Name:

Vorname:

Matrikelnummer:

Lösungsvorschlag

Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. P. Sanders 30.7.2013

Klausur Algorithmen I

Aufgabe 1.	Kleinaufgaben	15 Punkte
Aufgabe 2.	Dijkstras Algorithmus	10 Punkte
Aufgabe 3.	Zusammenhang	10 Punkte
Aufgabe 4.	Polyhalmadrom	10 Punkte
Aufgabe 5.	Minimale Spannbäume	8 Punkte
Aufgabe 6.	Hashing	7 Punkte

Bitte beachten Sie:

- Als Hilfsmittel ist nur **ein** DIN-A4-Blatt mit Ihren **handschriftlichen** Notizen zugelassen.
- **Schreiben** Sie auf **alle** Blätter Ihren Namen und Ihre Matrikelnummer.
- Merken Sie sich Ihre Klausur-ID für den Notenaushang.
- Die Klausur enthält 19 Blätter.
- Die durch Übungsblätter gewonnenen Bonuspunkte werden erst nach Erreichen der Bestehensgrenze hinzugezählt. Die Anzahl Bonuspunkte entscheidet nicht über das Bestehen.

Aufgabe		1	2	3	4	5	6	Summe
max. Punkte		15	10	10	10	8	7	60
Punkte	EK							
	ZK							

Bonuspunkte:	Summe:	**Note:**

Lösungsvorschlag

Aufgabe 1. Kleinaufgaben [15 Punkte]

a. Gegeben sind Teile der Union-Find Datenstruktur aus der Vorlesung. Fügen Sie Zeilen in den unten dargestellten Code ein, so dass die Union bzw. Link Operation immer den kleineren Baum (damit ist die Anzahl der Elemente der assoziierten Menge gemeint) an den größeren Baum anhängt.
Hinweis: dieses Vorgehen bezeichnet man auch als Union-by-Size.

[3 Punkte]

```
Class UnionFind(n : ℕ)
    parent=⟨1,2,...,n⟩ : Array [1..n] of 1..n

    Function find(i : 1..n) : 1..n
        if parent[i] = i then return i
        else return find(parent[i])

    Procedure link(i, j : 1..n)
        assert i und j sind Repräsentanten von verschiedenen Mengen

    Procedure union(i, j : 1..n)
        if find(i) ≠ find(j) then link(find(i), find(j))
```

Lösung

```
Class UnionFind(n : ℕ)
    parent=⟨1,2,...,n⟩ : Array [1..n] of 1..n
    size=⟨1,1,...,1⟩ : Array [1..n] of 1..n

    Function find(i : 1..n) : 1..n
        if parent[i] = i then return i
        else return find(parent[i])

    Procedure link(i, j : 1..n)
        assert i und j sind Repräsentanten von verschiedenen Mengen


        if size[i] < size[j] then
            parent[i] := j; size[j]+=size[i]
        else
            parent[j] := i; size[i]+=size[j]

    Procedure union(i, j : 1..n)
        if find(i) ≠ find(j) then link(find(i), find(j))
```

Lösungsende

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

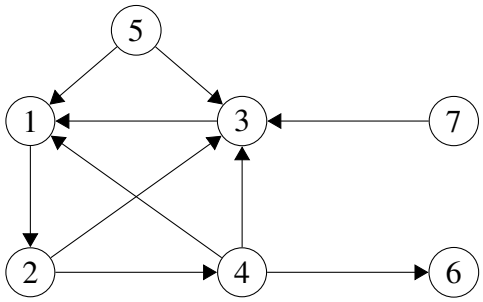
b. Ein gerichteter Graph $G = (V, E)$ ist durch folgendes Adjazenzarray abgespeichert. Zeichnen Sie den Graphen und nummerieren Sie die Knoten in Ihrer Zeichnung. [2 Punkte]

	1	2	3	4	5	6	7	8
V	1	2	4	5	8	10	10	11

	1	2	3	4	5	6	7	8	9	10
E	2	3	4	1	3	1	6	3	1	3

Lösung

Beispielsweise so:



Zwecks Korrektur ist es am leichtesten von jedem nummerierten Knoten die ausgehenden Kanten zu prüfen:
$1 \to 2$ $2 \to 3,4$ $3 \to 1$ $4 \to 1,3,6$ $5 \to 1,3$ $7 \to 3$

Lösungsende

c. Lösen Sie die beiden folgenden Rekurrenzen im Θ-Kalkül:

$$U(n) = n + 20U(n/4), \qquad U(1) = 5$$
$$V(n) = 3n + V(n/4), \qquad V(1) = 42$$

mit $n = 4^k$ und $k \in \mathbb{N}_{>0}$. [2 Punkte]

Lösung

$U(n) = \Theta(n^{\log_4 20})$ und $V(n) = \Theta(n)$.

