Name:

Vorname:

Matrikelnummer:

Klausur-ID:

Lösungsvorschlag

Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. P. Sanders 19.02.2019

Klausur Algorithmen II

Aufgabe 1.	Kleinaufgaben	10 Punkte
Aufgabe 2.	Approximationsalgorithmen: Set Cover	11 Punkte
Aufgabe 3.	Geometrische Algorithmen: Dreiecke und Punkte	8 Punkte
Aufgabe 4.	Flussalgorithmen: Bipartite Matchings	13 Punkte
Aufgabe 5.	Randomisierte Algorithmen: Skip Lists	9 Punkte
Aufgabe 6.	Stringalgorithmen: Permutiertes LCP Array	9 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 [10 Punkte]

a. Gegeben sei der unten abgebildete Radix Heap.
Geben Sie den Zustand nach einer sowie zwei `deleteMin()` Operationen an. [2 Punkte]

Bucket ID	-1	0	1	2	3
Radix Heap	1010_b			1110_b 1101_b 1100_b 1111_b	10000_b

Lösung

Nach erstem `deleteMin()`:

Bucket ID	-1	0	1	2	3
Radix Heap				1110_b 1101_b 1100_b 1111_b	10000_b

Nach zweitem `deleteMin()`:

Bucket ID	-1	0	1	2	3
Radix Heap		1101_b	1110_b 1111_b		10000_b

b. Ein *Reachability Oracle* für einen gerichteten Graphen $G = (V,E)$ sei eine Datenstruktur, die Erreichbarkeitsanfragen (`isReachable(u,v)`) in $\mathcal{O}(1)$ beantworten kann. Dabei ist `isReachable`$(u,v) =$ `true`, wenn in G ein Pfad von u nach v existiert und `false` sonst. Gegeben sei ein Algorithmus zur Berechnung eines Reachability Oracles für gerichtete *azyklische* Graphen $G = (V,E)$.
Skizzieren Sie unter Verwendung des Algorithmus für gerichtete azyklische Graphen einen Algorithmus für *beliebige* gerichtete Graphen, der in $\mathcal{O}(|V| + |E|)$ zusätzlicher Zeit ein Reachability Oracle berechnet. [2 Punkte]

Lösung

Wir berechnen mittels Tiefensuche in $\mathcal{O}(|V| + |E|)$ Zeit den Schrumpfgraphen des Eingabegraphen. Da dieser azyklisch ist, können wir dafür ein Reachability Oracle für gerichtete azyklische Graphen berechnen. Bei Anfragen ersetzen wir die angefragten Knoten durch ihre entsprechenden Knoten im Schrumpfgraphen.

c. Beschreiben Sie einen parallelen Sortieralgorithmus im CRCW-Modell im *combine* Modus mit Addition (das heißt, wenn mehrere Prozessoren eine Speicherstelle beschreiben, dann werden die geschriebenen Werte addiert). Bei Eingabe eines Arrays E der Größe n soll ihr Algorithmus mit n^2 Prozessoren in $\mathcal{O}(1)$ Zeit laufen.
Sie dürfen davon ausgehen, dass die Werte in E paarweise verschieden sind. Die Ausgabe soll in ein bereits angelegtes Array A geschrieben werden. Zusätzlich steht Ihnen ein weiteres Hilfsarray H der Größe n zur Verfügung, dessen Einträge alle 0 sind. [3 Punkte]

Lösung

Wir nummerieren die n^2 Prozessoren als (i,j) mit $i,j < n$. Prozessor (i,j) vergleicht die Elemente $E[i]$ und $E[j]$. Falls $E[i]$ größer ist als $E[j]$, dann schreibt er eine 1 an die Stelle $H[i]$. Durch den combine Modus enthält $H[i]$ danach also die Anzahl an Elementen, die kleiner sind als $E[i]$.
Danach müssen noch n Prozessoren das Ausgabearray beschreiben: Prozessor i schreibt $A[H[i]] := E[i]$.

d. Sei $f(n,k)$ die Laufzeit eines Algorithmus mit Eingabegröße n. Die Laufzeit hängt weiter von einem Parameter k ab. Geben Sie an, welche der folgenden Laufzeiten ein Problem *fixed-parameter-tractable* (FPT) machen. Begründen Sie ihre Antwort kurz. [3 Punkte]

Lösung

Ein Problem ist FPT, falls ein Algorithmus das Problem in $\mathcal{O}(f(k) \cdot p(n))$ löst, wobei f eine berechenbare Funktion, k der Parameter, p ein beliebiges Polynom und n die Eingabelänge ist.

