Name:

Vorname:

Matrikelnummer:

Klausur-ID:

Lösungsvorschlag

Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. P. Sanders 24.09.2019

Nachklausur Algorithmen II

Aufgabe 1.	Kleinaufgaben	9 Punkte
Aufgabe 2.	Prioritätslisten: Binomial Heaps	11 Punkte
Aufgabe 3.	Approximationsalgorithmen: Maximum Cut	8 Punkte
Aufgabe 4.	FPT Algorithmen: 3-Hitting Set	10 Punkte
Aufgabe 5.	Geometrische Algorithmen: Vereinigung von Rechtecken	11 Punkte
Aufgabe 6.	Flussalgorithmen: Matrix Rounding	11 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 **20 Blätter**.
- Zum Bestehen der Klausur sind 20 Punkte hinreichend.

Lösungsvorschlag

Aufgabe 1. Kleinaufgaben [9 Punkte]

a. Ein Algorithmus habe eine Laufzeit von $\mathcal{O}(f(n))$. Sein berechnetes Ergebnis weiche um den Faktor $g(n)$ vom optimalen Wert ab. Des Weiteren sei $0 < \varepsilon \leq 1$. Geben Sie für die folgenden Fälle an, ob ein PTAS, FPTAS oder APX vorliegt. Begründen Sie Ihre Antworten kurz.

1. $f(n) = \log(n^{\frac{1}{\varepsilon}}), g(n) = 1+\varepsilon$
2. $f(n) = \frac{1}{\varepsilon}^n, g(n) = 1+\varepsilon$
3. $f(n) = n^{100}, g(n) = 2$

[3 Punkte]

Lösung

1. $f(n) = \log(n^{\frac{1}{\varepsilon}}), g(n) = 1+\varepsilon$
   FPTAS, da eine $1+\varepsilon$-Approximation vorliegt und die Laufzeit polynomiell ist n und $\frac{1}{\varepsilon}$ ist (da $\log(n^{\frac{1}{\varepsilon}}) = \frac{1}{\varepsilon}\log n$).
2. $f(n) = \frac{1}{\varepsilon}^n, g(n) = 1+\varepsilon$
   Keine der drei Optionen, da die Laufzeit exponentiell in n ist.
3. $f(n) = n^{100}, g(n) = 2$
   APX, da der Approximationsfaktor konstant und die Laufzeit polynomiell in n ist.

b. Ein paralleler Sortieralgorithmus benötige zum Sortieren von n Elementen auf p Prozessoren $\mathcal{O}(\frac{n}{p}(\log n)^2)$ Zeit. Geben Sie die Arbeit (Work), den absoluten Speedup und die Effizienz an. Gehen Sie von vergleichsbasiertem Sortieren aus. [3 Punkte]

Lösung

- Work: $W = pT(p) = n(\log n)^2$
- Absoluter Speedup: $S = T_{seq}/T(p) = \frac{p}{\log n}$
- Effizienz: $E = S/p = \frac{1}{\log n}$

c. Es sei $\Sigma = \{a, b, c, d\}$. Dekomprimieren Sie den unten abgebildeten mittels Lempel-Ziv Kompression komprimierten Text. Geben Sie hierfür den dekomprimierten Text an, wenn anfangs $D = \Sigma$ mit der unten abgebildeten Kodierung gegeben ist. [3 Punkte]

Lösung

Komprimierter Text:	0	3	1	2	4	2	6	0	11	5
Dekomprimierter Text:	a	d	b	c	ad	c	bc	a	aa	db

Kodierung:

p	Code
a	0
b	1
c	2
d	3
ad	4
db	5
bc	6
ca	7
adc	8
cb	9
bca	10
aa	11
aad	12

Lösungsvorschlag

Aufgabe 2. Prioritätslisten: Binomial Heaps [11 Punkte]

Der Binomial Heap ist ein Vorgänger des Fibonacci Heaps. Die grundlegende Struktur dabei sind Binomialbäume, die wie folgt rekursiv definiert sind:

1. Ein Binomialbaum B_0 vom Rang 0 ist ein einzelner Knoten ohne Kinder.
2. Ein Binomialbaum B_k vom Rang k besteht aus einer Wurzel mit Kindern $B_0, \dots B_{k-1}$.

Es lässt sich zeigen, dass ein Binomialbaum mit Rang k genau 2^k Knoten enthält und Höhe k hat.
Ein Binomial Heap besteht aus Binomialbäumen, wobei jeder Knoten eines Binomialbaums einem Element im Heap entspricht. Zusätzlich werden folgende Invarianten aufrecht erhalten:

1. Jeder Binomialbaum erfüllt die Heapeigenschaft. Das heißt, dass der Schlüssel jedes Knotens (außer der Wurzel) größer oder gleich dem Schlüssel seines Elternknotens ist.
2. Für jeden Rang k existiert maximal ein Binomialbaum mit Rang k.