Lösungsende

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

d. Zeigen Sie oder widerlegen Sie, dass $f(n)+g(n) = \mathrm{O}(g(f(n)))$ gilt. [2 Punkte]

Lösung

Die Behauptung gilt nicht. Wähle $f(n) := \sqrt{n}, g(n) := n^2$. Dann gilt $f(n)+g(n) = \Omega(n^2)$. Es exisitert also ein $c_0 > 0$ und ein $n_0 \in \mathbb{N}$, so dass $f(n)+g(n) \geq c_0 n^2 \quad \forall n > n_0$. Weiter gilt $\mathrm{O}(g(f(n))) = \mathrm{O}(n)$. Annahme: $f(n)+g(n) \in \mathrm{O}(n)$. Dann exisitert ein $c_1 > 0$ und ein $n_1 \in \mathbb{N}$, so dass $f(n)+g(n) \leq c_1 n \; \forall n > n_1$. Damit erhält man $c_0 n^2 \leq c_1 n \; \forall n > \max(n_0, n_1)$. Durch Umformen erhält man $n \leq \frac{c_1}{c_0} \; \forall n > \max(n_0, n_1)$, ein Wiederspruch.

Lösungsende

e. Zeigen oder widerlegen Sie für jedes $k \in \mathbb{N}_+$, dass $\sum_{i=0}^{k} c_i n^i = \Theta(n^k)$ für $c_i \in \mathbb{N}, c_k \neq 0$ gilt. [2 Punkte]

Lösung

Die Behauptung gilt. Zeige zunächst $\sum_{i=0}^{k} c_i n^i = \mathrm{O}\left(n^k\right)$. Es gilt

$$\sum_{i=0}^{k} c_i n^i \leq \sum_{i=0}^{k} c_i n^k = n^k \sum_{i=0}^{k} c_i.$$

Wähle also $c = \sum_{i=0}^{k} c_i$ und z.B. $n_0 = 1$. Nun zeigen wir $\sum_{i=0}^{k} c_i n^i = \Omega(n^k)$. Es gilt

$$\sum_{i=0}^{k} c_i n^i \geq c_k n^k.$$

Wähle also $c = c_k$ und z.B. $n_0 = 1$. Damit folgt die Behauptung insgesamt.

Lösungsende

(weitere Teilaufgaben auf dem nächsten Blatt)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

f. Warum benötigt vergleichsbasiertes Sortieren im schlimmsten Fall mindestens $\Omega(n \log n)$ Vergleiche? Begründen Sie kurz. [2 Punkte]

Lösung

Für n Elemente gibt es $n!$ Permutationen, genau eine davon wendet der Sortierer an. Die Entscheidung, welche anzuwenden ist, braucht im schlimmsten Fall mindestens $\log_2(n!) = \Theta(n \log n)$ Vergleiche.

Lösungsende

g. Ordnen Sie die folgenden fünf Sortieralgorithmen aufsteigend nach worst-case Komplexität: Heap-Sort, Merge-Sort, Quick-Sort, Insertion-Sort und Bucket-Sort für Schlüsselwerte in $O(n)$. [2 Punkte]

Lösung

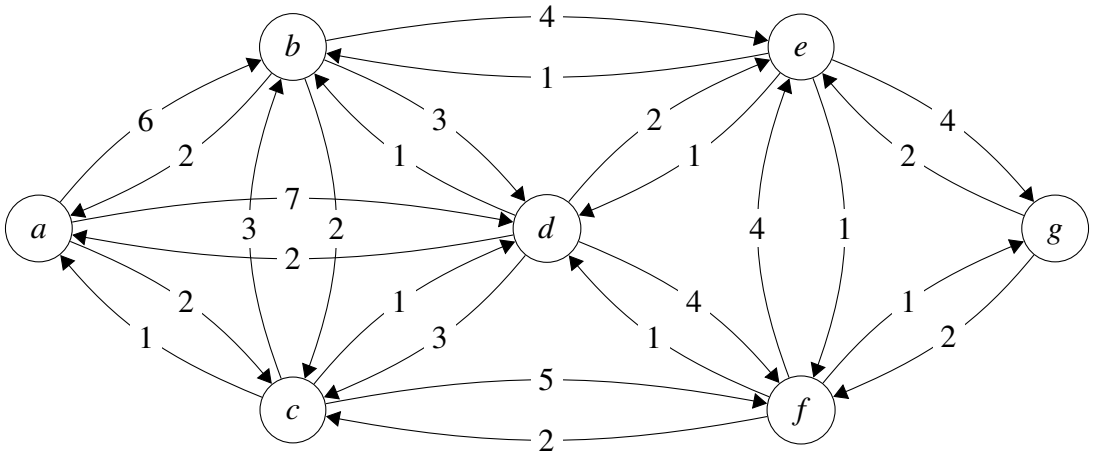
Bucket-Sort $O(n + |S|) = O(n)$, Heap-Sort $O(n \log n)$, Merge-Sort $O(n \log n)$, Quick-Sort $O(n^2)$, Insertion-Sort $O(n^2)$.