1. $f_1(n,k) = 2^k + n^2$:
   Das Problem ist fixed-parameter tractable, da $f_1(n,k) = 2^k + n^2 < 2^k \cdot n^2$. Hier ist also $f(k) = 2^k$ und $p(n) = n^2$.

2. $f_2(n,k) = 2^{k+\log n}$:
   Das Problem ist fixed-parameter-tractable, da $f_2(n,k) = 2^{k+\log n} = 2^k \cdot 2^{\log n} = 2^k \cdot n$. Hier ist also $f(k) = 2^k$ und $p(n) = n$.

3. $f_3(n,k) = 2^{k \log n}$:
   Das Problem ist nicht fixed-parameter-tractable, da $f_3(n,k) = 2^{k \log n} = (2^{\log n})^k = n^k$. Der Exponent von n wächst also mit k.

Lösungsvorschlag

Aufgabe 2. Approximationsalgorithmen: Set Cover [11 Punkte]

Das *Minimum Set Cover* Problem ist wie folgt definiert: Gegeben ist eine Menge X mit $|X| = n$ und eine Menge von Teilmengen $F = \{F_1, \ldots, F_m\}$ mit $F_i \subseteq X$ und $\bigcup_{F_i \in F} F_i = X$. Gesucht ist eine Teilmenge $S \subseteq F$ mit $\bigcup_{F_i \in S} F_i = X$ von kleinstmöglicher Kardinalität.

a. Sei $X = \{0,1,2,3,4,5,6,7,8,9\}, F = \{F_1, F_2, F_3, F_4, F_5\}$ mit
$F_1 = \{0,1,2,3\}$
$F_2 = \{2,3,4\}$
$F_3 = \{5,6,7\}$
$F_4 = \{5,8,9\}$
$F_5 = \{0,1,6,7\}$

Geben Sie ein Minimum Set Cover an. Begründen Sie, warum Ihr gefundenes Set Cover minimal ist. [3 Punkte]

Lösung

$S = \{F_2, F_4, F_5\}$
Da die Elemente 4 und 9 nur in jeweils einer Menge auftauchen, müssen diese Mengen (F_2 und F_4) in jedem Set Cover enthalten sein. Da noch nicht alle Elemente dadurch abgedeckt sind, muss noch mindestens eine weiter Menge gewählt werden. F_5 deckt alle verbleibenden Elemente ab.

b. Gegeben sei folgender Algorithmus zur Berechnung eines Set Covers:

Algorithmus 1 `ApproxMinSetCover(X, F)`

```
S ← ∅                                   ▷ Lösung
U ← X                                   ▷ Noch nicht abgedeckte Elemente
while U ≠ ∅ do
    Wähle T ∈ F, dass |T ∩ U| maximiert
    U ← U \ T
    S ← S ∪ {T}
return S
```

Geben Sie das Ergebnis von `ApproxMinSetCover` für X und F aus Teilaufgabe **a** an. Sollte bei der Auswahl von T Gleichstand zwischen zwei Mengen herrschen, wählen Sie die Menge, die in der Aufgabenstellung zuerst genannt wird. [2 Punkte]

Lösung

$S = \{F_1, F_2, F_3, F_4\}$

c. Sei k die Anzahl an Mengen, die für eine optimale Lösung benötigt werden. Zeigen Sie mittels vollständiger Induktion, dass am Ende der j-ten Iteration von `ApproxMinSetCover` gilt: $|U| \leq n(1 - 1/k)^j$. [4 Punkte]