Die Hauptoperation von Binomial Heaps ist die `merge`-Operation, welche zwei Binomial Heaps zu einem zusammenfasst und in $\mathcal{O}(\log n)$ Zeit läuft (wobei n die Summe der Anzahl der Elemente in beiden Heaps bezeichnet). Sie verwendet eine zusätzliche temporäre Variable `tmp`, die einen Baum zwischenspeichern kann. Es werden dabei die Binomialbäume beider Heaps in aufsteigender Reihenfolge traversiert. Es wird für jeden Rang k die Anzahl i der Bäume mit Rang k in beiden Heaps, sowie der Variablen `tmp` betrachtet und unterschieden:

$i = 0$: Nichts passiert

$i = 1$: Der entsprechende Baum wird in den neuen Heap übernommen und `tmp` ggf. geleert

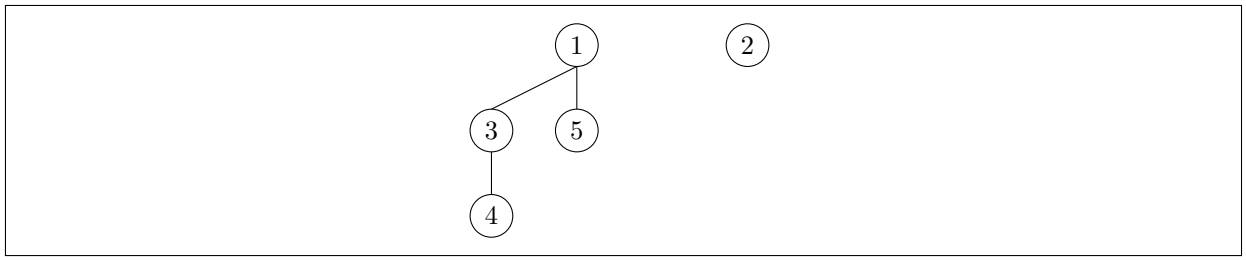
$i = 2$: Der Baum dessen Wurzel den größeren Schlüssel besitzt, wird als Kind an den Baum dessen Wurzel den kleineren Schlüssel besitzt angehängt. Damit entsteht ein Baum vom Rang $k+1$, der in `tmp` gespeichert wird.

$i = 3$: Der Baum, der aktuell in `tmp` gespeichert ist, wird in den neuen Heap übernommen und `tmp` geleert. Mit den beiden anderen Bäumen wird analog zum Fall für 2 Bäume von Rang k verfahren.

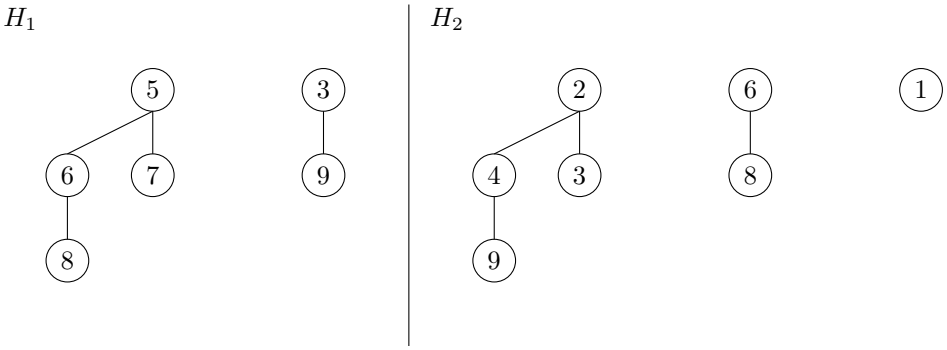
Beachten Sie, dass der maximale Rang eines Binomial Heaps mit n Knoten $\lfloor \log n \rfloor$ ist.
Die `insert`-Operation auf Heap H mit einzufügendem Element k erstellt einen neuen Binomial Heap N bestehend aus nur dem neuen Element und ruft die `merge`-Operation mit H und N als Parameter auf.

a. Zeichnen Sie den Zustand eines leeren Binomial Heaps nach dem Hinzufügen der Elemente $5, 1, 3, 4, 2$ (in dieser Reihenfolge). Sie müssen nur das Endergebnis angeben. [2 Punkte]