Lösungsende

Lösungsvorschlag

Aufgabe 2. Dijkstras Algorithmus [10 Punkte]

Gegeben sei der unten abgebildete gerichtete Graph $G = (V, E)$ mit Kantengewichten.



a. Führen Sie auf dem obigen Graphen den Algorithmus von Dijkstra mit Startknoten a durch. Verwenden Sie für jeden Schleifendurchlauf eine Zeile in folgender Tabelle und tragen Sie die Werte der dabei auftretenden Variablen ein. Die ersten beiden Zeilen sind bereits vorgegeben.

Die Zellen in Spalte Q sollen *alle in* der Queue gespeicherten Knoten v als Paare (p, v) mit Priorität p in Reihenfolge enthalten. Bei $d[\cdot]$ und $parent[\cdot]$ brauchen Sie nur *geänderte Werte* notieren. Wird ein Knoten v vom Algorithmus „*scanned*“ („*settled*“, „*set*“), so tragen Sie in $d[v]$ ein „s“ ein. [6 Punkte]

	geänderte Variablen $d[\cdot]$							geänderte Variablen $parent[\cdot]$						
vollständiger Inhalt Q	a	b	c	d	e	f	g	a	b	c	d	e	f	g
Initialisierung	0	∞	∞	∞	∞	∞	∞	a	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$(0, a)$	s	6	2	7					a	a	a			

Lösungsvorschlag

Fortsetzung von Aufgabe 2

Lösung

	geänderte Variablen $d[\cdot]$							geänderte Variablen $parent[\cdot]$						
Inhalt Q	a	b	c	d	e	f	g	a	b	c	d	e	f	g
Initialisierung	0	∞	∞	∞	∞	∞	∞	a	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$(0,a)$	s	6	2	7					a	a	a			
$(2,c),(6,b),(7,d)$		5	s	3		7			c		c		c	
$(3,d),(5,b),(7,f)$		4		s	5				d			d		
$(4,b),(5,e),(7,f)$		s												
$(5,e),(7,f)$					s	6	9						e	e
$(6,f),(9,g)$						s	7							f
$(7,g)$							s							

Lösungsende

b. Was ist die asymptotische worst-case Laufzeit von Dijkstras Algorithmus im O-Kalkül in Abhängigkeit von Knoten- und Kantenzahl, wenn ein binärer Heap verwendet wird? [1 Punkt]

Lösung

$O((|V|+|E|)\log|V|)$

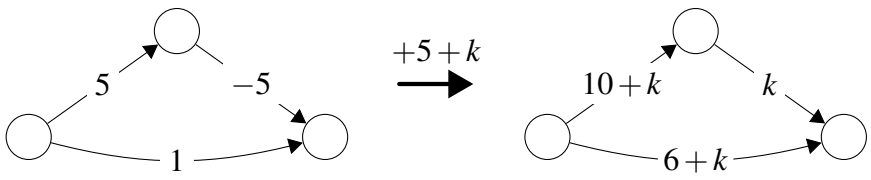
Lösungsende

c. Der Algorithmus von Dijkstra funktioniert nur für nicht-negative Kantengewichte. Es ist aber sicherlich möglich einen Graphen mit negativen Kantengewichten durch Addition einer Konstanten c zu allen Kantengewichten in einen Graph mit nicht-negativen Kantengewichten zu transformieren.

Wieso kann man den Algorithmus von Dijkstra mit dieser Technik nicht auf beliebige Kantengewichte verallgemeinern? Geben Sie ein Gegenbeispiel ohne negative Kreise an. [3 Punkte]

Lösung

Bei der Transformation verändern Sie sich die kürzesten Pfade im Graphen. Jeder Pfad wird um c mal die Anzahl enthaltener Kanten verlängert. Die Verlängerung hängt also von der Kantenzahl ab. Hierdurch können bisher kürzeste Wege im Graphen länger als andere werden, wie es in folgendem Beispiel für $k \geq 0$ passiert:



Lösungsende

Lösungsvorschlag

Aufgabe 3. Zusammenhang [10 Punkte]

Gegeben sei ein ungerichteter, zusammenhängender Graph $G = (V, E)$ und eine Zahl $k > 1$. Das Graphpartitionierungsproblem möchte eine Aufteilung der Knotenmenge V in k Teilmengen (Blöcke) $V_1, \ldots, V_k$ finden, die folgende Bedingungen erfüllt:

