Name:
Vorname:
Matrikelnummer:

Klausur-ID:

Lösungsvorschlag

Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. P. Sanders

13.3.2015

Klausur Algorithmen I

Aufgabe 1.	Kleinaufgaben	16 Punkte
Aufgabe 2.	Alle Wege	10 Punkte
Aufgabe 3.	Maximum-Subarray Problem	8 Punkte
Aufgabe 4.	Intervall-Graphen	13 Punkte
Aufgabe 5.	Leitelement	6 Punkte
Aufgabe 6.	Dijkstras Algorithmus	7 Punkte

Bitte beachten Sie:

- Als Hilfsmittel ist nur **ein** DIN-A4-Blatt mit Ihren **handschriftlichen** Notizen zugelassen.
- **Schreiben** Sie auf **alle Blätter** der Klausur und Zusatzblätter Ihre **Klausur-ID**.
- Merken Sie sich Ihre **Klausur-ID** auf dem Aufkleber für den Notenaushang.
- Die Klausur enthält 15 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		16	10	8	13	6	7	60
Punkte	EK							
	ZK							

Bonuspunkte:	Summe:	**Note:**

Lösungsvorschlag

Aufgabe 1. Kleinaufgaben [16 Punkte]

a. Die Methoden einer Union-Find Datenstruktur seien wie folgt realisiert (erste Version aus der Vorlesung):

```
Function find(i : 1..n)
    if parent[i] = i then return i
    else return find(parent[i])
Procedure link(i, j : 1..n)
    parent[i] := j
Procedure union(i, j : 1..n)
    if find(i) ≠ find(j) then link(find(i),find(j))
```

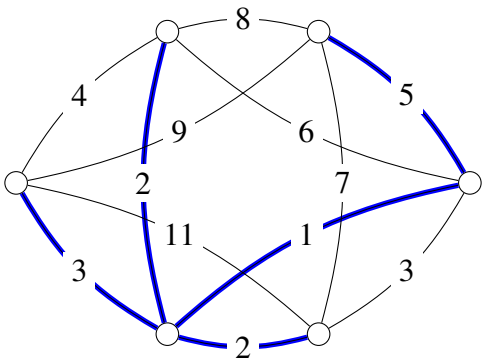
Welche worst-case Zeitkomplexität hat Kruskals Algorithmus mit dieser Union-Find Datenstruktur? [1 Punkt]

Lösung

$O(|E| \cdot |V| + |E| \log |E|)$, weil im schlimmsten Fall die Union-Find Datenstruktur zu einer verketteten Liste degeneriert, bei der bei jedem Aufruf von *find* die gesamte *parent*-Kette nach oben durchlaufen wird. Kruskals Algorithmus ruft pro Kante jeweils zweimal *find* auf den adjazenten Knoten auf.

Lösungsende

b. Markieren Sie in folgendem Graphen genau die Kanten, die zu einem *minimum spanning tree* (MST) gehören. [1 Punkt]



c. Zeigen oder widerlegen Sie, dass $\log_{\sqrt{n}} n = O(1)$ gilt. [1 Punkt]

Lösung

Wir haben

$$\log_{\sqrt{n}} n = \log_{n^{1/2}} n = \frac{\log_x n}{\log_x n^{\frac{1}{2}}} = \frac{\log_x n}{\frac{1}{2}\log_x n} = 2 = O(1)\,.$$

Lösungsende

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

d. Zeigen oder widerlegen Sie, dass $\frac{\log_2 n}{\log_{n/2} n} = O(1)$ gilt. [2 Punkte]

Lösung

Wir haben

$$\frac{\log_2 n}{\log_{n/2} n} = \frac{\frac{log_x n}{\log_x 2}}{\frac{\log_x n}{\log_x \frac{n}{2}}} = \frac{log_x n}{\log_x 2} \cdot \frac{\log_x \frac{n}{2}}{\log_x n} = \frac{\log_x \frac{n}{2}}{\log_x 2} = \log_2 \frac{n}{2} = \log_2 n - 1 \neq O(1)\,.$$

Lösungsende

e. Tragen Sie die folgenden Algorithmen und Datenstruktur-Operationen aus der Vorlesung entsprechend ihrer asymptotischer *worst-case Zeitkomplexität* in die Felder ein. [4 Punkte]

- *binarySearch()* auf einem Array mit festgelegter Größe n.
- *Dijkstra*s kürzeste Wege Algorithmus mit binärem Heap auf einem beliebigen ungerichteten Graphen mit n Knoten und $8n$ Kanten.
- *merge()* von zwei Listen der Länge n und $2n$.
- *last()* auf einer doppelt verketteten Liste der Länge n.
- *findComponents()* – Bestimmung aller Zusammenhangskomponenten in einem ungerichteten Graph mit $2n$ Knoten und n Kanten.
- *siftDown()* auf einem Heap mit n Elementen.
- *quickSort()* in der besten Variante aus der Vorlesung auf einem Array der Länge n.
- *insert()* in eine Hashtabelle mit linearer Suche.