Lösung

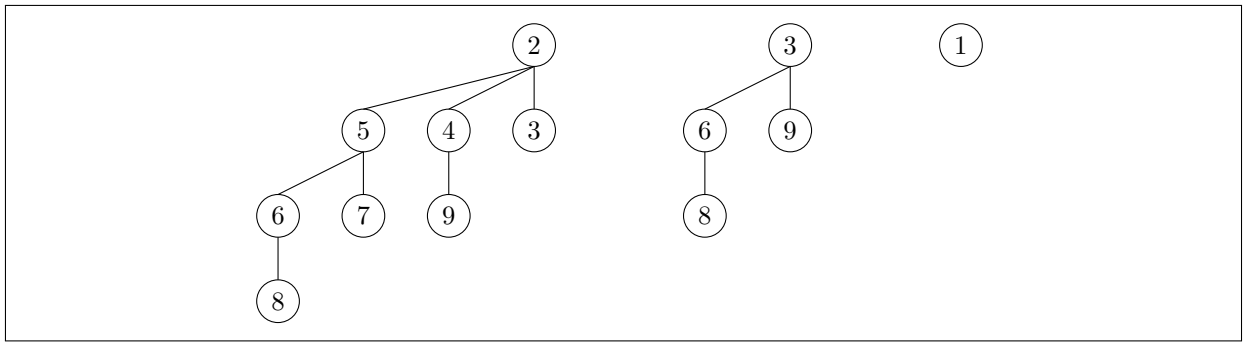


b. Zeichnen Sie das Ergebnis einer `merge`-Operation der beiden unten stehenden Binomial Heaps.



[2 Punkte]

Lösung



c. Zeigen Sie, dass eine Folge von n `insert`-Operationen $\mathcal{O}(n)$ Zeit benötigt. Gehen Sie dabei davon aus, dass Sie beim `merge` nur die Ränge betrachten müssen, für die auch ein Baum existiert.
Hinweis: Verwenden Sie eine amortisierte Analyse mittels der Kontomethode.

[2 Punkte]

Lösung

Wir analysieren mittels der Kontomethode, indem wir Tokens an den Wurzeln der Binomialbäume speichern. Bei jedem `insert` speichern wir einen Token an der neu erstellten Wurzel. Für das Zusammenfügen zweier Binomialbäume verbrauchen wir immer das Token der Wurzel mit dem größeren Schlüssel. Damit haben immer alle Wurzeln einen Token und für alle merge-Operationen wird mit Tokens gezahlt.

d. Geben Sie einen `decreaseKey`-Algorithmus mit Laufzeit in $\mathcal{O}(\log n)$ an, wobei n die Anzahl Elemente im Heap bezeichnet. Begründen Sie Ihre Laufzeit. [2 Punkte]

Lösung

Wir setzen den Schlüssel des betrachteten Elements auf den gewünschten Wert und vertauschen es so lange mit seinem Elternknoten, bis die Heapeigenschaft wiederhergestellt ist. Da Ein Binomialbaum mit Rang k Höhe k hat und der maximale Rang $\log n$ ist, müssen maximal $\log n$ Vertauschungen vorgenommen werden.

e. Geben Sie einen `deleteMin`-Algorithmus mit Laufzeit in $\mathcal{O}(\log n)$ an, wobei n die Anzahl Elemente im Heap bezeichnet. Begründen Sie Ihre Laufzeit. [3 Punkte]

Lösung

Durch Iterieren über alle Bäume im Heap finden wir in $\mathcal{O}(\log n)$ Zeit das kleinste Element im Heap (da der Heap aus maximal $\log n$ Bäumen besteht). Wir entfernen den gesamten Baum des kleinsten Elements aus dem Heap und erstellen einen neuen Heap bestehend aus den Kindern des Wurzelknotens. Dies ist auch ein gültiger Binomial Heap und besteht aus maximal $\log n$ Bäumen, da der maximale Rang $\log n$ ist. Danach vereinigen wir den alten mit dem neuen Heap mittels der `merge`-Operation, die $\mathcal{O}(\log n)$ Zeit benötigt.

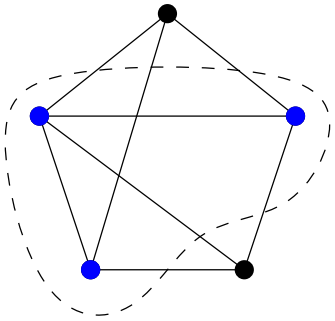
Lösungsvorschlag

Aufgabe 3. Approximationsalgorithmen: Maximum Cut [8 Punkte]

Gegeben sei ein ungerichteter ungewichteter Graph $G = (V,E)$. Ein maximaler Schnitt von G sei eine Partition $(S, V \setminus S)$ der Knotenmenge V, so dass der Wert des Schnittes $w(S) = |\{\{u,v\} \in E \mid u \in S \wedge v \in V \setminus S\}|$ maximal ist.

a. Gegeben sei der folgende Graph G. Geben Sie einen maximalen Schnitt $(S, V \setminus S)$ von G sowie dessen Wert an. [2 Punkte]

Lösung



Der Wert des maximalen Schnittes ist 6.

b. Gegeben sei folgender Algorithmus zur Berechnung eines maximalen Schnittes:

Algorithmus 1 `RandomApproxMaxCut(G)`

```
S ← ∅                                              ▷ Lösung
for v ∈ V do
    x ← Fairer Münzwurf                            ▷ 0 für Kopf, 1 für Zahl;
    if x = 0 then
        S = S ∪ {v}
return S
```