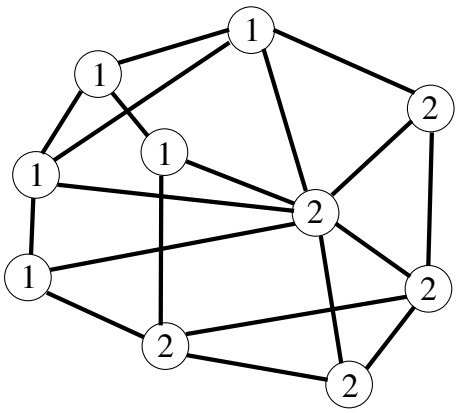
1. $\forall i \in \{1, \ldots, k\} : V_i \neq \emptyset$ – keine leeren Teilmengen
2. $\forall i, j \in \{1, \ldots, k\} : i \neq j \Rightarrow V_i \cap V_j = \emptyset$ – keine Überlappungen
3. $\bigcup_{i=1}^{k} V_i = V$ – volle Überdeckung
4. $\forall i \in \{1, \ldots, k\} : |V_i| \leq \lceil \frac{|V|}{k} \rceil$ – Balancebedingung

Die Zielfunktion ist häufig die Minimierung der Anzahl Schnittkanten $|\mathscr{E}|$ mit

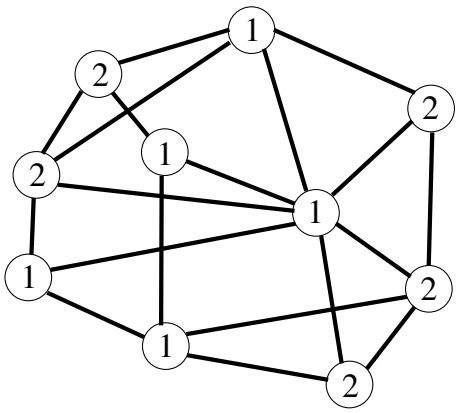
$$\mathscr{E} = \{\{u, v\} \in E \mid u \in V_i, v \in V_j, i \neq j\}.$$

In der Praxis werden aus verschiedenen Gründen häufig **zusammenhängende** Blöcke benötigt. Ein Block V_i heißt zusammenhängend, wenn der knoteninduzierte Teilgraph $G[V_i]$[1] zusammenhängend ist. Eine Partition heißt zusammenhängend, wenn alle Blöcke zusammenhängend sind.

a. Geben Sie jeweils für die folgenden beiden 2-Partitionen des Beispielgraphen an, ob alle Blöcke zusammenhängend sind. Begründen Sie jeweils kurz.
Hinweis: Die Blocknummer eines Knotens ist im Knoten dargestellt. [2 Punkt]



a)



b)

Lösung

Die Partition in a) ist zusammenhängend, da die knoteninduzierten Graphen $G[V_1]$ und $G[V_2]$ zusammenhängend sind (sie haben jeweils eine Zusammenhangskomponente).

Die Partition in b) ist nicht zusammenhängend, da der knoteninduzierte Graph $G[V_2]$ nicht zusammenhängend ist (er besteht aus zwei Komponenten).

Lösungsende

[1] $G[V_i] = (V_i, \{\{u, v\} \in E \mid u, v \in V_i\})$

Lösungsvorschlag

Fortsetzung von Aufgabe 3

b. Gegeben ist nun eine k-Partition $V_1, \ldots, V_k$ $(k > 1)$ von einem zusammenhängenden Graphen $G = (V, E)$. Geben Sie einen Algorithmus an, der in Zeit $O(|E|\alpha_T(2|E|, |V|))$ entscheidet, ob die Partition zusammenhängend ist. α_T bezeichnet dabei die Ackermann-Funktion.

Hinweis: Sie können davon ausgehen, dass man zu einem Knoten $v \in V$ in konstanter Zeit die Blocknummer abfragen kann. Weiterhin werden auch schnellere Algorithmen als Lösung akzeptiert. [6 Punkte]