$\Theta(1)$	last()
$\Theta(\log n)$	binarySearch(), siftDown()
$\Theta(n)$	merge(), insert(), findComponents()
$\Theta(n \log n)$	dijkstra()
$\Theta(n^2)$	quickSort()

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

f. Führen Sie least-significant-digit (LSD) Radixsort mit Dezimalziffern (Radix $K = 10$) auf dem folgenden Array durch. Tragen Sie nach jeder Runde das Array in eine der Schablonen ein, markieren Sie die *Bucketgrenzen der Ziffern* durch Trennstriche im Array und kennzeichnen Sie die zu bewertende Lösung deutlich. Rechenschritte zwischen den Arrays werden bei der Bewertung nicht berücksichtigt. [3 Punkte]

Eingabe:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
54	40	52	2	41	12	9	16	31	25	18	24	1	42	37	51

Bucket-Zähler zum Rechnen von Zwischenschritten (nicht bewertet):

0	1	2	3	4	5	6	7	8	9
1	4	4	0	2	1	1	1	1	1
0	1	5	9	9	11	12	13	14	15

Ergebnis nach der 1. Ziffer:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
40	41	31	1	51	52	2	12	42	54	24	25	16	37	18	9

Bucket-Zähler zum Rechnen von Zwischenschritten (nicht bewertet):

0	1	2	3	4	5	6	7	8	9
3	3	2	2	3	3	0	0	0	0
0	3	6	8	10	13	16	16	16	16

Ergebnis nach der 2. Ziffer:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	9	12	16	18	24	25	31	37	40	41	42	51	52	54

Weitere Kopie, falls benötigt:

Bucket-Zähler zum Rechnen von Zwischenschritten (nicht bewertet):

0	1	2	3	4	5	6	7	8	9

Ergebnis nach der Ziffer:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

(weitere Teilaufgabe auf dem nächsten Blatt)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

g. Entwerfen Sie eine FIFO Queue mit Hilfe von zwei Stacks S_1 und S_2, so dass die Operationen *enqueue* und *dequeue* in amortisert konstanter Zeit laufen. Gehen Sie davon aus, dass sie in konstanter Zeit die Anzahl der Elemente eines Stacks abfragen können. Begründen Sie kurz warum die geforderte Laufzeit erreicht wird. [4 Punkte]

Lösung

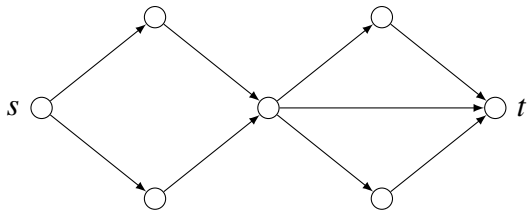
Seien die beiden Stacks S_1 und S_2. Die Operation $enqueue(v)$ führt immer $S_1.push(v)$ aus.
Die Operation *dequeue* gibt $S_2.pop()$ zurück, falls S_2 nicht leer ist. Falls S_2 leer ist, so wird der Stack S_1 auf den Stack S_2 zunächst übertragen. Solange S_1 nicht leer ist, führe $S_2.push(S_1.pop())$ aus (Bemerkung: dies invertiert die Reihenfolge der Stackelemente). Nun gibt der Algorithmus $S_2.pop()$ zurück.
Die amortisierte Laufzeit kann man kurz folgendermaßen begründen: Eine push Operation hat konstante Kosten. Eine pop Operation hat ebenfalls konstante Kosten, falls S_2 nicht leer ist. Im Fall das S_2 leer ist, entstehen kosten in Höhe von S_1.size(). Für jedes Element das sich in S_1 befindet muss jedoch vorher eine push Operation ausgeführt worden sein, so dass die man insgesamt amortisiert konstante Laufzeit erhält.