Zeigen Sie, dass der von `RandomApproxMaxCut(G)` berechnete Schnitt S einen erwarteten Wert von $w(S) = m/2$ besitzt. Wobei $m = |E|$ die Anzahl der Kanten im Graph ist. [2 Punkte]

Lösung

Sei $e_1, \ldots, e_m$ die Folge der Kanten von G ($|E| = m$). Des Weiteren sei für $i = 1, \ldots, m$ die Zufallsvariable X_i definiert durch

$$X_i = \begin{cases} 1 \text{ falls Kante } e_i \text{ die Mengen } S \text{ und } V \setminus S \text{ verbindet,} \\ 0 \text{ sonst.} \end{cases}$$

Da die Partition durch einen fairen Münzwurf bestimmt wird, ist die Wahrscheinlichkeit, dass Kante e_i die Mengen S und $V \setminus S$ verbindet $1/2$. Daher gilt

$$\mathbb{E}[X_i] = \frac{1}{2}.$$

Sei $W(S, V \setminus S)$ eine Zufallsvariable, die den Wert des Schnittes angibt. Es folgt aufgrund der Linearität des Erwartungswertes

$$\mathbb{E}[W(S, V \setminus S)] = \mathbb{E}[\sum_{i=1}^{m} X_i] = \sum_{i=1}^{m} \mathbb{E}[X_i] = m \cdot \frac{1}{2} = \frac{m}{2}.$$

c. Geben Sie einen deterministischen Algorithmus an, der eine $1/2$-Approximation des maximalen Schnittes berechnet. Die Laufzeit Ihres Algorithmus sollte in $\mathcal{O}(n+m)$ liegen, wobei n die Anzahl Knoten und m die Anzahl Kanten im Graph sind. Begründen Sie, dass Ihr Algorithmus eine $1/2$-Approximation des maximalen Schnittes berechnet.
Hinweis: Verwenden Sie einen Greedy-Algorithmus, der iterativ Knoten zu einer der beiden Seiten hinzufügt. [4 Punkte]

Lösung

Der Greedy-Algorithmus betrachtet die Knoten in einer beliebigen Reihenfolge $v_1, \ldots, v_n$. Er teilt die Knoten v_i der Menge S oder $V \setminus S$ zu je nachdem in welcher dieser Mengen v_i weniger Nachbarn hat. Seien $N(v_i) = \{u \in V \mid \{v_i, u\} \in E \wedge u \in \{v_1, \ldots, v_{i-1}\}\}$ die Nachbarn von v_i die bisher eingefügt wurden.
Somit trägt Knoten v_i mindestens $|N(v_i)|/2$ Kanten zum Schnitt bei. Da jede Kante genau einmal betrachtet wird, ist die Schnittgröße also mindestens $\sum_i |N(v_i)|/2 = m/2$.

Lösungsvorschlag

Aufgabe 4. FPT Algorithmen: 3-Hitting Set [10 Punkte]

Das NP-schwere Problem 3-Hitting Set für Eingabeparameter C, k ist wie folgt definiert: Für ein Universum S sei eine Menge von max. dreielementigen Teilmengen $C \subseteq \{\{a,b,c\} \mid a,b,c \in S\}$ der Größe n gegeben. Gefragt ist, ob es eine Teilmenge $S' \subseteq S$ mit $|S'| \leq k$ gibt, sodass $\forall c \in C : \exists s \in c : s \in S'$. Es soll also für jedes $c \in C$ mindestens eins der Elemente aus c auch in S' enthalten sein.

a. Sei $S = \mathbb{N}, C = \{\{1,2,3\},\{1,4,7\},\{2,4,5\},\{3,5,7\},\{5,6,8\},\{6,9\}\}$. Geben Sie ein Hitting Set (also S' aus der Definition oben) mit minimaler Größe an. [1 Punkt]

Lösung

$S' = \{3,4,6\}$ (Alternativ $\{1,5,6\}$)

b. Gegeben sei der unten abgebildete Pseudocode eines Kernbildungsalgorithmus für 3-Hitting Set, der das Problem in Polynomialzeit auf Größe $\mathcal{O}(k^3)$ reduziert. Hierfür sei $\#_C(s)$ (bzw. $\#_C(s_1,s_2)$) die Anzahl der Elemente aus C, in denen s (bzw. sowohl s_1 als auch s_2) enthalten sind. Beweisen Sie die Korrektheit des Algorithmus.

