|)$ Zeit einen MST von G berechnet. [5 Punkte]

Lösung

Sei $c : E \to \{1,3\}$ die Gewichtsfunktion von G. Man modifiziere den Jarnik-Prim Algorithmus nun wie folgt:

- Man ersetze die Priority-Queue (d.h. den binären Heap) Q durch zwei FIFO-Queues Q_1 und Q_3. Diese sollen als doppelt verkettete Listen realisiert sein, so dass Elemente in $O(1)$ Zeit auch **mitten** aus der FIFO-Queue entfernt werden können.
- Man lässt $Q.insert(s,0)$ weg, stattdessen durchläuft man alle Kanten $(s,v) \in E$. Für $c(s,v) = 1$ führe $Q_1.pushBack(v)$ aus, sonst führe $Q_3.pushBack(v)$ aus.
- Statt $deleteMin()$ führt man $Q_1.popFront()$ aus, sofern Q_1 nicht leer ist. Ansonsten führt man $Q_3.popFront()$ aus, wenn aber auch dieses leer ist, wird der Algorithmus beendet.
- Beim relaxieren einer Kante $(u,v) \in E$ werden Knoten v mit $v \notin Q_1$ und $v \notin Q_3$ in $Q_{c(u,v)}$ eingefügt, beim verringern eines Gewichtes (nur von 3 nach 1 möglich) entfernt man v aus Q_3 und führt $Q_1.pushBack(v)$ aus. Allerdings benötigt man hierzu auch ein Array $A[1..n]$, das zu jedem $v \in V$, welches sich in Q_3 befindet, an der Stelle $A[v]$ einen Handle auf das zugehörige Listenelement speichert.

Lösungsende

c. Argumentieren Sie kurz, warum Ihr Algorithmus aus Teilaufgabe **b.** das gewünschte Laufzeitverhalten aufweist. [2 Punkte]

Lösung

Jede Kante in E wird genau einmal betrachtet. Dabei wird jeweils höchstens ein *pushBack* ausgeführt und höchstens ein *remove* auf einer doppelt verketteten Liste. Außerdem wird für jeden Knoten höchstens einmal *popFront* ausgeführt und es ist ja $|V| = O(|E|)$. All die Operationen *pushBack*, *popFront* und *remove* benötigen jeweils $O(1)$ Zeit. Initialisieren von A benötigt $O(|V|) = O(|E|)$ Zeit.

Insgesamt braucht man also $O(|E|)$ Zeit.

Lösungsende