Lösung

1. Benutze die Union-Find Datenstruktur (mit Union by Rank und Pfadkompression) aus der Vorlesung folgendermaßen:

   - modifiziere die Struktur so, dass zu jedem Zeitpunkte gespeichert wird, wie viele Mengen vorhanden sind. Dies geht zum Beispiel durch Einführen eines Counters. Dieser Counter wird am Anfang mit der initialen Anzahl Mengen initialisiert und beim Vereinigen zweier Mengen stets um eins reduziert.
   - Start: jeder Knoten ist zunächst in seiner eigenen Menge UnionFind($|V|$)
   - iteriere über die Kanten $e = \{u, v\} \in E$: falls die beiden Endpunkte der Kante im gleichen Block sind, führe eine Link Operation mit Repräsentanten der beiden Mengen der Knoten aus (Vereinige die Mengen der Knoten). Mit anderen Worten, es werden nur Link Operation ausgeführt auf Kanten, die keine Schnittkanten.
   - Ende: wenn die Anzahl der Mengen in der UnionFind Datenstruktur k ist, gebe zusammenhängend aus, sonst unzusammenhängend.

```
Tc : UnionFind(|V|)                          // Modfizierte Union-Find Struktur
foreach {u,v} ∈ E do
    if Block[u] == Block[v] then
        A = Tc.find(u)                       // Repräsentant der Menge u
        B = Tc.find(v)                       // Repräsentant der Menge v
        if A ≠ B then
            Tc.link(A,B)
if k == Tc.number_of_sets() then return true else return false
```

2. Hinweis: es gibt auch Lösungen die das Problem in Zeit $O(|V| + |E|)$ lösen (z.B. mit Hilfe von Breitensuchen oder Tiefensuchen). Das war natürlich auch in Ordnung.

Lösungsende

c. Zeigen Sie, dass ihr Algorithmus aus Teilaufgabe **b.** das gewünschte Laufzeitverhalten aufweist. [2 Punkte]

Lösung

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

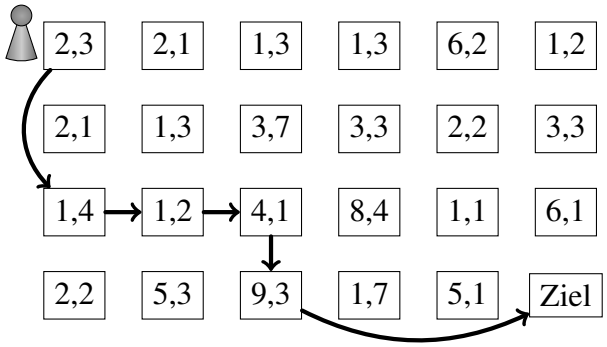
Der obige Algorithmus führt $2|E|$ Find Operationen und höchstens $|V|$ Link Operationen aus. Aus dem Satz von Tarjan und Seidel Sharir (siehe Vorlesung) folgt dann die Laufzeit.

Lösungsende

Lösungsvorschlag

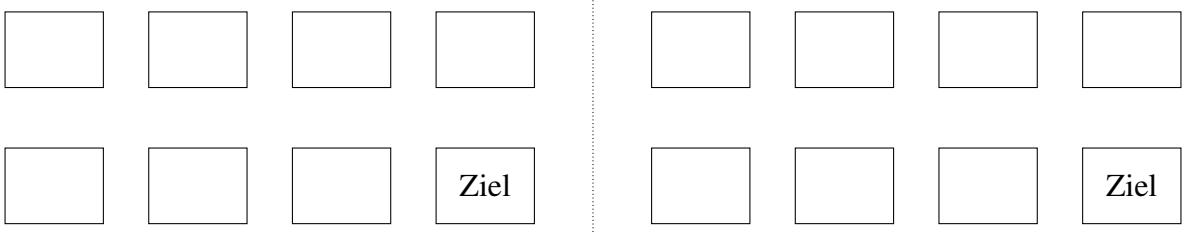
Aufgabe 4. Polyhalmadrom [10 Punkte]

Polyhalmadrom ist ein Spiel mit Karten und kleinen Figuren. Auf den Karten sind zwei Zahlen x_1, x_2 gedruckt und die Karten werden in einem rechteckigen Spielfeld mit m Zeilen und n Spalten zufällig angeordnet. Ein Spieler erhält eine kleine Figur, die zu Beginn auf die Karte am oberen linken Eck des Spielfelds gesetzt wird. Die Figur darf in jedem Zug wahlweise x_1 oder x_2 Karten weiterlaufen, aber nur nach rechts oder nach unten und nur innerhalb des Spielfeldes. Ziel des Spiels ist es, die Figur in **möglichst vielen Zügen** zur unteren rechten Ecke zu ziehen.



Beispiel eines Polyhalmadrom Spiels mit einer (nicht maximalen) Zugfolge

a. Zeigen Sie, dass eine *greedy-basierte* Suche, die nur die kleinere der beiden Zahlen betrachtet, nicht immer eine längstmögliche Zugfolge berechnet. Konstruieren Sie hierzu auf folgendem 2×4 Spielfeld ein Gegenbeispiel, in dem eine greedy-basierte Suche (siehe Pseudocode unten) eine suboptimale Zugfolge von Start bis Ziel berechnet. [2 Punkte]



Fall Sie beide Spielfelder verwenden, machen Sie deutlich welche Lösung zu werten ist.