[5 Punkte]

Algorithmus 2 `Kernbildung(C, k)`

while $\exists s_1, s_2 \in S : \#_C(s_1,s_2) > k$ **do**
  $C \leftarrow (C \setminus \{c \in C | s_1, s_2 \in c\}) \cup \{s_1, s_2\}$ ▷ $s_1 \neq s_2$
$k' \leftarrow k$
while $\exists s \in S : \#_C(s) > k^2$ **do**
  if $k' = 0$ **then return** false
  $C \leftarrow C \setminus \{c \in C | s \in c\}$
  $k' \leftarrow k' - 1$
if $|C| > k' \cdot k^2$ **then return** false
return (C, k')

Lösung

Erste Schleife: Zunächst betrachten wir zwei feste Elemente $s_1, s_2 \in S$: Angenommen es gäbe mehr als k Teilmengen, in denen s_1 und s_2 vorkommen. Dies impliziert, dass es mehr als k "dritte" Elemente in diesen Mengen gibt. Daher muss, um alle Teilmengen durch k Elemente abdecken zu können, mindestens eines der Elemente s_1 oder s_2 in S' enthalten sein (wir wissen jedoch nicht welches der beiden). Daher ersetzen wir alle Mengen, die s_1 und s_2 enthalten, durch eine einzige Menge $\{s_1, s_2\}$. Durch Iterieren über alle $\mathcal{O}(n^2)$ möglichen Kombinationen von Elementen, die in den Teilmengen in C vorkommen, läuft diese Schleife in Polynomialzeit.
Zweite Schleife: Nun betrachten wir ein festes Element $s \in S$: Angenommen s kommt in mehr als k^2 Mengen vor. Durch die erste Schleife wissen wir, dass s mit jedem anderen Element s_2 maximal k-mal gemeinsam vorkommen kann. Falls es also mehr als k^2 Teilmengen gibt, die s enthalten, könnten diese nicht abgedeckt werden, wenn s nicht in S' aufgenommen wird. Daher wird s in S' aufgenommen, weshalb anschließend nur noch ein Element weniger in der Lösung erlaubt ist.
Nun wissen wir, dass jedes Element in maximal k^2 Teilmengen enthalten ist. Da nur noch k' Elemente in S' aufgenommen werden können, können also maximal $k' \cdot k^2$ Teilmengen abgedeckt werden. Falls noch mehr Teilmengen verbleiben, gibt es also keine Lösung.

c. Beschreiben Sie kurz einen *fixed parameter tractable* (FPT) Algorithmus für 3-Hitting Set an. Begründen Sie, warum ihr Algorithmus fixed parameter tractable ist.

[2 Punkte]

Lösung

Finde einen Problemkern C', k' der Größe $\mathcal{O}(k^3)$. Dann prüfe für alle Kombinationen von k' Elementen aus $\bigcup_{c \in C} c$, ob sie eine Lösung ergeben.
Der Problemkern wird in Polynomialzeit gefunden. Danach ist die Problemgröße nur noch Abhängig von k. Damit ergibt sich ein FPT Algorithmus.

d. Zeigen oder widerlegen Sie, dass es einen Kernbildungsalgorithmus für 3-Hitting Set gibt, der einen Problemkern der Größe $\mathcal{O}(\log n)$ findet. (Unter der Annahme $P \neq NP$)

[2 Punkte]

Lösung

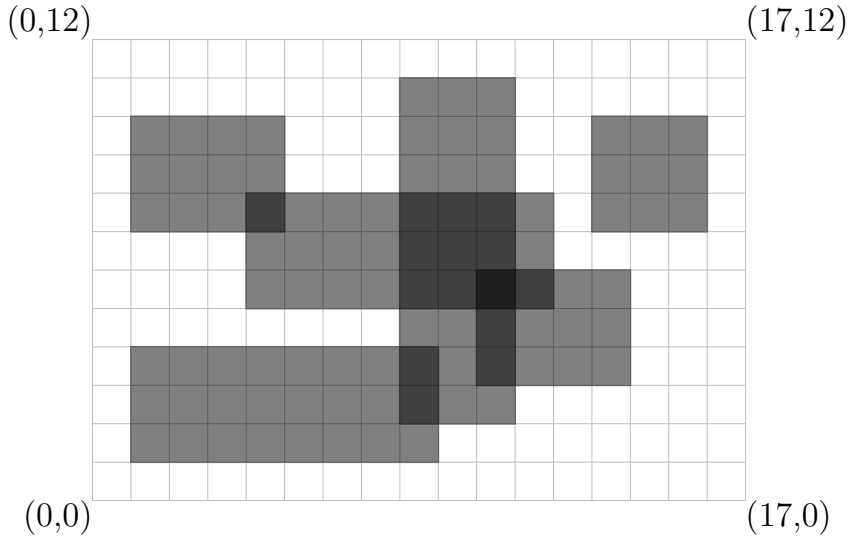
Angenommen es gäbe einen Problemkern der Größe $\mathcal{O}(\log n)$, dann können wir in Zeit $\mathcal{O}(2^{3 \log n} \log n) = \mathcal{O}(n^3 \log n)$ alle Kombinationen von Elementen in $\bigcup_{c \in C} c$ durchprobieren und hätte damit einen Polynomialzeitalgorithmus für 3-Hitting Set, da der Kernbildungsalgorithmus in Polynomialzeit läuft.