Lösungsende

Lösungsvorschlag

Aufgabe 2. Alle Wege [10 Punkte]

Betrachten Sie gerichtete azyklische zusammenhängende Graphen $G = (V, E)$ mit $V = \{1, 2, \ldots, n\}$.
a. Bestimmen Sie in folgendem Beispielgraph die Anzahl *aller* nicht notwendigerweise kantendisjunkter (paarweise verschiedener) Wege von s nach t. [1 Punkt]



6 Wege

b. Entwerfen Sie einen Algorithmus, der in $O(|V| + |E|)$ für zwei gegebene Knoten $s, t \in V$ die *Anzahl aller Wege* von s nach t bestimmt. Verwenden Sie hierbei das folgende Tiefensuchschema, und füllen Sie nur die unterstrichenen Prozeduren mit Pseudocode. [8 Punkte]

Tiefensuchschema für $G = (V, E)$ mit Startknoten $s \in V$
unmark all nodes
init(s, t); mark s; DFS$(\perp, s)$
`return` **result()**

```
Procedure DFS(u, v : NodeId)
    foreach (v, w) ∈ E do
        if w is not marked then
            traverseTreeEdge(v, w)
            mark w; DFS(v, w)
        else
            traverseNonTreeEdge(v, w)
    if u ≠ ⊥ then
        backtrack(u, v)
```

```
Procedure init(s, t : V):
p[v] := 0 für alle v ∈ V.
p[t] := 1
Procedure traverseTreeEdge(v, w : V):
No Operation.
Procedure traverseNonTreeEdge(v, w : V):
p[v] += p[w]
Procedure backtrack(u, v : V):
p[u] += p[v]
Function result() : ℕ:
return p[s].
```

Begründen Sie kurz die Korrektheit und Laufzeit Ihres Algorithmus.

Lösung

Wir berechnen nicht nur die Wege $s \rightsquigarrow t$, sondern für jeden v die Anzahl Wege $v \rightsquigarrow t$.
Zuerst stellen wir fest, dass sowohl in backtrack() als auch in traverseNonTreeEdge() die Werte $p[v]$ bzw. $p[w]$ zuvor final berechnet wurden. Bei backtrack() wurde $p[v]$ durch Betrachten aller Nachbarn gerade bestimmt, und in traverseNonTreeEdge() ist w immer ein Knoten der bereits von der Tiefensuche backtrackt wurde, da sonst ein Kreis im Graph wäre. Bereits nach der foreach Schleife über alle Nachbarn und vor backtrack() ist der Wert $p[v]$ final, da die Tiefensuche jeden Knoten genau einmal besucht und nur backtrack()/traverseNonTreeEdge() den entsprechenden Wert ändern.
Wir zeigen, dass nach Ausführen des Algorithmus $p[v]$ die Anzahl der Wege von v nach t ist. Der Wert $p[v]$ wird als Summe der $p[w]$ aller Nachbarn über die ausgehenden Kanten $v \rightarrow w$ berechnet. "tree"-Kanten werden dabei durch backtrack() und "nontree"-Kanten werden durch traverseNonTreeEdge() behandelt. Die Werte $p[w]$ werden im ersten Fall durch Rekursion berechnet und im zweiten Fall muss $p[w]$ bereits bestimmt sein, sonst hat man einen Kreis gefunden.
Die Laufzeit ist die der Tiefensuche, also $O(|V| + |E|)$.

Lösungsende

c. Beschreiben Sie kurz wie Sie Ihren Algorithmus aus Teilaufgabe b) erweitern würden, falls nicht bekannt ist ob der gerichtete Graph einen Kreis enthält. Nehmen Sie dabei an, dass Kreise nur auf Pfaden von s zu t liegen können. [1 Punkt]

Lösung

Während der Tiefensuche kann durch ein zusätzliches "already-backtrackt" Flag in traverseNonTreeEdge() geprüft werden, ob ein Kreis geschlossen wird. Ist in traverseNonTreeEdge() der Zielknoten noch nicht "gebacktrackt", dann wurde ein Kreis gefunden. In diesem Fall gibt es unendlich viele Pfade von s nach t.

Lösungsende

Lösungsvorschlag

Aufgabe 3. Maximum-Subarray Problem [8 Punkte]

Gegeben sei ein Array mit n ganzen Zahlen. Beim Maximum-Subarray Problem wird die maximale Summe eines *zusammenhängenden* Teilarrays $A[i..j]$ gesucht.

Enthält das Array nur positive Zahlen so ist das Problem einfach: die maximale Summe ist die Summe über alle Elemente. Enthält das Array ausschließlich negative Zahlen, soll das Ergebnis 0 sein. Schwierig wird das Problem dann, wenn das Array sowohl positive, als auch negative Zahlen enthält.

a. Markieren Sie im nachfolgenden Array das Teilarray, dessen Summe maximal ist und tragen Sie die maximale Summe in das vorgesehene Feld ein. [1 Punkt]