Lösung

Induktionsanfang ($j = 0$): Nach Definition von U sind vor der ersten Iteration n Elemente nicht abgedeckt.
Induktionsschritt ($j - 1 \curvearrowright j$): Zu Beginn der j-ten Iteration gilt nach Induktionsvoraussetzung $|U| \leq n(1 - 1/k)^{j-1}$. Da eine optimale Lösung alle n Elemente mit k Mengen abdeckt, kann sie insbesondere auch die noch verbleibenden Elemente mit maximal k Mengen abdecken. Es gilt also: Eine optimale Lösung verwendet eine Menge, die mindestens einen Anteil $1/k$ der verbleibenden Elemente abgedeckt. Da `ApproxMinSetCover` die Menge auswählt, die die meisten Elemente abdeckt, wird auch mindestens ein Anteil $1/k$ der verbleibenden Elemente abgedeckt. Damit gilt am Ende der j-ten Iteration: $|U| \leq n(1 - 1/k)^{j-1} \cdot (1 - 1/k) = n(1 - 1/k)^j$

d. Zeigen Sie, dass `ApproxMinSetCover` maximal $k \ln n + 1$ Mengen benötigt, wenn eine optimale Lösung k Mengen benötigt.
Hinweis: $(1-1/n)^n < 1/e$ für $n \in \mathbb{N}$ [2 Punkte]

Lösung

Für die Anzahl der nach $k \ln n$ Iterationen verbleibenden Elemente ergibt sich nach Teilaufgabe **c**:

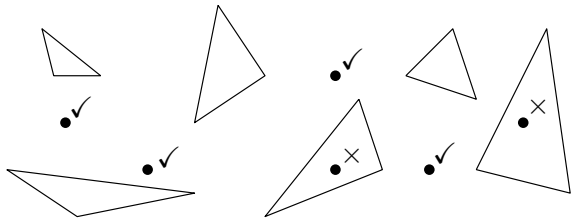
$$\begin{aligned} & n(1-1/k)^{k \ln n} \\ &= n((1-1/k)^k)^{\ln n} \\ &\leq n(1/e)^{\ln n} \\ &= 1 \end{aligned}$$

Das letzte verbleibende Element kann offensichtlich durch hinzufügen einer einzigen weiteren Menge abgedeckt werden.

Lösungsvorschlag

Aufgabe 3. Geometrische Algorithmen: Dreiecke und Punkte [8 Punkte]

Gegeben sei eine Menge S von n Dreiecken in der Ebene. Jedes Dreieck sei durch seine drei Eckpunkte p_1, p_2, p_3 definiert. Gehen Sie davon aus, dass die Seiten der Dreiecke sich nicht schneiden (oder berühren) und kein Dreieck in einem anderen Dreieck enthalten ist. Des Weiteren seien alle x- sowie y-Koordinaten der Punkte der Dreiecke paarweise verschieden. Zusätzlich sei P eine Menge von n Punkten in der Ebene.
Betrachten Sie dazu das folgende Beispiel:



a. Erweitern Sie den Sweepline-Algorithmus aus der Vorlesung, um alle Punkte aus P auszugeben, die außerhalb aller Dreiecke aus S liegen. Der Algorithmus soll höchstens $\mathcal{O}(n \log n)$ Zeit benötigen. Begründen Sie die Laufzeit Ihres Algorithmus. [6 Punkte]

Lösung

Wir erweitern den Algorithmus aus der Vorlesung zur Berechnung von Streckenschnitten, um für einen Punkt zu bestimmen, ob dieser innerhalb eines Dreiecks liegt. Hierzu verwenden wir eine vertikale Sweepline in Richtung aufsteigender x-Koordinaten.
Events: Events sind dann gegeben durch die Eckpunkte der Dreiecke aus S sowie der Punktemenge P. Wie in der Vorlesung verwenden wir eine Prioritätsliste Q für das Ermitteln des nächsten Events. Ist der nächste Event ein Eckpunkt eines Dreiecks aus S so unterscheiden wir dabei, ob es sich um den Anfangs-, Mittel- oder Endpunkt des Dreiecks handelt:

- Bei einem Anfangspunkt werden die beiden angrenzenden Segmente in den Sweepline-Status eingefügt.
- Bei einem Mittelpunkt wird das angrenzende Segment links des Punktes entfernt und das Segments rechts des Punktes eingefügt.
- Bei einem Endpunkt werden die beiden angrenzenden Segmente aus dem Sweepline-Status entfernt.