Lösungsvorschlag

Aufgabe 5. Geometrische Algorithmen: Vereinigung von Rechtecken [11 Punkte]

Gegeben sei eine Menge S von n achsenparallelen Rechtecken in der Ebene. Jedes Rechteck sei durch seinen linken unteren Eckpunkt p_1 und rechten oberen Eckpunkt p_2 definiert. Des Weiteren seien alle x- sowie y-Koordinaten der Eckpunkte der Rechtecke paarweise verschieden. Gehen Sie davon aus, dass Rechtecke sich überlagern können.

a. Gegeben sei die folgende Menge an Rechtecken S. Berechnen Sie den Flächeninhalt der Vereinigung der Rechtecke.



[1 Punkte]

Lösung

Der Flächeninhalt der Vereinigung der Rechtecke beträgt 92.

b. Gehen Sie zunächst vereinfachend davon aus, dass jedes Rechteck nie mehr als ein anderes Rechteck aus S scheidet. Erweitern Sie den Sweepline-Algorithmus aus der Vorlesung, so dass dieser den Flächeninhalt der Vereinigung der Rechtecke aus S in $\mathcal{O}(n \log n)$ Zeit berechnet. Begründen Sie die Laufzeit Ihres Algorithmus. [6 Punkte]

Lösung

Wir verwenden einen Sweepline-Algorithmus in Richtung aufsteigender x-Koordinaten. Hierzu werden die x-Koordinaten der Rechtecke aus S aufsteigend sortiert.
Sweepline-Status: Zur Verwaltung unseres Sweepline-Status verwenden wir einen balancierten Suchbaum T (AVL, Red-Black, ...). Der Suchbaum speichert dabei die aktuell von der Sweepline geschnittenen oberen bzw. unteren Seiten der Rechtecke sortiert nach ihrer relativen Ordnung auf der Sweepline.
Events: Events sind gegeben durch die linken bzw. rechten Seiten der Rechtecke. Für linke Seiten wird zunächst der horizontale Abstand zum letzten Event mit der zuvor berechneten Länge des Schnittes multipliziert und zum Flächeninhalt hinzugefügt. Anschließend wird die Länge des Schnittes aktualisiert. Hierbei wird zwischen linken und rechten Seiten unterschieden

- Linke Seiten: Hierfür werden die oberen und unteren Seiten des zum Event gehörigen Rechtecks in den Sweepline Status eingefügt. Hierbei wird für jede Seite überprüft, ob diese innerhalb bzw. außerhalb eines Rechtecks liegt. Liegen beide Seiten innerhalb des gleichen Rechtecks so muss der Schnitt nicht aktualisiert werden. Liegt nur die obere Seite innerhalb eines Rechtecks, so wird der Schnitt um die Distanz zwischen der unteren Seite des geschnittenen Rechtecks und der unteren Seite des neuen Rechtecks erweitert (analog für die untere Seite). Liegen beide Seiten außerhalb eines Rechtecks, so wird der Schnitt um die Distanz zwischen der oberen und unteren Seite des neuen Rechtecks erweitert.

- Rechte Seiten: Hierfür werden die oberen und unteren Seiten des zum Event gehörigen Rechtecks aus dem Sweepline Status entfernt. Liegen beide Seiten innerhalb des gleichen Rechtecks so muss der Schnitt nicht aktualisiert werden. Liegt nur die obere Seite innerhalb eines Rechtecks, so wird der Schnitt um die Distanz zwischen der unteren Seite des geschnittenen Rechtecks und der unteren Seite des neuen Rechtecks verringert (analog für die untere Seite). Liegen beide Seiten außerhalb eines Rechtecks, so wird der Schnitt um die Distanz zwischen der oberen und unteren Seite des neuen Rechtecks verringert.

Analyse: Das Sortieren der x-Koordinaten benötigt $\mathcal{O}(n \log n)$ Zeit. Insgesamt werden $\mathcal{O}(n)$ Events verarbeitet. Das Einfügen der oberen bzw. unteren Seiten der Rechtecke benötigt $\mathcal{O}(\log n)$ Zeit. Die Berechnung der Länge des Schnittes benötigt ebenfalls $\mathcal{O}(\log n)$ Zeit. Insgesamt ergibt sich damit eine Laufzeit von $\mathcal{O}(n \log n)$.

c. Gehen Sie nun davon aus, dass jedes Rechteck beliebig viele andere Rechtecke aus S schneiden kann. Erklären sie **kurz**, wie Sie ihren Algorithmus aus Aufgabenteil **b** erweitern müssten, damit dieser den Flächeninhalt der Vereinigung der Rechtecke aus S in $\mathcal{O}(n^2)$ Zeit berechnet. Begründen Sie die Laufzeit Ihres Algorithmus.
Hinweis: Verwenden Sie an jedem Event $\mathcal{O}(n)$ Zeit. [4 Punkte]