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

maximale Summe = ____

Lösung

1	2	3	4	5	6	7	8	9	10
-2	1	-3	**9**	**-5**	**2**	**7**	-5	4	-3

maximale Summe = 13

Lösungsende

b. Der nachfolgende Algorithmus löst das Maximum-Subarray Problem:

```
Function maxSubArraySum(A : Array [1..n] of ℤ) : ℤ
    maxSubArraySumRec(A, 1, n)

Function maxSubArraySumRec(A : Array [1..n] of ℤ, l : ℤ, r : ℤ) : ℤ
    if l > r then return 0
    if l = r then return max(0, A[l])
    m := (l + r)/2
    l_max := 0,   sum := 0
    for i := m downto l do
        sum := sum + A[i]
        l_max := max(l_max, sum)
    r_max := 0,   sum := 0
    for i := m + 1 to r do
        sum := sum + A[i]
        r_max := max(r_max, sum)
    c_max := l_max + r_max
    a_max := maxSubArraySumRec(A, l, m)
    b_max := maxSubArraySumRec(A, m + 1, r)
    return max(a_max, max(b_max, c_max))
```

In dieser Teilaufgabe interessieren wir uns für die *asymptotische Anzahl* von **max**(...) Operationen, die ein Aufruf von maxSubArraySum(A) für ein Array A mit n ganzen Zahlen benötigt.

Geben Sie hierfür eine Rekurrenz $T(n)$ an. Bestimmen und beweisen Sie für $T(n)$ eine geschlossene Form. Gehen Sie dabei davon aus, dass n eine Zweierpotenz ist. [3 Punkte]

Lösung

$T(1) = \mathrm{O}(1)$
$T(n) = 2\,T(\frac{n}{2}) + \mathrm{O}(n)$
Geschlossene Form nach Mastertheorem: $\mathrm{O}(n \log n)$

Lösungsende

c. Entwerfen Sie einen Algorithmus, der das Maximum-Subarray Problem in Zeit $\mathrm{O}(n)$ löst. [4 Punkte]

Lösung

Der gesuchte Algorithmus wurde bereits in der Übung vorgestellt:

```
Function maxSubArray(A : Array [1..n] of ℤ) : ℤ
    (best_max, endingHere_max) := 0
    for i := 1 to n do
        endingHere_max := max(endingHere_max + A[i], 0)
        best_max := max(best_max, endingHere_max)
    return best_max
```

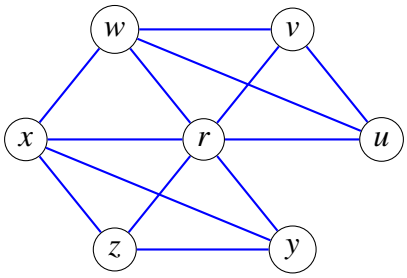
Lösungsende

Lösungsvorschlag

Aufgabe 4. Intervall-Graphen [13 Punkte]

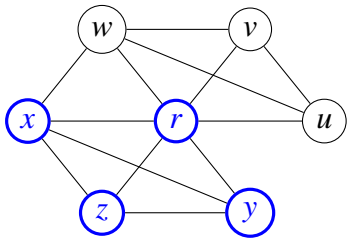
In der Vorlesung wurden Intervall-Graphen definiert als Graphen, deren Knoten Intervalle $[a,b] \subseteq \mathbb{R}$ sind, und zwischen zwei Knoten genau dann eine Kanten existiert, wenn sich die Intervalle überlappen. Formal: $G = (V,E)$ mit
$V = \{[a_1,b_1],\ldots,[a_n,b_n]\}$
$E = \{\{[a_i,b_i],[a_j,b_j]\} \mid [a_i,b_i] \cap [a_j,b_j] \neq \emptyset\}$

a. Gegeben sei folgender Intervall-Graph: $V = \{u,v,w,x,y,z,r\}$ mit $u = [-42,0]$, $v = [0,0]$, $w = [-1,1]$, $x = [\frac{2}{3},\frac{3}{2}]$, $y = [1.5,3]$, $z = [1.\overline{1},\pi]$ und $r = [-\infty,\infty]$. Zeichnen Sie die Kanten in den folgenden Graphen ein. [2 Punkte]



b. Eine *Clique* in einem Graphen $G = (V,E)$ ist eine Teilmenge $U \subseteq V$, in der jeder Knoten mit jedem Knoten verbunden ist. Eine *größte Clique* ist eine solche Teilmenge maximaler Kardinalität, das heißt es gibt keine größere Knotenteilmenge mit dieser Eigenschaft. Finden Sie *eine* größte Clique in dem oben angegebenen Graphen und markieren Sie die dazugehörigen Knoten deutlich. [1 Punkt]

Lösung



Knoten sind blau markiert.