Ist der nächste Event ein Punkt aus P so ermitteln wir mit Hilfe unseres Sweepline-Status die beiden Segmente direkt über- und unterhalb des Punktes. Anschließend überprüfen wir, ob die beiden Segmente zum gleichen Dreieck gehören. Ist dies der Fall, so liegt unser Punkt innerhalb eines Dreiecks und wir gehen zum nächsten Event. Ansonsten wird der Punkt ausgegeben.
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 Dreiecksegmente sortiert nach ihrer relativen Ordnung auf der Sweepline.
Analyse: Zur Berechnung der Dreiecksegmente benötigen wir $\mathcal{O}(n)$ Zeit. Für das Einfügen und Abfragen aller Events in Q ($3n$ Eckpunkte aus S und n Punkte aus P) benötigen wir $\mathcal{O}(n \log n)$ Zeit. In jeder Iteration wird für einen der insgesamt $4n$ Events eine der folgenden Änderungen an T durchgeführt:

- Bei einem Dreiecksevent werden entweder zwei Segmente in T eingefügt (Anfangspunkt), ein Segment entfernt und eins hinzugefügt (Mittelpunkt) oder zwei Segmente entfernt (Endpunkt).
- Bei einem Punktevent werden zwei Suchanfragen an T gegeben.

Alle diese Operationen benötigen $\mathcal{O}(\log n)$ Zeit. Alle sonstigen Berechnungen (Test ob Punkt zwischen zwei Segmenten) benötigen $\mathcal{O}(1)$ Zeit. Damit liegt die Gesamtlaufzeit des Algorithmus bei $\mathcal{O}(n \log n)$.

b. Erweitern Sie Ihren Algorithmus aus Teilaufgabe **a** möglichst effizient, so dass Dreiecke nun auch vollständig in anderen Dreiecken enthalten sein können. Welche Auswirkungen hat dies auf die Laufzeit Ihres Algorithmus? [2 Punkte]

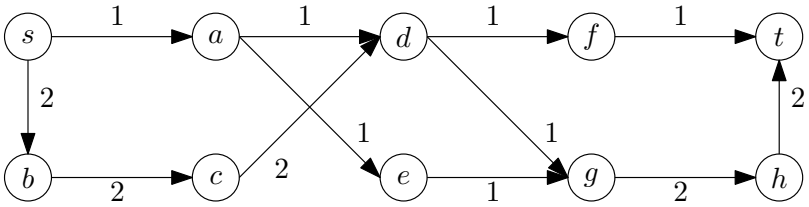
Lösung

Beobachtung: Ein Dreieck A liegt vollständig in einem anderen Dreieck B, genau dann wenn jeder Punkt der innerhalb von A liegt auch innerhalb von B liegt.
Aufgrund unserer Beobachtung genügt es für jeden Anfangspunkt eines Dreiecks zu testen, ob dieser innerhalb eines anderes Dreiecks liegt. Ist dies der Fall so kann das innere Dreieck ignoriert und die entsprechenden Eckpunkte aus Q entfernt werden. Die entsprechende Operation funktioniert analog zu den Punktevents aus Teilaufgabe **a**. Da alle benötigten Operationen eine Laufzeit von $\mathcal{O}(\log n)$ haben, ändert sich die Laufzeit des Algorithmus nicht.

Lösungsvorschlag

Aufgabe 4. Flussalgorithmen: Bipartite Matchings [13 Punkte]

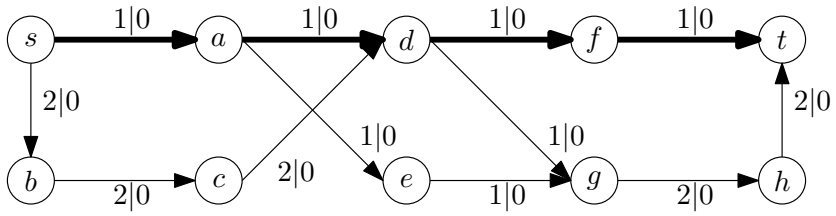
a. Betrachten Sie das unten abgebildete Flussnetzwerk. Die Kanten sind mit ihrer Kapazität beschriftet. Berechnen sie mit dem *Ford Fulkerson* Algorithmus den maximalen Fluss von Knoten s zu Knoten t. Augmentieren Sie jeweils den kürzesten Pfad im Residualgraphen. Geben Sie dabei für jeden Schritt den erhöhenden Pfad in Form einer Knotenliste und den Wert des erhöhenden Pfades an. Geben Sie zusätzlich den Wert des maximalen Flusses nach der Ausführung an.