Lösung

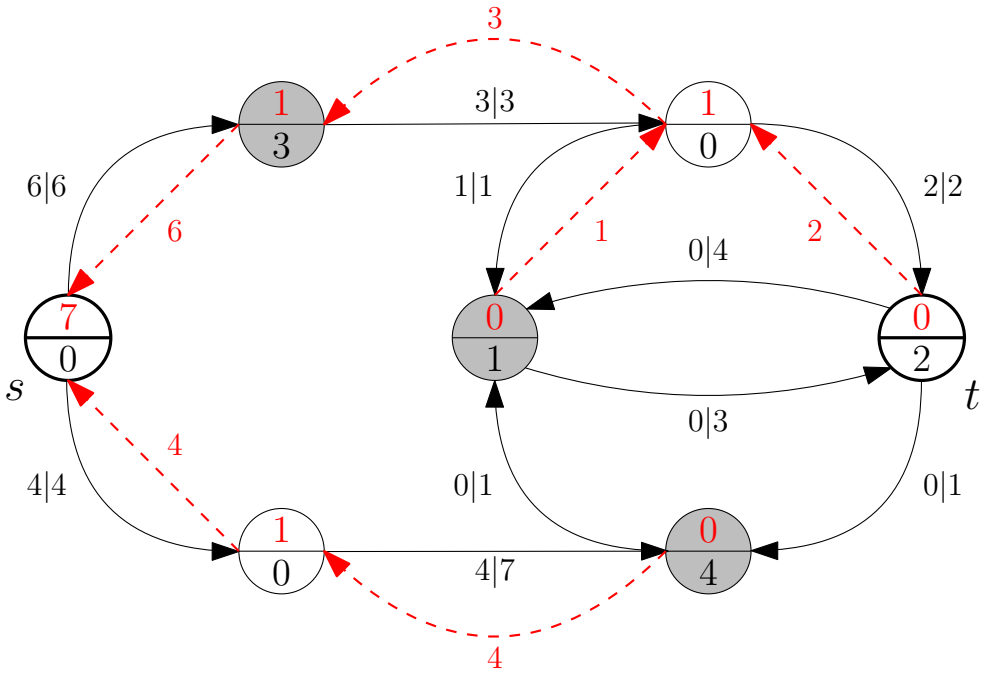
Hinweis: Zur besseren Lesbarkeit wird im Folgenden der gesamte Algorithmus angegeben. Das war zum Erreichen der vollen Punktzahl nicht notwendig!
Wir verwenden einen Kombination von Sweepline-Algorithmen in Richtung aufsteigender x-Koordinaten bzw. absteigender y-Koordinaten. Hierzu werden die x- bzw. y-Koordinaten der Rechtecke aus S aufsteigend sortiert. Der Algorithmus beginnt mit einer Sweepline in Richtung aufsteigender x-Koordinaten.
Sweepline-Status: Für den Zustand unserer Sweepline verwenden wir ein Boolean-Array das für jedes Rechteck angibt, ob es von der Sweepline geschnitten wird. Zudem speichern wir die Länge des Schnittes der Sweepline mit den aktiven Rechtecken.
Events: Für jedes Event wird zunächst der horizontale Abstand zum letzten Event mit der zuvor berechneten Länge des Schnittes multipliziert und zum Flächeninhalt hinzugefügt. Anschließend wird das zum Event gehörige Rechteck auf aktiv (linke Seite) bzw. inaktiv (rechte Seite) gesetzt und mithilfe der inneren Sweepline die Länge des Schnittes aktualisiert.
Innere Sweepline: Für die innere Sweepline verwenden wir einen Zähler, der angibt, wie viele Rechtecke aktuell von der Sweepline geschnitten werden. Für jedes Event (aktive Rechtecke) wird überprüft, ob der Zähler größer als 0. Ist dies der Fall, wird der vertikale Abstand zum letzten Event zur Länge des Schnittes hinzugefügt. Anschließend wird der Zähler inkrementiert (obere Seite) bzw. dekrementiert (untere Seite).
Analyse: Das Sortieren der x- bzw. y-Koordinaten benötigt $\mathcal{O}(n \log n)$ Zeit. Die Berechnung der Länge des Schnittes in der inneren Sweepline benötigt $\mathcal{O}(n)$ Zeit. Das Verarbeiten der Events der äußeren Sweepline benötigt ebenfalls $\mathcal{O}(n)$ Zeit. Insgesamt ergibt sich damit eine Laufzeit von $\mathcal{O}(n^2)$.