Lösungsende

c. Skizzieren Sie einen Algorithmus, der in $O(|V|\log|V|)$ die Kardinalität einer größten Clique in einem Intervall-Graphen findet. [2 Punkte]

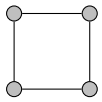
Lösung

Fügen alle Intervallgrenzen in ein Array ein und sortiere dieses aufsteigend. Iteriere über die aufsteigend sortierten Intervallgrenzen. Ist der aktuelle Eintrag eine öffnende Grenze, erhöhe einen Zähler 1, bei einer schließende Grenze, erniedrige um 1. Merke das Maximum dieser Zählvariable.

Lösungsende

d. Zeichnen Sie einen Graphen, der sich nicht als Intervall-Graph darstellen lässt. [1 Punkt]

Lösung

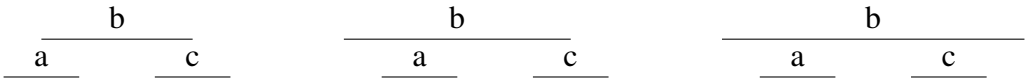


Lösungsende

e. Sei $G = (V,E)$ ein Graph. Eine *Sehne* in einem Kreis $C \subseteq V$ ist eine Kante s, die zwei Knoten des Kreises verbindet, ohne auf diesem Kreis zu liegen. Ein Kreis ist *sehnenfrei*, wenn er keine Sehne enthält. Beweisen oder widerlegen Sie: Jeder sehnenfreie Kreis in einem Intervall-Graphen enthält nicht mehr als drei Knoten. Hinweis: Stellen Sie Intervalle als Teilstrecken der Zahlengerade dar. [4 Punkte]

Lösung

Beweis: Sei C ein sehnenfreier Kreis in einem Intervall-Graphen $G = (V,E)$ mit $|C| = k \geq 4$. Sei $< a,b,c >$ ein Teilpfad von C der Länge 3. Es gibt obdA drei Möglichkeiten, diesen Teilpfad als Intervall-Graphen darzustellen:



In C muss c mit a über mindestens eine Kante verbunden sein, d.h. es gibt ein Intervall d welches a überlappt. Dieses darf b nicht überlappen, da C sonst die Sehne $\{d,b\}$ enthielte. Für die letzten beiden der obigen drei Fälle überlappt jedes Intervall, welches a überlappt, auch b, daher können wir diese ausschließen. Es muss also ein Intervall oder eine sich überlappende Folge von Intervallen geben, die für den ersten Fall a und c verbindet. Mindestens eines dieser Intervallen überlappt aber auch b, was im Widerspruch dazu steht, dass C sehnenfrei ist.

Lösungsende

f. Das angesehene Hotel „Moselschlösschen“ hat für die nächsten Monate n Buchungsanfragen erhalten. Diese liegen als Paare (b_i,e_i), $i \in \{1,n\}$ vor, die den jeweiligen An- und Abreisezeitpunkt festlegen. Der Manager des Hotels möchte aus Kostengründen für die Buchungen möglichst wenige Zimmer bereitstellen. Dank eines gewieften Studenten (der Teilaufgabe c gelöst hat) weiß er bereits, dass k Zimmer ausreichend sind. Jetzt muss er den Buchungen Zimmer zuordnen. Helfen Sie ihm, indem Sie einen Algorithmus mit Laufzeit $O(n \log n)$ skizzieren, der jeder Buchung genau ein Zimmer zuordnet, ohne ein Zimmer doppelt zu belegen. Sie können davon ausgehen, dass alle Zimmer gleichwertig sind. [3 Punkte]

Lösung

Wir führen ein Greedy-„Coloring“ durch: Verwalte die freien Zimmer in einer verketteten Liste und die Zuordnung in einem Feld Z der Länge n. Fügen $2n$ Tupel der Form (a_i,i) und (b_i,i) in ein Feld ein und sortiere nach den Zeitpunkten a_i,b_i. Durchlaufe das Feld. Für ein (a_i,i) nimm ein Zimmer aus der Liste und trage es in $Z[i]$ ein. Für ein (b_i,i) füge das Zimmer $Z[i]$ wieder in die Liste ein.

Lösungsende

Lösungsvorschlag

Aufgabe 5. Leitelement [6 Punkte]

Gegeben sei ein Array A, das n Elemente enthält. Ein Element m nennt man das *Leitelement* von A, wenn es mindestens $\lfloor \frac{n}{2} + 1 \rfloor$ mal in A vorkommt. Skizzieren Sie einen Algorithmus, der prüft ob es ein Leitelement gibt und dieses gegebenenfalls bestimmt. Sie dürfen annehmen, dass die Elemente in $O(1)$ Zeit vergleichbar und hashbar sind. Begründen Sie die Korrektheit, Laufzeit und den Platzverbrauch Ihrer Lösung.
Lösungen (samt Begründung) mit *deterministisch* $O(n)$ Zeit und $O(1)$ zusätzlichem Platz erhalten höchsten 6 Punkte, mit *erwartet* $O(n)$ Zeit und $o(n)$ Platz erhalten höchstens 4 Punkte, alle anderen Lösungen erhalten höchsten 2 Punkte.