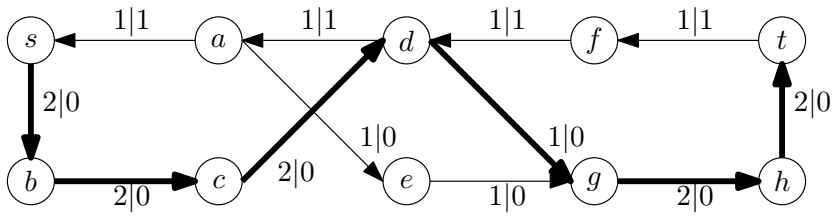
[3 Punkte]

Lösung

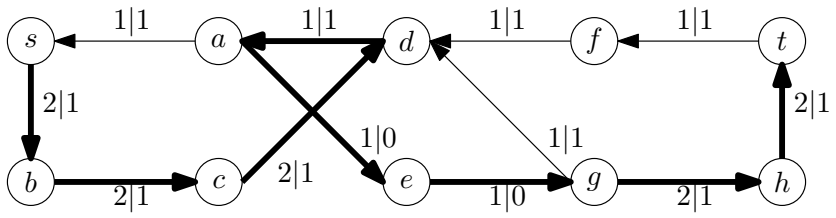
1. Erhöhender Pfad 1: $s \to a \to d \to f \to t \Rightarrow$ Wert 1



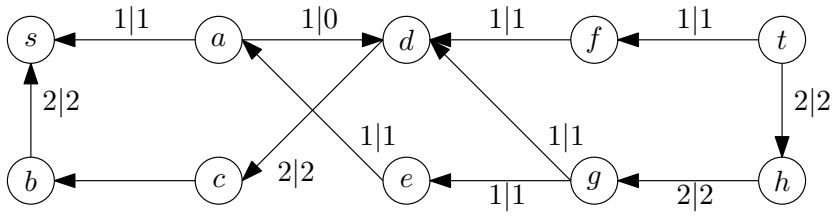
2. Erhöhender Pfad 2: $s \to b \to c \to d \to g \to h \to t \Rightarrow$ Wert 1



3. Erhöhender Pfad 3: $s \to b \to c \to d \to a \to e \to g \to h \to t \Rightarrow$ Wert 1



Folgendes Flussnetzwerk zeigt den maximalen Fluss:

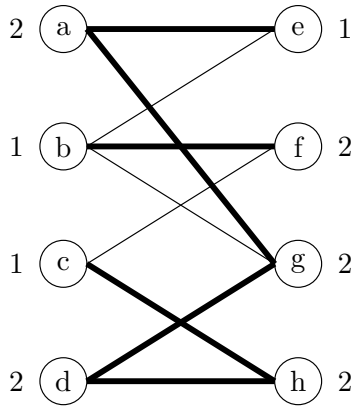


Wert des maximalen Flusses: 3

Anmerkung: Die gezeigten Residualgraphen sind nicht Teil der geforderten Lösung, sondern dienen ausschließlich zur Verdeutlichung derselben

b. Ein bipartiter Graph sei ein Graph dessen Knotenmenge V in zwei disjunkte Teilmengen U, W zerlegt werden kann, so dass keine zwei Knoten aus der gleichen Teilmenge adjazent sind. Gegeben sei ein bipartiter Graph $G = (U \cup W, E, b)$ mit natürlichen Knotenkapazitäten $b(v) \in \mathbb{N}$ für $v \in U \cup W$. Eine Kantenmenge $M \subseteq E$ sei ein bipartites b-Matching, wenn kein Knoten $v \in U \cup W$ zu mehr als $b(v)$ Kanten aus M inzident ist.
Ein b-Matching M sei maximal, wenn es kein b-Matching N mit $|N| > |M|$ gibt.
Geben Sie ein maximales b-Matching für den unten abgebildeten Graphen an. Begründen Sie außerdem, warum Ihr b-Matching maximal ist. [2 Punkte]