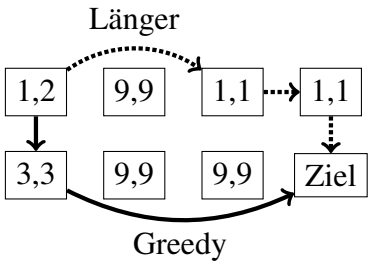
```
Function GreedyPolyhalmadrom(A : (m × n)-Matrix, i, j : ℕ, k : ℕ) : boolean
    if i > m ∨ j > n then return false
    if (i, j) = (m, n) then
        Print „Ziel vom Start in k Zügen erreichbar.“
        return true
    x := min(A[i, j].x1, A[i, j].x2)
    if GreedyPolyhalmadrom(A, i + x, j, k + 1) then
        Print „Zug (i, j) → (i + x, j)“
        return true
    if GreedyPolyhalmadrom(A, i, j + x, k + 1) then
        Print „Zug (i, j) → (i, j + x)“
        return true
    return false
```

Lösungsvorschlag

Fortsetzung von Aufgabe 4

Lösung

Folgendes Diagramm zeigt ein solches Gegenbeispiel:



Lösungsende

b. Geben Sie eine präzise Beschreibung eines Algorithmus an, der in $O(m \cdot n)$ Zeit immer eine längste Sprungfolge berechnet, oder feststellt, dass keine solche existiert.
Ein Algorithmus, der immerhin noch in $O(m^2 \cdot n)$ oder $O(m \cdot n^2)$ läuft, erreicht höchstens 3 Punkte. Eine Lösung ohne explizite Ausgabe einer längsten Zugfolge erreicht höchstens 5 Punkte. [8 Punkte]

Lösung

Wir verwenden dynamische Programmierung auf dem Spielfeld. Schlüssel zur Laufzeit $O(m \cdot n)$ ist es, in einer zweiten Matrix die Länge der längsten, aktuell bekannten Zugfolge zu speichern. In dem Algorithmus wird das Spielfeld dann zeilen- und spaltenweise (oder rekursiv BFS-artig) durchlaufen: in jedem Feld wird für alle möglichen aus diesem Feld ausgehenden Züge geprüft, für welchen Sprung eine längste Zugfolge zu Zielfeld entsteht. Man kann dies als Rekurrenz für eine Matrix L ausdrücken als

$$L[i,j] = \max \left\{ \begin{matrix} L[i+A[i,j].x_1, j], L[i+A[i,j].x_2, j], \\ L[i, j+A[i,j].x_1], L[i, j+A[i,j].x_2] \end{matrix} \right\} + 1\,,$$

wobei $L[i,j] := -\infty$ für Felder außerhalb des zulässigen Bereich und A die Sprungweite-Matrix sei.
Um nachher die Folge auch explizit und effizient berechnen zu können, speichert man neben der aktuellen Länge auch die Koordinaten des Vorgänger-Spielfeldes ab. Wegen der Abhängigkeiten in der Rekurrenz, muss das Spielfeld von unten nach oben und von rechts nach links berechnet werden:

Procedure Polyhalmadrom(A : $(m \times n)$-Matrix)
  Erzeuge Matrix $L \in \mathbb{Z}^{m \times n}$ der aktuell längsten Zugfolge und fülle mit Sentinels:
  $L[i,j] := -\infty \qquad \forall i = 1,\ldots,m \; \forall j = 1,\ldots,n.$
  Erzeuge uninitialisierte Matrix $P \in (\mathbb{Z} \times \mathbb{Z})^{m \times n}$ der Nachfolger-Spielfelder.
  $L[m,n] := 0$ // Distanz zum Zielfeld ist 0.
  for $i = m,\ldots,1$ **do**
    for $j = n,\ldots,1$ **do**
      if $i = m$ & $j = n$ **then skip**
      for $k \in \{1,2\}$ **do**
        $p := i + A[i,j].x_k; \qquad q := j + A[i,j].x_k$
        if $p \leq m$ & $L[i,j] < L[p,j] + 1$ **then** // Prüfe Zugfolge nach unten.
          $L[i,j] := L[p,j] + 1; \qquad P[i,j] := (p,j)$
        endif
        if $q \leq n$ & $L[i,j] < L[i,q] + 1$ **then** // Prüfe Zugfolge nach rechts.
          $L[i,j] := L[i,q] + 1; \qquad P[i,j] := (i,q)$
        endif
  if $L[0,0] < 0$ **do**
    Print „Keine Zugfolge von Start zu Ziel gefunden.“
  else
    Print „Längste Zugfolge enthält $L[0,0]$ Züge.“
    $(i,j) := (0,0)$. // Beginne am Ziel
    while $(i,j) \neq (m,n)$ **do**
      Print „Zug $(i,j) \to P[i,j]$“
      $(i,j) := P[i,j]$ // Wandere Kanten zum Ziel.