Lösung

Wir scannen in folgendem Algorithmus das Array und zählen den Überschuss k des aktuell häufigsten Elementes x. Hat man das häufigste Element gefunden, so muss man noch prüfen ob dieses mindestens $\lfloor \frac{n}{2} + 1 \rfloor$ mal vorkommt.

$k := 0, x := \perp$.
for $i = 0 \ldots n-1$ **do**
  if $k = 0$ **then**
    $x := A[i], k := 1$
  else if $x = A[i]$ **then** $k++$
  else $k--$
$k := 0$
for $i = 0 \ldots n-1$ **do**
  if $x = A[i]$ **then** $k++$
if $k \geq \lfloor \frac{n}{2} + 1 \rfloor$ **return** Leitelement x.
else **return** Kein Leitelement

Offensichtlich scannt der Algorithmus das Array genau zweimal und hat genau drei zusätzliche Variablen, also $O(n)$ Zeit und $O(1)$ Platz. Im ersten Teil des Algorithmus heben sich Paare verschiedener Array-Einträge gegenseitig auf. Daher bleibt in x am Ende eines der häufigsten Element übrig. Dieses muss man nur noch zählen.
Alternative Lösung: Verwende quickSelect aus der Vorlesung um ein Median-Element x mit Rang $\lceil \frac{n}{2} \rceil$ zu bestimmen. Wie oben muss man dann nur noch zählen ob das Element x häufig genug vorkommt. Die Laufzeit von quickSelect ist erwartet $O(n)$ mit $O(\log n)$ zusätzlichen Speicherplatz und der anschließende Scan $O(n)$ mit $O(1)$ Platz.
Weitere mögliche Lösungen sind:

1. Sortiere das Array mit Heapsort ($O(n \log n)$ Zeit und $O(1)$ Platz) und zähle beim Extrahieren ob ein Element häufig genug vorkommt.

2. Oder man kann eine Hashtabelle aufbauen und darin alle Element zählen, um dann ein Element zu finden, was häufig genug vorkommt. Dies benötigt aber $\Omega(n)$ Platz und erwartet $O(n)$ Zeit.

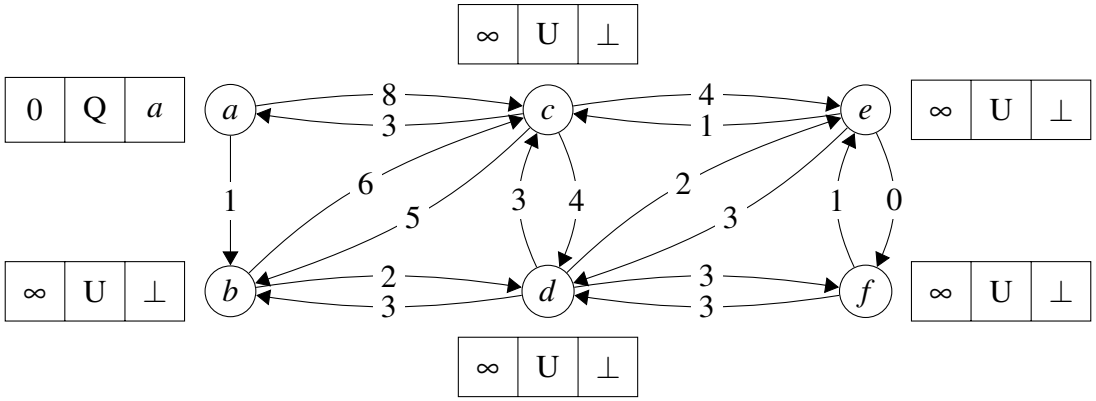
3. Ein einfacher quadratischer Algorithmus mit $O(1)$ Platz wäre für jedes Element zu zählen wie häufig es vorkommt, und nur das häufigste sich zu merken.