Lösung



Die angegebene Lösung ist maximal, da alle Knoten auf der linken Seite voll saturiert sind. Es kann also keine Lösung mit mehr gematchten Kanten existieren.

c. Geben Sie einen Algorithmus an, der mittels ganzzahliger maximaler Flüsse ein maximales b-Matching für einen bipartiten Graphen $G = (U \cup W, E, b)$ berechnet. Geben Sie hierfür insbesondere an, wie der gegebene Graph in ein Flussnetzwerk $H = (V', E', c)$ transfomiert wird. Geben Sie weiterhin an, wie der von Ihnen gefundene Fluss in ein maximales b-Matching umgewandelt wird. [3 Punkte]

Lösung

Der Algorithmus wandelt zunächst den gegebenen bipartiten Graphen $G = (U \cup W, E, b)$ wie folgt in ein Flussnetzwerk $H = (V', E', c)$ um:

- $V' = U \cup W \cup \{s, t\}$
- $E' = \{(u, w) | u \in U, w \in W, \{u, w\} \in E\} \cup (\{s\} \times U) \cup (W \times \{t\})$
- $\forall e \in E'$ setze $c(e) = \begin{cases} b(u) & e = (s, u) \\ b(w) & e = (w, t) \\ 1 & \text{sonst} \end{cases}$

Anschließend wird auf dem Flussnetzwerk H ein maximaler Fluss (z.B. via Ford-Fulkerson) berechnet. Der resultierende Fluss wird dann in ein maximales b-Matching umgewandelt, in dem alle saturierten Kanten von U nach W zum Matching hinzugefügt werden.

d. Beweisen Sie, dass Ihr Algorithmus aus Teilaufgabe **c** korrekt ist. [5 Punkte]

Lösung

- Unter der Transformation aus Teilaufgabe **c** gibt es eine bijektive Abbildung zwischen ganzzahligen Flüssen und bipartiten b-Matchings.
  Beweis: Für die eingehenden Kanten eines Knotens $u \in U$ gilt $c((s,u)) = b(u)$. Die ausgehenden Kanten entsprechen denen in E und haben Kapazität 1. Somit gilt nach Kapazitätskonformität und Flusserhalt, das ein Knoten $u \in U$ nicht mehr als $b(u)$ ausgehende saturierte Kanten besitzt. Analog gilt für Knoten $w \in W$ das diese nicht mehr als $b(w)$ eingehende saturierte Kanten besitzen. Somit ist für einen ganzzahligen Fluss das daraus berechnete Matching gültig. Entsprechend erhalten wir aus einem gültigen Matching einen ganzzahligen Fluss indem wir für Knoten $u \in U$ bzw. $w \in W$ den Fluss der eingehenden bzw. ausgehenden Kante auf die Anzahl der gematchten Kanten des Knotens setzen.

- Der Wert des Flusses stimmt mit der Kardinalität des entsprechenden Matchings überein.
  Beweis: Nach Flusserhalt entspricht der von s ausgehenden Fluss der Anzahl der saturierten (gematchten) Kanten zwischen U und W. Somit ist der Wert eines ganzzahligen Flusses gleich der Kardinalität des entsprechenden Matchings.

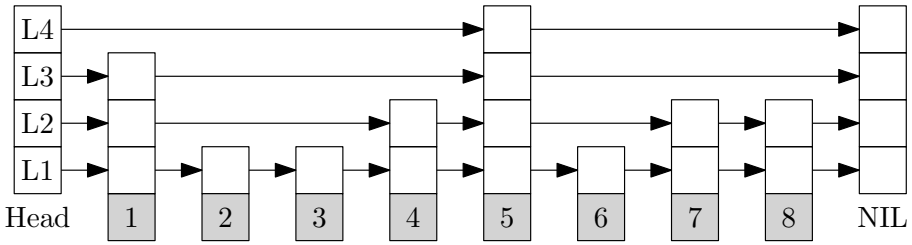
- Also liefert ein maximaler ganzzahliger Fluss ein maximales b-Matching.