Lösungsvorschlag

Aufgabe 6. Flussalgorithmen: Matrix Rounding [11 Punkte]

a. Gegeben sei das folgende unvollständige Flussnetzwerk mit Quelle s und Senke t. Es beschreibt einen Zwischenzustand des preflow-push Algorithmus. Die Knoten sind mit ihrem momentanen Überschuss im unteren Halbkreis beschriftet, die Kanten sind mit ihrem aktuellen Fluss (links) und ihrer Kapazität (rechts) beschriftet. Gehen Sie davon aus, dass immer der maximal mögliche Fluss über eine Kante gepusht wird. Weisen Sie allen Knoten ihr aktuelles Level zu, markieren Sie die aktiven Knoten und zeichnen Sie die Residualkanten mit deren Kapazitäten ein. [3 Punkte]

Lösung



Aktive Knoten sind grau hinterlegt. Einzutragende Residualkanten sind gestrichelt gezeichnet.

b. Gegeben sei eine reellwertige $p \times q$ Matrix $D = \{d_{i,j}\}$. Wir definieren $a_i = \sum_j d_{i,j}$ als die Summe der Einträge der i-ten Reihe und $b_j = \sum_i d_{i,j}$ als die Summe der Einträge der j-ten Spalte. Eine gültige Rundung von D sei eine Rundung (auf oder ab zur nächsten ganzen Zahl) der Einträge $d_{i,j}$, sowie der Summen a_i und b_j, so dass die Summe der gerundeten Einträge in jeder Reihe (Spalte) gleich der gerundeten Summe der Reihe (Spalte) ist.
Geben Sie für die folgende Matrix D eine gültige Rundung an. [2 Punkte]

3.14	6.80	7.30	17.24
9.60	2.40	0.70	12.70
3.60	1.20	6.50	11.30
16.34	10.40	14.50	

Lösung

Folgendes ist eine mögliche Lösung:

3	7	7	17
10	2	1	13
3	1	7	11
16	10	15	

c. Betrachten Sie folgende Erweiterung eines Flussnetzwerks $H = (V,E,c)$: Für jede Kante sei zusätzlich zur Kapazität $c(e)$ eine ganzzahlige untere Schranken $0 \leq l(e) \leq f(e) \leq c(e)$ gegeben. Gehen Sie im Folgenden davon aus, dass ein Algorithmus zur Berechnung eines gültigen maximalen Flusses auf einem erweiterten Flussnetzwerk $G = (V,E,c,l)$ gegeben ist.
Geben Sie an, wie sich eine reellwertige $p \times q$ Matrix D in ein ganzzahliges erweitertes Flussnetzwerk $H = (V,E,c,l)$ transformieren lässt. Geben Sie zusätzlich an, wie ein gültiger Fluss in Ihrem Netzwerk in eine gültige Rundung umgewandelt werden kann. Gehen Sie davon aus, dass für jede Eingabe eine gültige Lösung existiert.
Hinweis: Verwenden Sie als Knotenmenge $V = Zeilen \cup Spalten \cup \{s,t\}$. [6 Punkte]

Lösung

Der Algorithmus wandelt zunächst die gegebenen $p \times q$ Matrix $D = \{d_{i,j}\}$ sowie die Summen a_i, b_j wie folgt in ein Flussnetzwerk $H = (V,E,c,l)$ um. Hierfür verwenden wir die Knotenmenge $V = R \cup C \cup \{s,t\}$, d.h. Knoten entsprechen den Zeilen bzw. Spalten der Matrix. Anschließend wird für jeden Eintrag der Matrix eine Kante zwischen dem entsprechenden Zeilen- bzw. Spaltenknoten eingefügt. Die untere bzw. oberere Schranke der Kante ist gegeben durch die beiden möglichen Rundungen der dazugehörigen Zahl. Zusätzlich fügen wir Kanten zwischen s (t) und den Zeilenknoten (Spaltenknoten) ein. Diese entsprechen den gerundeten Zeilen bzw. Spalten und die untere bzw. obere Schranke werden entsprechend auf die möglichen Rundungen der Summen gesetzt. Durch den Flusserhalt und Kapazitätskonformität ist dadurch gewährleistet, dass die Summe der Einträge einer Zeile bzw. Spalte nicht die Gesamtsumme überschreitet.

- $R = \{1,\ldots,p\}, C = \{1,\ldots,q\}$
- $V = R \cup C \cup \{s,t\}$
- $E = (R \times C) \cup (\{s\} \times R) \cup (C \times \{t\})$
- $\forall e \in E$ setze $c(e) = \begin{cases} \lceil a_i \rceil & e = (s,i) \\ \lceil b_j \rceil & e = (j,t) \\ \lceil d_{i,j} \rceil & \text{sonst} \end{cases}$
- $\forall e \in E$ setze $l(e) = \begin{cases} \lfloor a_i \rfloor & e = (s,i) \\ \lfloor b_j \rfloor & e = (j,t) \\ \lfloor d_{i,j} \rfloor & \text{sonst} \end{cases}$

Anschließend wird auf dem Flussnetzwerk H ein maximaler Fluss berechnet. Der resultierende Fluss kann anschließend direkt in eine gültige Rundung umgewandelt werden, da jeder Eintrag (bzw. jede Summe) genau einer Kante im Flussnetzwerk entspricht. Der Wert des Eintrags (bzw. der Summe) entspricht dem Wert des Flusses des entsprechenden Kante.