Lösungsende

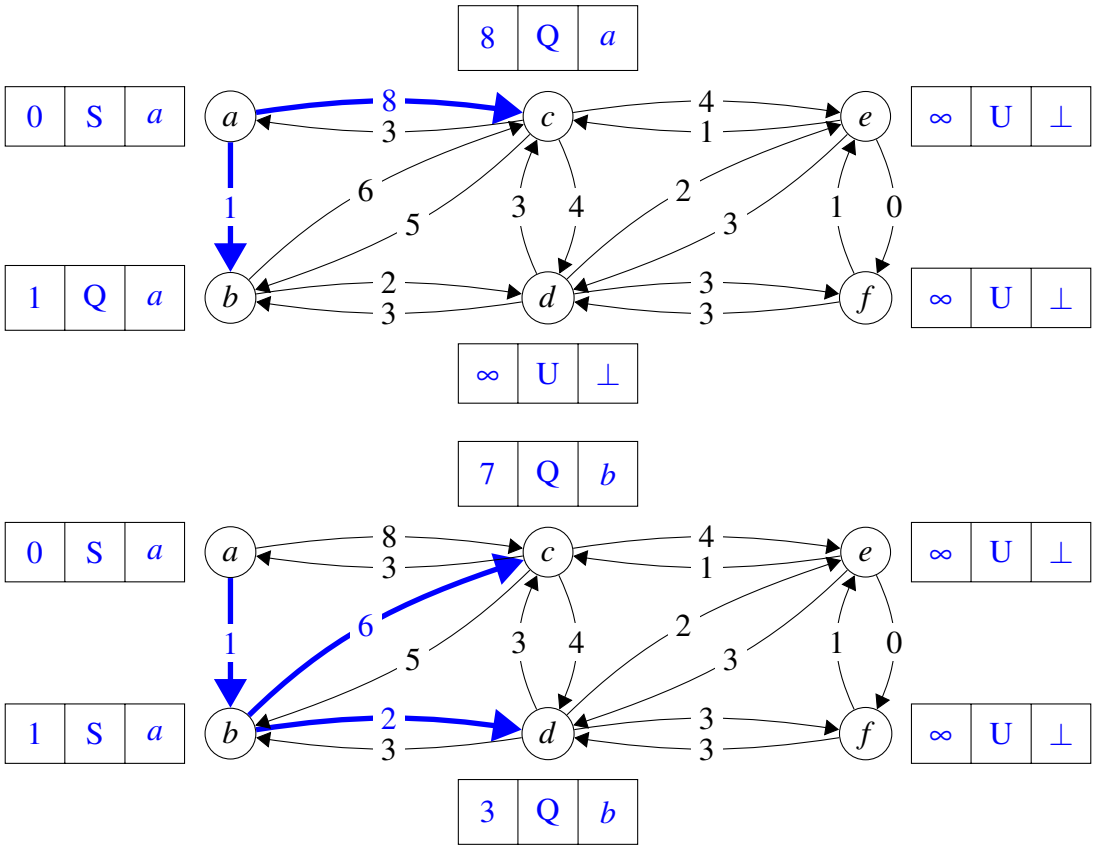
Lösungsvorschlag

Aufgabe 6. Dijkstras Algorithmus [7 Punkte]