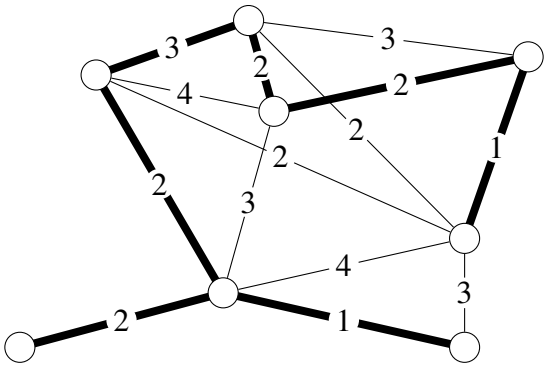
Das dynamische Programm berechnet die längsten Zugfolgen vom Ziel zu *jedem Feld*. Dies ist möglich da auch längste Pfade aus längsten Teilpfaden zusammengesetzt sind und Bellman's Optimalitätsprinzip anwendbar ist. Der obige Pseudocode ist offensichtlich in $O(m \cdot n)$ aufgrund der Schleifen und Speicherinitialisierungen.

Lösungsende

Lösungsvorschlag

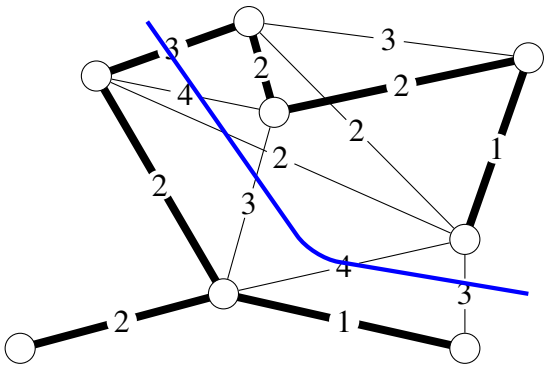
Aufgabe 5. Minimale Spannbäume [8 Punkte]

a. Nennen Sie die Schnitteigenschaft (cut property) minimaler Spannbäume und markieren Sie in folgendem Graphen einen Schnitt, mit dem das Gewicht des Spannbaums reduziert werden kann, oder zeigen Sie, dass kein solcher Schnitt existiert. [2 Punkte]



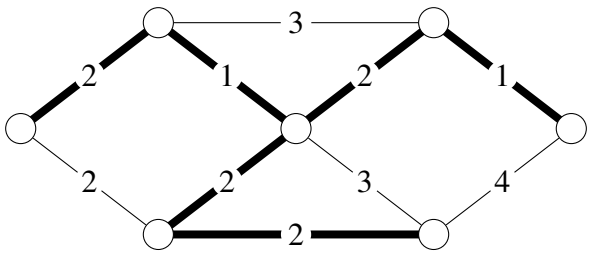
Lösung

Das Schnitteigenschaft wird in der Vorlesung als „Die leichteste Kante e in C kann in einem MST verwendet werden.“ angegeben. In dem Graphen gibt es einen Schnitt, bei dem nicht die leichteste Kante im MST liegt, sondern eine zweitleichteste. Dieser Schnitt verletzt die Schnitteigenschaft.



Lösungsende

b. Nennen Sie die Kreiseigenschaft (cycle property) minimaler Spannbäume und markieren Sie in folgendem Graphen einen Kreis, der diese Eigenschaft verletzt, oder zeigen Sie mit dieser Eigenschaft, dass der eingezeichnete Spannbaum minimal ist. [2 Punkte]



Lösung

Die Kreiseigenschaft wird in der Vorlesung als „Die schwerste Kante auf einem Kreis wird nicht für einen MST benötigt“ beschrieben.
In dem Graphen ist die Eigenschaft für alle Kanten erfüllt. Es gibt nur vier Kanten, die nicht im MST liegen. Die jeweils eindeutig bestimmten Kreise, die diese Kanten mit dem MST schließen, enthalten keine schwerere Kanten.