□

Lösungsvorschlag

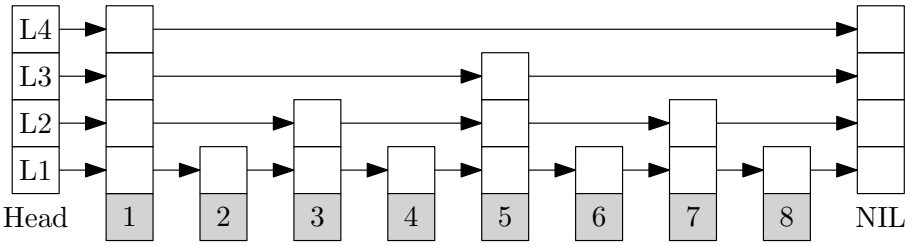
Aufgabe 5. Randomisierte Algorithmen: Skip Lists [9 Punkte]

Eine Skip List ist eine probabilistische Datenstruktur zur Speicherung einer geordneten Folge von n Elementen. Eine Skip List besteht aus mehreren Schichten. Die unterste Schicht (Schicht 1) ist eine geordnete (einfach) verkettete Liste, die alle Elemente enthält. Jede höhere Schicht bildet "Abkürzungen" für die Schichten darunter. Ein Element in Schicht i hat dabei Wahrscheinlichkeit p in Schicht $i+1$ aufzutreten. Zusätzlich besitzt die Liste ein Kopfelement (Head) und Endelement (NIL).
Betrachten Sie dazu das folgende Beispiel:



a. Konstruieren Sie eine Skip List für die Elemente $\{1,2,3,4,5,6,7,8\}$. Verwenden Sie als Ersatz für die Wahrscheinlichkeit p die Funktion $\texttt{Next}(x) := (x \mod 2^i = 1)$, die angibt ob ein Element x aus Schicht i in Schicht $i+1$ gelangt. Stoppen Sie dabei, sobald eine Schicht nur noch ein Element besitzt. [2 Punkte]

Lösung



b. Für die Suchoperation einer Skip List sei folgender Pseudocode gegeben:

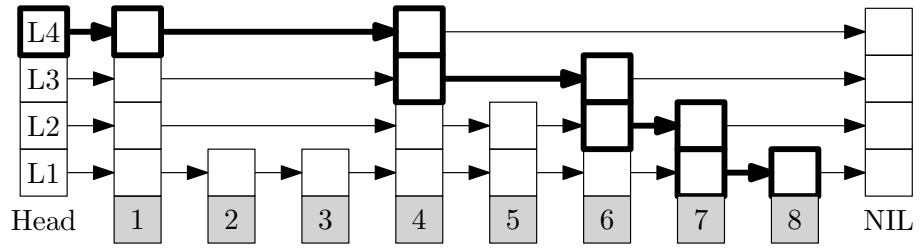
Algorithmus 2 `Suche(L: Skip List, k: Suchschlüssel)`

```
x ← L.head
for i ← L.max_level downto 1 do
    while x.forward[i].key < k do
        x ← x.forward[i]
x ← x.forward[1]
if x.key = k then return x.value
else return failure
```

Zeichnen Sie in die unten abgebildete Skip List den Suchpfad für das Element 8 ein.

[1 Punkt]

Lösung



c. Geben Sie eine geschlossene Formel $\#(n,l)$ für die erwartete Anzahl an Elementen in Schicht l an. [1 Punkt]

Lösung

Da jedes Element Wahrscheinlichkeit p besitzt, in die nächsthöhere Schicht zu gelangen, gilt für die erwartete Anzahl an Elementen pro Schicht

$$\#(n,l) = np^{l-1}$$

d. Geben Sie eine geschlossene Formel $L(n)$ für den Index der Schicht mit erwartet $1/p$ Elementen an. [2 Punkte]

Lösung

Nach Teilaufgabe **a** gilt $\#(n,l) = np^{l-1}$.