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



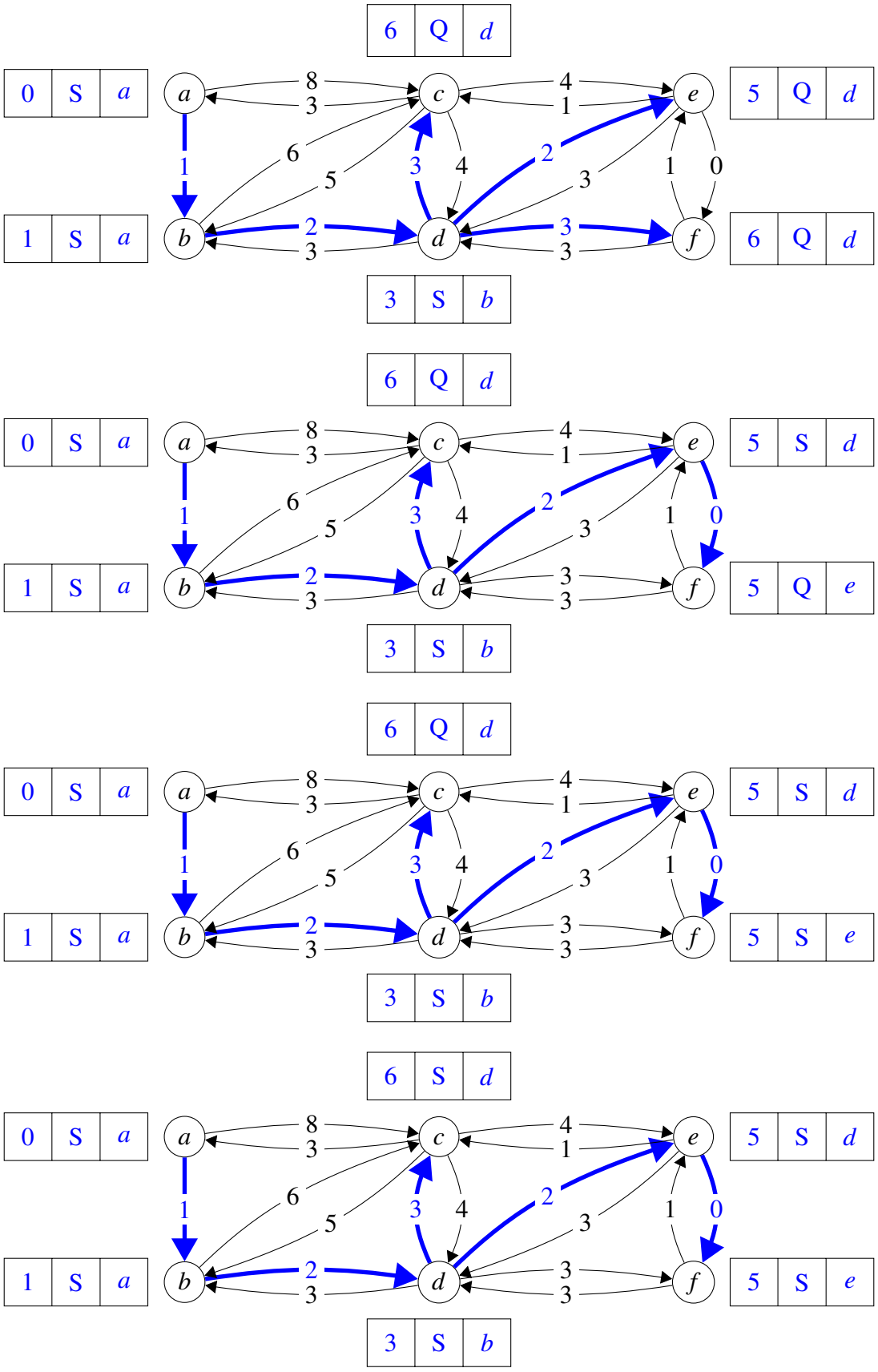
a. Führen Sie auf dem obigen Graphen den Algorithmus von Dijkstra mit Startknoten a durch. Geben Sie nach jedem Schleifendurchlauf für alle Knoten in den drei Kästchen die aktuelle Distanz, den Status des Knotens und den "Parent"-Knoten an. Bezeichnen Sie den Status mit einem 'Q', 'S' oder 'U', je nachdem ob der Knoten in der **Q**ueue ist, bereits **S**canned wurde oder noch **U**nerreicht ist. Verwenden Sie dazu die untenstehenden Kopien des Graphens. Die Initialisierung ist im obigen Graphen bereits vorgegeben. Beschriften Sie deutlich die Graphen, die gewertet werden sollen. Geben Sie gegebenenfalls die Schleifendurchlaufsnummern an. Sie erhalten weitere Blätter mit Graphen bei Bedarf von der Aufsicht. [4 Punkte]



(weitere Graphen auf dem nächsten Blatt)

Lösungsvorschlag

Fortsetzung von Aufgabe 6

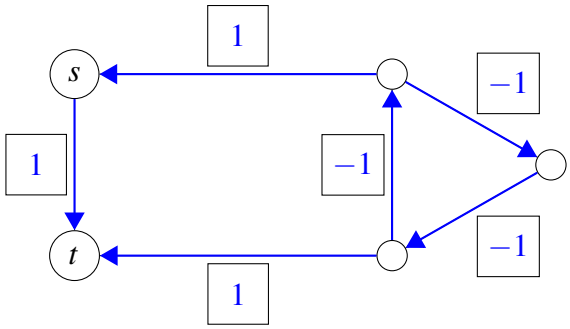


(weitere Graphen und Teilaufgaben auf dem nächsten Blatt)

Lösungsvorschlag

Fortsetzung von Aufgabe 6

b. Zeichnen Sie in folgendem Graphen mit festgelegtem Startknoten s und Zielknoten t *Kantenrichtungen und Gewichte* ein, so dass der Graph einen negativen Zyklus enthält und Dijkstras Algorithmus beim Scannen von t mit korrekter Distanz dennoch terminiert werden kann. [1 Punkt]



c. Welche Eigenschaft eines Problems wird bei dem Prinizip der dynamischen Programmierung ausgenutzt? [1 Punkt]

Lösung

Optimalitätsprinzip: Optimale Lösungen bestehen aus optimalen Lösungen für Teilprobleme.

Lösungsende

d. Nennen Sie drei Algorithmen, die auf dynamischer Programmierung basieren. [1 Punkt]

Lösung

Aufgabe 2, Aufgabe 3, Dijkstras Algorithmus, Bellman-Ford, Rucksack-Algorithmen aus der Vorlesung, Editierdistanz, etc.

Lösungsende