Lösungsende

c. Sei ein ungerichteter Graph $G = (V, E)$ mit Kantengewichten $w : E \to \mathbb{N}$ gegeben, in dem die Gewichte aller Kanten paarweise verschieden sind. Zeigen Sie, dass der Graph einen eindeutig bestimmten minimalen Spannbaum besitzt. [4 Punkte]

Lösung

Angenommen es gäbe zwei verschiedene MSTs T und T', dann gibt es eine Kante e in T, die nicht in T' ist. Durch Entfernen von e aus T entstehen zwei Teilbäume T_1 und T_2, deren Knotenmengen eine Schnitt S von G definieren. Da T' ein spannender Baum ist, muss es eine Kante $e' \in T'$ geben, die von S zerschnitten wird. OBdA sei e' leichter als e (sonst vertauschen wir die Rollen von (T, e) und (T', e')). Dann können wir aber einen leichteren spannenden Baum definieren, indem wir e in T durch T' ersetzen. Dies ist ein Widerspruch zu der Annahme, dass T ein MST ist.

Lösungsende

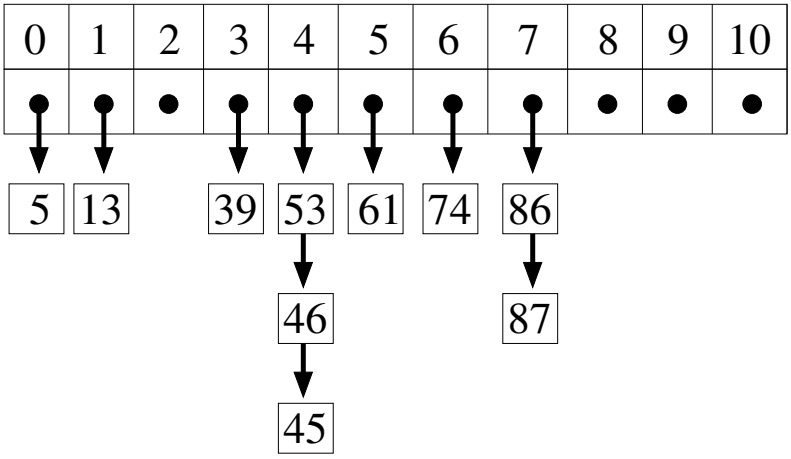
Lösungsvorschlag

Aufgabe 6. Hashing [7 Punkte]

Wir betrachten im Folgenden Hashtabellen mit n Buckets und zugehörigen Hashfunktionen[2],

$$h_n(x) = (x \text{ DIV } n) \text{ MOD } n\,.$$

Beispielsweise ist $h_{11}(74) = 6$. Zur Kollisionsauflösung werden einfach verkettete Listen verwendet. Folgende Hashtabelle hat beispielsweise die Größe $n = 11$ und Hashfunktion h_{11}:



a. Sei nun eine leere Hashtabelle mit $n = 11$ und Hashfunktion h_{11} gegeben.
Geben Sie eine Folge von *insert*-Operationen an, so dass die Tabelle nach Ausführen dieser Operationsfolge den obigen Zustand hat. [2 Punkte]

Lösung

Eine mögliche Lösung: 5, 13, 39, 45, 46, 53, 61, 74, 87, 86

Lösungsende

[2]Hierbei steht DIV für ganzzahlige Division und MOD für den Rest bei ganzzahliger Division.

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

Lösungsvorschlag

Fortsetzung von Aufgabe 6

b. Geben Sie für eine leere Hashtabelle der Größe n mit Hashfunktion h_n eine Folge von n **verschiedenen** *insert* Operationen und n **verschiedenen** *find* Operationen an, so dass die erwartete Laufzeit für die Operationsfolge nicht gilt. Begründen Sie kurz, warum Ihre Folge das gewünschte Verhalten liefert. [3 Punkte]

Lösung

Betrachte die Folge *insert(i)* mit $i = 0,\ldots,n-1$ und *find(i)* mit $i = 0,\ldots,n-1$. Die *insert* Operationen brauchen jeweils $O(1)$, insgesamt also $O(n)$, da nur der Kopfzeiger der Hashtabelle verändert wird. Nach den *inserts* sind im 0-ten Bucket die Einträge $n-1, n-2, \ldots, 1, 0$ und alle anderen Buckets sind leer. Die n *find*-Operationen benötigen also $\sum_{i=0}^{n-1}(i+1) = \sum_{i=1}^{n} i = \frac{n\cdot(n-1)}{2}$ Operationen, da die i-te Operation $n-i+1$ Schritte in der verketten Liste läuft. Die Gesamtkosten der *find* Operation liegt also in $\Theta(n^2)$, was nicht der erwarteten Laufzeit von n mal $O(1)$ also $O(n)$ entspricht.

Lösungsende

c. Nennen Sie zwei Vorteile von Hashing mit verketteten Listen gegenüber Hashing mit linearer Suche. [2 Punkte]

Lösung

- insert in $O(1)$
- Kollisionen wirken sich nicht auf anderen Buckets aus.
- referentielle Integrität
- Leistungsgarantien mit universellem Hashing

Lösungsende