$$\begin{aligned} np^{L(n)-1} &= \frac{1}{p} \\ n &= p^{-L(n)} \\ \lg n &= L(n) \lg \frac{1}{p} \\ L(n) &= \lg_{1/p} n \end{aligned}$$

e. Im Folgenden betrachten wir den Suchpfad der in Teilaufgabe **b** definierten Operation in umgekehrter Richtung (d.h. ausgehend von Schicht 1). Die Richtung des Suchpfades ist damit nach links und nach oben.
Geben Sie eine geschlossene Formel für die erwartete Länge eines Suchpfades an, der von Schicht 1 bis Schicht k verläuft. Die Länge des Suchpfades sei dabei durch die Anzahl der Linksbewegungen definiert. Gehen Sie davon aus, dass der Anfang der Liste nie erreicht wird.
[3 Punkte]

Lösung

Wir nehmen an, das die Schicht eines Elements erst dann bestimmt wird wenn wir es während unserer Traversierung betrachten.
Da jedes Element mit einer Wahrscheinlichkeit p eine Schicht aufsteigt, müssen wir auf jeder Schicht erwartet $1/p$ Linksbewegungen durchführen bis wir in eine höhere Schicht gelangen. Aufgrund der Linearität des Erwartungswertes, ist der Erwartungswert der Summe der Linksbewegungen auf allen Ebenen gleich der Summe der Erwartungswerte auf den einzelnen Ebenen. Von Schicht 1 bis zu Schicht k ergibt sich damit eine erwartete Suchpfadelänge von $(k-1)/p$.

Lösungsvorschlag

Aufgabe 6. Stringalgorithmen: Permutiertes LCP Array [9 Punkte]

Im Folgenden betrachten wir Φ und das *permutierte LCP Array* (PLCP) eines Textes T, das wie folgt definiert ist:

$$\Phi[i] = \begin{cases} SA[SA^{-1}[i]-1] & \text{wenn } SA^{-1}[i] \neq 1 \\ 0 & \text{sonst} \end{cases}$$

$$PLCP[i] = \begin{cases} |lcp(S_i, S_{\Phi[i]})| & \text{wenn } \Phi[i] \neq 0 \\ -1 & \text{sonst} \end{cases}$$

Zur Erinnerung: $LCP[i] = \begin{cases} |lcp(S_{SA[i-1]}, S_{SA[i]})| & \text{wenn } i \neq 1 \\ -1 & \text{sonst} \end{cases}$

a. Bestimmen Sie für $T = \texttt{banana}$ die Arrays SA^{-1}, Φ und $PLCP$ [5 Punkte]

i	S_i	$SA[i]$		$LCP[i]$	
1	banana	6	a	−1	a
2	anana	4	ana	1	ana
3	nana	2	anana	3	anana
4	ana	1	banana	0	banana
5	na	5	na	0	na
6	a	3	nana	2	nana

Lösung

i	$SA^{-1}[i]$	$\Phi[i]$	$PLCP[i]$
1	4	2	0
2	3	4	3
3	6	5	2
4	2	6	1
5	5	1	0
6	1	0	−1

b. Zeigen Sie, dass $PLCP[i] = LCP[SA^{-1}[i]]$. Sie dürfen dafür die Sonderfälle in den obigen Definitionen ignorieren. Betrachten Sie also nur die Fälle, für die gilt: $SA^{-1}[i] \neq 1$. [2 Punkte]

Lösung

$$LCP[SA^{-1}[i]] = |lcp(S_{SA[SA^{-1}[i]-1]}, S_{SA[SA^{-1}[i]]})| = |lcp(S_{\Phi[i]}, S_i)| = PLCP[i]$$

c. Vervollständigen Sie den folgenden Algorithmus zu einem Linearzeitalgorithmus zur Berechnung des PLCP, indem sie die ausgelassenen Zeilen ausfüllen. [2 Punkte]

Lösung

Algorithmus 3 `PLCP` (T, SA, SA^{-1}) mit $|T| = n$

$h \leftarrow 0$
$PLCP \leftarrow$ Array der Größe n
for $i = 1, \ldots, n$ **do**
  if $SA^{-1}[i] \neq 1$ **then**
    $j \leftarrow \boxed{SA[SA^{-1}[i]-1]}$
    while $T[i+h] = T[j+h]$ **do**
      $h{+}{+}$
    $\boxed{PLCP[i] \leftarrow h}$
    $h \leftarrow \max(0, h-1)$
  else
    $PLCP[i] \leftarrow -1$
return $PLCP$